FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Peer-to-peer Backup Application
Student:	David Košťál
Supervisor:	Ing. Jiří Dostál, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2023/2024

Instructions

Backups are an important problem that is often overlooked because of their complexity. One way to simplify the process is to create a user-friendly application that automatically creates an encrypted backup of one's data to another user's computer.

The goal of this thesis is to work towards that goal and explore creating a peer-to-peer backup application. It will allow its users to back up their data to other users while simultaneously using some of their free disk space to store backups of others in exchange. The implementation will have two parts – a client application that will create backups and handle the transfer of data and a central server that will facilitate the discovery of clients. The server only has access to the metadata of clients and not the data that is being backed up.

1. Review existing similar software, e.g., CrashPlan Home or BuddyBackup.
2. Make a threat model.
3. Design a security architecture and protocols.
4. Implement a simple, functional application to demonstrate the concept.
5. Client-side application features: backups to an automatically selected peer for users in the same network segment, backup encryption, and authentication. (Support for backups to any internet-connected computer is a possible future extension that requires implementing NAT traversal).
6. Server-side application features: client discovery and backup management.

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 3 January 2023 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Peer-to-peer Backup Application

David Košťál

Department of Computer Systems
Supervisor: Ing. Jiří Dostál, Ph.D.

May 11, 2023

Contents

List of Figures

List of Tables

I would like to thank my supervisor, Ing. Jiří Dostál, Ph.D., for his guidance and helpful advice throughout the whole process. I'd also like to thank my friend ∼Nebula∼ for working with me on pluto, the project that started this thesis, as well as his help with teaching me Rust. Last but not least, I would like to thank my friends and especially my family for their invaluable support during my whole studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 11, 2023

Abstract

This bachelor's thesis explores creating a peer-to-peer backup application that allows users to create a backup to other randomly selected users — in exchange for some of their free disk space. The theoretical part of this thesis involves analyzing the requirements, creating the design, and examining solutions to its deficiencies. The implementation part features a functional proof-of-concept application that showcases the concept.

Keywords peer-to-peer data backup, client application, server application, security analysis, peer-to-peer communication, data encryption, Rust

Abstrakt

Tato bakalářská práce zkoumá možnost vytvoření peer-to-peer zálohovací aplikace, která umožňuje jejím uživatelům vytvoření zálohy k ostatním náhodně vybraným uživatelům, výměnou za část volného místa na disku. Teoretická část této práce zahrnuje analýzu požadavků, vytvoření návrhu a diskuzi nad řešením jeho nedostatků. Implementační část obsahuje funkční ukázkovou implementaci, která demonstruje koncept této technologie.

Klíčová slova peer-to-peer zálohování dat, klientská aplikace, serverová aplikace, bezpečnostní analýza, peer-to-peer komunikace, šifrování dat, Rust

Abbreviations

AES	Advanced Encryption Standard
API	Application Programming Interface
CA	Certificate authority
CDC	Content Defined Chunking
CDN	Content Delivery Network
CSPRNG	Cryptographically secure pseudorandom number generator
CSS	Cascading Style Sheets
DoS	Denial of service
GCM	Galois/Counter Mode
HKDF	HMAC-based Extract-and-Expand Key Derivation Function
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ICE	Interactive Connectivity Establishment
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
JSON	JavaScript Object Notation
KDF	Key derivation function
NAT	Network address translation
PKI	Public key infrastructure
RSA	Rivest-Shamir-Adleman
STUN	Session Traversal Utilities for NAT
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TRNG	True random number generator
TURN	Traversal Using Relays around NAT
UI	User interface
UPnP	Universal Plug and Play

Chapter 1
Introduction

People often don't pay enough attention to making proper backups, with one of the reasons being that they take significant effort. However, data loss can have disastrous consequences. There are many ways one can lose data. Some examples include natural disasters, theft, malware, or simply hardware failure. Even larger companies and professional environments have problems with insufficient or unreliable backups [1, 2]. Home users and small companies often pay even less care to back up their data.

In this thesis, I will research a solution that has the potential to make backups simpler and less expensive: a peer-to-peer backup application. Its function is to back up local data to other automatically selected users — and provide free disk space for backups of those users in return.

A peer-to-peer solution is more affordable than commonly used solutions like physical media (such as external hard drives) and cloud services. It could also be easier to set up and use. However, this method also has disadvantages, such as concerns about backups being available for restoration, concerns about privacy, and the complexity of finding peers. I will mitigate some of these problems, with a particular focus on security risks, and discuss the solutions for other issues.

I originally got the idea to create an application for peer-to-peer backups in a video by the YouTube channel Linus Tech Tips [3]. After that, I started working on the project as an open-source application with a friend — in our free time [4].

While doing that, we realized that our scope is large and it is a complex problem that would justify writing a bachelor's thesis. Thanks to the project, we gained a lot of experience, and I can reuse some of it in this thesis.

I want to pursue this topic because the result could be helpful to me, my surroundings, and the wider community. Solving security challenges and creating useful programs are some of my personal interests.

Chapter 2
Goals

In this chapter, I will describe the goals and non-goals of this thesis in further detail.

My thesis aims to create foundations for developing a user-friendly version of said peer-to-peer backup application. The first chapter of the theoretical part will be dedicated to researching existing solutions in this area. That will be followed by performing an analysis to create the application. That will include making a threat model, designing the security architecture, and developing protocols and data storage formats.

The implementation part will entail creating a proof-of-concept implementation consisting of a client and a server component. I will compare available technologies and pick the most suitable ones. The client-side application needs to support creating simple backups for users in the same network segment. Those backups will need to be correctly encrypted and authenticated.

The reason for the initial version only supporting backups in the same network segment is that trying to bypass various networking restrictions (such as NATs or firewalls) is a nontrivial problem, but one that already has proper solutions — which are widely used and would not add much to researching this topic.

The server component will ensure that clients can discover peers and find a backup partner. It will also keep track of created backups and the computers that store them so that the backups can be later easily restored. The server will be centralized but cannot access unencrypted backup content.

Due to time constraints, I will not be able to create a robust, production-ready application. It will also be necessary to exclude some nice-to-have features, such as a polished user interface or automatic continuous backups with file watching.

2.1 Client functionality

After the first start, the client will give the user the option to either restore an existing account or register a new one. Registering will create a new identity while using an existing one will allow the user to restore a previous backup.

When creating a backup, the client application will be able to gather files from a folder and find the changes that occurred since the last backup. The changed data will be packaged, encrypted, and transferred to a peer — which the server will automatically

assign. The client will also allow viewing the state of backups and display additional logs.

The client will naturally also be able to receive backups from other clients. It connects with peers directly and transfers/receives the encrypted backup data. The application will keep track of how much data was received from each peer. When restoring a backup from a fresh installation, the client will request data about peers from the server, try to connect to all computers that store our data, and download the files.

2.2 Server functionality

All clients are going to connect to a central server. It will keep a database of all the clients and peers that store their backed up data to enable restoring backups.

Its other function is to match clients that are requesting to create a backup. It is going to keep track of requests for backups and match peers based on how much space they are requesting.

Chapter 3
Existing solutions

In this chapter, I will examine existing similar services in this space, as well as other relevant systems.

In my research, I discovered two services that have functionality comparable to the goals of this thesis, both of which are no longer operating. I will also look at other software worth investigating.

3.1 CrashPlan for Home

The first comparable piece of software was CrashPlan for Home by Code42. It was a full-featured backup application supported on Windows, Mac, and Linux. Users were able to make local, offsite, and cloud backups. Local and offsite backups were a free feature for all users. Cloud backups were a subscription service that enabled storing an unlimited amount of data on a central server using a CrashPlan user account. Local backups allowed users to use their external media, such as hard drives [5].

The offsite backup destination is the most interesting feature for the purposes of this thesis. It allowed backing up to a friend's computer by getting a unique code from them [6]. CrashPlan for Home was discontinued on October 23, 2018 [7].

3.1.1 Backup

CrashPlan used real-time file watchers to watch changes continuously. It also performed deduplication on parts of files (blocks). These blocks were compressed and subsequently encrypted and transmitted to the destination. Users with a paid subscription were able to pick separate sets of files to include in backups and configure different destinations for each set. There were extensive configuration options for selecting how many historical versions to keep and which files to exclude.

The application also allowed for *backup seeding*, meaning users could transfer initial backed-up data to the destination using physical media and only use the internet for subsequent changes [8, 9].

3.1.2 Security

CrashPlan for Home had various data encryption configurations, all using the Blowfish algorithm. 128-bit keys were used for users without a subscription, with 448-bit keys reserved for paying customers. Users could pick from three types of encryption key storage. The default one was called standard, where the data encryption key was stored on Code42's servers for an easy restoration option.

A more secure configuration was offered, where the key stored on the server was also encrypted with an additional *archive* password. There also was an option of a security question, which was a way to restore a forgotten archive password. The most secure option was using a custom key, which was not stored on any servers. It was possible to use a passphrase, generate a new random key, or import an existing one [10, 11].

3.1.3 Summary

CrashPlan for Home was a powerful backup application that offered a lot of functionality and flexibility but might not always have been the easiest to use. Peer-to-peer backup functionality wasn't the headline feature, but it was a good option for users who wanted to configure it.

3.2 BuddyBackup

The second comparable service was BuddyBackup, a free Windows program launched in 2010 [12]. The application's main focus was peer-to-peer data backup to friends (called *buddies*). However, it also allowed backups to USB drives or other external locations as virtual buddies. Peers were added using a username they picked when creating a BuddyBackup account. The program had an advanced integration with Windows Explorer [13]. It was discontinued on May 31, 2022 [14].

3.2.1 Backup

This application, similar to CrashPlan, also uses real-time file scanning and allows continuous backups. With BuddyBackup, the user will configure a set of files to back up, maximum backup space (upper limit for how much data can be backed up), and backup safety (number of copies that are spread among different buddies). All buddies combined will then be allowed to back up the same amount of data as the user's backup space. Adding a buddy will automatically allow them to make backups.

The application also keeps local backups apart from just sending data to peers. That allows for fast restoration of earlier versions of files (which are also backed up to buddies) or files that have been deleted altogether. Backups are incremental, so only byte-level differences are transferred.

Restoration of files was made easier by including a guest mode, which users could run on their buddies' computers and access their files saved on that device. An option for seeding backups is also offered here [13].

3.2.2 Networking

BuddyBackup had a number of methods to allow users to connect to each other. If a direct connection didn't succeed, BuddyBackup tried to open ports on the router automatically, using the UPnP protocol. If UPnP isn't enabled or users are behind

a NAT or have other networking restrictions, the application offered a feature called Buddy Connect. This feature allowed users without network restrictions to act as a relay for their buddies or even all users, depending on the configuration. Thanks to that, even users without direct options could connect and transport backups [13].

3.2.3 Security

Encryption keys were stored on the central server, encrypted with the user's password. By default, user passwords were also saved on the server (for easy recovery), but this behavior could be disabled. The AES-256 algorithm was used for encrypting backed-up data. For authenticating buddies, asymmetric cryptography was used, in particular, the RSA-2048 algorithm [13].

3.2.4 Summary

BuddyBackup offered a comprehensive and free solution for peer-to-peer backups. It also seems to have been user-friendly and easy to configure.

3.3 Other software

3.3.1 Pluto

The project that my friend and I started as a predecessor to this thesis was called pluto. Its goal was to be a user-friendly application for peer-to-peer backups, but the implementation never reached a functional stage. However, working on the project was a valuable source of experience [4].

For pluto, I designed a backup storage format (explained in detail in chapter 5) that allows for efficient packaging of files, and their encryption and compression. The code for this format is adapted and brought into this thesis as well.

3.3.1.1 Networking

Its networking solution of choice is a publish-subscribe protocol called MQTT, which is especially popular in low-power applications such as Internet of Things. The protocol is based around publishing and subscribing to path-like *topics*. MQTT server, called *broker*, automatically delivers a message to all clients which are subscribed to a certain topic whenever a message is posted there. Subscriptions to topics can also include wildcards, so a client can subscribe to all topics under a certain path. The protocol is message-based and doesn't have a request-response flow [15].

This has some advantages — e.g., the message handling server is separate from the application server, and broadcast messages are straightforward. But the complexity of using MQTT in our desired way outweighs the advantages, which leads to my decision to use a different networking solution in this thesis.

3.3.2 Syncthing

Syncthing is a powerful open-source application for peer-to-peer continuous data synchronization. Its primary purpose is not to create backups, so it doesn't have all features typical for such usecases. However, it supports file retention and, as a relatively recent feature, experimental end-to-end encryption of synchronized data, making it suitable

for mirroring data to peers that are not fully trusted. It also features a mechanism for automatically finding the best network paths between peers, including using relays — which could be an inspiration for later versions of the application created by this thesis [16, 17].

3.3.3 Nextcloud Backup app

Nextcloud is a self-hosted open-source solution for file storage and collaboration, similar in functionality to the likes of Google Drive. It offers a feature in their Backup app, to create an encrypted backup of an entire Nextcloud instance to a different instance [18]. This can make it easy for individual users running their own Nextcloud installations to create backups to, for example, their friends who also use Nextcloud. However, the goal of this solution is different from the goal of this thesis, which is aimed at individual desktop users.

3.3.4 Storj

Storj is a service that offers distributed object storage (with an Amazon S3-compatible API), mainly to companies looking for an alternative to traditional cloud object storage. The saved files are encrypted using AES-256-GCM and split up into parts using Reed-Solomon erasure coding for redundancy. After that, all the parts are sent to distributed voluntary nodes, which get paid for storing files and used bandwidth [19].

The idea of using erasure codes for saving files to a distributed storage can be helpful for future versions of this project. The ratio used by Storj is 29 data blocks + 51 parity blocks for a total of 80 blocks stored. Reed-Solomon coding enables restoring data with, in this case, up to 51 blocks of either type lost [19, 20].

3.3.5 Restic

Restic is a popular open-source solution for creating backups. It doesn't directly support backups to other peers using the application, but one has the choice of many backup backends, including local folders, SFTP, or Amazon S3-compatible cloud services [21].

I am mentioning its project because I took inspiration from its efficient way of packaging and storing backup data, which resulted in some of the ideas described in section 5.3 [22].

3.4 Current situation

I looked at two main services that used to provide peer-to-peer backup functionality, as well as other related software. None of the explored software supports backups to random peers, which is one of the main goals of this thesis. At present, there are no working programs fulfilling the same goals — that I am aware of.

It's worth noting that there are many other ways for users to create safe offsite backups with robust applications such as the aforementioned restic. However, that requires significantly more effort than simply installing a program at both ends or even just installing it for oneself and using an automatically selected peer.

In conclusion, the idea that this thesis pursues is relatively unique. There is a lot of inspiration to be taken from the existing projects. However, some features, like backups to automatically selected peers, must be designed from scratch.

Chapter 4
Analysis

In this chapter, I will go over the analysis performed prior to creating the core design and explain why some key choices were made. The bulk of final design decisions is described in chapter 5.

