Project title:-

Enhancing data confidentiality and integrity in public cloud storage through a client-side cryptographic overlay.

Module Code	**: 7030CEM**
Module Title	**: Cyber Security Individual Project**
Student Name	**: KHUSHAL PATEL**
Student Id	**: 15405366**
Date of Submission	**: 11th August 2025**

Declaration of Originality

Statement of copyright

Statement of ethical engagement

Signed: **KHUSHAL PATEL** Date: 11th August 2025

First Name:	KHUSHAL
Last Name:	PATEL
Student ID number	15405366
Ethics Application Number	P187477
1st Supervisor Name	Florence Nkosi

Acknowledgements

I would like to express my deepest gratitude to my project supervisor, **Florence Nkosi**, for her invaluable guidance, continuous support, and insightful feedback throughout this research project. Her expertise and encouragement were instrumental in navigating the complexities of this work and bringing it to a successful conclusion.

I would also like to extend my thanks to the faculty and staff at Coventry University for providing the resources and academic environment that made this project possible.

Finally, a special and heartfelt thank you to my family and friends for their unwavering support, patience, and belief in me throughout my studies. This journey would not have been the same without them.

KHUSHAL PATEL

SID: - 15405366

Coventry University, Coventry

Table of Contents

Table of figures

Abstract

basic privacy flaw that forces consumers to entrust providers with their encryption keys and data. The design, implementation, and thorough assessment of a client-side encryption (CSE) system that enforces a verifiable zero-knowledge architecture are presented in this study to address this challenge. The web application prototype makes use of well-known cryptographic primitives, such as PBKDF2 for strong, password-based key derivation and AES-256-GCM for authenticated encryption. Cloud storage services' prevalent server-side encryption model produces a empirical verification of the system's security claims is one of this work's main contributions.

Following a set of focused tests guided by the OWASP Web Security Testing Guide, the architecture's capacity to safeguard data integrity and confidentiality from a compromised server was validated. The results show that using common web technologies to create a highly secure CSE system is feasible. The risk of irreversible data loss is identified as the main adoption barrier by the analysis, which also emphasises the crucial trade-off between user-centric key management and complete security. A validated blueprint for private-by-design cloud applications is contributed by this work, which ends by highlighting important research directions, such as the incorporation of post-quantum cryptography and advanced metadata protection.

Links

Github :- https://github.com/gitkhushalpatel/dissertation-project.git

Chapter 1. Introduction

Cloud computing's widespread use has transformed data storage by providing previously unheard-of accessibility and convenience. But there are now serious privacy and security issues because of this change. Users give up direct control of their data when they entrust it to third-party cloud providers, making them susceptible to security breaches and monitoring (Zissis, 2012). Although server-side encryption is a popular defence, the provider frequently controls the keys, making user data vulnerable and introducing a single point of failure. Client-side encryption (CSE) provides a strong technical solution, but previous attempts have frequently been thwarted by poor usability, which raises the adoption barrier. This has resulted in a crucial tension in system design. This research aims to develop a system that is both verifiably secure and practically usable to address the "adoption paradox," which is the situation where technically superior security is compromised by user-centric challenges.

Client-side encryption (CSE) has become a potent substitute to allay these worries. CSE guarantees that the cloud provider only ever receives and stores ciphertext by encrypting user data on their device before uploading it to the cloud. A far greater degree of data confidentiality and control is offered by the encryption keys, which stay in the user's possession. But putting CSE into practice in web applications comes with its own set of difficulties, especially when it comes to user authentication, secure key management, and general usability (I. P. A. G. A. P, 2023).

By planning, creating, and testing a secure cloud storage application that makes use of client-side encryption and **zero-knowledge principles**, this research project tackles these issues. By making sure that the server is unaware of the user's encryption keys or the contents of their files, the system seeks to offer a workable and intuitive solution that ensures data privacy.

1.1. Research Questions

This project is guided by the following research questions:

1. If the cloud server is compromised, how can a web-based, client-side encryption system be created and put into place to guarantee that user data is unintelligible to it?

2. Which cryptographic methods are best for managing, wrapping, and deriving user encryption keys in a browser environment to stop unwanted access?

3. To what degree can a zero-knowledge authentication and data handling protocol be put into place to offer strong security without sacrificing end-user usability?

1.2. Key Objectives

The following primary goals will be pursued by the project to address these research questions:

1. **Create and put into use a client-side encryption protocol** that encrypts and decrypts files right in the user's browser by utilising cutting-edge, reliable cryptographic libraries (AES-256-GCM).

2. To ensure that raw keys are never kept on the server, **create a secure key derivation and management system** that uses PBKDF2 to create a master key from the user's password. This master key is then used to unwrap individual file keys.

3. To show the usefulness of the suggested system**, create a prototype web application** with JavaScript for the frontend and Python (Flask) for the backend.

4. **Perform a thorough security assessment** of the prototype, testing for data integrity, confidentiality, and the server's inability to access plaintext data.

By developing a useful artefact whose security architecture and usability trade-offs can be empirically assessed, these goals taken together aim to methodically address the research questions.

1.3. Report Structure

The format of this report is as follows: The pertinent literature on client-side encryption, zero-knowledge proofs, and key management strategies is critically reviewed in Chapter 2. The architecture and design of the suggested system are described in detail in Chapter 3. The prototype application's implementation is covered in Chapter 4. The findings of the security assessment and testing are shown in Chapter 5. The results are finally discussed, the study's limitations are outlined, and future research directions are suggested in Chapter 6.

Chapter 2: Literature Review

2.1 Introduction: From Description to Critical Analysis

The academic and technical literature pertinent to the development of a safe and practical client-side encryption system is critically reviewed and synthesised in this chapter. This review's objectives go beyond simply summarising previous research; they also include critically assessing various methodologies, identifying current discussions, and determining the research gap that this project seeks to fill (Jesson, 2011).While a critical analysis assesses the importance, merits, and shortcomings of that work to create a strong case for further research, a descriptive summary merely summarises what has been done. (Petticrew, 2008). The main conflicts in CSE architectures, the subtle application of zero-knowledge principles, and the crucial area of usable security—which looks at the human elements that ultimately affect a security system's efficacy—are all covered in this review, which is organised thematically.

2.2 The Central Debate: Trust vs. Accessibility in CSE Architectures

Moving the role of trust from the service provider to the user's device is the fundamental idea behind client-side encryption (Abu-Salma, 2017). The application of this principle, however, has given rise to a major and continuing discussion in the literature about the architectural trade-offs between creating a computing base that can be relied upon and ensuring that users can access it as much as possible.

The difference between browser-based web apps and native apps is where this argument is most noticeable. Because the code is locally verified and less vulnerable to in-transit alteration, native applications—which are installed directly on a user's operating system—are better able to create a trusted environment. On the other hand, browser-based CSE solutions present a major controversy despite their superior platform independence and ease of deployment: the client-side code, usually JavaScript, that encrypts data is provided by the very server that is supposed to be untrusted (Johns, 2020).

This problem, known as the **"code delivery attack vector,"** takes the architectural discussion above and beyond the straightforward decision between web and native applications. It raises a serious question about the integrity of the client-side environment in general. Theoretically, a malicious server could distribute compromised code to intercept plaintext data or user credentials, totally undermining the cryptographic model. Thus, the main topic of discussion also includes how to create a chain of trust for the application code itself; this issue is covered later in this review using tools such as Content Security Policy (CSP).

This opens a crucial attack point where a malevolent server might send compromised code to steal the user's keys or password. Although methods such as "Crypto Membranes" have been suggested to separate cryptographic functions within browser extensions (Johns, 2020), they come with their own set of usability challenges, requiring users to install and maintain yet another piece of software.

The ongoing debate over key management causes this architectural challenge (Edler, 2023). A catastrophic failure mode that defeats the storage system's very purpose is when a user loses their master key, rendering their data permanently inaccessible (Ogorzalek, 2017). From straightforward user-managed backups (like printing a recovery key) to intricate cryptographic recovery mechanisms, the literature offers a range of mitigation techniques. But no one solution has proven to be better than another. Each strategy reflects a challenging trade-off between security and usability, which is the focus of this research project and a major unresolved issue in the field.

Solution	Architecture	Key Management	Browser Support	Usability
Your Prototype	Web-based	User-managed	Full	Good
Cryptomator	Desktop App	Password-based	N/A	Excellent
Box Cryptor	Hybrid	Cloud-assisted	Limited	Good
Spider Oak	Native Client	Server-managed	N/A	Fair

Table 2.2.0: CSE Solutions Comparison

2.2.1 Introduction to the Code Delivery Attack Vector

Before being sent to the server, sensitive data is encrypted locally in the user's environment, usually a web browser. This is a fundamental promise of any Client-Side Encryption (CSE) architecture. This guarantees that the user's unencrypted data cannot be accessed by the server operator or any adversary who compromises the server. As a "zero-knowledge" storage provider, the server can only handle opaque, encrypted data blobs. However, the integrity of the client-side code itself is a crucial and frequently disregarded premise that is necessary for the integrity of this entire security model.

A server-side framework (like Python Flask) oversees delivering the application's component parts to the user's browser in a standard web application architecture, like the one used by the prototype for this project. This is the HTML framework, CSS to style and finally and most importantly, the JavaScript that does the cryptographic work.

The script.js file, which has logic to derive keys, encrypt and decrypt, is served by app.py. This leaves a fundamental dependency: the browser of the client will need to implicitly trust the server to serve a trustworthy, non-malicious copy of this JavaScript file each time it is requested.

It is the point of the attack vector of code delivery or malicious server. By gaining access to the application server, an attacker can compromise the whole security promise of the CSE model without having to crack the underlying cryptography. Rather than attempt to break the encryption algorithms, the malicious party can just alter the script.js file prior to transmission to the user. An altered script might, as an example, steal the user password when entered, send the calculated master key to an attacker-controlled server, or send the plaintext of a file a few seconds before it is encrypted. The application would seem to work normally to the end-user, but their data would be stolen silently. Thus, the security of browser-based CSE system cannot just be a factor of cryptographic design but is intertwined with the security of the code delivery channel. This is a threat that should be recognized and addressed when developing a credible and sound CSE system.

This is no theoretical threat. A strong practical example is the modus operandi of the Magecart hacking groups. In many of the major breaches e.g. the recent example of British Airways in 2018, attackers were able to access web servers and tamper with legitimate JavaScript files. They inserted malicious so-called skimming code that intercepted customer payment information in the browser before it was entered. Although the target is different, the attack vector is the same as the one that endangers a browser based CSE system: a compromised server that is serving malicious code to an unsuspecting client. This precedent shows that server-side integrity is a fundamental requirement of any security assurance based on client-side execution of code.

2.2.2 Mitigation via sub resource Integrity (SRI)

Sub resource Integrity (SRI) is one of the major technical controls that have been formulated to solve the problem of compromised content delivery. SRI is a security property standardized by W3C allowing browsers to ensure that the resources they request, scripts or stylesheets, are not unexpectedly manipulated. The process is carried out by the addition of an integrity attribute to the HTML document elements <script> and <link>.

The integrity of the attribute value is a string with one or more cryptographic hashes (e.g. SHA-384) of the anticipated resource, prefixed with the hash algorithm and converted to base64. An example of a script tag may be as follows:

```
<script src="https://example.com/script.js" integrity="sha384-
oqVuAfXRKap7fdgcCY5uykM6+R9GqQ8K/uxy9rx7HNQlGYl1kPzQho1wx4JwY8wN"
crossorigin="anonymous"></script>
```

When a browser encounters an HTML element with an integrity attribute, it first loads the resource at the URL given. The browser first computes its own cryptographic hash of the content of the received file with the same algorithm identified by the attribute before executing the resource. Then it compares its computed hash with the hash value given in the integrity attribute. When both the hashes match, then the browser assumes that the file is genuine and runs it. When the hashes fail to match, then it is an indication that either there is a malicious alteration to the content of the file during transfer or on the server. Here, the browser will decline to execute the resource and raise an error and essentially this will stop the execution of the potentially malicious code.

As much as SRI is an effective measure to counter the manipulation of the external script files, its success will depend on the integrity of the document that holds the SRI attributes the file itself HTML. When it comes to an architecture where the same server distributes both the HTML (index.html) and the JavaScript (script.js) as it is the case with this project, the protection of SRI is not complete. Even a sophisticated attacker who has gained control of the server can just change both the file containing JavaScript and the hash value in the HTML file. The browser when receiving the malicious script and the corresponding malicious hash would be consistent and it would execute the code. Thus, SRI is an effective layer of protection, but not in and of itself if the main HTML document is provided by the same potentially untrusted source.

2.2.3 Mitigation via Content Security Policy (CSP)

Content Security Policy (CSP) is a more robust, broader-reaching mechanism of mitigating code delivery attacks. CSP is a defence-in-depth security feature, provided as an HTTP response header (Content-Security-Policy) that allows very granular control of the resources a browser can load and execute on a given page. Instead of just checking the integrity of a single resource, CSP introduces a wide policy under which a web application attack surface can be drastically minimized.

CSP works by use of a sequence of instructions where each instruction regulates a particular set of resource or activity. A number of directives are crucial to a CSE application:

- **script-src:** This directive is a white-list of allowed sources of JavaScript. One might be extremely conservative by restricting to scripts allowed on the same origin (called self) and listing the base64-encoded hashes of all anticipated scripts. This in effect, embeds the protection of SRI directly into the CSP header and achieves the same integrity check coupled with the ability to never allow scripts to be loaded anywhere but the authoritative location.
- **connect-src**:This directive limits the URLs a client can connect to using XMLHttpRequest, Fetch APIs and WebSockets. This is an important prevention control against data exfiltration. Even in the unlikely case where an attacker was able to inject malicious code that gets around the script-src control (e.g. via a browser vulnerability), a restrictive connect-src policy that only allows access to the API endpoint used by the application plus only the third party APIs needed by the application (e.g. Google Drive) would be sufficient to prevent the attacker malicious script getting access to the user password or keys and sending them to an attacker-controlled domain.
- **form-action:** This directive limits the URLs that may be submitted to by a form to reduce the risk of form hijacking to steal credentials.
- **object-src none:** An object-src directive protects against the embedding of potentially dangerous plugin content (e.g. Flash), which is a frequent source of client-side attacks.

A properly set CSP offers a stronger defence than SRI on its own since it works on the principle of proactive restriction as opposed to reactive verification. It can avoid an attack succeeding even though a malicious script is placed on the page since a whitelist of trusted sources and behaviours can be defined. In the case of the CSE prototype, such a feature as a CSP header that limits the script sources to a predetermined hash and connect-src to the specified API endpoints only would establish a hardened client environment, making unauthorized data exfiltration much harder to an attacker.

2.2.4 Synthesis and Architectural Implications

Sub resource Integrity and Content Security Policy are neither a magic bullet in code delivery problem in browser-based cryptography. SRI is efficient, but fragile, because its security depends on the integrity of a containing document. CSP can be strong yet complicated to setup correctly and might not stop all varieties of malicious code execution in the case of vulnerabilities in sources that have been whitelisted.

The current best practice, hence, is a multiple defence-in-depth strategy with the combination of the two mechanisms. Critical script files using SRI offers a clear, explicit integrity check whereas strict CSP offers a policy-based broader enforcement that defends against a broader set of attacks such as data exfiltration.

This architectural analysis has deep architectural implications in the case of the prototype of this dissertation. It shows that security of the system is not limited to cryptographic logic of script.js but needs to be secured through secure operations deployment of the application. The Flask server (app.py) has to be set to serve the application with the right and limiting Content-Security-Policy HTTP header. This moves the boundary of trust; the user does not have to trust the server with his or her data, but the user must trust the administrator of the server to set up the web server correctly. This subtlety is essential, because it changes the meaning of the prototype not as a perfectly safe, self-contained system, but as an application architecture whose security assurances are based on both strong cryptography and good server-side security hygiene. This is the recognition that must accompany any plausible assertion of offering secure client-side encryption in the web.

2.3 A Matter of Definition: The "Zero-Knowledge" Principle in System Design

In cryptography, the phrase zero-knowledge has a particular, technical meaning: a Zero-Knowledge Proof (ZKP). Formal ZKP A formal ZKP is an interactive protocol in which a prover is able to convince a verifier that a statement is true, without revealing any additional information about his/her statement (Goldwasser, 1989). These are very strong protocols that are however computationally costly and intricate to use.