4.1 Threat model

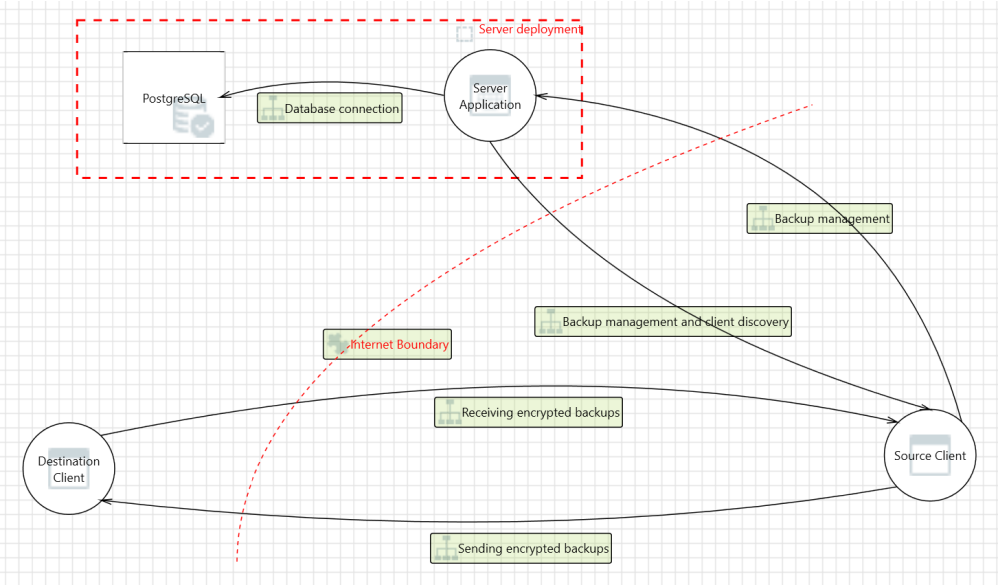

Figure 4.1 Threat model diagram.

An essential part of creating a secure system is designing a threat model and considering possible mitigations for potential security vulnerabilities. The tool used to help model vulnerabilities was Microsoft Threat Modeling Tool, using the STRIDE model [23]. The full report is in Appendix A. This section includes the key insights.

The application consists of two main components — the client and the server.

Clients communicate with each other to transport backups and also communicate with the server to facilitate client discovery and store information essential for restoring backups. The data that mainly need protecting are the user backups.

4.1.1 Communication

The server is considered to be a trusted source of data. All clients are considered inherently untrusted by the server and other clients. However, the local storage that the client application relies on is considered trusted — if a threat actor gained access to the files of a user account, they could read or modify the backup data directly. Attacking the backup application would not create any additional threats. Similarly, the database used by the server application is trusted — it's deployed in a controlled and isolated environment. For that reason, client-server and peer-to-peer communication are the data paths that need protecting. The mechanisms used to protect communication are explained in section 5.2.

4.1.2 Backup data

Concerning backups, the key goals are to ensure the integrity and confidentiality of the data. Maximum data availability is not a goal of the current model application. The protection of peers from receiving illegal or otherwise harmful data sent by attackers bypassing the encryption is also not considered entirely solved. Still, it uses a mitigation described in subsection 5.4.3. Solutions for these issues are further explored in chapter 7.

The integrity and confidentiality of backups are ensured by using a storage format explained in Figure 5.6. It encrypts the contents of files (including their hashes, which are used for storage deduplication purposes — with one exception explained in the following paragraphs), names, sizes, and other metadata of files. Deduplication means that we try to minimize the number of duplicate data, primarily across file changes to facilitate incremental backups, but also across different files.

Counts or sizes of individual files are also not revealed directly. However, the storage format can disclose some non-critical information. The absolute size of backups is inevitably revealed. The incremental nature of the backup will also reveal the magnitude of files modified over the time period since the last backup. Due to the side effects of the storage format design (packfile header sizes and index file sizes hinting at the number of blobs), the receiving peer or an intercepting party can make an estimate about the nature of the backup, mainly the rough file count.

And while the blob hashes, which could in some cases reveal the contents of the files, are normally encrypted, there is one exception. After a backup is completed, the client sends a hash of the backup root directory to the server. It's unlikely to reveal any information in real usage, but it's a potential weak spot. However, it could be mitigated with extra encryption.

4.2 Backup storage format

I had several requirements for the backup storage format in this application. It needs to be secure and efficient. And importantly, it must be designed for incremental backups with incremental file transport, meaning that the application will only add (or delete) files rather than directly modify existing ones. We don't have direct access to the filesystem on the other peer, and the connection might not be very reliable, so I have chosen to work with entire files only.

The format needs to store the entire folder user wishes to back up, including the structure, the contents of the files, and the metadata (such as modification times). For efficiency, we also only want to store the minimum amount of changes since the last backup. Saving entire changed files would not achieve that goal, so we will need to work not with entire files but their parts. However, transferring only parts of files individually would be inefficient, given that they can be small.

The storage format of the backup program restic already solves a large portion of this problem. Files are split into chunks, each directory and file data is serialized, and all of these pieces of data are stored in bigger packaged files called *Packs* [22]. The backup storage format in this application uses ideas from restic. There are however many differences, such as the index being simpler and strictly incremental (data from newer index files always overwrites the older ones). This format is described in 5.3.

4.3 Network architecture

With the overall architecture of the network, there was a choice between using a fully decentralized network or a partially decentralized one employing a central server. While a fully decentralized model is resilient, and there are no operating costs for a central server, it comes with numerous problems and is challenging to design. Using a central server will lead to better reliability and a simpler design.

Considering networking protocols for this application, there's a need to support transactional requests like logging in — but also respond to real-time events like connecting with a peer that requested a backup. I also want to consider using standard, widely used protocols when possible to make the implementation simpler and reduce the risk of security problems.

For transactional requests from client to server, HTTPS is used for its extreme popularity. WebSocket is used for real-time notifications from server to client. WebSocket is a protocol (based on TCP) that is also widely used in browsers and thus has extensive support in network infrastructure (like CDNs and reverse proxies) and software libraries. It's message based, making it well-suitable for this application. The authentication and encryption of the traffic between clients and the server are ensured by TLS, with the server having a set domain and a valid certificate.

4.3.1 Peer-to-peer connections

The core functionality of this software is to make backups to other users. Therefore, we need a way to transport data between individual users. The simplest way to implement this would be to relay all peer-to-peer communication through central servers. However, this comes with several drawbacks: operating this service would use a large amount of bandwidth, users would likely experience slower speeds, and the server would have access to more data than is required for running the service.

Therefore, it's desirable to allow peers to transport backups directly. This is generally a relatively complicated problem, but our requirements allow us to only be able to communicate within the same network segment and disregard real-world networking problems like NATs or firewalls. We can then connect peers directly over a standard TCP socket to a local IP address instead of a more complicated discovery protocol. The details of this problem and the possible solutions are explained in chapter 7.

However, even with this simplification, it's still necessary to establish a connection and properly secure it from tampering. There is no standard transport security for

peer-to-peer communication with random clients. PKI is not a viable option in a P2P environment. For that reason, I've designed a protocol that solves this problem. This protocol is also based on WebSocket to make the implementation easier.

The protocol only supports transporting encrypted backup files and acknowledgment messages. The design is simple — it only ensures reliability and is not designed for best efficiency. Security is ensured by wrapping the file in a message, which adds a replay protection header and a signature on the data header. The message is signed by the private key of the sending peer, ensuring that we are communicating with the party we have negotiated. The details are explained in section 5.2.

4.4 Account management

The applications I investigated in chapter 3 use a traditional account system. It's a working model, but the user experience can be improved while preserving maximum privacy. Having a password reset option would necessarily mean that the backup key would need to be saved on the central server. However, the server should not be able to decrypt the backup.

Having a password that cannot be reset is also problematic. Forgotten passwords are a common occurrence — according to [24], the action of resetting a password on a website was done 15% as often as logging in normally. In this case, this problem will be even more apparent. The password would be utilized rarely, only when backups need to be restored. And the consequences of forgetting the password are devastating since the backup couldn't be restored.

Another risk associated with traditional user accounts are weak passwords. A password policy is always a tradeoff between password security and memorability. For offline cracking of potentially very sensitive data, a strict policy would likely be necessary. An alternative approach to deriving a key from a password using a KDF (and eliminating the possibility of offline cracking) would be to store the encryption key using a hardware security module deployed on the server, which requires a password to release the key. That, however, needs users to have more trust in the central server and also is out of scope for this thesis.

For these reasons, I've chosen to use a randomly generated value (the *root secret*) as a basis for the backup encryption key, as well as authentication keys. The users are forced to write down that value in a safe place, so it won't be forgotten when backups need to be restored. However, just showing the root secret to the user in a usually formatted way, such as hexadecimal, is annoying and prone to errors when writing it down.

To help solve that, the root secret is presented to the user in the form of a BIP39 mnemonic. BIP39 is a user-friendly format to display entropy, designed for the Bitcoin project. For a 256-bit value, like the root secret, it consists of 24 English words. Those encode the value itself and a checksum. Using common words from a fixed wordlist — which excludes easily mistaken words — will significantly lower the risk of users mistyping the phrase and being unable to restore backups later [25].

In conclusion, having a single root secret serve as a user identity is positive for both security and user convenience. Random generation ensures that a suboptimal password will never weaken backup encryption. Since the root secret never leaves the client device, it also guarantees that the central server or other users cannot decrypt any data.

Chapter 5
Design

In this chapter, I will explain the fundamental design principles behind essential aspects of the application — mainly around backup key management and the security, packaging, and transport of backups.

5.1 Identity management and authentication

The software's identity management is based on having a single random secret value called the *root secret*. This value is generated by a TRNG (true random number generator) on the first start and saved to disk (see Figure 5.1).

After that, and on every subsequent start, the root secret is used to seed a CSPRNG (cryptographically secure pseudorandom number generator) based on the ChaCha20 cipher. ChaCha20 is a well-respected stream cipher (used, for example, in TLS 1.3 [26]), making it very suitable as a base for a PRNG that is both fast and cryptographically secure.

The reason behind using a PRNG and not using the root secret directly is twofold. The root secret should stay as short as possible for convenience, and at the same time, the values derived from the secret should remain as independent as possible. That way, if either key is compromised or either algorithm is discovered to be weak, the other key is unaffected. And thanks to the random number generator, the design can be updated to generate more values. This can be useful, for example, when the currently used algorithms have a vulnerability and need to be replaced. The new algorithm can then be used with an independent, new, safe secret.

The seeded CSPRNG is then used to generate separate secrets for an authentication Ed25519 keypair and a *backup primary key*, a value from which symmetric backup keys are derived. Ed25519 is an elliptic-curve-based public key signature system. It's a widely used algorithm used by a large amount of popular software, such as OpenSSH [27]. The targeted security is stronger than RSA-2048 while providing very compact 256-bit public keys — used as client identifiers in this software [28].

5.1.1 Public key authentication

After the client generates a root secret and a derived keypair, it will be used to register and subsequently authenticate with the central server. The public key serves as a

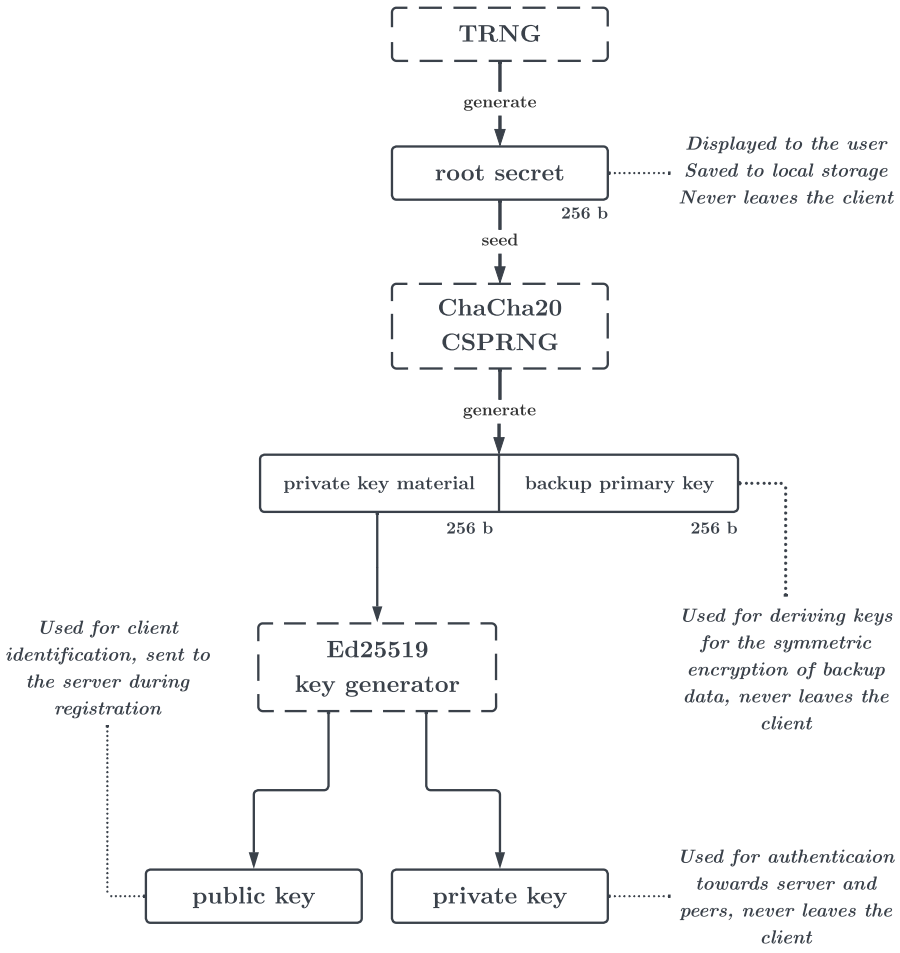

Figure 5.1 Secrets management hierarchy.

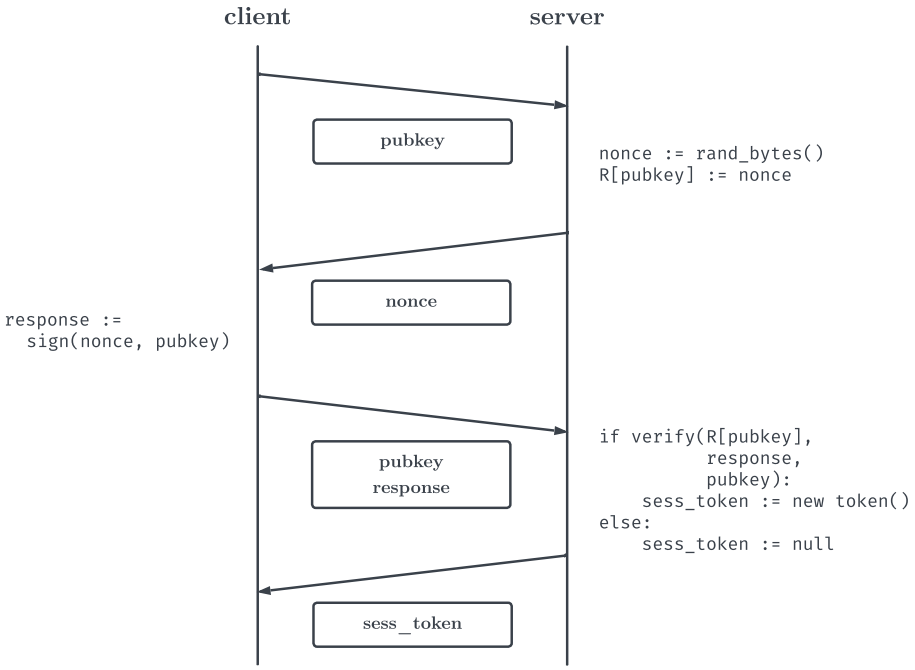

Figure 5.2 Challenge-response authentication flow.

unique identifier for a certain client, both for the server and other clients. The private key is also used for securing transport between clients. All messages sent by a client to its peers will be signed by its private key to ensure that they haven't been tampered with.

Authentication is performed in a challenge-response scheme (see Figure 5.2), where the client requests a challenge from the server, which replies with a random nonce. The nonce is then signed by the client's private key and sent to the server along with its public key to verify its identity. If the verification succeeds, the client will receive a temporary session token — valid for no longer than 24 hours — used for subsequent authenticated requests. Whenever it becomes invalid, the client will simply reauthenticate using this scheme.

5.2 Networking

As per the specifications in the assignment, this application currently only supports the transfer of backups between peers in the same network segment (each peer must be able to connect to each other directly over TCP). This is implemented by using a custom protocol based on WebSocket. Authentication and encryption are done on the application layer.

The connection between peers is always established via the server. It starts with a client sending an initialization message with the public key of the client it wants to connect to and a randomly generated nonce. The server then sends the public key of

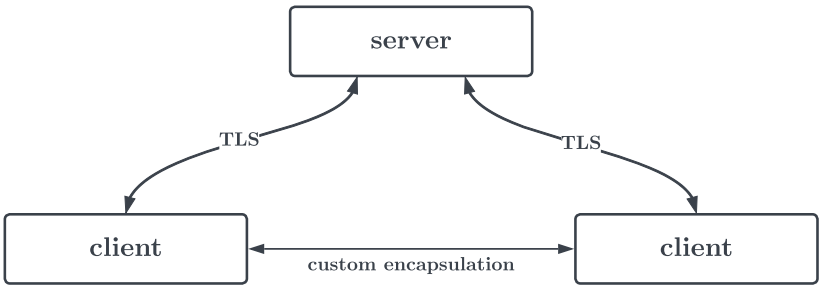

Figure 5.3 Integrity/confidentiality layers in network communication.

the requesting client and nonce to the destination. If the peer wants to accept the connection (i.e., is aware of the requesting peer), it sends a confirmation message to the server with its IP address (which needs to be on the same network) and a random port. The server finally sends the IP address of the destination peer to the requesting peer, and the original peer can now establish the connection to the IP address (the process is shown in Figure 5.4).

After the connection is established, the peer that accepted the connection is waiting for an initialization message from the requesting peer. The initialization message can either request to restore a backup or request to transport new files. If the requesting peer sends a request to restore a backup, the accepting peer will start sending all files that the requesting peer has stored at that location. Otherwise, if the requesting peer sends a request to send new files, the accepting peer will wait for these and store them.

After the connection is initialized, the peer that is transporting files sends encapsulated messages. The peer that is receiving them sends acknowledgment messages for a particular message sequence number whenever it receives and saves the file. These messages are visualized in Figure 5.5.

5.2.1 Security

The protocol is designed for transporting encrypted backup data, so confidentiality is ensured on the backup layer (see subsection 5.3.3). These encrypted backups are the data that is subsequently saved to disk.

The integrity of the received data is ensured by encapsulating encrypted backup data in a message that adds a header with a message sequence number and a random nonce, and a signature over the header and data. This additional header is designed to prevent replay attacks. The signature mitigates tampering with messages.

The public keys of the clients and the random nonce are relayed through the server (over a TLS-secured connection) as part of the handshake that connects the source and destination peer (see Figure 5.4). The nonce is initially generated by the client that is requesting the connection. The message sequence number is a counter that starts from zero and increments with each sent message.

These two values, together with the signature, ensure that messages cannot be replayed, sent out of order, individually withheld, tampered with, or outright forged. TCP guarantees the correct order on the network layer, so messages will normally be delivered in the correct order.

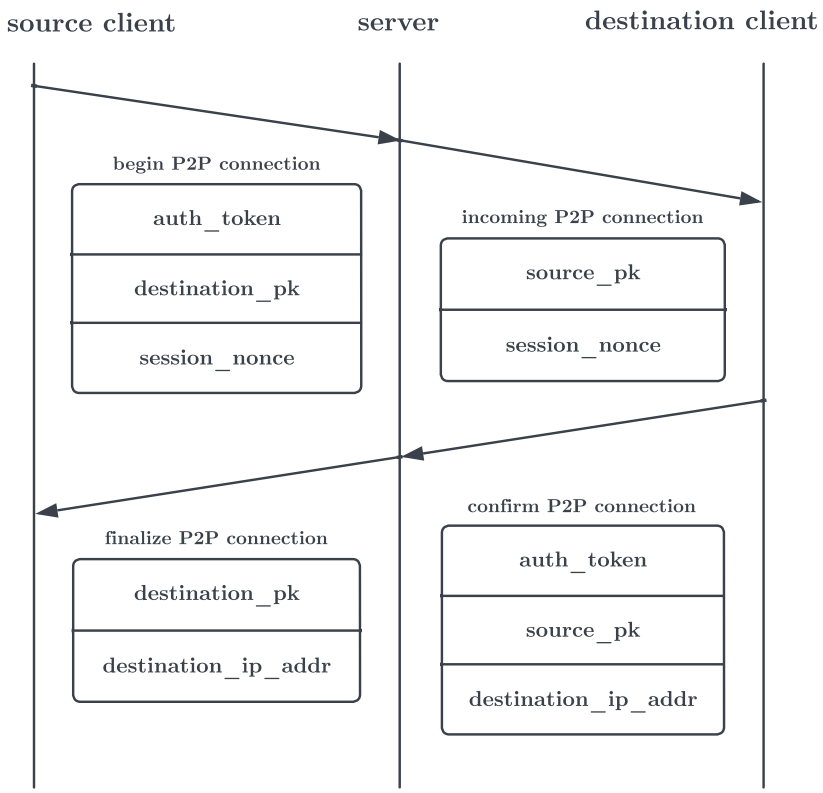

Figure 5.4 Server-facilited peer-to-peer connection handshake.

5.3 Backup format

The security of the backups is paramount, and storage efficiency is also a significant concern. Therefore, the backup storage format for this application is custom, using many ideas from the backup program restic [22].

5.3.1 Incremental backups and file chunks

Having incremental backups that conserve as much space as possible is essential, especially for a peer-to-peer backup application constrained by individual users' storage space and bandwidth. Simply backing up changed files only is not sufficient. Therefore need to use a more granular approach.

All files in the backup directory will be split into *chunks*. Smaller files under a certain threshold will consist of one chunk, and larger files will be split into several chunks. A chunk (sized up to 3 MiB) is the smallest data unit considered for incremental backups. That way, if a small amount of data is changed in a large file, there's no need to save and transport the whole file. The data is split into chunks using a Content Defined Chunking (CDC) algorithm.

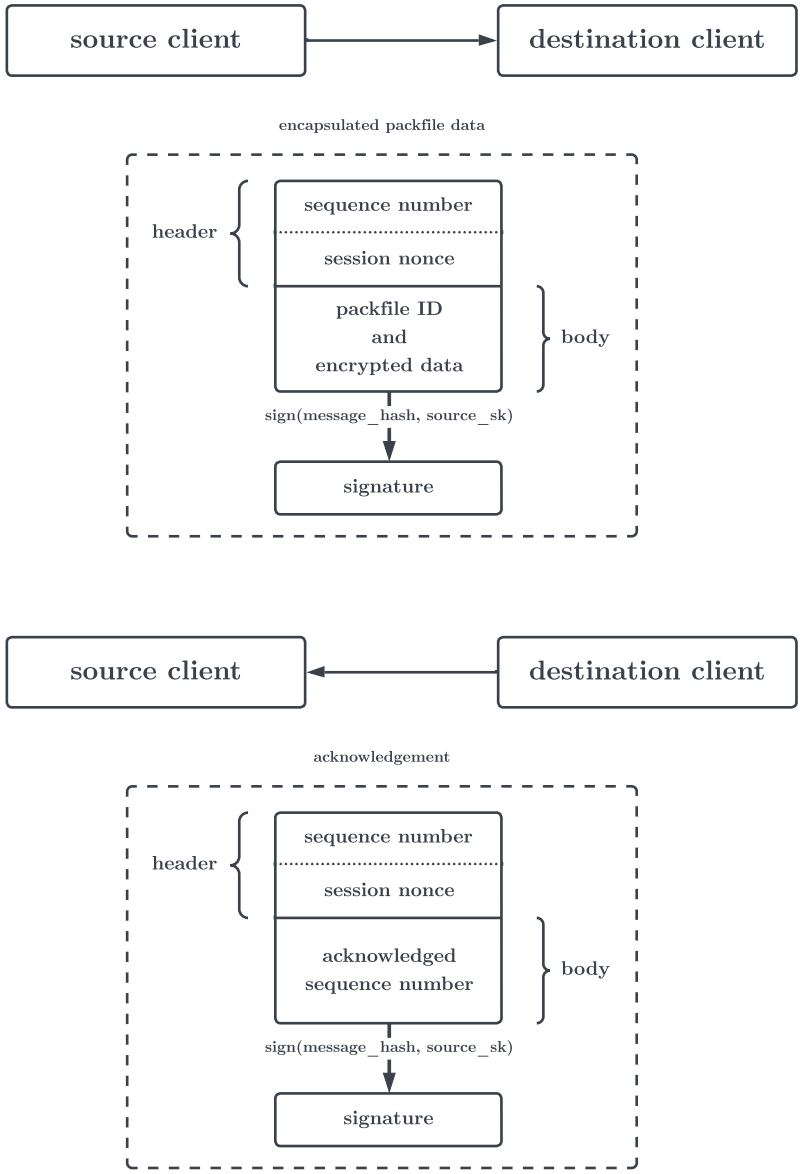

Figure 5.5 Encapsulation in the peer-to-peer file transport protocol.

5.3.1.1 Content Defined Chunking

The naive, traditional way to split files into parts would be simply dividing them into chunks of fixed length. However, this approach has a major weakness — chunk boundaries can be shifted very easily by just adding or removing data and thus changing the length of a chunk. This is especially inefficient when the length of a chunk changes near the start of the file. In that situation, the hashes of almost all chunks would change, and most data must be retransmitted.

CDC algorithms solve the problem of unnecessary chunk boundary shifting. They try to split the files at the same boundaries every time, even if the length of the chunks changes (if some data is inserted/removed, up to a certain limit). That allows for more efficient deduplication and incremental backups because the number of changed chunks is greatly minimized. I have chosen to use the FastCDC algorithm. It's a modern state-of-the-art CDC algorithm that implements this concept in an efficient way.

The general idea behind CDC algorithms is using a rolling hash function with a sliding window (FastCDC uses Gear hash) on the contents of the file, declaring a chunk boundary whenever the hash has a certain predefined property (e.g., it starts with a set number of zeros). This randomness is the reason why the length of chunks can vary significantly, although advanced algorithms like FastCDC take measures to prevent excessive size differences [29, 30].

5.3.2 Directory/file trees and blobs

Since files are backed up in the form of chunks, we need a way to reconstruct them. Similarly, we need to keep track of the directory structure. For this reason, we are introducing *trees*. Trees come in two types, file trees and directory trees.

File trees contain a list of chunks that make up the actual file data. They also hold metadata about a file, including the name and creation/modification timestamps. Directory trees contain the list of file trees representing files contained in that folder. Similarly, they also contain metadata. For storing in a backup, trees are serialized in an efficient and safe binary format called bincode [46].

Serialized trees and file chunks are together called *blobs*. Blobs are the pieces of data that the entire backup format is based on. In trees, they are represented as a hash of their contents. The cryptographic hashing function of choice for blobs is BLAKE3 because of its outstanding performance [31]. Deduplication of blobs is based on comparing their hashes — if the hashes are identical, blobs are also assumed to be identical. Since the hash output size is 256 bits long, the chance of collision is extremely low. For a mere 10^{-18} chance of getting a collision, it would be necessary to generate $4.8 \cdot 10^{29}$ blobs [32].

5.3.3 Packfiles

Blobs are stored in *packfiles*. A single packfile holds one or more blobs, a header, and the header length. A packfile header contains as many entries as the number of blobs stored within the packfile. Each entry holds the type (chunk/tree), hash, length, and offset from the start of the file for a single blob. When a packfile is read, we first decode the header and then retrieve the desired blobs based on the data in the header. During the backup process, the program will create packfiles and send them to the destination, deleting them afterward.

The main purpose of using packfiles is to normalize the size of stored data. Blobs don't have a minimum length since every file or directory will generate at least one

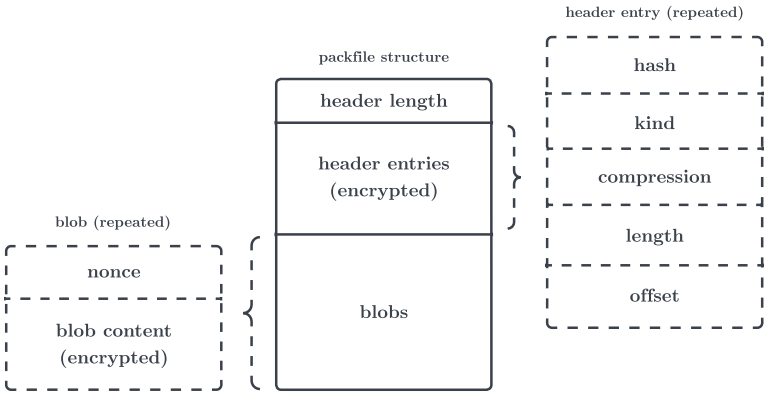

Figure 5.6 Packfile structure.

blob. Storing chunks individually could lead to inefficient transfer of many small files — so we will bring smaller chunks together. The packfile generator targets a maximum size and count of packfiles, so if a certain blob is big enough, it will get its own packfile. Other packfiles can have many small blobs. As a result, all packfiles will have a similar size (usually megabytes, up to 12 MiB).

The packfile layer is where the data encryption is performed. Chunks are encrypted individually using AES-256-GCM. Each chunk has a random nonce stored along with the data itself in the packfile (concatenated as `nonce || data`). The key used to encrypt a chunk is derived using HKDF from its hash and the backup primary key. The header is also encrypted with the same algorithm. A nonce for the header is also randomly generated, and this value is used as a packfile ID (its file name) to prevent reusing a nonce (packfile IDs are unique). The key used for the header is derived from a unique constant value and the backup primary key. The header length is stored as a plain number and is not encrypted. Encrypting blobs individually has advantages for parallelization, faster seeking, and easier packfile restructuring.

Compression is also a part of the packfile format. It uses Zstandard, a state-of-the-art algorithm offering an excellent compression ratio and speed balance [33]. Compression is performed on each blob individually before encryption.

5.3.4 Packfile index

With the current design, finding a blob would involve scanning the headers of all packfiles, which would be slow and inefficient. For that reason, we are adding an index, which is a separate set of files. Index files only contain mappings of blob hashes to packfile IDs.

Index uses a simple custom incremental storage format. The data is stored in a directory containing index files named with increasing sequential numbers. Each file contains one or more mappings of blob hashes to packfile IDs. Like the other storage formats, the data is also serialized with bincode. The index is also always stored on the origin machine, unlike packfiles — it's used during the process of creating a backup.

When working with the index, all the files are deserialized and loaded into memory.