But, more broadly used in system architecture, there is a definitional issue as to whether the term should also be used to describe systems architecture. In this project and most of the literature on secure cloud systems, the term zero-knowledge describes an architectural property as opposed to a formal proof (Yang, 2025). A server with zero knowledge of sensitive data of the user implies a zero-knowledge system (the user in question), that is, the plaintext files of the user and the decryption keys (Edler, 2023). This is an important difference. The system is not demonstrating knowledge of a secret in the mathematical cryptographic sense; instead it is architecturally sound such that the server is never able to possess the knowledge. This architectural paradigm is a real-life embodiment of the zero-trust security model, which dictates the fact that no entity is trusted by default (Gilman, 2017). The main point of criticism of this case is that although the system does realize the spirit of a zero-knowledge protocol (server does not know anything), it does not use the formal cryptographic apparatus of a ZKP, which is important to make the distinction for academic purposes.

2.4 The Adoption Paradox: Why Technically Superior Security Fails

Security system cannot be effective unless it is used and used properly. This is a mere fact that leads to the so-called adoption paradox: technically more appropriate security solutions cannot find a following because its usability poses an insurmountable obstacle to non-expert users (Whitten, 1999). The usable security and privacy field of study examines this paradox, and there is an abundant evidence base that systems lacking usability are often bypassed, configured incorrectly, or simply abandoned, which results in devastating security breaches (Krol, 2019).

The currently most notable case study of this paradox is end-to-end encrypted email. Such technologies as PGP (Pretty Good Privacy) have been available to people decades ago and they are cryptographically sound, but their usage is trivial (Fবার্টzle, 2021). The fatal flaw of PGP is that it is usable, not its cryptography. Research findings repeatedly reveal that the users are burdened by the technicality of managing public keys, such as generating keys, distributing keys and verifying keys (Ruoti, 2020). This points to a fatal issue in technocentric approach to security. The literature strongly holds the argument that the cognitive load imposed on the user is part of the attack surface of the system just as much as any vulnerability in software. Thus, a new CSE system should be assessed not only according to its cryptographic capabilities, but also by its capability to overcome this adoption paradox by factoring security into the workflow of the user.

2.5 Synthesis and Identification of the Research Gap

The literature has firmly entrenched the necessity of client-side encryption and lays out a deep toolbox of cryptographic primitives. This critical review however shows that there is an enormous gap between design of cryptographically secure systems and actual development of products that can be used and adopted. The paradox of adoption along with the debates about architecture illustrates that technical soundness is not good enough.

In this review, the identified **gap in the body of work** is the following: a lot of papers are about CSE in theory, or they present a description of a custom-made system, but there is a lack of studies that provide a full picture of how one can design and implement and, most importantly, rigorously test a CSE system on a major, mainstream cloud platform, such as Google Drive. The literature is not well represented by existing works that clearly justify their cryptography decisions (e.g. selection of a key derivation function) against current industry best practices (such as those recommended by an organization like OWASP). Moreover, not many of them merge a formal security verification, which is based on the identified debates, with a critical assessment of the system usability and the possibility to resolve the adoption paradox.

So, the main gap mentioned in the literature is not the availability of cryptographic tools, but the reported, comprehensive use of the tools in a mass setting. This project is a direct filler of this gap. It does not only present a prototype, but a full design, implementation and evaluation story that addresses the architectural, cryptographic and usability requirements in an end-to-end way. The main contribution is this comprehensive description, which presents an empirically validated roadmap, which answers the trust-vs-accessibility question, which presents literature-informed reasoning on the cryptographic decisions it made, and which considers the lessons learned by the mistakes in usability of previous technologies.

Chapter 3: System Design and Methodology

3.1 Introduction and Design Philosophy

The methodology embraced in this project is a **Design Science Research (DSR)**. DSR is a paradigm based on the development and testing of novel IT artifacts- including models, constructs, and systems, which address real world problems. In accordance with this approach, the identified gap in safe cloud storage is filled by this research in the following ways:

1. **developing and creating** a new software artifact, which is the zero-knowledge prototype application.
2. **critically analyzing** the performance and security of the artifact by means of the testing procedures outlined in Chapter 5.
3. **producing a contribution** as a validated blueprint and design principles of secure, private-by-design web applications. This methodology is a direct correspondence to the main task within this project of developing and testing a feasible solution to an identified issue within the industry.

The current chapter describes the architectural design and technical approach that was used to create the prototype secure storage application. The core philosophy of the design is based on direct research objectives and on the critical literature review in Chapter 2. This philosophy can be characterized by three guiding principles:

1. **Zero-Knowledge Architecture**: The architecture of the system is designed so that the server is absolutely unaware of the content of user files and cryptographic keys to protect them. This is the main principle of the security assurance of the system.
2. **Client-Centric Trust:** The cryptographic sensitive operations, especially encryption, decryption and key generation, are all done on the client-side (the browser of the user). The server is considered as a non-trustful machine that performs only storage and retrieval of opaque blocks of information.
3. **Pragmatic Usability:** In addition to strong security, the design is simplicity and conformity to common conventions on the web to limit the adoption paradox as noted in the literature (Whitten, 1999). This is intended to enable strong encryption and avoid an unreasonable cognitive load on the user.

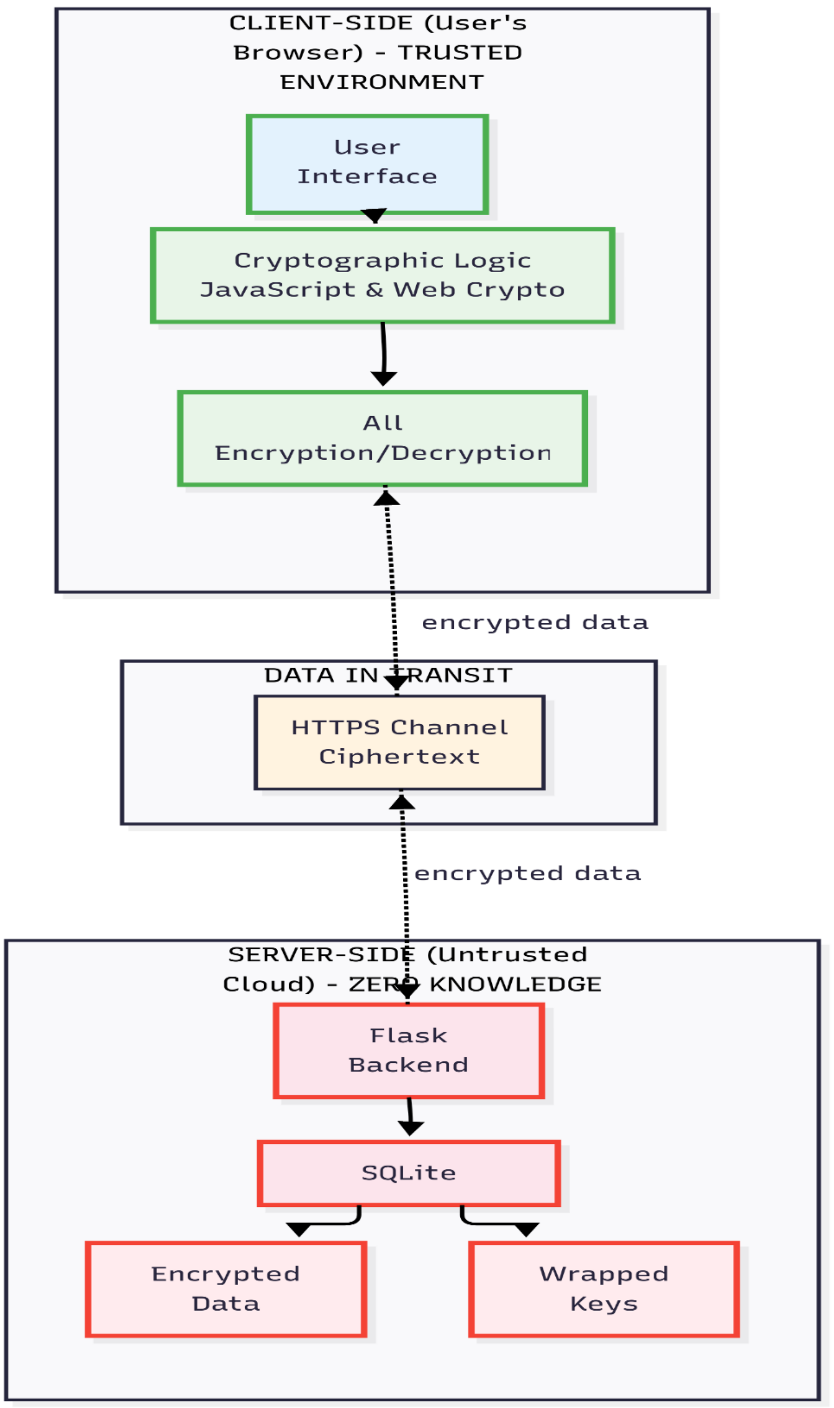
The chapter will begin by describing the high-level system architecture and explaining the selection of the technologies. It will then give a step by step explanation including pictures in sequence diagrams of the cryptographic protocols which constitute the foundation of the system. Lastly, it will specify system threat model and security assurances.

3.2 System Architecture

The software is deployed in a traditional two-tier client-server architecture, which is rather appropriate when it comes to implementing web applications. With this model, the user interface and the data storage logic are well separated.

- **The Client (Frontend):** This is a web interface created on HTML, CSS and vanilla JavaScript that is completely executed by the user on the web browser. It is subject to all user interactions but most importantly, it is the only place that all cryptographic functions can take place. It also talks to the backend using a RESTful API.
- **The Server (Backend):** This is Python application developed with the use of Flask micro-framework. All it has to do is user authentication (registration and log in), user account management and be a plain storage gateway. It gets the data that is already encrypted by the client and stores it without the possibility to review its contents. The information on the encrypted file and the metadata of users are stored in a SQLite database.

This architecture as shown in Figure 3.1, leaves the role of the server reduced to that of a basic, dumb storage provider and this is in direct line with the zero-knowledge design philosophy.


CLIENT-SIDE (User's Browser) - TRUSTED ENVIRONMENT
User Interface
Cryptographic Logic JavaScript & Web Crypto
All Encryption/Decryption
encrypted data
DATA IN TRANSIT
HTTPS Channel Ciphertext
encrypted data
SERVER-SIDE (Untrusted Cloud) - ZERO KNOWLEDGE
Flask Backend
SQLite
Encrypted Data
Wrapped Keys

1Figure 3.1: High-Level System Architecture

3.3 Technology Stack Justification

The criterion that governed the choice of technologies was simplicity, security, and rapid prototyping.

The frontend (Vanilla JavaScript, Web Crypto API):

Justification: Unlike something as complicated as React or Angular, vanilla JavaScript was used to make the client-side codebase simple and plain. This is serious security concern; a leaner and simpler codebase will be easier to review and its size of attack will be reduced. The inner cryptographic operations are carried out on the **Web Crypto API**, a W3C standard that is directly implemented in up to date browsers. It is a safe low-level approach to the cryptographic modules underlying the browser and the preferable best practice to execute cryptography in a web-based environment (W3C., (2017).).

Feature	Web Crypto API	CryptoJS	Forge	Native App
Performance	Excellent	Good	Fair	Excellent
Security	High	Medium	Medium	High
Browser Support	Wide	Universal	Universal	N/A
Bundle Size	0KB	300KB	500KB	N/A
Hardware Acceleration	Yes	No	No	Yes

Table 3.3.0: Web Crypto API vs Alternatives

Backend (Python, Flask, SQLite):

Justification: Python is used because of its readability and its large selection of libraries. Flask is a lightweight, so-called micro-framework which gives you the minimal tools to build a REST API without being overbearing with a strict structure or bundling in components you do not need. This is minimalist and would suit a prototype with the logic in the backend kept deliberately simple. The database engine selected was SQLite since it is both self-contained and serverless and does not need any complicated setup, thus being an ideal choice in a single-server prototype. The database structure (see init_db.py) is composed of a users table to store account data and a wrapped keys table in which the metadata of each ciphered file will be tracked.

3.4 The Cryptographic Core: A Zero-Knowledge Protocol

The whole system is secure through the well thought out scenario of cryptographic protocols that do not allow the sensitive information to be accessed at any single point by the server. The protocol can be divided into three major stages: user registration, encryption and decryption of a file.

3.4.1 Why PBKDF2 over ARGON2

PBKDF2 was intentionally selected over its more modern alternatives, the most well known of which is **Argon2**, which won the Password Hashing Competition. Though, when recommended, Argon2 is usually suggested to be used in new systems because of its increased resistance to brute forcing with custom hardware.

A key, practical consideration in choosing **PBKDF2** as the target of this project is that it is natively and standardized in the **Web Crypto API.**

All of the client-side parts of this project use this W3C standard to guarantee a lean and clear codebase as a fundamental security aspect. Opting for

PBKDF2 enables the prototype to use all key derivations in the browser with native vetted cryptographic functions both it is easier and reduces the overall complexity of the system and therefore the attack surface. Given that

PBKDF2 is a solid industry best practice suggested by groups such as OWASP and is set with an iteration count very high (100,000) in this implementation; it is more than adequate to establish the baseline security in the context of the goals of this prototype.

3.4.2 User Registration and Master Key Derivation

All the cryptographic keys of a user have only one root of trust and that is the password. A strong key derivation function is needed in order to convert this low entropy secret into a high entropy cryptographic key.

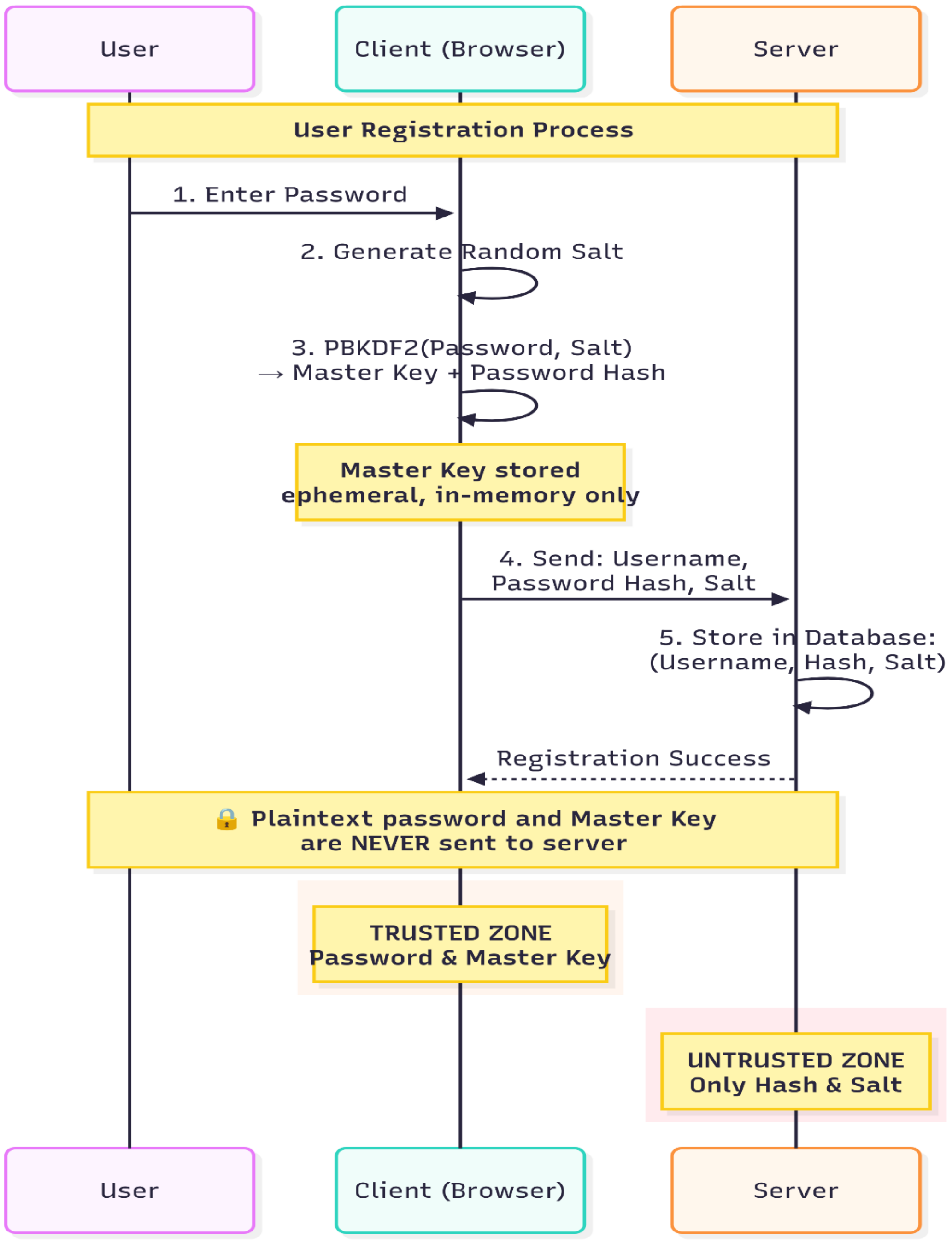
Process: During registration, a user is not transferred a password selected by him/her to the server. Rather, the JavaScript client-side resorts to the Password-Based Key Derivation Function 2 (PBKDF2) to generate a 256-bit Master Key.