Newer index files (the ones with higher sequence numbers) always take precedence over the older ones. The index is append-only, so this is the way to keep it updated as new packfiles are created. As a next step in loading, mappings are sorted by the blob hash. Now we can find out where a blob is saved or if it exists at all with binary search in $\log n$ time. Searching for blobs is an operation that is used frequently — both for deduplication in backups (finding whether the blobs already exist) and for locating the correct file during restores. Writing is always done by creating new files instead of modifying existing ones.

The index is encrypted on a file-by-file basis. Each file is encrypted with AES-256-GCM. The nonce is a null-padded sequential number of the file (which is unique), while the key is once again derived with HKDF from a string constant and backup primary key.

5.3.5 Snapshots

Whenever a backup is run, a new snapshot is created. A snapshot is identified by a hash of a blob that stores the backup root directory tree. This allows us to easily keep track of the state of the backup directory at any given point.

When restoring a backup, we retrieve the blob with the hash of the snapshot ID we want to restore. That blob stores a directory tree, which we will treat as a root of the backup and recursively unpack all data that it references. Only the changes are stored with each backup, so directories with no changes will still be represented by the same blob hash.

5.4 Other aspects

5.4.1 Peer discovery

One of the requirements from the assignment is that our backup destination peer has to be automatically selected. For that reason, we need to create mechanisms to initially find suitable peers to back up to, keep track of used storage, and retrieve our connections if we need to restore the backups.

Initial peer matching is performed on the server. The server receives *storage requests* from clients for a certain amount of space — and will either put them in a queue or take existing requests from the queue and match them with the incoming request. When a new request comes in with the queue empty, it will simply be enqueued. If the queue already has requests waiting, the algorithm will remove them, subtracting the requested size from the incoming request. If the whole size of the incoming request hasn't been fulfilled, the process continues. The algorithm is done when the queue is emptied, or the incoming request is fulfilled. If the size doesn't match exactly, the remaining size from either the incoming or the already enqueued requests is returned to the queue.

5.4.2 Server metadata storage

For users to be able to restore backups after a system failure and the loss of all application data, the server needs to store some amount of metadata. The metadata will be retrieved by the client when trying to restore a backup. Since the user identity is tied to the root secret, inputting the generated mnemonic from the previous installation will allow the application to present with the same credentials and retrieve the metadata.

The server currently stores all peers that have had storage matched together, and when a peer requests to restore a backup, the server will send a list of these. This could, however, be improved to make restores more reliable (discussed in section 7.3). The other type of metadata stored by the server is snapshot hashes, which represent the state of the folder at the time of backup. These are also required to successfully restore a backup, the unpacking mechanism needs to know the root directory. At present, the server will always send the latest snapshot has for a given client. Snaphost hashes are uploaded by clients to the server whenever they complete a backup and transport all data to peers.

5.4.3 Malicious data mitigation

Having untrusted peers send data to other users comes with many risks, which this thesis attempts to mitigate. One of the more complicated problems is mitigating the risk of peers bypassing the standard protocol and sending malicious data instead of regular encrypted backups. Problematic data includes illegal content, possessing which could endanger the receiving user. Another example is data designed to exploit the software of a receiving peer, ranging from annoyances to more serious attacks. An example of an annoyance can be a snippet that is harmless but triggers an alert in an antimalware scanner on the receiving user's computer. A more serious problem could arise if, for example, the victim runs software with a vulnerability that is triggered by specific data in a file.

The mitigation used in the current design is obfuscating the received data before saving them on disk. This prevents the attacker from having control over the data that is being saved to storage. The obfuscation consists of generating a random 4-byte key, unrelated to the root secret, on the first start of the application. The received data is then, in 4-byte chunks, XOR'd with the obfuscation key, after which it's finally saved to the disk. Before sending the data back, this obfuscation is undone by following the same process. The effectiveness and implications of this mitigation are discussed in subsection 7.2.1.

Chapter 6
Implementation

In this chapter, I will review the technology and library choices and key points of the client and server implementations. The entire codebase, as well as the generated code documentation (with comments), is included in the attached files.

The goal of the implementation is to serve as an experiment and a model for a production-ready user-friendly peer-to-peer backup application. Therefore, it's likely for it to be extended, and technology choices should help that goal whenever possible. One of my goals also was cross-platform support for the client application, so the underlying technologies are also chosen based on that goal (it's currently tested on Linux and Windows, but should work on other platforms as well).

6.1 Programming languages

My programming language of choice for the vast majority of the application across both server and client is Rust. The only component that uses a different language is the web-based user interface for the client, which is implemented in JavaScript and HTML. I will explain this choice in the rest of this section.

The implementation has a client and a server part, so it was necessary to accommodate this property in the programming language selection. A possibility was to choose different languages for these two parts, but using a single language suited for both makes cross-communication, shared code, and the whole development process easier.

The client functionality, by its nature, often needs concurrent operations. Especially when it concerns networking, it may be necessary to communicate with numerous parties simultaneously. For example, when creating a backup and transferring data to a peer (or multiple peers at once), we also receive data from other peers simultaneously. While doing that, a different peer might request a connection to restore their data. Ideally, we should be able to perform all these operations at once. Real-time networking capabilities are also a big bonus since many events happen asynchronously.

Another significant consideration for this thesis is security. Picking a language that is not memory safe would significantly increase the risk of severe security vulnerabilities and decrease reliability. However, memory safety is not the only consideration in the area of security. One of the other considerations is threading safety. It is a challenging

problem, but some languages are better at helping prevent race conditions than others. Among other notable language features, their type system greatly impacts security. Weakly typed languages have an increased risk of introducing type-conversion-related bugs and vulnerabilities.

To pick a language for the implementation, I also need to have a good degree of familiarity with it. For that reason, besides Rust, I mainly considered C, C++, Python, PHP, and JavaScript. I am not familiar enough with other popular, potentially suitable languages such as C# or Java. From the languages that were taken into consideration, C and C++ have safety problems and are generally unergonomic for such an application. PHP is not suitable for desktop applications and lacks real-time networking capabilities. JavaScript is similarly not designed for desktop applications running in the background and offers lacking language features.

Python is also an acceptable choice. However, it's slower and less robust than Rust. Additionally, on the client side, Python applications are harder to distribute and harder to implement efficiently. On the server side, it offers less scalability.

6.1.1 The Rust programming language

Rust best fulfills all the requirements. It's a cross-platform language that works well for both desktop and server applications and supports advanced asynchronous programming. Security is also a major focus of Rust — it separates *safe* code from *unsafe* code. It features a powerful compiler that statically verifies safe code for memory and type safety (with strict type checking) at compile time.

Thanks to that, the safe subset of Rust doesn't allow out-of-bounds memory access, use-after-free, and other memory-related vulnerabilities (even without a garbage collector). The type checker also prevents data races, which is helpful since the implementation features a lot of threaded code. Unsafe code in Rust must be clearly marked and delimited and can be avoided almost entirely for most applications, including this one. This gives the programmer great security guarantees but also the flexibility to write code that cannot be statically checked to be memory safe — when necessary [34].

Using Rust also provides some additional benefits. It offers built-in comfortable tooling with access to many good quality 3rd party libraries and produces easy-to-distribute, self-contained executables. The language is also designed for speed and memory efficiency, which is useful for a program intended to run in the background.

However, it also has its shortcomings. Some of the notable ones include long compile times and a large compilation cache. Another disadvantage is that the static compile-time checks can be overly strict and occasionally slow down the development. It also is a fast-moving language, so some language features are underdeveloped and less polished.

6.2 Network communication

The library for running an HTTP server and handling incoming requests on the server side, as well as serving the user interface on the client side is *poem* [47]. For sending HTTP requests, I use a library called *reqwest* [48]. The WebSocket protocol, in both parts of the application, is handled by the *tokio-tungstenite* library [49].

All of these projects are popular and well-respected libraries in their respective areas. Implementation of TLS in both server and client relies on *Rustls*. This modern and mature TLS library is not burdened by legacy protocols and is implemented in Rust, taking advantage of its safety [50].

The messages are serialized, across the board, using *serde* [51]. Serde is a powerful data serialization framework for Rust, which supports safely and easily encoding/decoding data in various formats. Data between server and client is utilizing the JSON format using the *serde_json* [52] backend. JSON is not as efficient as the *bincode* [46] format used in peer-to-peer communication. However, debugging is easier, and the higher overhead is negligible in client-server messaging.

6.3 Server

The server application is relatively simple, as its only function is facilitating communication between peers. The server runs a server with regular HTTP API endpoints and a WebSocket server. It's designed to run behind a reverse proxy (such as nginx) which will handle HTTPS connections and certificates — building support for HTTPS directly into the server is not advantageous since the reverse proxy is more configurable and integrates with software like Certbot (used for automatically issuing HTTPS certificates with services like Let's Encrypt).

The functionality of storage requests and managing sessions and login challenges for clients is implemented with in-memory structures. Since this data is expected to be ephemeral, I used in-memory data to reduce the implementation complexity and database overhead. The list of clients, backups, and matched storage are saved to a persistent PostgreSQL database. The server application automatically creates the database structure if it does not exist.

The server also keeps track of WebSocket connections of clients in memory, with a hash map of client IDs and connection objects. Using this map, the server can easily dispatch messages to clients. Closing of the connections is listened to by the server and will trigger the removal of the connection from the map.

6.4 Client

The client portion of the application is divided into several modules. However, those are still tightly integrated, as the process of creating backups and communication are intertwined. Those modules include backup creation itself, P2P communication, server communication, managing local configuration, and serving the UI.

The client part of the implementation is using SQLite [53] as its backend for storing structured local data, including configuration and information about peers. I chose SQLite for its universality, powerful querying, and resiliency.

6.4.1 Backup process

The backup creation process consists of packaging files and, at the same time, sending them to peers. This is implemented by using two asynchronous tasks, which can run on separate threads and work simultaneously. The synchronization between those parts is performed using a shared global state. The packing process will generate files until it reaches a threshold of maximum temporary file size — to not excessively use disk space. If the threshold is exceeded, the packing task pauses and waits for data to be sent to peers.

The packing task will first scan the backup path, build an in-memory directory hierarchy, and put directories into a queue, with the deepest levels of directories at the start — the hashes of upper levels depend on lower levels due to the hash-based

storage format. In a second pass, the packer will take directories from the queue and process (chunk, compress, encrypt) files in the directory. After all files in a directory are processed, all the hashes of their trees are known, allowing us to build the directory tree and calculate its hash. This process continues until we get the hash of the root directory, which is when the backup is completed, and a snapshot is created.

The sending part continuously sends any available files to peers, attempting to establish new connections if they don't exist already. It will first try to send to currently connected peers — if none are connected, then it will attempt to establish a connection with a known peer that has storage. If both strategies fail, it will send a storage request and wait for it to be fulfilled, connecting to the newly discovered peer.

The backup process is relatively complex and utilizes a degree of parallelization, offering great performance even with a large number of small files. However, at present, it's lacking in robustness. Since it utilizes in-memory caching and file buffers, it's vulnerable to getting into an inconsistent state when the application is terminated while a backup is in progress. For a more production-ready version, it's necessary to better handle shutdowns and detect an inconsistent state.

6.4.2 Restoration

The restoration process will request metadata (snapshot hash and all contacted peers) from the server, then attempt to connect to each peer and retrieve all the files it possesses (from the contacting peer). After all the files are received, the unpacker starts retrieving blobs, starting from the root hash. In the process, it creates all the stored directories (decoding their structure from blobs and continuing recursively), reassembles files, and sets their metadata.

At this time, the restoration process is vulnerable to a denial of service attack by a peer that sends the packfiles back by sending a large number of files since the receiving peer does not know the backup size. This can be mitigated by saving this information to the server and validating it while restoring.

6.4.3 Networking and identity management

The application needs to communicate with peers and with the central server. The peer-to-peer part of the networking code is responsible for retrieving a local IP address and opening a random port when being requested to open a connection. When requesting a connection, it will connect to the address it received from the server. It's also responsible for enforcing the communication protocol described in section 5.2.

The server-client section of the code involves making HTTP requests and a connection to the WebSocket server, handling and dispatching all incoming messages. When receiving a message related to peer-to-peer communication, it's relayed to the P2P component. It also handles reconnecting in case the connection terminates, and performing authentication again if the authentication token expires.

Managing the secrets and keys required to authenticate or encrypt data is performed by a separate part responsible for deriving the key on-the-fly and keeping them in memory.

6.4.4 User interface

The user interface in the client is based on web technologies and runs in a browser. Upon startup, the client launches a local HTTP server and displays a link to open the

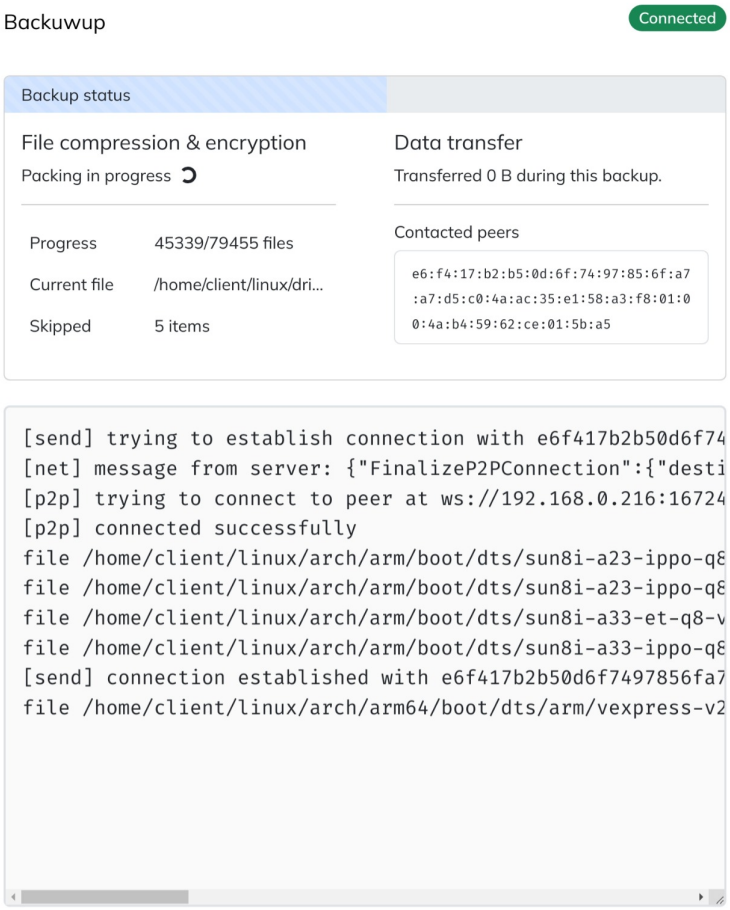

Figure 6.1 Screenshot of the user interface with a backup in progress.

web interface. This solution is chosen because of simple cross-platform support, fast and easy development, and flexibility — such as headless control from other computers.

The portion that runs in a browser is made in JavaScript and HTML, utilizing the Vue.js [54] frontend library to make it possible to easily and reliably update the state with new real-time data. Bootstrap CSS framework is used to style the interface [55]. Updating the UI's state and sending log messages is facilitated by sending messages over a WebSocket connection. Triggering actions in the client application, such as making backups, is also realized by sending a WebSocket message, from the browser to the client application.

Chapter 7
Results

In this chapter, I will discuss the usability and problems of the final design and implementation — as well as explore the possible solutions and a direction for the future.

From the start of working on this thesis, it was clear that the resulting design would not be perfect. On top of that, during the research and the development of this application, various aspects transpired differently than I expected. Creating a reliable and secure peer-to-peer backup application has proven to be complicated. There are numerous open questions that might be worth researching, some of which I will discuss in this chapter.