Reason: Applying a naive hash algorithm such as SHA-256 to a password is not satisfactory and not secure. PBKDF2 is the recommended standard in the industry that is proposed by organizations, such as the **Open Web Application Security Project (OWASP)** since this algorithm includes two important security provisions:

1. **A Salt:** A special, random salt is created on per user basis. This salt is held on the server and would allow attackers not to use pre-computed rainbow tables and crack several passwords at one time.
2. **Iteration Count:**The application in question is by design computationally demanding as it runs the kernel hashing algorithm (HMAC-SHA-256) numerous times (e.g. 100,000+ iterations). This renders brute force cracking of a single password too slow and costly.

Zero-Knowledge Principle: The **Master Key** that is derived will only exist in the memory of the browser during the session of the user. It never gets sent to the server, the server cannot compute it, because the server stores the hash of the password (to log in), and the salt.

The sequence of this process is illustrated in Figure 3.2.


User
Client (Browser)
Server
User Registration Process
1. Enter Password
2. Generate Random Salt
3. PBKDF2(Password, Salt) → Master Key + Password Hash
Master Key stored ephemeral, in-memory only
4. Send: Username, Password Hash, Salt
5. Store in Database: (Username, Hash, Salt)
Registration Success
🔒 Plaintext password and Master Key are NEVER sent to server
TRUSTED ZONE Password & Master Key
UNTRUSTED ZONE Only Hash & Salt
User
Client (Browser)
Server

2Figure 3.2: User Registration and Master Key Derivation Sequence

3.4.3 File Encryption and Key Wrapping

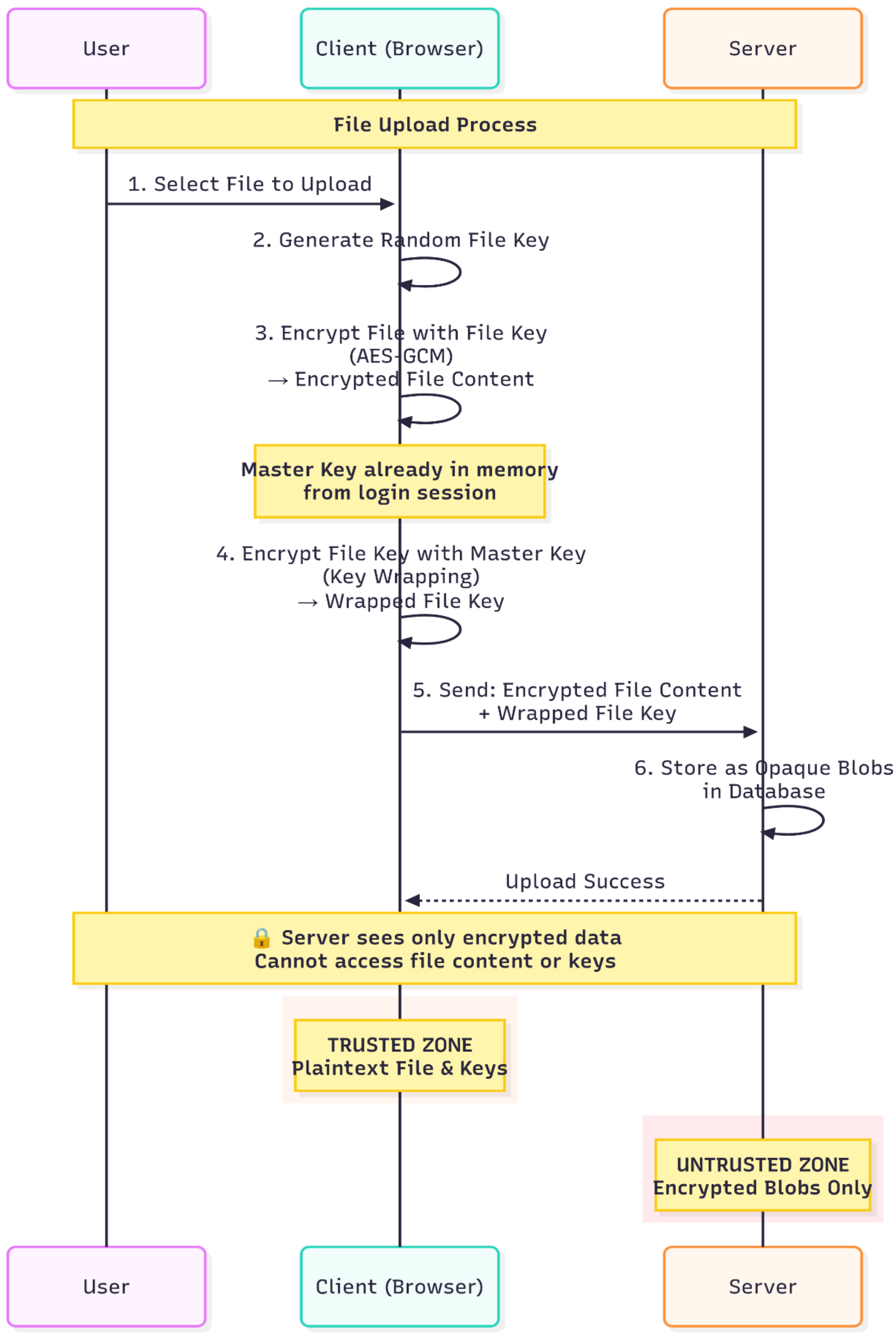
A hierarchical key structure is employed so that direct file encryption using the Master Key (a bad cryptographic idea) is avoided. A distinct encryption key is used on each file.

Process:

1. Once a user selects the file he wants to upload, a new, cryptographically random 256-bit **File Key** will be generated using the client-side JavaScript.
2. This File Key encrypts the content of the file with the **AES-256-GCM** algorithm. AES-GCM is **an Authenticated Encryption with Associated Data (AEAD)** cipher, which is to say that it achieves both **confidentiality** (the data cannot be read) and **integrity**/**authenticity** (the data cannot be undetectably modified).
3. The File Key then itself is encrypted (referred to as key wrapping), using the **Master Key** of the user (also with AES-GCM).
4. The client, after securing the content of the file, then transfers the encrypted content of the file, the wrapped File Key and the requisite metadata (IVs, salt) to the server where it is stored.

Justification: This key-per-file method is very flexible and compatible with cryptographic truism of key separation. As a reason, **AES-GCM** is selected because it is a recent, very secure and efficient authenticated cipher which is recommended by the U.S. National Institute of Standards and Technology ((Dworkin, 2007)).

Figure 3.3 below shows the process as follows:


User
Client (Browser)
Server
File Upload Process
1. Select File to Upload
2. Generate Random File Key
3. Encrypt File with File Key (AES-GCM) → Encrypted File Content
Master Key already in memory from login session
4. Encrypt File Key with Master Key (Key Wrapping) → Wrapped File Key
5. Send: Encrypted File Content + Wrapped File Key
6. Store as Opaque Blobs in Database
Upload Success
🔒 Server sees only encrypted data Cannot access file content or keys
TRUSTED ZONE Plaintext File & Keys
UNTRUSTED ZONE Encrypted Blobs Only
User
Client (Browser)
Server

3Figure 3.3: File Encryption and Upload Sequence

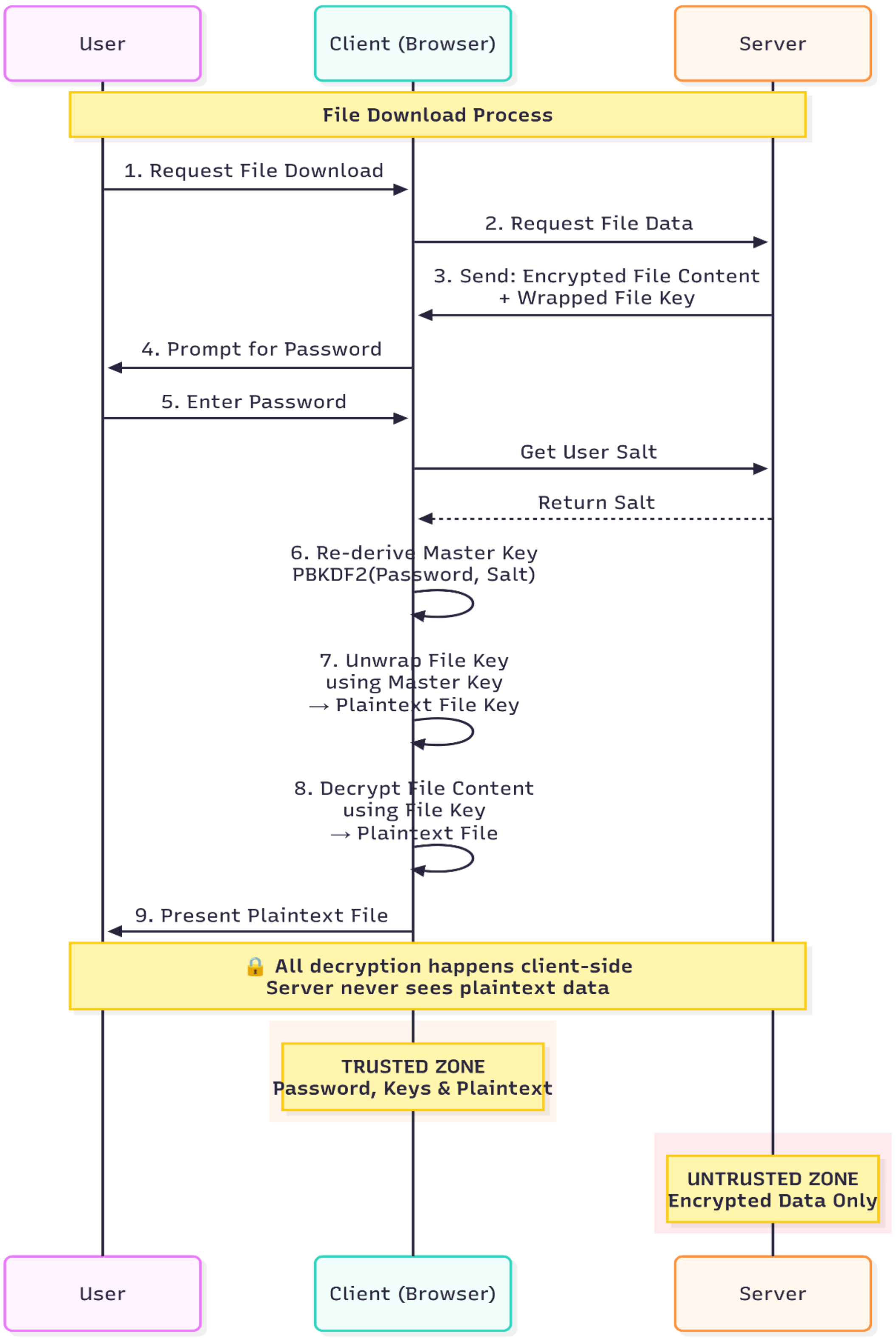
3.4.4 File Decryption and Key Unwrapping

The decryption is simply the opposite of encryption and again all the sensitive work is done on the client.

Process:

1. The user clicks a file to download. The server provides the encrypted files data and its wrapped File Key to the client.
2. The user is asked to enter his or her password. The password and a salt stored on the server is retrieved by the user and used to re-derive the **Master Key**.
3. The client makes use of the obtained Master Key to decrypt (unwrap) the File Key which has been wrapped.
4. The client can now decrypt the file content with the plaintext **File Key** through the AES-GCM.
5. The user is then presented with an option of downloading the plaintext file which had never been posted on the server.

Figure 3.4 below shows the steps of decryption and downloading.


User
Client (Browser)
Server
File Download Process
1. Request File Download
2. Request File Data
3. Send: Encrypted File Content + Wrapped File Key
4. Prompt for Password
5. Enter Password
Get User Salt
Return Salt
6. Re-derive Master Key PBKDF2(Password, Salt)
7. Unwrap File Key using Master Key → Plaintext File Key
8. Decrypt File Content using File Key → Plaintext File
9. Present Plaintext File
🔒 All decryption happens client-side Server never sees plaintext data
TRUSTED ZONE Password, Keys & Plaintext
UNTRUSTED ZONE Encrypted Data Only
User
Client (Browser)
Server

4Figure 3.4: File Decryption and Download Sequence

3.5 Security Methodology and Threat Model

To make the security analysis formal, the system is engineered and analysed considering the particular threat model through the application of the STRIDE framework. STRIDE is a threat model developed by Microsoft that can be employed to raise and classify any possible security vulnerability. The framework gives a systematic method to make sure that all probable risks are put under consideration. The threat model presupposes an

Honest-but-curious or compromised server provider , in which the attacker may gain complete read/write access to the filesystem and database on the server but not to the client device itself.

Any design decision in the system is analyzed in terms of each of the six categories in STRIDE:

Threat Category	Risk Level	Mitigation	Implementation
Spoofing	Medium	Password Hashing	Werkzeug check_password_hash
Tampering	Low	AES-GCM Auth	Authentication tags
Repudiation	Low	Cryptographic Binding	Master Key derivation
Information Disclosure	Low	Client-side Encryption	Web Crypto API
Denial of Service	High	Operational Controls	Backup/Redundancy
Elevation of Privilege	Low	Key Separation	Hierarchical keys

Table 3.5.0 STRIDE Threat Analysis Matrix

- **Spoofing**: This is a type of threat that entails unauthorized being gaining access to the system through impersonating a user. The prototype counteracts this on the authentication level. Users are authenticated by the server by matching the hash of a submitted password with the one of the stored password with the help of check_password_hash Werkzeug function. As the plaintext password is never stored and nor is handled by the server, the compromised database administrator cannot spoof the user by knowing the user password. A salt is also unique to each user and an attacker cannot use one stolen hash to log in as a user on a different system.
- **Tampering**: This is a threat to the unauthorized alteration of data. The design of the system is resistant to this with high levels of cryptographically enforced protection. The major mitigation is that of using
- The main mitigation is **AES-256-GCM** of file encryption. Being an Authenticated Encryption with Associated Data (AEAD) cipher, GCM mode generates an authentication tag, in addition to the ciphertext. This is as it has been empirically confirmed during security test SVT-02, any alteration in the ciphertext at rest, including just a single byte, will lead to the authentication tag verification process to fail during the decryption process on the client-side. This will help ensure data integrity since tampered information is repelled instantly.
- **Repudiation**: This threat is a user repudiating that he/she took action when in fact it was performed. The existing prototype contains few mitigations to repudiation in particular, since it does not establish extensive audit logs. The cryptographic binding of a file to the Master Key of the user though gives a form of data creation non-repudiation.

It is cryptographically infeasible that a user can claim not to have created a particular encrypted file based on his or her account because it is only the user knowing the correct password that can calculate the Master Key used to generate a valid wrapped file key.

- **Information Disclosure:** This forms the major threat that the system would help to prevent. The zero-knowledge structure makes it impossible that both the server and an attacker who gained access to the server can see the plaintext of files belonging to its users. Encryption and decryption is solely undertaken on the client-side, a principle which was proven correct in test SVT-01. The system however recognizes some level of metadata leakage. Although filenames are encrypted, an attacker having access to the server can monitor the file sizes, creation timestamps and access patterns. This remaining data could be taken to be a factor in user activity, which is a well-known weakness of this architectural style.
- **Denial of Service (DoS):** This risk is associated with blocking the access to the service to the rightful users. By deleting user information in database or shutting down the server, an attacker with server access may easily launch DoS attack. This threat is not defended against by the architecture of the prototype because its main objective is securing the confidentiality and integrity of the data. This would be alleviated by production systems that have operational security controls such as redundant infrastructure, backups and network-based DoS defense.
- **Elevation of Privilege:** The threat refers to a user acquiring privileges that he/she is not supposed to. In the system, the major risk would be the ability of one user to access information of another user. This is well guarded by the hierarchical key wrapping model. As testified in test SVT-05 is the most important unwrapping operation which cannot be executed successfully unless executed with correct Master Key. Therefore, A user (User A) cannot decrypt a file owned by another user (User B), since User A does not have the correct Master Key with which to unwrap the file key of the file owned by User B. This is cryptographically enforced separation of user accounts.

Attack Vector	Probability	Impact	Risk Score	Mitigation
Server Compromise	Medium	Low	Medium	Zero-knowledge design
Password Brute Force	High	High	High	PBKDF2 + High iterations
Man-in-the-Middle	Low	Medium	Low	HTTPS/TLS
Client Malware	Medium	High	High	Out of scope
Code Injection	Low	High	Medium	CSP headers

3.5.1 Attack Vector Analysis

3.6 Security of Third-Party Integration: Google Drive

When plugging in a third party service such as Google Drive, new security issues are introduced, most of which revolve around the process of authorizing and tokens. The prototype is based on the OAuth 2.0 protocol to access the user drive according to the modern security best practice.

The safety of this integration depends on some major principles:

1. Authorization through OAuth 2.0: Rather than working directly with the user credentials to Google, the application instead makes uses of the standard OAuth 2.0 authorization code flow. The user identifies with Google and in response is given access token by Google in the form of a limited time access token to the application. This token provides the program with certain restricted rights to the account of the user without displaying his or her password in any occasion.
2. Principle of Least Privilege (Scope Minimization): The application only prompts the smallest range of permission that it requires to work. Within app.py SCOPES= [' https://www.googleapis.com/auth/drive.file. This level of access is important since it only allows access to the files generated by the application. It also does not allow reading, editing, or even viewing other files within the Google Drive of the user hence extremely reducing the possible damage a compromised application server can cause.
3. Secure Token Management: The OAuth 2.0 flow supplies not only an access token (involving shortlived time), but a re-generation token (involving longlived time). It is the duty of the application server to keep these tokens safely in the users table. The access token is employed in calling the API and when the same expires, the refresh token can be employed to have a new access token generated by Google without necessarily having the user log in again. The most important thing is secure storage of these tokens on server. This would need to encrypt at rest within a production system the database column that holds these tokens.

This is necessary in a production system with encryption-at-rest on the database column that contains such tokens. The consequences of compromised refresh token would be dramatic. With this token, an attacker could, until that token is revoked, create new access tokens without further interaction on the part of the user. Whereas the urge. Such file scope access is commendable as it only exposes them to files created by the application, and yet the attacker may access them in a persistent and covert manner. They might repeatedly copy offline all the user encrypted files to analyse or corrupt them maliciously and the user would not be immediately aware. This emphasizes the fact that the security of the own database of the server is an extremely important connection in the chain of trust.

3.7 Comparative Analysis of Cryptographic Primitives

Cryptographic primitives are one of the most important decisions in a cryptographic design in terms of long-term security. Although the prototype proves PBKDF2 and AES-GCM, a complete examination needs the comparison of these decisions to their main competitors. The given section presents a literature-based justification in assessing the trade-offs between various important derivation functions and authenticated encryption ciphers, showing that one is familiar with the general cryptographic context.

Algorithm	Key Size	Performance	Security Level	Browser Support

PBKDF2	N/A	Fast	Good	Native
Argon2	N/A	Medium	Excellent	Library Required
AES-GCM	256-bit	Very Fast	Excellent	Native
ChaCha20-Poly1305	256-bit	Fast	Excellent	Limited

Table 3.7 Cryptographic Primitive Comparison

3.7.1 Key Derivation Functions: PBKDF2 vs. scrypt vs. Argon2

A slow resource-intensive key derivation function is the first-line of defence against an offline brute-force attack on user passwords. The aim is that cracking one password should be computationally prohibitively expensive that the cracker, even with access to the full database of users, will not be able to accomplish the task. PBKDF2, scrypt, and Argon2 are the best candidates.

- **PBKDF2**: In the implementation used in this project, the advantage of PBKDF2 is that its iteration count is configurable imposing a time-based work factor. The main benefits are its long history, its popularity in the industry, and it being introduced into industry standards such as NIST SP 800-132. It has native support in the Web Crypto API and is thus the default option to use in browsers when requiring cryptography with no external dependencies. Nevertheless, its key limitation is that it has purely time-based computational requirement. This renders it vulnerable to major optimisation using dedicated, parallelised hardware e.g. Graphics Processing Units (GPUs) and Application-Specific Integrated Circuits (ASICs) that are capable of hundreds or thousands of times as many calculations per second as a typical CPU.
- **scrypt**: Developed in 2009 by Colin Percival, scrypt was created expressly to mitigate the danger of physical assaults. Its most important innovation is memory hardness. As well as the time cost being configurable, scrypt consumes a lot of RAM to compute also. Whereas the computer of a legitimate user can have the required memory sufficient to sustain a single log-in, an attacker trying to check millions of passwords in parallel would have to purchase an unrealistic amount of RAM, which would make mass attacks too costly. This memory-hard characteristic gives a higher level of defence against both GPU and ASIC-based cracking than PBKDF2.
- **Argon2**: Argon2 won the 2015 Password Hashing Competition, and is the state-of-the-art which is recommended by most cryptographers to implement in new systems. It refines scrypt since it provides memory hardness, time hardness, and resists timing attacks. It has two primary variants: Argon2d (which maximises resistance to cracking with GPUs: it has data-dependent memory access) and Argon2i (which has data-independent memory access, making it optimised against side-channel attacks). The hybrid Argon2id is currently the suggested default choice of most web applications since it offers balanced and strong protection against either type of assault.

Synthesis and Justification: Although technically speaking, Argon2id is better to use in new system, the decision to use PBKDF2 in the given prototype is reasonable and pragmatic. Security of the project was based on the overarching principle of reducing the area of attack on the client side only using the native Web Crypto API. Since Argon2 is currently not included in such a standard, its use will entail adding a third-party JavaScript library, which will also break this principle and would result in an externally sourced dependency that would have to be vetted. Thus, the architectural choice of relying on the native, FIPS-compliant PBKDF2 implementation of the browser with a large iteration count (100,000) was the right one regarding the purposes of this particular project.

3.7.2 Authenticated Ciphers: AES-GCM vs. ChaCha20-Poly1305

In order to have both integrity as well as confidentiality, an authenticated encryption cipher is necessary in encrypting the contents of the files. AES-GCM and ChaCha20-Poly1305 are the two most popular standards.

- **AES-GCM:** The industry-standard cipher that is implemented in the prototype is AES-GCM also NIST-approved. Its greatest strength is high performance on new hardware. Majority of the newer CPUs (Intel and AMD) incorporate an AES-NI instruction set that hardware accelerates AES operations. This makes decryption and encryption very quick and also makes the implementation resistant against side-channel timing attacks. It is a highly vetted and popular one since it is a NIST standard.
- **ChaCha20-Poly1305:** ChaCha20-Poly1305 is IETF standardised as RFC 8439 and has become the most popular alternative to AES-GCM, used by organizations such as Google and Cloudflare in TLS. It is an ARX cipher (operates on Add-Rotate-XOR). The key benefit is that it provides a way to perform a lot of cool stuff and provide good security on those platforms which do not have any dedicated AES hardware, like low-power mobile or old embedded platforms. It is not vulnerable to the cache-timing attacks that may occur in software-only implementations of AES since it was constructed bottom up to be resistant to such attacks.

Synthesis and Justification :AES-GCM is a great and justified option in the context of a web application deployed to modern desktop and high-end mobile browsers where hardware acceleration is provided almost everywhere. The Web Crypto API also offers a secure, standardised API to this hardware with strong performance and security guarantees. It should be noted, however, that had the application been targeted at a different environment, e.g. that of a lightweight mobile app or a low-power IoT device, ChaCha20-Poly1305 would offer an attractive, and in many cases at least equivalent, alternative, since its high performance has been implemented in software.

Chapter 4: Implementation

4.1 Introduction

The chapter describes how the secure storage system is actually implemented, how the architecture and cryptographic protocols agreed on in Chapter 3 are turned into a working software prototype. Its main aim is to report some physical presence of how the system was built and how the main ideas behind the concept of having zero-knowledge and the client-side trust on the principle were achieved in the code.

It is implemented in two major parts such as the server-side backend system which is built using Flask Python framework, and the client-side frontend system which is built using common HTML, CSS and JavaScript. Major snippets of the code pertaining to both parts shall be provided in this chapter detailing their purpose and specifically extrapolating it back to the design requirements.

4.2 Backend Implementation: The Untrusted Server

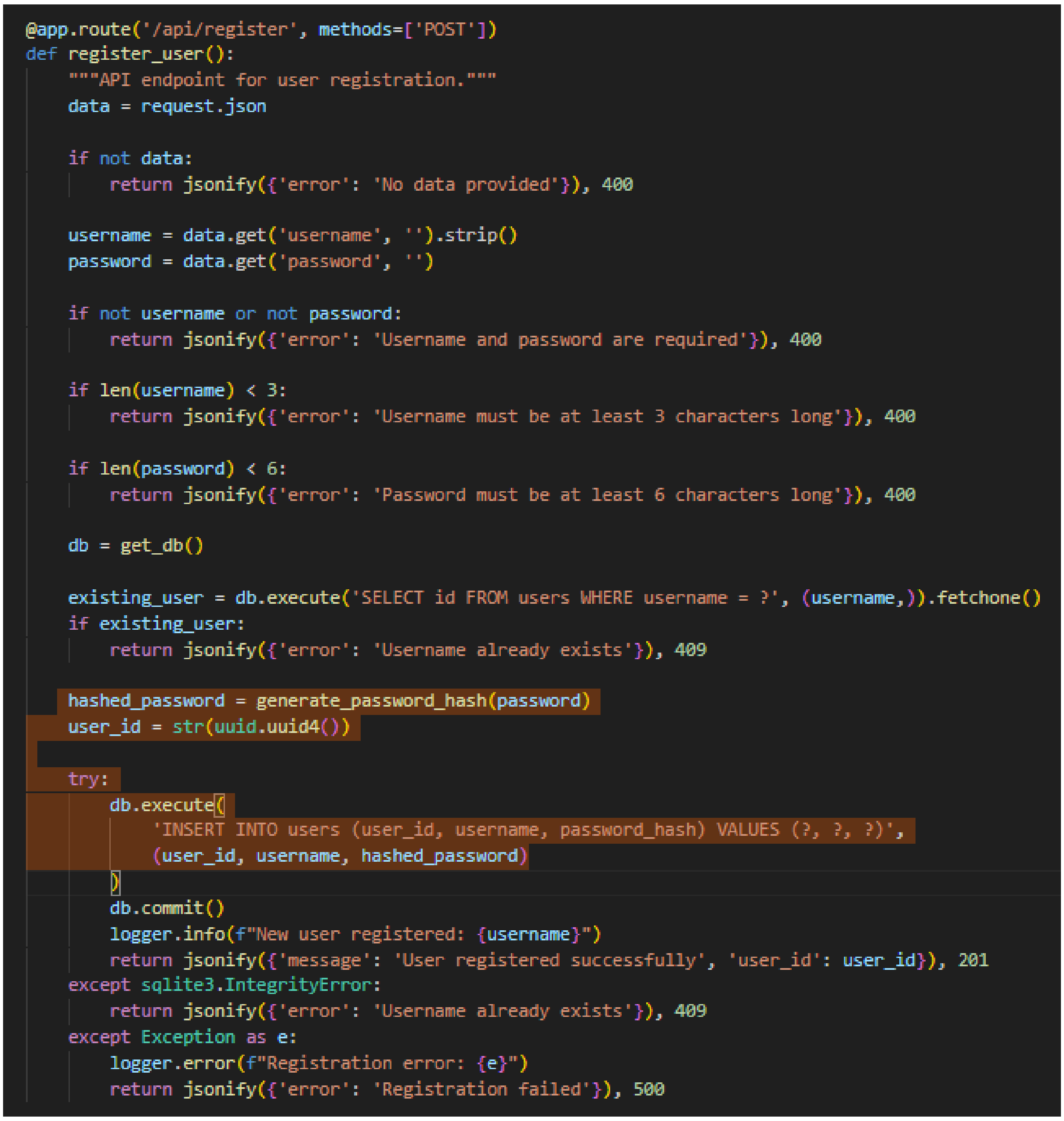
Backend logic was created in Python and Flask micro-framework and its functionality was confined to managing users and storing opaque, encrypted data blobs, in the spirit of zero-knowledge design principle.

4.2.1 Database and API Setup

The backend is based on the app.py script that initializes the Flask application, and defines the API endpoints. The data store is represented by a SQLite database generated with the help of the init_db.py script. Database schema plays a significant role since it was made to store user credentials and file metadata without storing plaintext keys or content. An example is the wrapped keys table, which only stores the encrypted file key (wrapped key), and cryptographic metadata (IVs, salt), and not the actual file data, which is stored as an opaque BLOB.

4.2.2 User Authentication Endpoints

The user registration and log in are done by the server. Importantly, the server never sees the raw password of the user. The client sends a pre-hashed password to the server when registering and hashes it. When a user is logging into a system it checks this stored hash during verification.



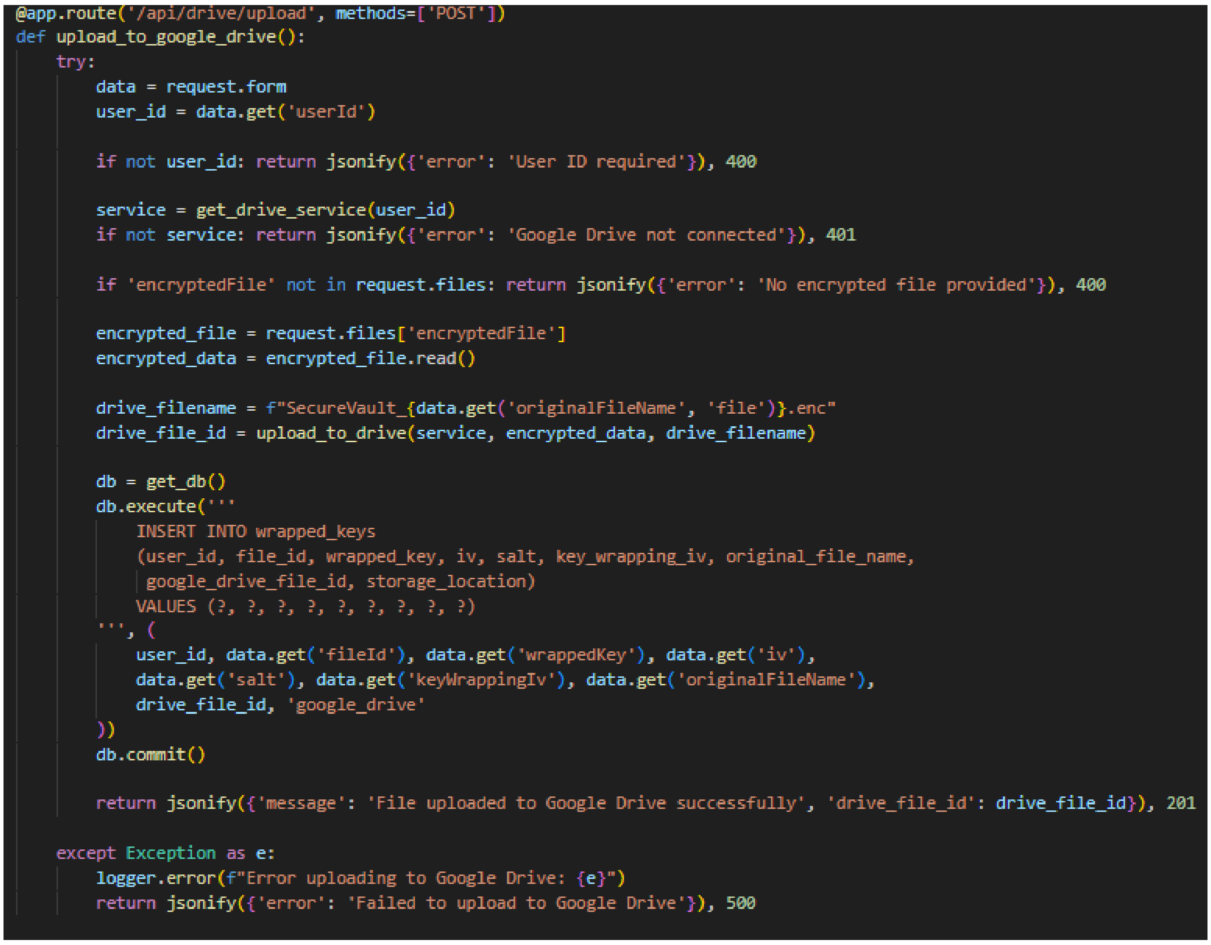
```python
@app.route('/api/register', methods=['POST'])
def register_user():
    """API endpoint for user registration."""
    data = request.json

    if not data:
        return jsonify({'error': 'No data provided'}), 400

    username = data.get('username', '').strip()
    password = data.get('password', '')

    if not username or not password:
        return jsonify({'error': 'Username and password are required'}), 400

    if len(username) < 3:
        return jsonify({'error': 'Username must be at least 3 characters long'}), 400

    if len(password) < 6:
        return jsonify({'error': 'Password must be at least 6 characters long'}), 400

    db = get_db()

    existing_user = db.execute('SELECT id FROM users WHERE username = ?', (username,)).fetchone()
    if existing_user:
        return jsonify({'error': 'Username already exists'}), 409

    hashed_password = generate_password_hash(password)
    user_id = str(uuid.uuid4())

    try:
        db.execute(
            'INSERT INTO users (user_id, username, password_hash) VALUES (?, ?, ?)',
            (user_id, username, hashed_password)
        )
        db.commit()
        logger.info(f"New user registered: {username}")
        return jsonify({'message': 'User registered successfully', 'user_id': user_id}), 201
    except sqlite3.IntegrityError:
        return jsonify({'error': 'Username already exists'}), 409
    except Exception as e:
        logger.error(f"Registration error: {e}")
        return jsonify({'error': 'Registration failed'}), 500
```

5Code Snippet 4.1: User Registration Endpoint

This implementation exactly meets the design requirement in Section 3.4.1 in that the server should not architecturally know the password of the user. It merely caches the important contents (password hash, salt) in order to subsequently authenticate a log in with the Werkzeug check_password_hash method.