7.1 Networking

The network connection between the clients and the server uses standard protocols and works fine. However, the peer-to-peer connection protocol is more complex and has some unique drawbacks, which are discussed below.

7.1.1 Efficiency

The current design of the peer-to-peer backup transport protocol requires establishing two separate connections if we want to transport files from both peers at the same time (i.e., both peers are creating a backup). That would ideally be consolidated, and the protocol would reuse an existing connection.

The transport acknowledgment messages are implemented on a simple sequential basis where we wait for an acknowledgment before sending the next file. That reduces the throughput of the backup transport but provides crucial information about whether sending a file has succeeded and, therefore, we can delete it. This could be improved, for example, by implementing a sliding window acknowledgment message technique.

7.1.2 NAT traversal

The obvious main problem with the peer-to-peer networking solution in the current design is the inability to communicate with peers under network restrictions (such as a

Port restricted NAT	58.86%
Symmetric NAT	17.06%
Unknown	13.20%
Full cone NAT	5.98%
Not behind a NAT	2.09%
Restricted NAT	1.12%
Symmetric UDP firewall	1.69%

Table 7.1 NAT types breakdown, collected from Syncthing users [36].

NAT), as outlined in the assignment. For the application to be usable in the real world, with how networking is currently done, it would be mandatory to have an automatic ability to connect with peers, regardless of their network situation.

NAT is a networking solution that translates IP addresses by modifying packets passing through a router. The kind of NAT that is widely deployed and that we are going to focus on is called *one-to-many* NAT. It maps private IP addresses of many devices on a certain network to one public IP address. Its primary purpose is to alleviate the problem of IPv4 address space exhaustion. NAT is overwhelmingly used with IPv4, its use with the modern IPv6 protocol is fortunately rare. There are several types of one-to-many NAT, each with different restrictions, but the general problem is that peers cannot connect directly to each other [35].

7.1.2.1 NAT types

The most problematic kind of NAT is called *symmetric* NAT. With this type of address translation, only an external computer that receives data from an internal host can send data back. Both peers being behind separate symmetric NATs is the worst case for establishing a connection — in that case, communication is impossible without a relay (middle-man) server that is not behind a NAT and can establish TCP/IP connections directly. Both peers will then have to connect to the relay, sending data from one peer to the other. These techniques for establishing P2P connections are generally called NAT traversal [35].

For other types of commonly used NAT methods, there are lesser restrictions on which data is allowed to be transferred. This allows us, in many cases, to use techniques for establishing a direct connection between peers, with some help from external servers. This is a complex process, but it's worth considering. According to data collected from the users of Syncthing, an application for peer-to-peer data syncing (mentioned in chapter 3), only about 18% of users had to use relay servers, despite over 80% being reported as using NAT [36].

7.1.2.2 NAT traversal techniques

This problem already has comprehensive solutions that are used by various software. One of those is the ICE (Interactive Connectivity Establishment) protocol. The basic principle of ICE is as follows. The protocol starts by collecting possible candidate IP addresses. These include LAN-bound address, the public address of the NAT, or an address allocated by a relay server. To discover the public address of a NAT, ICE

uses STUN (Session Traversal Utilities for NAT). STUN is a standard mechanism for detecting the types of NAT and requires running a public server that is contacted by peers that want to connect. The STUN server can then perform a set of checks and may output candidate addresses (the NAT public address) for use by ICE.

For the last category of addresses, ICE uses the TURN (Traversal Using Relays around NAT) protocol. TURN is a type of protocol for relay servers and therefore requires a lot of bandwidth on the part of the server operator. However, it works in all cases, even when both peers are using symmetric NAT. After gathering all candidate adressess, ICE will then use STUN to perform connectivity checks on all pairs of candidate addresses of both peers — in a specific order to discover the best option. When a connectivity check succeeds, ICE has concluded, and we have a working route.

There are also other procedures that could be performed — one of them is using the UPnP protocol to directly try opening ports on supported routers. This might fail in many cases when the router NAT is also behind a carrier NAT (or UPnP is not supported), but it doesn't require any external servers and allows for a direct connection [39, 38, 40, 41, 37].

7.1.3 Privacy

Another concern raised by direct peer-to-peer communication is that users can see each other's IP addresses directly. However, in an environment where there's no additional personally identifiable information about the other user — since backups are encrypted, and users are identified by random keys — I wouldn't consider this to be a threat. One solution for this potential issue would be to exclusively communicate using a central server, which would most likely not be viable for a free service due to significant bandwidth usage. A different solution would be to use community-run relays, which might be more financially feasible but still require trusting a 3rd party, and the transfer speed would likely be slow.

7.1.4 Summary

In conclusion, the networking part of the application requires a sizable amount of work to allow for seamless peer-to-peer backup transport — NAT traversal in IPv4 is a complex subject. The current design would need to be amended to support NAT type detection, enhance the protocol for establishing P2P connections, and add support for NAT traversal and relaying data, possibly using ICE. Improving the throughput by enhancing the acknowledgment messages in the transport protocol would also be beneficial.

7.2 Backup storage

The backups storage format, as designed, works properly and allows for good efficiency. There are still some problems, however, which are discussed below.

7.2.1 Malicious data

One of the issues of storing data from random users is that they could try to bypass the encryption and send backup data that is malicious in some way. This problem, as well as the mitigation in place, is explained in subsection 5.4.3. The currently used

mitigation is effective in masking the data on disk, protecting users from attackers trying to, for example, coax the antivirus program into showing an alert.

However, this protection is unlikely to be effective if the attacker tries to frame the victim for storing illegal content (e.g., child sexual abuse material) because on-disk obfuscation is easy to reverse. On top of that, this data would be detectable on the network layer — there's no extra encryption on top of the packfile encryption. To help with network-level detection, it would be possible to introduce encryption on the network layer as well since all the keys are already in place.

7.2.1.1 Plausible deniability

One of the approaches that can be taken to alleviate the problem with illegal data is to aid the victim's plausible deniability and make all backup data signed with the sender's private key (and save it to disk with the signature), on top of just the message being signed. The signatures would be checked before saving the data since the receiving peer possesses the public key of the sender.

This mechanism wouldn't be very helpful just by itself with anonymous keypairs since anyone would be simply able to create multiple identities. Nevertheless, with a certain kind of real-world identity (such as a phone number) tied to a user's keypair could make this approach practical. With this solution, the victim will have proof that this data was created by someone else.

7.2.1.2 Anomaly detection

Another approach to tackling this issue is to use some form of anomaly detection. The backup format, as designed, will only contain encrypted data. Therefore, we could attempt to take steps toward detecting whether the received data is actually encrypted. This does not have a trivial solution — while encrypted data will necessarily have very high entropy, many other types of data will also have high entropy (namely data using lossless or lossy compression, typical in media files).

Nevertheless, there are more advanced methods of data classification apart from just simple entropy detections. One of these methods is called EnCoD, a neural network-based approach. It classifies different file kinds, including encrypted data, lossless compression formats like ZIP and RAR, and media files like JPEG or PNG. The classifier reaches a detection accuracy of over 90% for encrypted files, on file parts of just 2 KB (and improves further with larger samples) [42]. Running such a classifier on the client for received files and marking peers with high rates of unexpected results could be a good indicator.

Heuristic detection systems will, unfortunately, always have weaknesses and a way to bypass them — especially if the inner workings are well known. But if we tie this detection technique in with the reputation system described in subsection 7.3.3 (since the transport of backups is peer-to-peer), it could serve as a valuable indicator of a peer's trustworthiness.

7.2.1.3 Summary

Combining the existing obfuscation approach along with the above-discussed signature and detection approaches can form a viable framework for protecting users from being attacked by storing malicious data. However, this subject still needs further research.

7.2.2 Data retention

As currently implemented, the backup format only adds new data and never removes old snapshots (described in section 5.3). That allows the user to access the entire history of backups, but if a user has a longer-running backup, it might stop being practical. Therefore, having a mechanism to clean older, unneeded data would significantly improve usability. Due to the peer-to-peer nature of this application, we don't have fast access to the filesystem of the peers — so this feature is relatively complicated to design properly.

A possible way to allow for garbage collection of old data would require locally saving (along with the index, which is already saved) the list of peers which hold particular packfiles. During the backup process, the program would then be able to collect hashes of all blobs needed by the latest snapshot (since all the files exist locally). Allowing to keep more than the last snapshot would also require saving this list of hashes locally.

After that, cross-referencing with the index, we could determine which sent packfiles have exclusively or mostly orphaned blobs. Packfiles which contain exclusively orphaned blobs can simply be deleted (by sending a message to the storing peer). For other packfiles, we can use a certain threshold for the orphaned peers. If their number crosses a set value, the packfile will be individually retrieved. Then, the blobs will be copied into a new packfile (along with any possible new blobs), and the original packfile will be ordered for deletion. Following that, a new index file will be created, where the entries for identical blob hashes will automatically supersede old ones.

This mechanism doesn't allow for garbage collection in the index, but it's unlikely to have a significant footprint. 2 500 000 blobs (up to 2 500 000 files or up to about 5 TiB of data) only take up about 110 MiB of the index. Either way, the index only contains redundant data and can be rebuilt from packfiles. Overall, I would consider this to be a viable way to implement the cleanup of old data, with some parameters still needing investigation.

7.3 Resiliency and reliability

As I discussed in chapter 3, the idea of randomly chosen peers for storing backups appears to be novel. Therefore, the current design has several drawbacks. The restoration process is unambiguously fragile, as it relies on all nodes a client has ever communicated with to be online to restore any data. Destination peers are currently able to falsify actually storing data. The strategy for choosing peers only takes into consideration the recency of a storage request — no other factors are considered. There is also no way to manually choose peers (friends) we might want to back up to. All storage is also treated with a 1:1 ratio — we cannot provide to someone freely nor choose the number of copies of our data. This section discusses solutions related to these deficiencies.

7.3.1 Storage proofs

In the current design, it's possible for peers to claim that they store backups while actually not saving them to disk. That is a significant problem with an enormous impact. It would allow a malicious user to use space and store backups on drives of honest users while not contributing anything — and when an honest user attempts to restore a backup, the malicious user will not give out any data (since they don't even store them).

A mitigation for this issue is adding a *storage proof* system where the storing peer would be required to prove that it truly stores the data. A good way to achieve this is to periodically send *challenges*. These consist of requesting cryptographic hashes of random byte ranges of the stored files. The reason for using hashes of random ranges is twofold — there's no way to calculate the hash without actually having the file, and it maximizes storage and bandwidth efficiency.

The storage proof challenges will be generated by the peer that is creating a backup. The client, during the creation of the encrypted packfiles, generates random ranges of bytes and calculates cryptographic hashes on the generated files before sending and deleting them. These challenges — one consisting of the file name, the byte range, and the hash — will be saved to local storage. The peer that created a backup will then periodically send these prepared byte ranges and file names to the destination peer. If the other client responds with the correct hash, the challenge is successful. Otherwise, the user is alerted that there's a problem. This system can also be tied in with the reputation system mentioned subsection 7.3.3, where successfully responding to challenges will increase the reputation of a peer.

The peer that is backing up will only be able to generate a finite number of challenges — but thanks to using hashes, these don't take up a significant amount of space. However, it will be necessary to have challenges ready for a reasonable timeframe. At the same time, the trust in the destination peer is initially low. Therefore, initially, the challenges will be sent often, with the frequently later being reduced as the trust in a peer increases.

7.3.2 Data storage redundancy

There's undeniably a need for some form of redundancy in a peer-to-peer backup application, but the design and implementation of such a solution were outside this thesis's scope. The easiest solution for redundancy would be to simply send each file to multiple peers. This would result in an efficiency ratio of $\frac{1}{n}$ and allow $n-1$ peers to disappear before losing any data, where n is the number of peers storing the files. However, more efficient ways exist to achieve a good degree of redundancy at the cost of more complexity.

7.3.2.1 Erasure codes

Erasure codes are a better way to handle redundancy with greater flexibility and efficiency. They are a form of error correction code that can recover from bit erasures. Erasure codes split the data into parts, and a certain amount of these might then be deleted — and the code will still be able to recover the data. The amount of redundant parts depends on the configuration and the code [43].

Reed-Solomon codes are a broadly used type an erasure code, which are also very suitable for this application. They are a type of linear code and have very favorable properties. Reed-Solomon codes would enable us to flexibly split each backed up packfile into n data parts and k parity parts. Each part would then be transported to a different peer, so the entire file would be spread among $n+k$ peers. We can then afford to lose up to k parts (any of them) and still recover the entire file. The efficiency ratio would then be $\frac{n}{k}$ [20]. The ratio could even be dynamic and vary depending on the reputation of peers available for storage. I would suggest splitting the file into about 5 parts with 2–4 parity parts. This would be a very valuable addition, but the exact way to implement this needs further analysis.

7.3.3 Peer reputation

At present, peers can register into the system without any vetting and immediately start sending storage requests. That can lead to many problems. For example, a malicious peer could perform a DoS attack on the storage request queue — by repeatedly sending large storage requests which would match other peers and subsequently refuse to connect to those peers, preventing all from creating backups. Another possible problem is peers who go offline often, have poor internet uplink, or intentionally refuse to give out backup data. These problems could be alleviated by introducing a peer reputation system.

The idea for this system is to have a score for each registered peer. The score would start at a low value, growing with the reliability of the peer. One of the possible components is the time that the peer spends online and connected to the server. Online time would slowly increase the score, while being offline would decrease it. This portion of the score could be handled by the server by checking whether the connection is alive or not. Manipulating this is possible by being connected to the server, while not actually communicating with peers. However, that shouldn't have a significant effect in combination with other components of the score.

One of the other significant components would be a score that signifies whether a peer receives and actually stores backups. This portion of the reputation score would work along with the storage proof system. However, designing a robust peer-to-peer reputation is complicated. Using ideas from [44], we can use a protocol built on top of storage proofs. A peer that is responding to a storage proof request will send a ticket signed with its private key. The peer that sent the request will then verify whether the proof is valid and send the ticket to the server, which will then increment the score for the peer that served the valid response. The peer that sent the storage proof would be able to withhold sending the ticket, but it would not give it any advantages. On top of that, the protocol might give the peer sending the ticket a small increase in its reputation as well, as a reward.

A severe problem with such a reputation score system is collusion among friendly peers, which would be able to fake creating backups and unfairly increment each other's scores. One of the most well-known systems to implement such a score that combats this problem is called EigenTrust [45]. Nevertheless, that is beyond the scope of this thesis, and the implementation of a score such as that needs further research.

7.3.4 Storage requests and peer discovery

The current solution for finding peers to make backups is a simple queue with expiration. The requests are inserted as they are received by the server and are fulfilled in order. This process is applied to all storage requests, including if we are already connected to a peer. Such a solution works; however, it's prone to problems, including those described in the previous sections.

Matching peers fairly is a nontrivial problem. During testing, I encountered problems with peers not being able to complete backups — in a situation where there simply weren't any other users who also wanted to backup up data. This problem is amplified if peers have wildly different backup sizes. However, this issue should get less impactful with scale and more advanced peer matching. Unfortunately, with a one-to-one ratio (or other fixed ratio) for storage matching, this will always be a weak point of such a peer-to-peer system.

Additionally, allowing already connected peers to renegotiate and increase their storage allocation would reduce fragmentation and increase efficiency. Another signifi-

cant improvement to the matching system would be allowing one to add certain peers manually. That would enable users to back up to friends specifically, bypassing the problems with peer matching and reducing the likelihood of the peer being offline.