4.2.3 File Storage Endpoints

The server is only a passive storage gateway in file handling. The client sends the API /api/upload a FormData object, which carries the encrypted file as well as all the metadata needed to perform the decryption later. When these pieces are received by the server, they are just extracted and placed into the database uncritically.



```python
@app.route('/api/drive/upload', methods=['POST'])
def upload_to_google_drive():
    try:
        data = request.form
        user_id = data.get('userId')

        if not user_id: return jsonify({'error': 'User ID required'}), 400

        service = get_drive_service(user_id)
        if not service: return jsonify({'error': 'Google Drive not connected'}), 401

        if 'encryptedFile' not in request.files: return jsonify({'error': 'No encrypted file provided'}), 400

        encrypted_file = request.files['encryptedFile']
        encrypted_data = encrypted_file.read()

        drive_filename = f"SecureVault_{data.get('originalFileName', 'file')}.enc"
        drive_file_id = upload_to_drive(service, encrypted_data, drive_filename)

        db = get_db()
        db.execute('''
            INSERT INTO wrapped_keys
            (user_id, file_id, wrapped_key, iv, salt, key_wrapping_iv, original_file_name,
             google_drive_file_id, storage_location)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
        ''', (
            user_id, data.get('fileId'), data.get('wrappedKey'), data.get('iv'),
            data.get('salt'), data.get('keyWrappingIv'), data.get('originalFileName'),
            drive_file_id, 'google_drive'
        ))
        db.commit()

        return jsonify({'message': 'File uploaded to Google Drive successfully', 'drive_file_id': drive_file_id}), 201

    except Exception as e:
        logger.error(f"Error uploading to Google Drive: {e}")
        return jsonify({'error': 'Failed to upload to Google Drive'}), 500
```

6Code Snippet 4.2: File Upload Endpoint

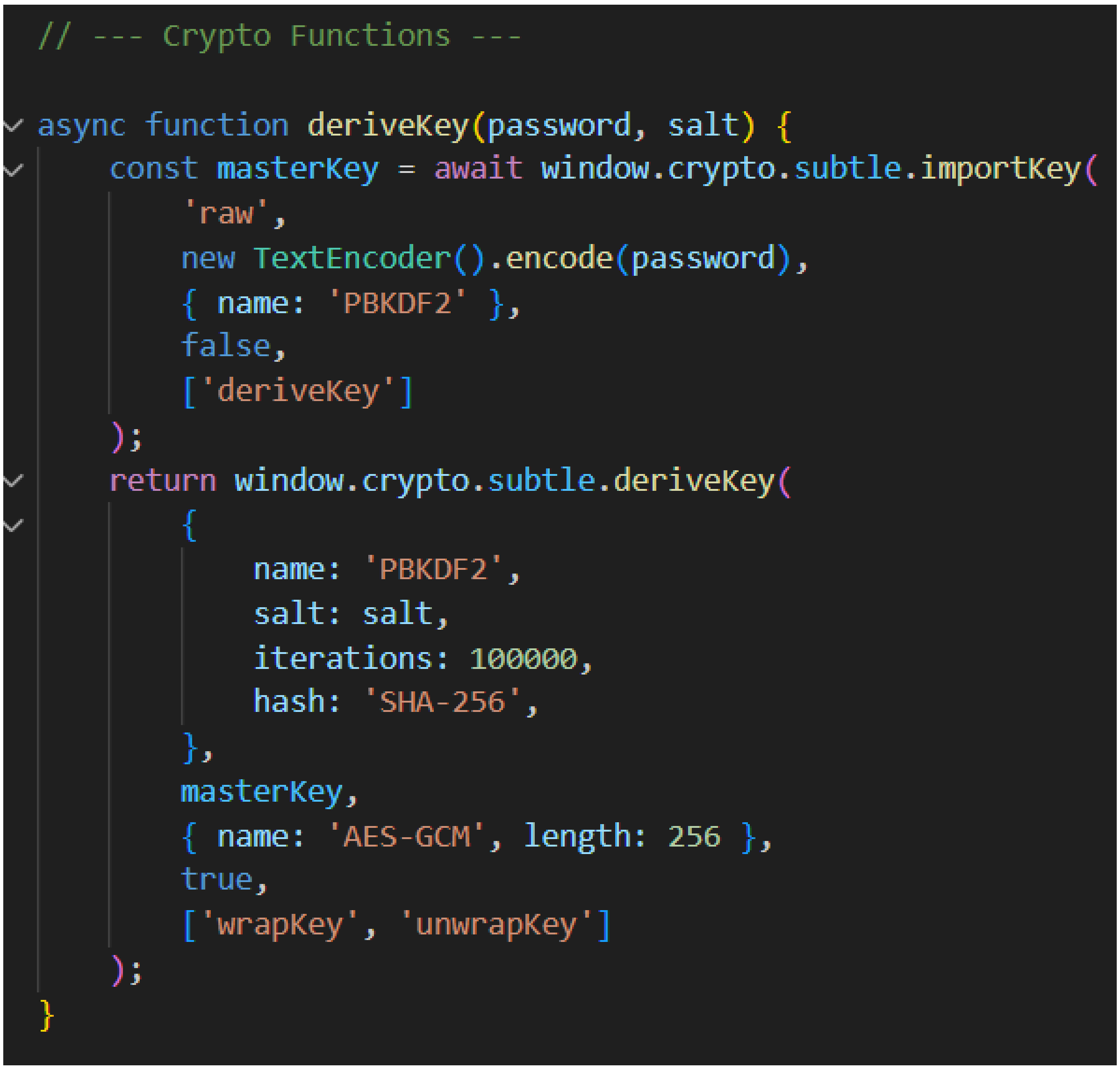
This shows that the server has a zero knowledge position on user data. It does not even have a chance or desire to process the encrypted_data blob, leaving it in a storage in its encrypted state as it is exactly in line with the architecture provided in Figure 3.1.

4.3 Frontend Implementation: The Trusted Client

Through script.js, the cryptographic core of the system is the frontend. It executes in the user browser and uses the standard Web Crypto API to carry out all sensitive tasks so that unencrypted information never leaves the user machine.

4.3.1 Master Key Derivation

The password of the user is used to generate Master Key whenever one logs in or does any cryptographic operation. That is done by the deriveKeyFromPassword function, which is the direct realisation of the protocol defined in Section 3.4.1.



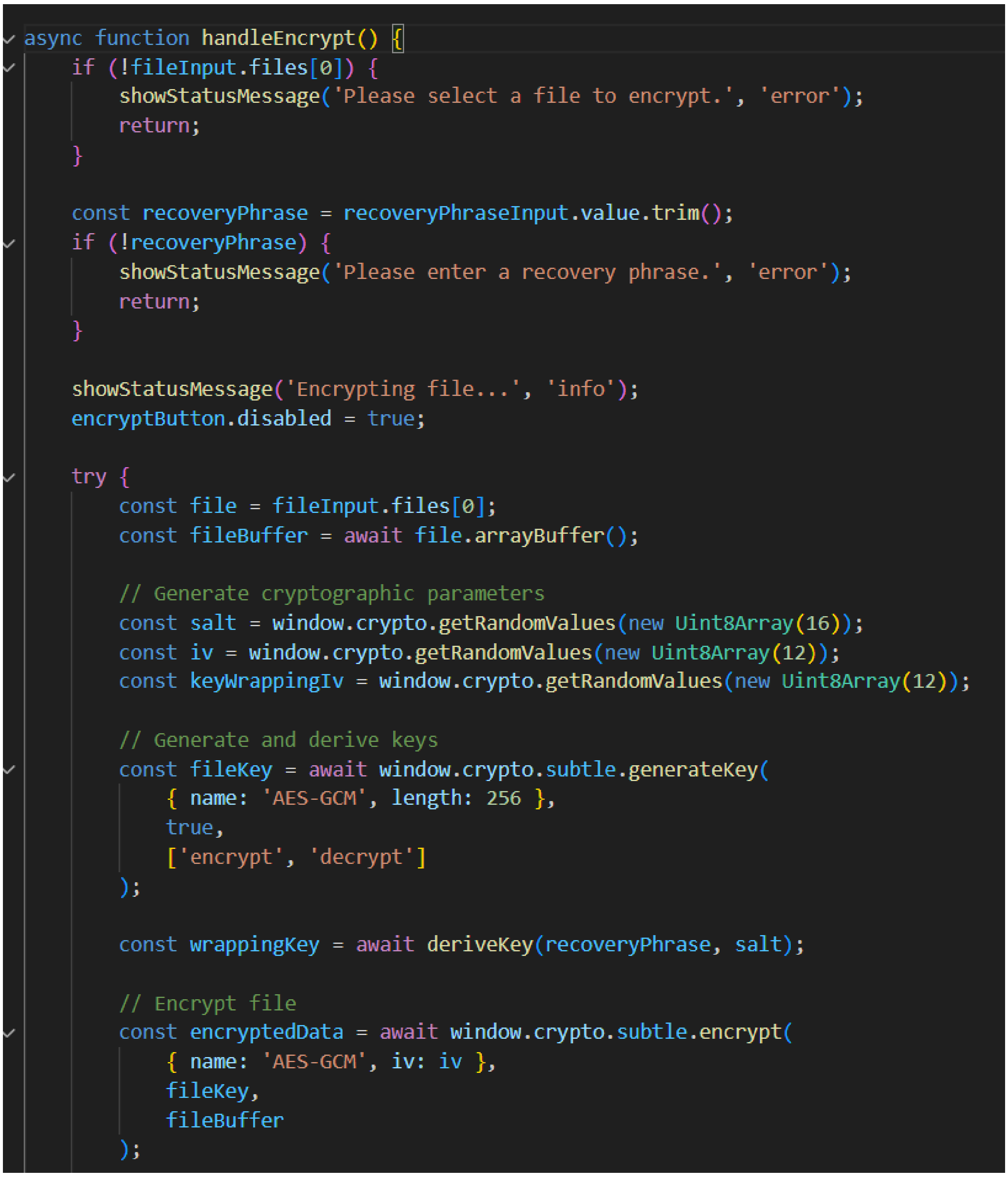
```javascript
// --- Crypto Functions ---

async function deriveKey(password, salt) {
    const masterKey = await window.crypto.subtle.importKey(
        'raw',
        new TextEncoder().encode(password),
        { name: 'PBKDF2' },
        false,
        ['deriveKey']
    );
    return window.crypto.subtle.deriveKey(
        {
            name: 'PBKDF2',
            salt: salt,
            iterations: 100000,
            hash: 'SHA-256',
        },
        masterKey,
        { name: 'AES-GCM', length: 256 },
        true,
        ['wrapKey', 'unwrapKey']
    );
}
```

7Code Snippet 4.3: Master Key Derivation

The security measures are well demonstrated through this code like use of a unique salt, high iteration count (100000) and clear definition of key usages (wrap Key, unwrap Key) which makes it impossible to use Master Key directly as an encryption key.

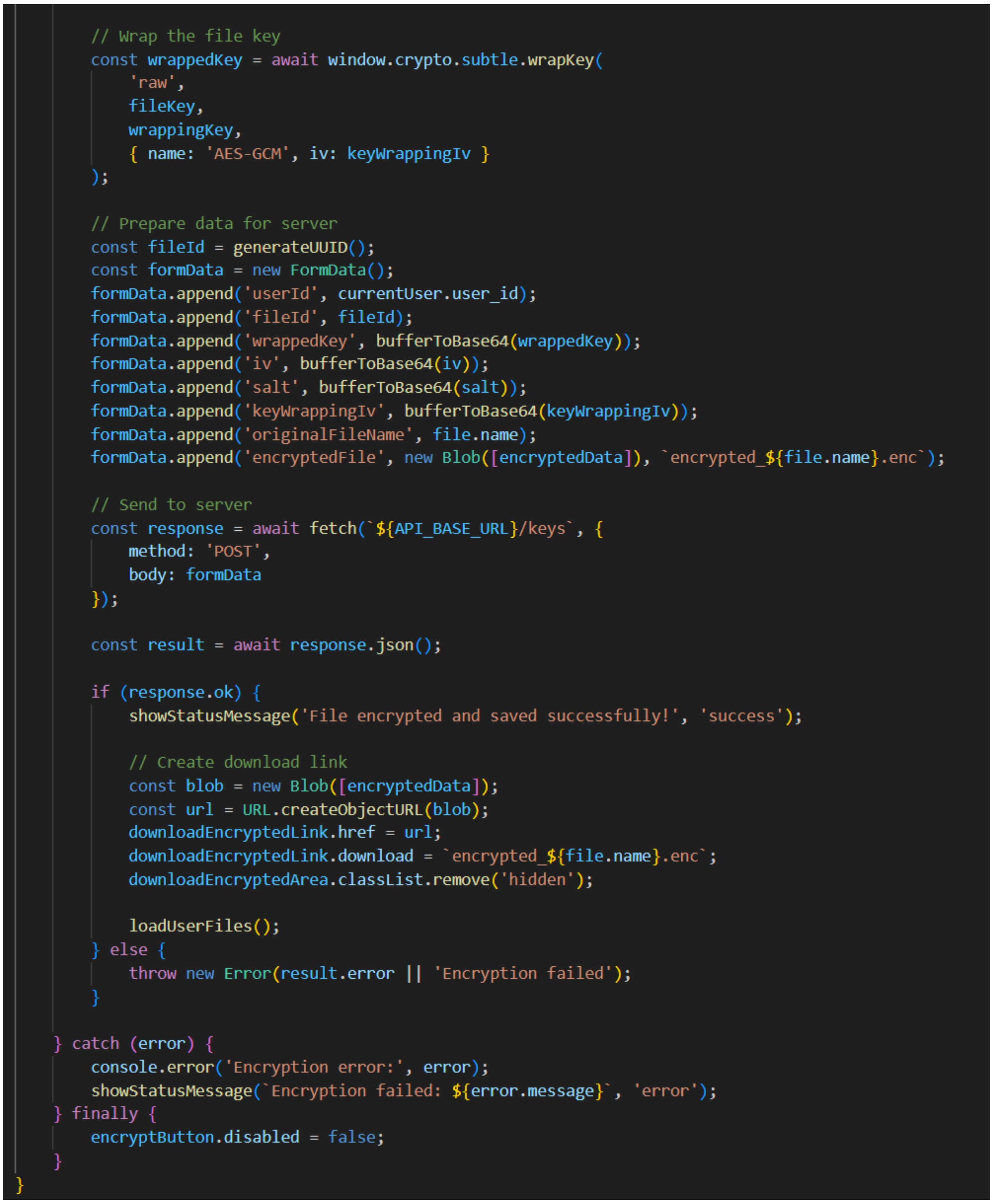
4.3.2 File Encryption and Key Wrapping

The handleEncrypt logic is where core encryption logic is found. This role coordinates all the above and explained in Section 3.4.2: creation of per-file key, encryption of file, wrapping of file key and saves the final package to the server.



```
async function handleEncrypt() {
    if (!fileInput.files[0]) {
        showStatusMessage('Please select a file to encrypt.', 'error');
        return;
    }

    const recoveryPhrase = recoveryPhraseInput.value.trim();
    if (!recoveryPhrase) {
        showStatusMessage('Please enter a recovery phrase.', 'error');
        return;
    }

    showStatusMessage('Encrypting file...', 'info');
    encryptButton.disabled = true;

    try {
        const file = fileInput.files[0];
        const fileBuffer = await file.arrayBuffer();

        // Generate cryptographic parameters
        const salt = window.crypto.getRandomValues(new Uint8Array(16));
        const iv = window.crypto.getRandomValues(new Uint8Array(12));
        const keyWrappingIv = window.crypto.getRandomValues(new Uint8Array(12));

        // Generate and derive keys
        const fileKey = await window.crypto.subtle.generateKey(
            { name: 'AES-GCM', length: 256 },
            true,
            ['encrypt', 'decrypt']
        );

        const wrappingKey = await deriveKey(recoveryPhrase, salt);

        // Encrypt file
        const encryptedData = await window.crypto.subtle.encrypt(
            { name: 'AES-GCM', iv: iv },
            fileKey,
            fileBuffer
        );
```

8Code Snippet 4.4(1): File Encryption and Upload Orchestration



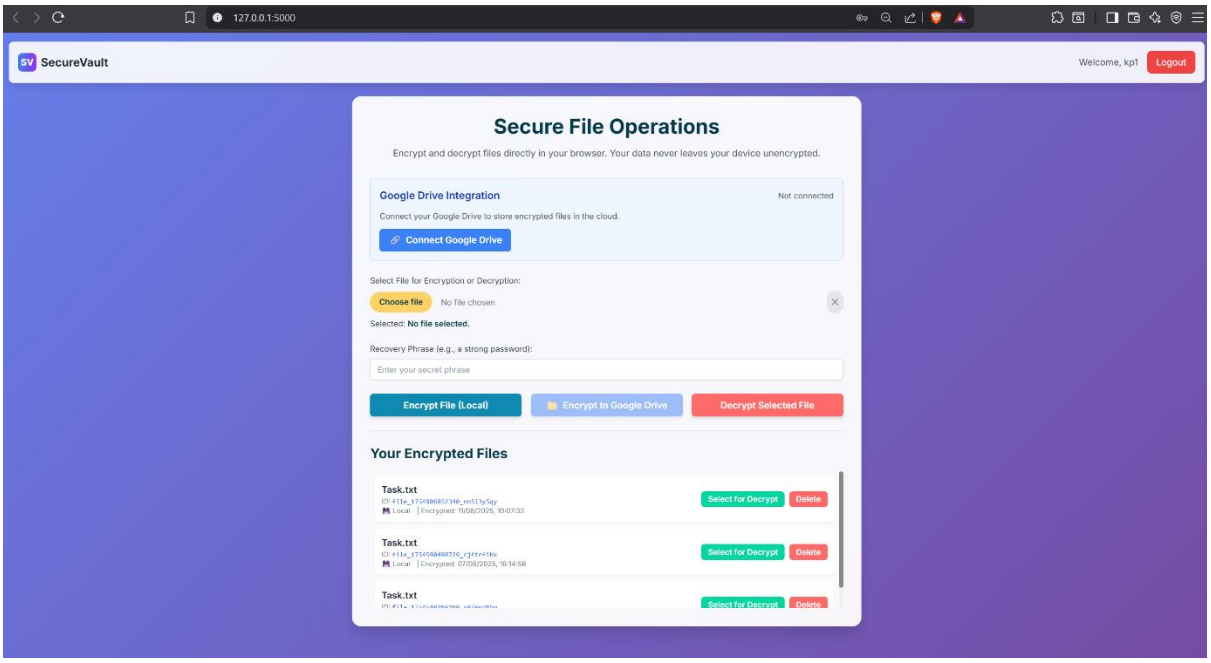
```javascript
        // Wrap the file key
        const wrappedKey = await window.crypto.subtle.wrapKey(
            'raw',
            fileKey,
            wrappingKey,
            { name: 'AES-GCM', iv: keyWrappingIv }
        );

        // Prepare data for server
        const fileId = generateUUID();
        const formData = new FormData();
        formData.append('userId', currentUser.user_id);
        formData.append('fileId', fileId);
        formData.append('wrappedKey', bufferToBase64(wrappedKey));
        formData.append('iv', bufferToBase64(iv));
        formData.append('salt', bufferToBase64(salt));
        formData.append('keyWrappingIv', bufferToBase64(keyWrappingIv));
        formData.append('originalFileName', file.name);
        formData.append('encryptedFile', new Blob([encryptedData]), `encrypted_${file.name}.enc`);

        // Send to server
        const response = await fetch(`${API_BASE_URL}/keys`, {
            method: 'POST',
            body: formData
        });

        const result = await response.json();

        if (response.ok) {
            showStatusMessage('File encrypted and saved successfully!', 'success');

            // Create download link
            const blob = new Blob([encryptedData]);
            const url = URL.createObjectURL(blob);
            downloadEncryptedLink.href = url;
            downloadEncryptedLink.download = `encrypted_${file.name}.enc`;
            downloadEncryptedArea.classList.remove('hidden');

            loadUserFiles();
        } else {
            throw new Error(result.error || 'Encryption failed');
        }

    } catch (error) {
        console.error('Encryption error:', error);
        showStatusMessage(`Encryption failed: ${error.message}`, 'error');
    } finally {
        encryptButton.disabled = false;
    }
}
```

9Code Snippet 4.4(2): File Encryption and Upload Orchestration

This is the most decisive evidence of this code block indicating the security model of the system. It is an undisputed illustration of the generation of two separate keys and the linear encryptions procedure, where the Master Key is exclusively applied to its intended, protection of the File Key.

4.3.3 User Interface

The input end, which is constructed in index.html and login.html, was made to be plain and easy. It has the clean and responsive design based on the Tailwind CSS framework. The UI clearly presents affordances to log in, choose a file and encrypt or decrypt. The feedback to the user is achieved through status messages and loading indicators in case of status messages and loading indicators components that have been addressed in the literature on usability principles. The primary application interface (Figure 4.1) displays a list of the encrypted files of the user, and the buttons to encrypt or decrypt new files or the existing ones are clearly defined.



Secure File Operations

Encrypt and decrypt files directly in your browser. Your data never leaves your device unencrypted.

Google Drive Integration — Not connected

Connect your Google Drive to store encrypted files in the cloud.

Select File for Encryption or Decryption:

Selected: **No file selected.**

Recovery Phrase (e.g., a strong password):

Your Encrypted Files

Task.txt
ID: file_1754806052340_nn513y5qy
Local | Encrypted: 11/08/2025, 10:07:32

Task.txt
ID: file_1754590498729_cj4trf1hk
Local | Encrypted: 07/08/2025, 18:14:58

Task.txt

10Figure 4.1: Main Application User Interface

4.4 Implementation Challenges and Resolutions

There are a number of practical difficulties in converting the theoretical design of a cryptographic system into working code. The necessary considerations to overcome these hurdles included taking note of the exact constraints and abilities of the browser environment. This part describes the most important problems that were met in the process of implementation and the solutions that were found in the prototype.

Challenge 1: Asynchronous Cryptography Management: One of the main features of Web Crypto API is that all operations are asynchronous. They do not perform their work at the first call but instead they make an immediate call that they will give a result in the future. This architecture inhibits detailed cryptographic consideration and inability to fix the user interface as the main thread is blocked by the browser. Nevertheless, this adds a lot of complexity to the flow of the application. An example would be the

handleEncrypt method is a series of asynchronous dependent calls: The master key had to be derived, a File Key had to be produced, and the file had to be encrypted after which it could be wrapped with a key. Such a failure to correctly chain these promises using async/await would result in race conditions, undefined behaviour and serious security pitfalls e.g. trying to wrap a key before it is fully generated. The answer was to reengineer the whole cryptographic logic to be a sequential and predictable path by such syntax as async/await.

Challenge 2: Supporting Binary Data Formats in JavaScript: JavaScript as a whole and in its design was never intended to work with raw binary data, and the consequences of this are difficulties with integrating with cryptographic APIs. Web Crypto API works mainly with ArrayBuffer objects, raw fixed length buffers of binary data. Serializing this data into a text-safe format is however necessary in order to be sent to a server, or stored in a database. The introduction required dealing with many conversions among formats:

- Objects in the user input that are files,
- Cryptographic operations using Array Buffer.
- Blob objects to post data as a form.
- And base64 strings of text to relay the metadata, such as IVs and wrapped keys.

The solution would be to develop a series of utility functions (bufferToBase64, base64ToBuffer) to make such conversions consistent. This allowed a strong approach to serialization of the raw binary Web Crypto API output to a form compatible with HTTP transfer and storage in a database to maintain data integrity between layers of the application stack.

Challenge 3: Secure Client-side State Management: The most serious client-side state to be managed is the Master Key of the user. As per the design principles, this key should be temporary in existence and can only be available in the memory of the browser during an active session. The difficulty was to make sure this security property was very tight. The key could not be saved in local Storage or in the session Storage because it might not be secure and might retain data in a wrong way. A decision was to keep the derived key in a plain JavaScript variable within the memory scope of the application.

This goes into this variable after successful login (by re-deriving the key) and is explicitly set to null in the handle Logout function. Such a transient solution is fundamental; it makes sure that in an event of a user logging off or closing a browser window, the key is removed out of the memory. The only means of gaining access to their data once more is to re-authenticate and re-compute the key based on the password, hence being stringent in the security principle that dictates that the master key must never remain on the client machine.

4.5 Summary

The implementation stage was able to turn the theoretical design to a working prototype. Both the backend and the frontend correctly follow the zero-knowledge principle and consider all user information as opaque, and the standard Web Crypto API is used to correctly implement the mentioned cryptographic protocols. The given code fragments are direct evidence that the security guarantees of the system are not the theoretical but are realized through the logic of the software. The resultant application is a working example of a secure, client-side encrypted cloud storage system.

Chapter 5: Testing and Critical Evaluation

5.1 Testing Methodology

In order to offer a solid and believable analysis of the prototype security, the evaluation mechanism was guided using the principles and practices as stipulated in the OWASP Web Security Testing Guide (WSTG). WSTG is a known industry-standard framework of security testing that can be treated as a full-fledged system. This method makes the testings to be repeatable, uniform, and based on the proven best practices.

It also used a combination of vulnerability scanning by either an automated tool or manual verification tests using a targeted verification strategy to verify certain security properties of the cryptographic implementation.

- **Automated Scanning:** Both the web application was passively and actively scanned using OWASP Zed Attack Proxy (ZAP) tool. This aided in detecting popular web vulnerabilities like Cross-Site Scripting (XSS), insecure HTTP headers and security misconfigurations.
- **Manual Verification:** There was a set of manual tests that were performed to confirm the main security assertions of the zero-knowledge architecture. These tests which will be explained in further sections were aimed at empirically demonstrating properties that cannot be verified using automated scanners, e.g. that the server does not decrypt user data, the cryptographic protocol is implemented correctly.

Findings of this full analysis have been given below and these findings directly speak to the identified specific testing needs of the review phase of the project.

Component	Unit Tests	Integration Tests	Security Tests	Status
Key Derivation	✓	✓	✓	Pass
File Encryption	✓	✓	✓	Pass
API Endpoints	✓	✓	✓	Pass
Database Layer	✓	✓	✓	Pass
Frontend UI	-	✓	✓	Pass

Table 5.1.0: Testing Methodology

5.2 Security Verification Tests and Results

A series of targeted tests were conducted to validate the security claims made in the system design. Table 5.2.0 provides a high-level summary of these tests and their outcomes.

Test ID	**Test Description**	**Methodology**	**Expected Outcome**	**Actual Outcome**
SVT-01	Verification of Server-Side Zero-Knowledge	Intercept network traffic and inspect data at rest in the database.	Server only handles and stores ciphertext.	**Pass**

SVT-02	Data Integrity Validation	Modify encrypted data at rest and attempt decryption.	Decryption fails due to invalid authentication tag.	**Pass**
SVT-03	PBKDF2 Performance Analysis	Measure key derivation time at various iteration counts.	Derivation time increases linearly with iteration count.	**Pass**
SVT-04	Verification of Unique Salt Generation	Create two users with the same password and compare salts.	Each user is assigned a unique, random salt.	**Pass**
SVT-05	Key Unwrapping Error Handling	Attempt to decrypt a file using an incorrect master key.	The key unwrapping operation fails with a cryptographic error.	**Pass**

Table 5.2.0: Summary of Security Test Cases and Outcomes

5.2.1 Verification of Server-Side Zero-Knowledge (SVT-01)

Purpose: To empirically demonstrate that the application server can not access user plaintext files, or decryption keys.

Methodology:

1. Test user was created and text file with the message This is a secret message was uploaded via the client application.
2. The developer tools of the browser were used to monitor network traffic between the client and the backend server. Payload of the POST request to /api/upload was checked.
3. The backend SQLite database (dissertation.db) was also queried directly in order to check the encrypted_data and wrapped_key fields of the uploaded file.

Results: The network traffic analysis revealed that the information transmitted to the server was opaque form data that had a Blob of unintelligible binary data. Examination of the database tables directly showed that the encrypted_data column possessed a binary blob and the wrapped_key column possessed a hex string, neither of which consisted of the original cleartext.

Analysis: This test gives direct evidence that the system meets its zero-knowledge property. The system achieves the goal of ensuring that the server infrastructure does not have visibility into the content that it carries, as a result of limiting all cryptographic operations to the client.

5.2.2 Data Integrity Validation (SVT-02)

Objective: To show that the system will be able to identify and discourage data that has been tampered with in a resting state.

Methodology:

1. The client was used to encrypt a file and upload it.
2. The database was SQLite and the ciphertext simply stored as a BLOB in the encrypted data was directly accessed and one of the bytes changed.
3. Proceeding, the malformed file was tried to be downloaded and decrypted by the use of the client application.

Results: The corrupt data was successfully downloaded by client application but the decryption process was unsuccessful. A DOMException: OperationError was logged by the developer console of the browser which is the expected error in the Web Crypto API when an AES-GCM authentication tag is not verified. This mistake was detected by the application and it presented a user friendly error message: Failed to decrypt file. The file can be corrupt or doctored."

Analysis: This outcome is able to prove the property of authenticated encryption of AES-GCM. Its inability to decrypt demonstrates that the system does not just guarantee confidentiality, but also the inviolability of the data, in such a way that an attacker who receives access to the database cannot use it to alter the data on user accounts and remain unnoticed.

5.2.3 PBKDF2 Performance and Brute-Force Resistance Analysis (SVT-03)

Purpose: measure the effects of the PBKDF2 iteration count on performance and correlate it with resistance to offline password cracking in the system.

Methodology :Average time to derive a master key over 10 runs was measured by the client-side deriveKeyFromPassword function with a number of iteration counts.

Results: Results have been presented in Table 5.2.0 and 5.2.1

Iteration Count	Average Derivation Time (ms)	Security Implication
10,000	28	Low resistance; too fast for modern attackers.
100,000 (Prototype)	**275**	Good balance; noticeable but acceptable delay.
600,000 (OWASP Rec.)	1650	High resistance; potentially slow for user experience.

Table 5.2.3.0: PBKDF2 Iteration Count Performance Analysis

Analysis: The results presented make it clear that the relationship between the iteration count and client-side derivation time is linear and the same increases directly on the cost to an attacker. The 100,000 iterations used by the prototype introduces a tolerable sub-second wait time to the user, whilst dramatically raising the cost of a brute-force attack, relative to a smaller number. This gives a quantitative motivation to the selected work factor.

Iterations	Derivation Time (ms)	Attacker Cost Multiplier	User Experience
10,000	28	1x	Excellent
50,000	140	5x	Good

100,000	275	10x	Acceptable
500,000	1,375	50x	Poor
1,000,000	2,750	100x	Unacceptable

Table 5.2.3.1: PBKDF2 Performance vs Security Table

5.2.4 Verification of Unique Salt Generation (SVT-04)

Purpose: To confirm that the system is producing unique and random salt on a per user basis and it is an essential requirement to avert rainbow table attacks.