7.3.5 Summary

In the current state, the action of restoring a backup only works under ideal conditions. Adding support for storage proofs, redundancy (ideally with erasure codes), a reputation score system, and improved storage requests (which support these features) would improve the reliability immensely. With these enhancements, users would be able to have trust in the backups.

Chapter 8
Conclusion

In this thesis, I have performed research on existing solutions. Following that, I created a threat model and designed the application's protocols and storage formats. Based on those designs, I implemented a basic demonstration application that is able to create (and restore) securely encrypted backups to randomly selected peers in the same network segment.

With that, the goals of the thesis have been fulfilled successfully. However, this topic is complicated — and, as expected, the created application is not suitable for production use in its current state. Moreover, I have uncovered some deficiencies in the original design. These shortcomings and the respective solutions are discussed in detail in chapter 7.

Given all the work done on this topic, I see peer-to-peer backups (including those to randomly selected users) as a field that deserves further research — which could lead to an application that actual users could utilize. Compared to other popular forms of backups, like cloud storage or external drives, they can be a zero-cost solution and have a very low barrier of entry. Therefore, they can help users by making backing up data easier. However, some disadvantages of a peer-to-peer solution include difficulties with matching peers fairly, poor compatibility with ever more popular mobile devices, the need to own disks to store backups of other users, or the complexity of developing such a solution.

There is a lot of potential for future enhancements, including implementing the numerous design improvements discussed in the previous chapter and making the application more user-friendly — with features like automatic continuous backups and improving the user interface. More exploration can also be done to improve the system's decentralization and not rely on a central server for metadata storage and peer discovery.

Appendix A

Threat model

This appendix contains the Microsoft Threat Modeling Tool output, with all the generated and custom threats.

Mitigated Weak Authentication Scheme (Receiving encrypted backups)

Category Information Disclosure

Priority High

Description Custom authentication schemes are susceptible to common weaknesses such as weak credential change management, credential equivalence, easily guessable credentials, null credentials, downgrade authentication or a weak credential change management system. Consider the impact and potential mitigations for your custom authentication scheme.

Justification Mitigated by using randomly generated keys with state-of-the-art algorithms.

Mitigated Collision Attacks (Receiving encrypted backups)

Category Tampering

Priority High

Description Attackers who can send a series of packets or messages may be able to overlap data. For example, packet 1 may be 100 bytes starting at offset 0. Packet 2 may be 100 bytes starting at offset 25. Packet 2 will overwrite 75 bytes of packet 1. Ensure you reassemble data before filtering it, and ensure you explicitly handle these sorts of cases.

Justification Mitigated by using a higher-level message-based network protocol.

Needs Investigation Bogus Data Entries to Central Server (Backup management)

Category Denial of Service

Priority Low

Description Clients could spam the Central Server with bogus data like client registrations or snapshots to perform DoS attacks.

Justification The central server currently doesn't have rate limiting, but it can be achieved using a reverse proxy.

Mitigated Replay Attacks (Receiving encrypted backups)

Category Tampering

Priority High

Description Packets or messages without sequence numbers or timestamps can be captured and replayed in a wide variety of ways. Implement or utilize an existing communication protocol that supports anti-replay techniques (investigate sequence numbers before timers) and strong integrity.

Justification Replay attacks on data between clients are mitigated by adding a header to each message with a sequence number and a random nonce negotiated over a TLS connection with the server.

Mitigated Weak Authentication Scheme (Sending encrypted backups)

Category Information Disclosure

Priority High

Description Custom authentication schemes are susceptible to common weaknesses such as weak credential change management, credential equivalence, easily guessable credentials, null credentials, downgrade authentication or a weak credential change management system. Consider the impact and potential mitigations for your custom authentication scheme.

Justification Mitigated by using randomly generated keys with state-of-the-art algorithms.

Mitigated Collision Attacks (Sending encrypted backups)

Category Tampering

Priority High

Description Attackers who can send a series of packets or messages may be able to overlap data. For example, packet 1 may be 100 bytes starting at offset 0. Packet 2 may be 100 bytes starting at offset 25. Packet 2 will overwrite 75 bytes of packet 1. Ensure you reassemble data before filtering it, and ensure you explicitly handle these sorts of cases.

Justification Mitigated by using a higher-level message-based network protocol.

Mitigated Replay Attacks (Sending encrypted backups)

Category Tampering

Priority High

Description Packets or messages without sequence numbers or timestamps can be captured and replayed in a wide variety of ways. Implement or utilize an existing communication protocol that supports anti-replay techniques (investigate sequence numbers before timers) and strong integrity.

Justification Replay attacks on data between clients are mitigated by adding a header to each message with a sequence number and a random nonce negotiated over a TLS connection with the server.

Mitigated Elevation Using Impersonation (Sending encrypted backups)

Category Elevation Of Privilege

Priority High

Description Destination Client may be able to impersonate the context of Source Client in order to gain additional privilege.
Justification The clients are authenticated towards the central server with their private key.

Not Applicable Source Client Process Memory Tampered (Sending encrypted backups)

Category Tampering
Priority High
Description If Source Client is given access to memory, such as shared memory or pointers, or is given the ability to control what Destination Client executes (for example, passing back a function pointer.), then Source Client can tamper with Destination Client. Consider if the function could work with less access to memory, such as passing data rather than pointers. Copy in data provided, and then validate it.
Justification Memory pointers are not passed over the network.

Mitigated Elevation Using Impersonation (Receiving encrypted backups)

Category Elevation Of Privilege
Priority High
Description Source Client may be able to impersonate the context of Destination Client in order to gain additional privilege.
Justification The clients are authenticated towards the central server with their private key.

Not Applicable Destination Client Process Memory Tampered (Receiving encrypted backups)

Category Tampering
Priority High
Description If Destination Client is given access to memory, such as shared memory or pointers, or is given the ability to control what Source Client executes (for example, passing back a function pointer.), then Destination Client can tamper with Source Client. Consider if the function could work with less access to memory, such as passing data rather than pointers. Copy in data provided, and then validate it.
Justification Memory pointers are not passed over the network.

Mitigated Allowed Backup Size Spoofing (Receiving encrypted backups)

Category Denial of Service
Priority Medium
Description Source client could send more data than would normally be allowed.
Justification Destination client will check the size of received files on a client-by-client basis.

Mitigated Bogus Data Entries Between Clients (Receiving encrypted backups)

Category Denial of Service
Priority Medium
Description Client could connect to a peer and try to send bogus data to perform DoS attacks.

Justification Connection has to be established with the cooperation of the server and both clients. The client will check that the peer is known — only then is the connection allowed. The client will not allow receiving over a negotiated amount of data.

Needs Investigation Spoofing of Encrypted Data (Sending encrypted backups)

Category Repudiation
Priority Medium
Description Source client might be able to spoof the data that it sends to the destination which stores it. That can be abused for exploiting the victim with malicious code or framing the victim for storing illegal data.
Justification The data is obfuscated on disk to prevent the source to have control over data on disk of the destination. However, this is a complex issue and there are other possible mitigations.

Mitigated Impersonation of Other Clients (Backup management)

Category Information Disclosure
Priority Low
Description Clients may be able to impersonate other clients to retrieve a list of backups.
Justification Mitigated by authenticating the clients with a private key.

Mitigated Weak Encryption (Sending encrypted backups)

Category Information Disclosure
Priority High
Description The destination might be able to decrypt the data if the encryption is not adequate.
Justification Mitigated by using state-of-the-art algorithms and random keys.

Mitigated Backup Tampering (Receiving encrypted backups)

Category Tampering
Priority High
Description Retrieved backups might have been tampered with.
Justification Mitigated by using an authenticated encryption scheme.

Needs Investigation Destination Client Spoofing Storage (Receiving encrypted backups)

Category Spoofing
Priority High
Description The destination client might not be storing the backup that it receives, leading to loss of data for the source and spoofed capacity for the destination.
Justification Could be mitigated using a challenge system, where the destination has to prove that it stores the data. Source will request hashes of certain ranges of data, which are impossible to respond to without actually having the data.

Not Applicable Backup Retrieval Impossible (Receiving encrypted backups)

Category Denial of Service

Priority Low

Description The destination client may be offline or refuse to give out backups.

Justification Out of scope.

Not Applicable Source Client Process Memory Tampered (Backup management)

Category Tampering

Priority High

Description If Source Client is given access to memory, such as shared memory or pointers, or is given the ability to control what Server executes (for example, passing back a function pointer.), then Source Client can tamper with Server. Consider if the function could work with less access to memory, such as passing data rather than pointers. Copy in data provided, and then validate it.

Justification Memory pointers are not passed over the network.

Mitigated Replay Attacks (Backup management)

Category Tampering

Priority High

Description Packets or messages without sequence numbers or timestamps can be captured and replayed in a wide variety of ways. Implement or utilize an existing communication protocol that supports anti-replay techniques (investigate sequence numbers before timers) and strong integrity.

Justification Mitigated using TLS for communication between clients and server.

Mitigated Collision Attacks (Backup management)

Category Tampering

Priority High

Description Attackers who can send a series of packets or messages may be able to overlap data. For example, packet 1 may be 100 bytes starting at offset 0. Packet 2 may be 100 bytes starting at offset 25. Packet 2 will overwrite 75 bytes of packet 1. Ensure you reassemble data before filtering it, and ensure you explicitly handle these sorts of cases.

Justification Mitigated using TLS for communication between clients and server.

Mitigated Authenticated Data Flow Compromised (Backup management)

Category Tampering

Priority High

Description An attacker can read or modify data transmitted over an authenticated dataflow.

Justification Mitigated using TLS for communication between clients and server.

Mitigated Weak Authentication Scheme (Backup management)

Category Information Disclosure

Priority High

Description Custom authentication schemes are susceptible to common weaknesses such as weak credential change management, credential equivalence, easily guessable credentials, null credentials, downgrade authentication or a weak credential change management system. Consider the impact and potential mitigations for your custom authentication scheme.

Justification Mitigated using a randomly generated key as a credential.

Not Applicable Elevation Using Impersonation (Backup management)

Category Elevation Of Privilege

Priority High

Description Server may be able to impersonate the context of Source Client in order to gain additional privilege.

Justification Each client has its own public key, and anyone can be a client.

Not Applicable Server Process Memory Tampered (Backup management and client discovery)

Category Tampering

Priority High

Description If Server is given access to memory, such as shared memory or pointers, or is given the ability to control what Source Client executes (for example, passing back a function pointer.), then Server can tamper with Source Client. Consider if the function could work with less access to memory, such as passing data rather than pointers. Copy in data provided, and then validate it.

Justification Memory pointers are not passed over the network.

Mitigated Elevation Using Impersonation (Backup management and client discovery)

Category Elevation Of Privilege

Priority High

Description Source Client may be able to impersonate the context of Server in order to gain additional privilege.

Justification Server is authenticated using a valid TLS certificate with a set domain.

Mitigated Weak Authentication Scheme (Backup management and client discovery)

Category Information Disclosure

Priority High

Description Custom authentication schemes are susceptible to common weaknesses such as weak credential change management, credential equivalence, easily guessable credentials, null credentials, downgrade authentication or a weak credential change management system. Consider the impact and potential mitigations for your custom authentication scheme.

Justification Server has a valid TLS certificate and a set domain.

Mitigated Replay Attacks (Backup management and client discovery)

Category Tampering

Priority High

Description Packets or messages without sequence numbers or timestamps can be captured and replayed in a wide variety of ways. Implement or utilize an existing communication protocol that supports anti-replay techniques (investigate sequence numbers before timers) and strong integrity.

Justification Mitigated using TLS for communication between clients and server.

Mitigated Collision Attacks (Backup management and client discovery)

Category Tampering

Priority High

Description Attackers who can send a series of packets or messages may be able to overlap data. For example, packet 1 may be 100 bytes starting at offset 0. Packet 2 may be 100 bytes starting at offset 25. Packet 2 will overwrite 75 bytes of packet 1. Ensure you reassemble data before filtering it, and ensure you explicitly handle these sorts of cases.

Justification Mitigated using TLS for communication between clients and server.

Mitigated Potential SQL Injection Vulnerability for PostgreSQL (Database connection)

Category Tampering

Priority High

Description SQL injection is an attack in which malicious code is inserted into strings that are later passed to an instance of PostgreSQL for parsing and execution. Any procedure that constructs SQL statements should be reviewed for injection vulnerabilities because PostgreSQL will execute all syntactically valid queries that it receives. Even parameterized data can be manipulated by a skilled and determined attacker.

Justification Mitigated by exclusively using prepared statements.

Not Applicable Spoofing of Destination Data Store PostgreSQL (Database connection)

Category Spoofing

Priority High

Description PostgreSQL may be spoofed by an attacker and this may lead to data being written to the attacker's target instead of PostgreSQL. Consider using a standard authentication mechanism to identify the destination data store.

Justification The connection is set up with a internal local Docker network on the server.

Needs Investigation Potential Excessive Resource Consumption for Server Application or PostgreSQL (Database connection)

Category Denial Of Service

Priority Medium

Description Does Server Application or PostgreSQL take explicit steps to control resource consumption? Resource consumption attacks can be hard to deal with, and there are times that it makes sense to let the OS do the job. Be careful that your resource requests don't deadlock, and that they do timeout.

Justification Requests to the database are using a fixed number of pooled connections. However rate limiting is currently not implemented on the server application.

Mitigated Elevation by Changing the Execution Flow in Source Client (Backup management and client discovery)

Category Elevation Of Privilege

Priority High

Description An attacker may pass data into Source Client in order to change the flow of program execution within Source Client to the attacker's choosing.

Justification Client application performs extensive input validation and is designed to use safe data processing methods.

Mitigated Source Client May be Subject to Elevation of Privilege Using Remote Code Execution (Backup management and client discovery)

Category Elevation Of Privilege

Priority High

Description Server Application may be able to remotely execute code for Source Client.

Justification Client application performs extensive input validation and is designed to use safe data processing methods.

Not Applicable Data Flow Backup management and client discovery Is Potentially Interrupted (Backup management and client discovery)

Category Denial Of Service

Priority Low

Description An external agent interrupts data flowing across a trust boundary in either direction.

Justification Out of scope.

Mitigated Potential Process Crash or Stop for Source Client (Backup management and client discovery)

Category Denial Of Service

Priority Low

Description Source Client crashes, halts, stops or runs slowly; in all cases violating an availability metric.

Justification The server is considered trusted, however all incoming data is properly validated.

Needs Investigation Potential Data Repudiation by Source Client (Backup management and client discovery)

Category Repudiation

Priority Low

Description Source Client claims that it did not receive data from a source outside the trust boundary. Consider using logging or auditing to record the source, time, and summary of the received data.

Justification More work could be done to log client communications to the central server.

Mitigated Elevation by Changing the Execution Flow in Server Application (Backup management)

Category Elevation Of Privilege
Priority High
Description An attacker may pass data into Server Application in order to change the flow of program execution within Server Application to the attacker's choosing.
Justification Server application performs extensive input validation and is designed to use safe data processing methods.

Mitigated Server Application May be Subject to Elevation of Privilege Using Remote Code Execution (Backup management)

Category Elevation Of Privilege
Priority High
Description Source Client may be able to remotely execute code for Server Application.
Justification Server application performs extensive input validation and is designed to use safe data processing methods.

Not Applicable Data Flow Backup management Is Potentially Interrupted (Backup management)

Category Denial Of Service
Priority Low
Description An external agent interrupts data flowing across a trust boundary in either direction.
Justification Out of scope.

Mitigated Potential Process Crash or Stop for Server Application (Backup management)

Category Denial Of Service
Priority Medium
Description Server Application crashes, halts, stops or runs slowly; in all cases violating an availability metric.
Justification The server validates all incoming requests. In case of a crash, the supervisor will restart the application. The application is designed to safely exit in case of an unhandled error. Running slowly could be mitigated with more advanced logging.

Not Applicable Potential Data Repudiation by Server Application (Backup management)

Category Repudiation
Priority Low
Description Server Application claims that it did not receive data from a source outside the trust boundary. Consider using logging or auditing to record the source, time, and summary of the received data.

Justification The server is trusted.

Needs Investigation Potential Data Repudiation by Source Client (Receiving encrypted backups)

Category Repudiation

Priority Low

Description Source Client claims that it did not receive data from a source outside the trust boundary. Consider using logging or auditing to record the source, time, and summary of the received data.

Justification More work could be done to log client communications to the central server.

Mitigated Potential Lack of Input Validation for Source Client (Receiving encrypted backups)

Category Tampering

Priority High

Description Data flowing across Receiving encrypted backups may be tampered with by an attacker. This may lead to a denial of service attack against Source Client or an elevation of privilege attack against Source Client or an information disclosure by Source Client. Failure to verify that input is as expected is a root cause of a very large number of exploitable issues. Consider all paths and the way they handle data. Verify that all input is verified for correctness using an approved list input validation approach.

Justification Received messages are properly validated for authenticity and the client is designed to use safe data processing methods.

Mitigated Data Flow Sniffing (Receiving encrypted backups)

Category Information Disclosure

Priority High

Description Data flowing across Receiving encrypted backups may be sniffed by an attacker. Depending on what type of data an attacker can read, it may be used to attack other parts of the system or simply be a disclosure of information leading to compliance violations. Consider encrypting the data flow.

Justification The backup data is always encrypted on the application layer.

Mitigated Potential Process Crash or Stop for Source Client (Receiving encrypted backups)

Category Denial Of Service

Priority Medium

Description Source Client crashes, halts, stops or runs slowly; in all cases violating an availability metric.

Justification The client validates that messages are of a valid size. Receiving data is capped by negotiated storage space. Data restoration is rate limited.

Mitigated Spoofing the Source Client Process (Receiving encrypted backups)

Category Spoofing

Priority High

Description Source Client may be spoofed by an attacker and this may lead to information disclosure by Destination Client. Consider using a standard authentication mechanism to identify the destination process.

Justification All messages are signed with a private key of the respective peer.

Not Applicable Data Flow Receiving encrypted backups Is Potentially Interrupted (Receiving encrypted backups)

Category Denial Of Service

Priority Low

Description An external agent interrupts data flowing across a trust boundary in either direction.

Justification Out of scope.

Mitigated Source Client May be Subject to Elevation of Privilege Using Remote Code Execution (Receiving encrypted backups)

Category Elevation Of Privilege

Priority High

Description Destination Client may be able to remotely execute code for Source Client.

Justification Client application performs extensive input validation and is designed to use safe data processing methods.

Mitigated Elevation by Changing the Execution Flow in Source Client (Receiving encrypted backups)

Category Elevation Of Privilege

Priority High

Description An attacker may pass data into Source Client in order to change the flow of program execution within Source Client to the attacker's choosing.

Justification Client application performs extensive input validation and is designed to use safe data processing methods.

Mitigated Spoofing the Destination Client Process (Sending encrypted backups)

Category Spoofing

Priority High

Description Destination Client may be spoofed by an attacker and this may lead to information disclosure by Source Client. Consider using a standard authentication mechanism to identify the destination process.

Justification All messages are signed with a private key of the respective peer.

Mitigated Potential Lack of Input Validation for Destination Client (Sending encrypted backups)

Category Tampering

Priority High

Description Data flowing across Sending encrypted backups may be tampered with by an attacker. This may lead to a denial of service attack against Destination Client or an elevation of privilege attack against Destination Client or an information disclosure by Destination Client. Failure to verify that input is as expected is a root cause of a very large number of exploitable issues. Consider all paths and the way they handle data. Verify that all input is verified for correctness using an approved list input validation approach.

Justification Received messages are properly validated for authenticity and the client is designed to use safe data processing methods.

Needs Investigation Potential Data Repudiation by Destination Client (Sending encrypted backups)

Category Repudiation

Priority Low

Description Destination Client claims that it did not receive data from a source outside the trust boundary. Consider using logging or auditing to record the source, time, and summary of the received data.

Justification More work could be done to log client communications to the central server.

Mitigated Data Flow Sniffing (Sending encrypted backups)

Category Information Disclosure

Priority High

Description Data flowing across Sending encrypted backups may be sniffed by an attacker. Depending on what type of data an attacker can read, it may be used to attack other parts of the system or simply be a disclosure of information leading to compliance violations. Consider encrypting the data flow.

Justification The backup data is always encrypted on the application layer.

Mitigated Potential Process Crash or Stop for Destination Client (Sending encrypted backups)

Category Denial Of Service

Priority Medium

Description Destination Client crashes, halts, stops or runs slowly; in all cases violating an availability metric.

Justification The client validates that messages are of a valid size. Receiving data is capped by negotiated storage space. Data restoration is rate limited.

Not Applicable Data Flow Sending encrypted backups Is Potentially Interrupted (Sending encrypted backups)

Category Denial Of Service

Priority Low

Description An external agent interrupts data flowing across a trust boundary in either direction.

Justification Out of scope.

Mitigated Destination Client May be Subject to Elevation of Privilege Using Remote Code Execution (Sending encrypted backups)

Category Elevation Of Privilege
Priority High
Description Source Client may be able to remotely execute code for Destination Client.
Justification Client application performs extensive input validation and is designed to use safe data processing methods.

Mitigated Elevation by Changing the Execution Flow in Destination Client (Sending encrypted backups)

Category Elevation Of Privilege
Priority High
Description An attacker may pass data into Destination Client in order to change the flow of program execution within Destination Client to the attacker's choosing.
Justification Client application performs extensive input validation and is designed to use safe data processing methods.

Appendix B

User guide

This appendix contains the guide for the client application and a guide for running the server. It is also available in a convenient website form in the attachment.

B.1 User guide

This section will explain how to install and use the client application.

B.1.1 Installation

The package contains precompiled executables for Linux (built on Fedora Linux 37, it might not run on other distributions) and Windows (tested on Windows 10). The executables are self-contained, only relying on standard system dynamically linked libraries. When on supported systems, simply run the executable in a terminal, where a first start guide should be displayed. After setting up the application for the first time, the web user interface will be available.

B.1.1.1 Building from source

The application can also be built from source. To build it, you will need the Rust toolchain. However, it's easy to install by clicking the link and following the guide.

After installing the Rust toolchain, the application (including all Rust libraries) automatically be built and started by running the following command. Make sure that you are in the `client` folder.

Compile-time configuration constants can be changed in the `defaults.rs` file.

```
cargo run --release
```

B.1.2 Usage

The client application needs to connect to a server. The default server address is `127.0.0.1:9999`, but it can be overridden with the environment variable `SERVER_ADDR`.

By default, the clients enforce TLS on HTTPS/WebSocket connections with the server. This can be overridden by setting the `USE_TLS` environment variable to `1`.

Important: TLS is only designed to be disabled for testing purposes. It's mandatory to use TLS in production.

Backuwup client expects to run in a terminal environment and with ANSI color escapes supported (works on common Linux and modern Windows versions). When the application is started for the first time, a command line first start guide will show up. The user can choose to either set up the application from scratch (when creating a new backup) or from an existing mnemonic (for restoring an existing backup). When starting from scratch, it's necessary to save the displayed mnemonic to be able to perform a restore later if the application data is destroyed.

After the first setup guide is completed, a web user interface is started. To access it, simply open the displayed link in a browser. The default address of the web user interface is `http://127.0.0.1:3000` (and is accessible only on the local computer). This can be overridden by the environment variable `UI_BIND_ADDR`. For example, setting it to `0.0.0.0:3001` will change the port to `3001` and allow access from outside the local computer.

The web user interface offers easy access to main functions with buttons at the top and displays basic progress graphically. Below the panel at the top, there's a window that shows logs. The terminal window where the application is running will show even more detailed logs.

B.1.2.1 Backups and restores

Before starting a backup or a restore, a path needs to be set. That can be done in the right section of the user interface.

Backups After a backup is started, backuwup will begin scanning the path and packaging files for transport.

Storage requests After starting a backup, the client will attempt to negotiate storage space with other peers (which is done via the server). If no peers are available for a backup, the backup will be paused, and the client will wait. Packaging files and transporting them will be done simultaneously, up to a certain specified maximum size of local files.

The client will send a storage request to the server right after a backup starts. If there's no other request received by the server, it will be put in a queue and wait for storage requests from other clients. After the server receives other requests, they will be matched. If one request is for a greater amount of space than the other, it will only match up to the smaller size, and the unfulfilled part will get enqueued again and wait for other requests to come in.

> Note for testing
>
> For creating backups, it's easiest to have two clients which have data of about the same size in their backup path. After starting a backup on one client, one can see that files start getting packaged and then the backup pauses. Starting a backup on the second client will then send another storage request to the server, they will get matched, and both clients will be able to complete the backup and successfully transport data to the other peer.
> Having datasets with sizes that differ too much will result in the backup never

being completed due to not having a storage partner. The matching system is based on a one-to-one storage ratio (with a certain buffer).

After a backup is completed, the snapshot ID of the completed backup will be sent to the server.

Peer-to-peer communication The clients can only connect to each other if they are on the same local network and a firewall is not blocking a direct connection. The backuwup client will attempt to get the local IP address and a random port, which will be relayed through a server.

Restores Triggering a backup restore will first request and retrieve all files from all contacted peers. After all packaged data is retrieved, it will be unpacked into the backup path.

Currently, when restoring a backup, backuwup will attempt to contact **all** peers with any negotiated storage, no matter how many files were saved to that peer. For that reason, a client needs to be able to connect to all previously used peers to successfully restore a backup.

B.1.3 Notes

Application data is stored in the paths shown in the following tables. These paths can be overridden by setting the environment variables `CONFIG_DIR` or `DATA_DIR`.

Linux

Type	Path
Configuration files	`$XDG_CONFIG_HOME/backuwup` or `$HOME/.config/backuwup`
Data from other clients and temporary backup/restore files	`$XDG_DATA_HOME/backuwup` or `$HOME/.local/share/backuwup`

Windows

Type	Path
Configuration files	`%LocalAppData%/backuwup`
Data from other clients and temporary backup/restore files	`%LocalAppData%/backuwup`

B.2 Server setup guide

This section will explain how to run the required server application.

B.2.1 Using Docker Compose

The easiest way to run the server is to use the bundled configuration files for Docker Compose on Linux. Docker will automatically build and launch a container with the

server executable and the required PostgreSQL database. To use this method, please ensure you have Docker and Docker Compose installed and are in the root folder of the implementation code.

Now, run the following command:

```
docker-compose up --build
```

After the build process, the server should now be running locally with port `9999`.

If you want to destroy all data and run everything again, run this command:

```
docker-compose down
```

B.2.2 Manual build process

It's also possible to run the server without Docker. The server binary can be compiled and launched in an equivalent way to the client binary, except it depends on PostgreSQL development libraries instead of SQLite.

The PostgreSQL database needs to be run and managed separately. Credentials for the database can be set up in the `server/.env` file.

B.2.3 Production deployment

For production deployment, it's recommended to use the Docker setup, along with a reverse proxy (like nginx) that provides TLS support with a valid certificate. Using TLS in production is mandatory.

Bibliography

1. Postmortem of database outage of January 31. *GitLab* [online]. 2017 [visited on 2023-02-12]. Available from: https://about.gitlab.com/blog/2017/02/10/postmortem-of-database-outage-of-january-31/.
2. PALMER, Danny. Ransomware: How the NHS learned the lessons of WannaCry to protect hospitals from attack. *ZDNET* [online]. 2021 [visited on 2023-02-12]. Available from: https://www.zdnet.com/article/ransomware-how-the-nhs-learned-the-lessons-of-wannacry-to-protect-hospitals-from-attack/.
3. LINUS TECH TIPS. *3nm CPUs Are Coming! – WAN Show July 2, 2021* [video]. 2021 [visited on 2023-02-12]. Available from: https://www.youtube.com/watch?v=NCYNftA4EYM&t=1750. From 29:10 to 33:37.
4. KOŠŤÁL, David; ∼NEBULA∼. *pluto* [comp. software]. 2022 [visited on 2023-03-17]. Available from: https://github.com/profi248/pluto.
5. *Online Data Backup | Offsite, Onsite & Cloud | Crashplan* [online]. 2016 [visited on 2023-02-14]. Available from: https://web.archive.org/web/20160811085303/https://www.crashplan.com/en-us/. Originally available at https://www.crashplan.com/en-us/, snapshot from 2016-08-11.
6. Backing Up To A Friend's Computer – Code42 Support. *Code42* [online]. 2016 [visited on 2023-02-13]. Available from: https://web.archive.org/web/20160803190612/http://support.code42.com/CrashPlan/4/Backup/Backing_Up_To_A_Friends_Computer. Originally available at http://support.code42.com/CrashPlan/4/Backup/Backing_Up_To_A_Friends_Computer, snapshot from 2016-08-03.
7. BARBOSA, Greg. CrashPlan for Home being discontinued, refers customers to Carbonite – 9to5Mac. *9to5Mac* [online]. 2017 [visited on 2023-02-14]. Available from: https://9to5mac.com/2017/08/22/crashplan-home-being-discontinued/.
8. How Backup Works – Code42 Support. *Code42* [online]. 2016 [visited on 2023-02-14]. Available from: https://web.archive.org/web/20160830054040/http://support.code42.com/CrashPlan/4/Backup/How_Backup_Works. Originally available at http://support.code42.com/CrashPlan/4/Backup/How_Backup_Works, snapshot from 2016-08-30.