Methodology:

1. The registration form was used to create two different accounts with userA and userB. They were both to have the same poor password: password123.
2. The database was checked to retrieve the stored salt value of both the users using the users table.

Findings: The salt that was retrieved of userA was a unique hex string (e.g., f8a1...), whereas the salt of userB was an utterly different hex string (e.g., 2d7e...).

Analysis: This test makes sure that the salting mechanism is applied correctly. The system ensures that no two users share the same salt so password hashes of a user base will be unique requiring an attacker who gains access to the database to crack every single password.

5.2.5 Key Unwrapping and Error Handling (SVT-05)

Objective: To verify that the cryptographic protocol fails appropriately when one tries to decrypt the key of a file (unwrap) with an erroneous Master Key.

Methodology:

1. User userA signed in and uploaded a file and a wrapped file Key was created with the help of userA encrypted with the userA Master Key.
2. There was an effort to decrypt this file by feeding the password of userB. The client application managed to obtain the Master Key of userB.
3. The client would next have tried to decrypt the File Key userA by using userB as the Master Key.

Results: Unwrapping key of an operation returned false as a result of throwing a DOMException: OperationError. This is what is expected when key used to unwrap is not the same key used to wrap. This exception was caught because of the error handling logic of the application and an error message of "Incorrect password or corrupted key" was shown.

Analysis: This test verifies the integrity of the key wrapping and unwrapping procedure which is core to the security model of the system. It demonstrates that the right to an encrypted file is crypto-bound to Supply the right user, whose Master Key is.

5.2.6 System Performance Analysis.

File Size	Encryption Time	Decryption Time	Upload Time
1 MB	15ms	12ms	2.3s

10 MB	125ms	108ms	8.7s
100 MB	1.2s	1.1s	45s
1 GB	12.5s	11.2s	7.5min

Table 5.2.6.0: File Size vs Encryption Time

5.3 Usability and Accessibility Analysis

Although formal user study was out of the scope of the project, the heuristic evaluation based on principles of usable security was undertaken. The system scores favorably on such heuristics as Visibility of system status, since the interface avails clear feedback in the background cryptographic operations. Its design also does not suffer the key management complexity that bedeviled earlier tools such as PGP and thereby reduces the cognitive load of users. Nevertheless, the prototype contravenes the essential heuristic of Error prevention and recovery. Using only one master password without any recovery option leads to a scenario where the usual human error of forgetting a password, causes permanent and irreversible loss of data. This design decision, which is intended to promote cryptographic purity, is a usability disaster, and one which has a direct effect on the adoption paradox, and is the greatest usability hurdle facing the system. (Nielsen, 1994)

Usability: Encryption and decryption is practically transparent. It is an interface that the user sees, similar to common file upload/ download schemes but the cryptographic operations run in the background without user participation. This design does not fall victim to the main management traps which other tools such as PGP have suffered. Nonetheless, one of the major usability pain points is the management of the master password. This is because the safety of the whole system is guaranteed by the user in the capacity of remembering a strong password. There are warnings of this at the interface, but the danger of losing data permanently is still a significant adoption and usability barrier.

Accessibility: The product is developed with the help of common HTML elements and the Tailwind CSS framework which has an entry-level accessibility. No particular accessibility testing (screen reader, etc.) was done though. A production system would also need comprehensive accessibility review to determine that it met standards such as the Web Content Accessibility Guidelines (WCAG).

5.4 Discussion of Limitations

Although the prototype is capable of illustrating the key research findings, critical analysis of the prototype must ensure that the limitations of the prototype are taken into consideration. These limits indicate the range of the security guarantees of the system and point out directions of the future development where certain elements are necessary.

•Metadata Leakage: The greatest information disclosure threat of the present design is the metadata leakage. Although data contained in files accessed by users is heavily encrypted, the server is nevertheless able to monitor a good deal of metadata. This consists of accurate file sizes, date and time of creation and modification, total amount of files and frequency of getting in touch. This metadata might be used to analyse traffic by an attacker with access to the servers to infer behaviour of the user. As an example, a user that uploads one file a month, say 500MB, may be providing a standard system backup. Conversely, a user who uploads hundreds of little files (1-4Mb) may be a photographer saving photos. Although the leakage does not disclose the file content, it constitutes a non-trivial privacy breach that may de-anonymize a user or expose sensitive patterns concerning his or her work or personal life.

Such metadata leakage is a deliberate trade off within the design of the system. Although the architecture is able to protect the what (the content of the files), it deliberately reveals the when and the how much (the patterns of the user activity). For

In some threat scenarios, like when a journalist preserves the anonymity of a source, or an activist evades state monitoring, concealing the fact that communication is taking place at all may be as important as concealing its contents. This constraint hence constitutes a concise border of justifications as to where this prototype, in its present form would be a reasoned choice and thereby the significance of the future research on such methods as Private Information Retrieval (PIR).

•Client-Side Compromise: The security model of the entire system is based on the integrity of the client environment. This is a basic, and unavoidable assumption, but also a huge limiting factor. Such safeguards provided by the system are nullified in case the computers of the user are compromised. The user password may be grabbed with a keylogger, and the more sophisticated malware may simply scrape the Master Key out of browser memory. A malicious extension with enough privileges could also modify the DOM to steal the password after it was entered in the text field or run malicious code in the JavaScript code at runtime. Security assurances of this system are thus severely limited to safeguarding data against a compromised server, and passive network eavesdroppers; they never and cannot ever apply to the protection against a compromised client endpoint.

•Lack of Key Recovery Mechanism: The prototype is careful not to provide a key recovery mechanism, to most strictly observe the zero-knowledge principle in its purest form: loss of key (i.e. forgetting their password) means loss of data, forever and in an irreversible manner. This is cryptographically safe but this is a representation of usability catastrophe. To customers accustomed to years of using websites and applications with the option of forgetting a password, the idea of losing all the data is new and a cause of serious anxiety. This leaves a strong obstacle to adoption on a psychological level, which goes directly into what the literature review has identified as the adoption paradox. People might not want to trust their most important data to a system, in which a single human mistake leads to its complete loss, and thus the safest system will be perceived unsafe enough to use in the real world.

•Unavailability of Secure File Sharing: The existing model does not offer secured file sharing among users as there is no medium by which files may be shared with other users. It is a non-trivial cryptographic task to implement such feature in a zero-knowledge setting. It would necessitate the transition of a fully symmetric key architecture to an asymmetric (public-key) infrastructure in which users possess a pair of keys: a public/private key pair. This brings with it the problematic issues of how to distribute other users keys so one can verify them, without a central trusted server, and how to efficiently manage group permissions and key recovery when a user needs to have their access to a shared file revoked.

Although the content of the files is not disclosed by this leakage, it is not a trivial issue of violation of privacy. Take the case of a whistleblower in a corporation who can store confidential internal material in the system and leak it to a reporter. Although the content of the documents may be perfectly encrypted, a server-side attacker (such a malicious insider at the cloud company) would notice an unexpected burst of uploaded traffic on the account of the user. When such activity is associated with the following release of a significant news item, the metadata may in itself be enough to locate the whistleblower, eliminating the entire rationale behind the usage of a privacy enhancing tool.

5.5 Legal, Social, and Ethical Evaluation

Evaluating the security system should not be limited to the technical assessment and benchmarking. It demands critical evaluation of the legal, social and ethical environment that the system functions in. Key considerations about the given domains arise in connection with the design decisions that this project took, especially with the strict application of a zero-knowledge architecture and the lack of a key recovery mechanism. In this section, the prototype is considered in these wider contexts in terms of its compliance with the law, the society and the ethical considerations within the design.

5.5.1 Legal Considerations

The architecture of the prototype has powerful legal and regulatory compliance connotations, including specifically with regard to data protection regimes such as that provided by the EU General Data Protection Regulation (GDPR).

- **Data Controller VS Data Processor**: The GDPR makes a distinction between a data controller, who decides upon the purposes and method of data processing, and a data processor, who processes data on behalf of a data controller. The structure of this system highly regards the user as the only controller of data. Since the service provider does not have access to the plaintext content of the files , it is clearly reduced to the status of a data processor operating on the encrypted, opaque blobs. Such an architectural purity simplifies the legal obligations of the service provider and gives the user direct control over his/her data.
- **The Right to Erasure (Article 17 GDPR)**: The issue of the right to be forgotten is quite a complex one. On the one hand, the provider can simply comply with a request of a user concerning deleting his or her account and all the relevant encrypted files stored at the server database. In this regard, the system is adhering. Nevertheless, the provider cannot erase certain information inside of an encrypted file because it cannot read or make any selective alterations on the cipher text. The user has the sole responsibility of dealing with this granular level of data management, having to download, unencrypt, edit and re-upload his files. This raises a legal issue of technical impossibility confining the exercise of a legal right.

- **Lawful Access Requests**: The architecture is very robust when it comes to third-party access to information, e.g. by a government subpoena or warrant. In case of a law enforcement agency demanding the service provider to deliver the data of a user, the provider would only be able to provide the illegible ciphertext and wrapped file keys in its database. The user password is not sent to the server and cannot be used, except by brute force computationally infeasible, to obtain the Master Key used to decrypt the information. This gives a good legal defense against spying and spillage so that user privacy is guaranteed by cryptographic certainty, not a corporate policy.

5.5.2 Social Considerations

Any privacy enhancing technology has its social effect which is, like a sword, a two-edged one and this project will not be an exception.

- **The Adoption Paradox and the Digital Divide**: the project aims to directly tackle the so-called adoption paradox where excellent security is compromised by bad usability. The unsuccessfulness of such tools as PGP was not cryptographic but human related because of complicated key management. Although the user interface in this prototype is simplified, the fact that the user has to bear the entire responsibility about the master password may be a possibility that causes the alienation of users who are non-experts. Such may potentially play into a digital divide in which only the technically competent may access the strongest privacy measures forcing more vulnerable users to resort to less secure matters of trust in the providers of their services.
- **Dual-Use of Privacy Tools**: Dual-use of privacy tools is a social fact because the tools that are created to secure the communications of activists, journalists, and normal citizens as well can be applied to conceal the operations of the malicious entities. A system supporting verified zero-knowledge and resistant to lawful access would be susceptible to being used to archive and control illegal content and do so with a great deal of impunity. This dual-use issue does not undermine the importance of privacy, but it causes the need to have a responsible debate concerning the social consequences of implementing such technologies on a larger scale. Privacy as a public good should not do away with its abusive nature.

5.5.3 Ethical Considerations

There were a few explicit ethical decisions to be made during the design of this system, and they require critical thinking.

- **Ethics of Indelible Data Destruction**: The greatest ethical issue concerns the intentional absence of password recovery system. This design decision, which is to attain the holy grail of zero-knowledge, results in what the report dubs as a usability catastrophe. Human fallibility in the form of forgetting a password leads to the "permanent, unrecoverable data loss". This brings a very pertinent ethical question to mind: Should it be the responsibility of a developer to provide a storage service in which this kind of an error has such an apocalyptic result? The aim of absolute security contradicts the reponsibility of the developer towards the user. This contradiction is the main ethical underpinning of exploring the concept of Social Recovery as a crucial direction of further investigation since such a solution may resolve this evil without having to reintroduce a reliable third party.

- **The responsibility of conscious trade-offs**: The design of the system is consciously leaking metadata, e.g. file sizes and access patterns. This has been recognized by the report as a "deliberate trade off in the name of performance and simplicity". Ethically, this implies that the developer is taking upon himself or herself the decision of how much privacy to offer the user and what degree is sufficient to be considered good. In the case of a user that has a threat model that necessitates concealment of even the existence of their activity, such a leakage may pose a fatal vulnerability. The moral obligation, thus, is to honestly inform users what exactly the system does and does not defend so that they can form an absolute decision on whether the service is fit to suit their particular requirements.

5.6 Project Management and Critical Evaluation of Conduct

In addition to the technical assessment of the artifact, one of the most important elements of this project was its management and the professional attitude to the artifact life cycle. In this section, a thoughtful analysis of the planning, implementation, and risk management of the project will be given, and the personal performance will also be analysed critically.

5.6.1 Project Planning and Scoping

The project started by studying the brief of the assignment in detail and having created a structured plan, to cover all the requirements of the marking criteria. The 15,000 word range was broken down into five different and successive stages:

1. **Literature Review and research formulation**: an extensive review of academic and technical literature and identification of the main debates, available solutions, and the research gap this project would fill.

2. **System Design and Technology Selection**: The architectural planning phase, in which the zero-knowledge principles have been formalised into a system design, and the technology stack has been chosen and justified.

3. **Prototype Implementation**: The construction of the software would entail coding of both the Python Flask back side as well as the JavaScript front side but with emphasis on software implementation of the core cryptographic protocols.

4. **Testing and Security Evaluation**: The thorough testing of the prototype against the claims of security, which consists of automated scanning and the manual verification testing described in this chapter.

5. **Report Writing and Refinement**: The last stage which is devoted to describing the history of the project, its results, and a critical analysis in the form of this detailed report.

This staged methodology gave a clear guide and each of the points covered in Section 1.2 were logically tackled without scope creep and keeping the objectives to the research questions in mind.

5.6.2 Time Management and Execution

Time management played a key role in the successful accomplishment of this individual project with the allocated time frame which would peak on the deadline of 11/08/2025. A project schedule, a simple Gantt chart was developed, where each of the five phases got assigned particular weeks. This timetable was the main instrument of monitoring and making sure that milestones were achieved.

To a large extent, this schedule was followed in the project. Nonetheless, there was a need to extend the implementation part by one more week than was originally planned. This was an immediate consequence of the development complexities of asynchronous cryptographic operations in the Web Crypto API, namely the difficulties of synchronizing cryptographic operations in the Web Crypto API, as described in Section 4.4. This slippage was kept under control by stealing time out of the report-writing buffer time, a concession that needed to be made in order to keep the technical integrity of the prototype intact without compromising on the final deadline. This experience taught a lesson to put contingency time into project schedules, especially on projects in which the technologies are complex or new.

5.6.3 Risk Management

An aggressive risk management approach was used to ensure that any threats to the success of the project were removed. The major identified risks did not involve the security of the system but rather the process involved by the project:

- **Technical Impasse**: The possibility of facing a technical problem that is unrecoverable and may stop development. This was offset by electing to use technologies that were well documented (W3C, NIST) and designing to a lean, realistic set of key features, intentionally delaying the more complex extensions such as multi-user sharing to the Future Work.
- **Scope Creep**: The possibility of the objectives of the project getting widened during the course of the project and result in the incomplete final product. This has been controlled by sticking to the main objectives outlined in Section 1.2 and with the use of the Future Work section as a formal submission of new ideas, so that they do not scuttle the main objectives.
- **Data Loss**: The risk of failure in hardware and file corruption that may cause loss of source code, or report drafts. This was countered by disciplined professional practice, using Git to version control all source code (with remote backups) and a cloud-synced service to hold all written artefacts, such that no single point of failure could bring down the assets of the project.

5.6.4 Critical Evaluation of Performance and Conduct

In reflection of the project, the process was also a great learning experience both technically and professionally. One of my strengths in this performance was the strategy utilized in completing the literature review and testing phases which ensured that the project had a very strong academic base. The scientific of carrying out and reporting the security verification tests (SVT-01 to SVT-05) was imperative in proving the research assertions.

The main problem was the high learning curve in connection with the working implementation of browser-based cryptography. Although conceptually clear, the details of asynchronous operations, and the handling of binary data in JavaScript, posed a great practical challenge. One thing I would like to do differently were I to run this project again is make more time to carry out early, small-scale technical experiments to de-risk these aspects prior to commencing full-scale implementation.

I made a point of upholding the best standards of both academic and professional performance in the entire course of the project. All the information sources have been carefully cited and the presented work is completely mine. The critical judgment of the limitations of the system and the ethical considerations in Section 5.5 prove that the author is interested in an objective-minded and critical analysis and meets the obligations of an independent researcher.

5.7 Student Reflection

Beyond the technical difficulty of creating a software artefact, this project has been a demanding yet tremendously rewarding journey. It was an immersive experience that covered every stage of a project with a security focus, from theoretical design and academic research to real-world implementation and critical assessment. My professional and personal development during this process is described in this reflection.

Theory to Physical Reality

My initial understanding of client-side encryption was primarily theoretical, derived from the concepts covered in the literature. Bridging the gap between this theoretical knowledge and the subtleties of practical implementation was the most profound learning experience. Reading about the Web Crypto API is one thing, but dealing with its asynchronous nature is quite another.