9. Backup Settings Reference – Code42 Support. *Code42* [online]. 2016 [visited on 2023-02-14]. Available from: https://web.archive.org/web/20160731060142/http://support.code42.com/CrashPlan/4/CrashPlan_App_Reference/backup_Settings_Reference. Originally available at http://support.code42.com/CrashPlan/4/CrashPlan_App_Reference/backup_Settings_Reference, snapshot from 2016-07-31.
10. Security Settings Reference – Code42 Support. *Code42* [online]. 2016 [visited on 2023-02-14]. Available from: https://web.archive.org/web/20160803014604/http://support.code42.com/CrashPlan/4/CrashPlan_App_Reference/Security_Settings_Reference. Originally available at http://support.code42.com/CrashPlan/4/CrashPlan_App_Reference/Security_Settings_Reference, snapshot from 2016-08-03.
11. Archive Encryption Key Security – Code42 Support. *Code42* [online]. 2016 [visited on 2023-02-14]. Available from: https://web.archive.org/web/20160803022809/http://support.code42.com/CrashPlan/4/Configuring/Archive_Encryption_Key_Security. Originally available at http://support.code42.com/CrashPlan/4/Configuring/Archive_Encryption_Key_Security, snapshot from 2016-08-03.
12. BuddyBackup software and website online | BuddyBackup Blog. *BuddyBackup* [online]. 2010 [visited on 2023-02-14]. Available from: https://web.archive.org/web/20200929190658/https://blog.buddybackup.com/2010/04/19/buddybackup-software-and-website-online/. Originally available at https://blog.buddybackup.com/2010/04/19/buddybackup-software-and-website-online/, snapshot from 2020-09-29.
13. BuddyBackup User's Guide. *BuddyBackup* [online]. 2010 [visited on 2023-02-13]. Available from: https://web.archive.org/web/20220527024833/https://www.buddybackup.com/download/BuddyBackupManual.pdf. Originally available at https://www.buddybackup.com/download/BuddyBackupManual.pdf, snapshot from 2022-05-27.
14. BuddyBackup is closing down | BuddyBackup Blog. *BuddyBackup* [online]. 2022 [visited on 2023-02-14]. Available from: https://web.archive.org/web/20220526091140/https://blog.buddybackup.com/2022/02/21/buddybackup-is-closing-down/. Originally available at https://blog.buddybackup.com/2022/02/21/buddybackup-is-closing-down/, snapshot from 2022-05-26.
15. Introducing the MQTT Protocol – MQTT Essentials: Part 1. *HiveMQ* [online]. 2015 [visited on 2023-03-17]. Available from: https://www.hivemq.com/blog/mqtt-essentials-part-1-introducing-mqtt/.
16. Syncthing: Open Source Continuous File Synchronization [comp. software]. 2023 [visited on 2023-04-28]. Available from: https://github.com/syncthing/syncthing.
17. BORG, Jakob. Untrusted Device Encryption — Syncthing documentation [online]. 2021 [visited on 2023-04-28]. Available from: https://docs.syncthing.net/specs/untrusted.html.
18. POORTVLIET, Jos. Nextcloud introduces peer-to-peer backup app for private instances, beta available now [online]. 2021 [visited on 2023-04-27]. Available from: https://nextcloud.com/blog/press_releases/pr20211101/.

19. *Storj – Make the world your data center* [online]. 2023 [visited on 2023-04-28]. Available from: https://www.storj.io/.

20. BEACH, Brian. Erasure Coding: Backblaze Open Sources Reed-Solomon Code [online]. 2015 [visited on 2023-04-28]. Available from: https://www.backblaze.com/blog/reed-solomon/.

21. restic: Fast, secure, efficient backup program [comp. software]. 2023 [visited on 2023-04-28]. Available from: https://github.com/restic/restic.

22. *References — restic 0.15.1 documentation* [online]. 2018 [visited on 2023-04-19]. Available from: https://restic.readthedocs.io/en/stable/100_references.html.

23. Threats – Microsoft Threat Modeling Tool [online]. 2022 [visited on 2023-04-27]. Available from: https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats.

24. FLORENCIO, Dinei; HERLEY, Cormac. A Large-Scale Study of Web Password Habits. In: *Proceedings of the 16th International Conference on World Wide Web*. Banff, Alberta, Canada: Association for Computing Machinery, 2007, pp. 657–666. WWW '07. ISBN 9781595936547. Available from DOI: 10.1145/1242572.1242661.

25. PALATINUS, Marek; RUSNAK, Pavol; VOISINE, Aaron; BOWE, Sean. BIP-39 [online]. 2013 [visited on 2023-04-27]. Available from: https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki.

26. RESCORLA, Eric. *The Transport Layer Security (TLS) Protocol Version 1.3* [RFC 8446]. RFC Editor, 2018. Request for Comments, no. 8446. Available from DOI: 10.17487/RFC8446.

27. sshd(8) — OpenBSD manual pages [online]. 2023 [visited on 2023-04-27]. Available from: https://man.openbsd.org/sshd.

28. BERNSTEIN, Daniel J.; DUIF, Niels; LANGE, Tanja; SCHWABE, Peter; YANG, Bo-Yin. Ed25519: high-speed high-security signatures [online]. 2017 [visited on 2023-04-27]. Available from: https://ed25519.cr.yp.to/.

29. XIA, Wen; ZOU, Xiangyu; JIANG, Hong; ZHOU, Yukun; LIU, Chuanyi; FENG, Dan; HUA, Yu; HU, Yuchong; ZHANG, Yucheng. The Design of Fast Content-Defined Chunking for Data Deduplication Based Storage Systems. *IEEE Transactions on Parallel and Distributed Systems*. 2020, vol. 31, no. 9, pp. 2017–2031. Available from DOI: 10.1109/TPDS.2020.2984632.

30. restic · Foundation - Introducing Content Defined Chunking (CDC) [online]. 2015 [visited on 2023-04-21]. Available from: https://restic.net/blog/2015-09-12/restic-foundation1-cdc/.

31. O'CONNOR, Jack; AUMASSON, Jean-Philippe; NEVES, Samuel; WILCOX-O'HEARN, Zooko. BLAKE3 [online]. 2021 [visited on 2023-04-23]. Available from: https://raw.githubusercontent.com/BLAKE3-team/BLAKE3-specs/master/blake3.pdf.

32. WIKIPEDIA CONTRIBUTORS. *Birthday attack — Wikipedia, The Free Encyclopedia* [online]. 2023 [visited on 2023-04-20]. Available from: https://en.wikipedia.org/w/index.php?title=Birthday_attack&oldid=1147539823.

33. COLLET, Yann; KUCHERAWY, Murray. *Zstandard Compression and the 'application/zstd' Media Type* [RFC 8878]. RFC Editor, 2021. Request for Comments, no. 8878. Available from DOI: 10.17487/RFC8878.

34. THE RUST PROJECT DEVELOPERS. *The Rustonomicon* [online]. 2023 [visited on 2023-04-22]. Available from: https://doc.rust-lang.org/stable/nomicon/.

35. WIKIPEDIA CONTRIBUTORS. *Network address translation — Wikipedia, The Free Encyclopedia* [online]. 2023 [visited on 2023-04-30]. Available from: https://en.wikipedia.org/w/index.php?title=Network_address_translation&oldid=1152346420.

36. *Syncthing Usage Reports* [online]. 2023 [visited on 2023-04-30]. Available from: https://data.syncthing.net/.

37. WIKIPEDIA CONTRIBUTORS. *Internet Gateway Device Protocol — Wikipedia, The Free Encyclopedia* [online]. 2022 [visited on 2023-04-30]. Available from: https://en.wikipedia.org/w/index.php?title=Internet_Gateway_Device_Protocol&oldid=1118754023.

38. ROSENBERG, Jonathan; KERÄNEN, Ari; LOWEKAMP, Bruce; ROACH, Adam. *TCP Candidates with Interactive Connectivity Establishment (ICE)* [RFC 6544]. RFC Editor, 2012. Request for Comments, no. 6544. Available from DOI: 10.17487/RFC6544.

39. KERÄNEN, Ari; HOLMBERG, Christer; ROSENBERG, Jonathan. *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal* [RFC 8445]. RFC Editor, 2018. Request for Comments, no. 8445. Available from DOI: 10.17487/RFC8445.

40. WIKIPEDIA CONTRIBUTORS. *STUN — Wikipedia, The Free Encyclopedia* [online]. 2023 [visited on 2023-04-30]. Available from: https://en.wikipedia.org/w/index.php?title=STUN&oldid=1152282467.

41. WIKIPEDIA CONTRIBUTORS. *Traversal Using Relays around NAT — Wikipedia, The Free Encyclopedia* [online]. 2023 [visited on 2023-04-30]. Available from: https://en.wikipedia.org/w/index.php?title=Traversal_Using_Relays_around_NAT&oldid=1152277782.

42. GASPARI, Fabio De; HITAJ, Dorjan; PAGNOTTA, Giulio; CARLI, Lorenzo De; MANCINI, Luigi V. *EnCoD: Distinguishing Compressed and Encrypted File Fragments*. 2020. Available from arXiv: 2010.07754 [cs.CR].

43. WIKIPEDIA CONTRIBUTORS. *Erasure code — Wikipedia, The Free Encyclopedia* [online]. 2023 [visited on 2023-04-30]. Available from: https://en.wikipedia.org/w/index.php?title=Erasure_code&oldid=1140623712.

44. GUPTA, Minaxi; JUDGE, Paul; AMMAR, Mostafa. A Reputation System for Peer-to-Peer Networks. *Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video*. 2003. Available from DOI: 10.1145/776322.776346.

45. KAMVAR, Sepandar D.; SCHLOSSER, Mario T.; GARCIA-MOLINA, Hector. The Eigentrust Algorithm for Reputation Management in P2P Networks. In: *Proceedings of the 12th International Conference on World Wide Web*. Budapest, Hungary: Association for Computing Machinery, 2003, pp. 640–651. WWW '03. ISBN 1581136803. Available from DOI: 10.1145/775152.775242.

Libraries

46. OVERBY, Ty; RIORDAN, Zoey; TRANGAR. *bincode 1.3.3* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/bincode`.
47. ALEX; SUNLI. *poem 1.3.55* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/poem`.
48. MCARTHUR, Sean. *reqwest 0.11.14* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/reqwest`.
49. GALAKHOV, Alexey; ABRAMOV, Daniel. *tokio-tungstenite 0.18.0* [comp. software]. 2022 [visited on 2023-04-22]. Available from: `https://crates.io/crates/tokio-tungstenite`.
50. CTZ; OCHTMAN, Dirkjan. *rustls 0.21.1* [comp. software]. 2023 [visited on 2023-05-08]. Available from: `https://crates.io/crates/rustls`.
51. TOLNAY, David. *serde 1.0.152* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/serde`.
52. TOLNAY, David. *serde_json 1.0.93* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/serde_json`.
53. HIPP, Richard. *SQLite* [comp. software]. 2023 [visited on 2023-05-03]. Available from: `https://www.sqlite.org/index.html`.
54. YOU, Yuxi (Evan). *Vue 3.2.74* [comp. software]. 2023 [visited on 2023-04-23]. Available from: `https://vuejs.org/`.
55. THE BOOTSTRAP AUTHORS. *Bootstrap v5.3.0-alpha3* [comp. software]. 2023 [visited on 2023-04-23]. Available from: `https://getbootstrap.com`.
56. ALLAN. *dotenvy 0.15.7* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/dotenvy`.
57. RUIZ, Mariano. *sum-queue 1.0.0* [comp. software]. 2021 [visited on 2023-04-22]. Available from: `https://crates.io/crates/sum-queue`.
58. ARCIERI, Tony. *aes-gcm 0.10.1* [comp. software]. 2022 [visited on 2023-04-22]. Available from: `https://crates.io/crates/aes-gcm`.
59. TOLNAY, David. *anyhow 1.0.69* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/anyhow`.
60. TOLNAY, David. *async-trait 0.1.68* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/async-trait`.

61. ROOSE, Steven. *bip39 2.0.0* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/bip39`.
62. O'CONNOR, Jack. *blake3 1.3.3* [comp. software]. 2022 [visited on 2023-04-22]. Available from: `https://crates.io/crates/blake3`.
63. APARICIO, Jorge. *cast 0.3.0* [comp. software]. 2021 [visited on 2023-04-22]. Available from: `https://crates.io/crates/cast`.
64. MANNING, Age. *delay_map 0.3.0* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/delay_map`.
65. RONACHER, Armin; SUNKARA, Pavan Kumar. *dialoguer 0.10.3* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/dialoguer`.
66. SOC. *dirs 5.0.0* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/dirs`.
67. LOVECRUFT, isis agora. *ed25519-dalek 1.0.1* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/ed25519-dalek`.
68. RAIN. *enable-ansi-support 0.2.1* [comp. software]. 2022 [visited on 2023-04-22]. Available from: `https://crates.io/crates/enable-ansi-support`.
69. FIEDLER, Nathan. *fastcdc 3.0.2* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/fastcdc`.
70. CRICHTON, Alex. *filetime 0.2* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/filetime`.
71. KURILENKO, Denis. *fs_extra 1.3.0* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/fs_extra`.
72. CRICHTON, Alex; CRAMER, Taylor; ENDO, Taiki. *futures 0.3.26* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/futures`.
73. CRAMER, Taylor; ENDO, Taiki. *futures-util 0.3.26* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/futures-util`.
74. HARDY, Diggory. *getrandom 0.2.8* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/getrandom`.
75. KOKAKIWI. *hex 0.4.3* [comp. software]. 2021 [visited on 2023-04-22]. Available from: `https://crates.io/crates/hex`.
76. FILIPPOV, Vlad; PAVLOV, Artyom. *hkdf 0.12.3* [comp. software]. 2022 [visited on 2023-04-22]. Available from: `https://crates.io/crates/hkdf`.
77. B., Namkhai. *human_bytes 0.4.1* [comp. software]. 2022 [visited on 2023-04-22]. Available from: `https://crates.io/crates/human_bytes`.
78. BLUSS; WRENN, Jack. *itertools 0.10.5* [comp. software]. 2022 [visited on 2023-04-22]. Available from: `https://crates.io/crates/itertools`.
79. BORAI, Esteban. *local-ip-address 0.5.1* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/local-ip-address`.
80. REIZNER, Yevhenii. *memmap2 0.5.10* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/memmap2`.
81. JAM1GARNER. *owo-colors 3.5.0* [comp. software]. 2022 [visited on 2023-04-22]. Available from: `https://crates.io/crates/owo-colors`.

82. GOREGAOKAR, Manish. *pathdiff 0.2.1* [comp. software]. 2021 [visited on 2023-04-22]. Available from: `https://crates.io/crates/pathdiff`.

83. KARPPILA, Hannes. *portpicker 0.1.1* [comp. software]. 2021 [visited on 2023-04-22]. Available from: `https://crates.io/crates/portpicker`.

84. HARDY, Diggory. *rand_chacha 0.3.1* [comp. software]. 2021 [visited on 2023-04-22]. Available from: `https://crates.io/crates/rand_chacha`.

85. JOHN, Peter. *rust-embed 6.4.2* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/rust-embed`.

86. ARCIERI, Tony; PAVLOV, Artyom. *sha2 0.10.6* [comp. software]. 2022 [visited on 2023-04-22]. Available from: `https://crates.io/crates/sha2`.

87. BONANDER, Austin; LECKEY, Ryan. *sqlx 0.6* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/sqlx`.

88. TOLNAY, David. *thiserror 1.0.39* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/thiserror`.

89. LERCHE, Carl; RYHL, Alice. *tokio 1.25.0* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/tokio`.

90. LERCHE, Carl; RYHL, Alice. *tokio-stream 0.1.12* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/tokio-stream`.

91. BURY, Alexandre. *zstd 0.12.3* [comp. software]. 2023 [visited on 2023-04-22]. Available from: `https://crates.io/crates/zstd`.

92. ADAMS, Vernon. *Mulish – Google Fonts* [font]. 2021 [visited on 2023-04-28]. Available from: `https://fonts.google.com/specimen/Mulish`.

93. THE MOZILLA FOUNDATION. *Fira Code – Google Fonts* [font]. 2023 [visited on 2023-04-28]. Available from: `https://fonts.google.com/specimen/Fira+Code`.

Attachment contents

Implementation

- `readme.txt` ..... a summary of the attachment contents
- `src` ..... the full implementation source code
- `bin` ..... executables for the implementation
- `guide` ..... a website form of guide in Appendix B
- `doc` ..... generated source code documentation

Thesis

- `thesis.pdf` ..... the thesis in the PDF format
- `src` ..... the LaTeX source code of this thesis