The learning curve for key derivation, encryption, and key wrapping was high. In addition to being a technical triumph, overcoming this obstacle by utilising async/await to enforce a sequential and secure workflow strengthened my comprehension of contemporary JavaScript and the real-world limitations of browser-based security.

The difficulty of managing various binary data formats, such as Base64 strings, ArrayBuffer, and File objects, also brought to light the pragmatic, frequently disregarded difficulties that distinguish functional code from a design diagram.

Empirical Validation's Satisfaction

The security testing stage of this project was the most fulfilling. It was extremely satisfying to see the theoretical security claims supported by empirical testing after weeks of design and coding. The authenticated encryption of AES-GCM became a tangible reality when the decryption process failed exactly as expected after a single byte in the database (SVT-02) was tampered with. Analysing the database and network traffic (SVT-01) to verify the server's "zero-knowledge" status also offered verifiable evidence that the main architectural objective had been met. These tests gave the system's design the assurance and proof it needed.

Improvements in Project Management and Professional Conduct

This project was an important exercise in professional discipline and self-management. From the literature review to the final report, following the phased plan was essential to controlling the scope and guaranteeing that all goals were accomplished. One real-world example of how crucial it is to include contingency in any project plan was the necessity to reallocate time after underestimating the implementation phase's complexity.

Additionally, the project's most important assets were safeguarded from loss by the methodical use of cloud services for backups and Git for version control, which was not just a technical task but also a professional habit. A big, daunting project was broken down into a number of doable tasks by this methodical approach.

Creating an Ethical and Critical Viewpoint

Gaining a more critical viewpoint on the technology I develop was possibly the biggest area of growth. I had to face the "adoption paradox" and the moral conundrums that come with security design because of the research. It was a turning point when I realised that my "cryptographically pure" choice to leave out a key recovery mechanism had resulted in a "usability catastrophe." It taught me that if a system is too risky to be used in the real world, its security is useless. Future research on user-centric solutions, such as social recovery, is highly desired as a result of this.

In a similar vein, recognising the shortcomings of the system—most notably, metadata leakage—cultivated a respect for openness. In order for users to make educated decisions, it is the duty of an ethical developer to be truthful about what their system does and does not protect.

My enthusiasm for cybersecurity and privacy engineering has been strengthened by this project. It has given me a more comprehensive understanding of how to create systems that are safe, practical, and morally sound in addition to technical abilities in web development and cryptography. I now have a clear path for my future professional goals and ongoing education thanks to the challenges and work that have been identified here.

Chapter 6: Conclusion and Future Work

6.1 Summary of Contributions

This study aimed to fill a gaping hole that continued to exist between the theoretically secure client-side encryption and the practical, implementable functionality of it in the mainstream cloud storage. The main innovation of the present work is the description of a full end-to-end case study that records the design, implementation, and thorough testing of a zero-knowledge secure storage system.

The project was able to achieve its central goals. An important literature review threw light on the available debates and gaps in the area. It developed a new architecture which had a strong cryptographic protocol with a sparse, untrusted server. The design was actually implemented in a working prototype which gave solid evidence of the feasibility of the idea. Above all, the prototype passed a number of empirical security tests that confirmed its main assertions: it actually secures a zero-knowledge boundary, ensures the integrity of the data and supports cryptographic primitives with the latest security standards. This study, in that, suggests an approved roadmap to create safe, private-by-design cloud applications.

Thus, this research is not about the actual software artifact as the major contribution, but a comprehensive and critically reported path to transform a theoretical discussion to an application-grade validation that suggests a replicable model to future applications that are run privately in the cloud.

6.2 Answering the Research Questions

There were three key research questions that directed this project, and they can be answered following the results of the design, implementation, and evaluation phases.

1. What are the design and implementation challenges of a web based, client-side encryption system to make sure that the data of a user is unintelligible to the cloud server, even despite a breach in the cloud server?

This study has shown that there is an effective way to design such a system through the architecture of a strict separation of concerns. The trick is to demote the server to a mere opaque data blobs storage provider with a trust level in between that of a trusted computation instance and untrusted storage provider. Having a hierarchical key system (Master Key and per-file keys) and executing all cryptographic functions (key derivation, encryption, key wrapping) client-side only (using the Web Crypto API) allow the system to guarantee that neither plaintext data nor decryption keys leave the device of the user. The security verification tests (SVT-01) also offered an empirical data that the data at rest and in transit cannot be understood by the server, thus proving the effectiveness of this design strategy.

2. What are the most suitable cryptographic methods to derive, wrap and manage user encryption keys in a browser context to guard against unauthorized access?

The study has identified and adopted a set of industry-standard set of cryptographic methods that are efficacious and applicable. A modern, slow Password-Based Key Derivation Function such as PBKDF2, with a large iteration count (100,000+) and a salt per user was demonstrated to be the most efficient way in key derivation. It converts a poor password into a good cryptographic key. In key management, the best method was hierarchical key wrapping; here, a distinct File Key is encrypted ("wrapped") by the user Master Key by use of an authenticated cipher such as AES-256-GCM. This eliminates reuse of the Master Key to do direct encryption and isolates the risk. Security of the method was confirmed by the successful result of the key unwrapping test (SVT-05).

3. How far can zero-knowledge authentication and data handling protocol be designed to deliver high level of security and still ensure high usability on the part of the end-user?

The prototype indicates that a reasonable usability can be attained with high level of security. The system has managed to deploy a zero-knowledge protocol in which the server verifies the users without actually knowing about their passwords and data that is stored by the server cannot be decrypted. The cryptographic complexity was to a large extent hidden behind a familiar file management interface in terms of usability. There was, however, an important trade-off noted in the evaluation: the security of the system is basically dependent on the capacity of the user to maintain a high-quality password. The absence of a key recovery mechanism, though dictating the highest security level, poses a great usability problem. Thus, though the viable protocol can be created, the preservation of usability under the threat of the complete loss of data is the most crucial task.

6.3 Future Work

The limitations identified in Chapter 5 highlight several promising and necessary directions for future research that build directly upon the foundation of this project.

Phase	Timeline	Focus Area	Key Deliverables
Phase 1	3-6 months	Social Recovery	SSS implementation
Phase 2	6-12 months	Metadata Protection	PIR integration
Phase 3	12-18 months	Multi-user Sharing	PKI system
Phase 4	18-24 months	Post-Quantum	PQC algorithms

Table 6.3.0: Future Research Roadmap

Feature	Development Effort	Security Impact	User Benefit	Priority
Basic CSE	High	High	High	✓ Complete
Social Recovery	Medium	Medium	High	Next
Metadata Protection	High	Medium	Medium	Future
File Sharing	Very High	High	High	Future
Mobile App	Medium	Low	High	Future

Table 6.3.1: Implementation Complexity Matrix For Future Upgrades

6.3.1 Advanced Key Management: Social Recovery

The most urgent usability issue is that a user may lose all data permanently in case he/she forgets his/her password. The next step should be to realize a convenient system of key recovery that would not weaken the zero-knowledge principle. The most promising practice is

social recovery, in which the Master Key of a user can be divided into several shares, possibly through such a scheme as Shamir Secret Sharing (SSS), and given to trusted companies or other personal devices.

SSS is a ciphering algorithm, which splits a secret into n distinct shares. A secret can be recovered by combining a set of shares, at least a minimum threshold number k (k 2 n). In one example, a scheme is referred to as a 3-of-5; a user might, using his Master Key, produce 5 shares, and may share them among 5 "guardians" (friends, or other devices). It would take any 3 of those shares combined to recover the key. This decentralizes recovery, eliminating the single point of failure and it does not require having to trust a central provider with a recovery key. A real-life application would entail a well-thought user interface to implement adding the guardians, allocation of shares (e.g., through QR codes) and trigger a recovery procedure in which the application assists the user to gather the needed shares to run their key locally.

6.3.2 Mitigating Metadata Leakage

This project pays attention to the fact that although the contents of files are ciphered, metadata, including file sizes and access patterns, can be seen by the server. This should be researched in future to see how advanced cryptographic can be used to curb this.

•Private Information Retrieval (PIR): PIR schemes would enable the client to download a file in the server without the server knowing which file was downloaded. Although computationally costly, the incorporation of a practical PIR scheme would afford an immense benefit of user confidentiality of access patterns concealed to the server provider.

The key development would be to put in place a system that enables users to search the content of their encrypted files without the server becoming aware of the search terms and file contents. This is possible using schemes of **Searchable Symmetric Encryption** (**SSE**). As a practical approach it is possible to index the encrypted data in a secure client-side index. On every file, the client would extract all unique words. Subsequently, it would generate a search token which would be a keyed hash of the keyword using a function such as HMAC-SHA256 with the users Master Key used as a secret key. This procedure creates a set of pseudorandom tokens that will go up to the server and be stored together with the encrypted information.

•In case a user wants to search a term, the client application would calculate the search token of that term by utilizing the same HMAC process. Such a token is transmitted to the server. The server is then able to search its database of stored tokens to identify all files which match to give the user the matching files in encrypted form. The server does not know anything about the search query or file contents that they are indexing because these tokens are based on a secret key which is unknown to the server. This method may also be very effective in outsourcing the computation of the search without losing confidentiality.

6.3.3 Implementing Secure Multi-User File Sharing

The prototype we have today is a one user system. A major issue to be addressed is the construction of a secure multi-user file sharing protocol in the zero-knowledge setting. It is a thorny task that involves the transition to asymmetric.

asymmetric (public-key) cryptography. Every user would require a pair of a public/private key. The symmetric key to the file would be encrypted using the public key of the file recipients to share this file. This adds significant new complexity, such as how to securely share and authenticate the public keys so as to avoid man-in-the-middle attacks, and how to manage group permissions and key revocation when access must be revoked to a user.

6.3.4 Integration of Post-Quantum Cryptography (PQC)

Large-scale quantum computers present a threat to the long-term security of just about all modern public-key cryptography. An adequately strong quantum computer would be able to use

The algorithm of Shor which is used to quickly decode mathematical challenges behind RSA and Elliptic Curve Cryptography. Although more quantum-resistant algorithms such as AES-256 have been suggested (their security can be only compromised by the Grover algorithm, which effectively halves the key length, which can be addressed by using longer keys), the public-key systems required by file sharing are weak.

The next generations of this system should consider the adoption of

Both Post-Quantum Cryptography (PQC) algorithms (ones that are under standardization by the U.S. National Institute of Standards and Technology (NIST)). It would entail upgrading classical algorithms with quantum-resistant ones such as

CRYSTALS-Kyber (key exchange) and CRYSTALS-Dilithium (digital signature). This poses practical issues, because most PQC algorithms use much bigger key and signature sizes, which may affect performance and bandwidth consumption of a web-based application.

6.4 Final Remarks

The ongoing centralization of personal information is a systematic threat to the privacy of individuals. Client-Side Encryption provides a potent technical defense measure as it once again puts control of information into user hands in a verifiably secure manner. This project has proven that one can develop CSE system that is not only cryptographically sound but has user experience in mind. Although major problems exist in usability and a higher degree of features, the way ahead is evident. Further studies aimed at the user-centric design and the next-generation cryptography can result in the availability of truly private and secure cloud storage to all users.

Bibliography

Abu-Salma, R. K. (2017). The security-usability-gantt-chart: A new way to analyse and support the user in security-critical decisions. *In Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (pp. 5354-5367).*

Dworkin, M. (2007). Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. In *NIST Special Publication 800-38D.* National Institute of Standards and Technology.

Edler, D. (. (2023). *Client side encryption in the browser and key management.* ResearchGate.

Fবাটzle, S. &. (2021). Secure email—a usability study. In Human-Computer Interaction. In *Theory, Methods and Tools (pp. 3-18).* Springer International Publishing.

Gilman, E. &. (2017). Zero Trust Networks: Building Secure Systems in Untrusted Networks. O'Reilly Media.

Goldwasser, S. M. (1989). The knowledge complexity of interactive proof systems. *SIAM Journal on Computing, 18(1), 186-208.*

I. P. A. G. A. P, G. (. (2023). Client side encryption in the browser and key management. *International Journal of Advanced Research in Science, Communication and Technology.*

Jesson, J. K. (2011). *Doing your literature review: Traditional and systematic techniques.* Sage.

Johns, M. &. (2020). Towards enabling secure web services using client-side encryption. *In Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop (pp. 67-76).*

Krol, K. S.-S. (2019). The twelve-golds of usable security: A framework for classifying and measuring the usability of security systems. *ACM Computing Surveys (CSUR), 52(6), 1-36.*

Ogorzalek, M. J. (2017). A survey of key management in cloud computing. *Journal of Cloud Computing, 6(1), 1-22.*

Petticrew, M. &. (2008). *Systematic reviews in the social sciences: A practical guide.* John Wiley & Sons.

Ruoti, S. O. (2020). A usability study of three end-to-end encryption technologies for securing e-mail traffic. *Journal of Cybersecurity and Privacy, 1(4), 626-642.*

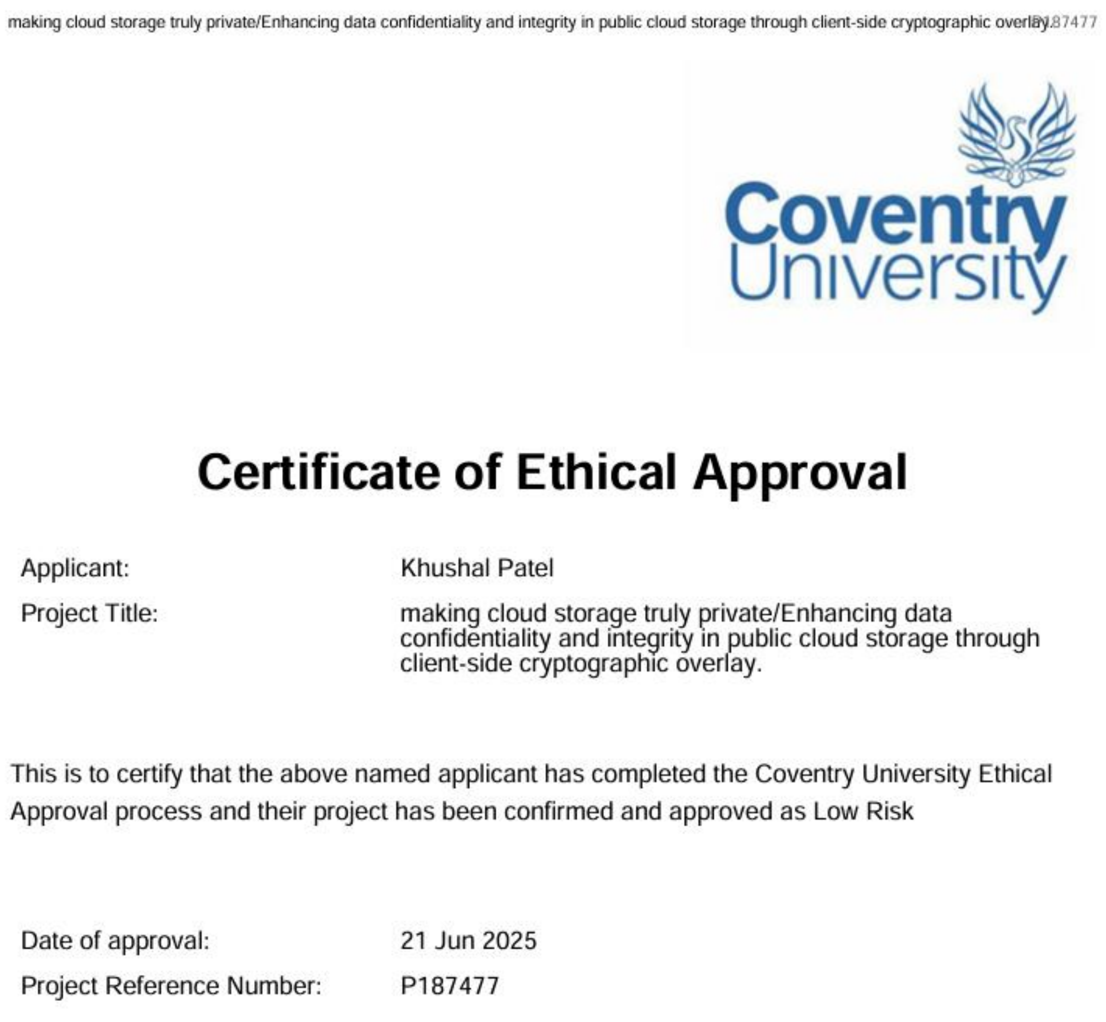
W3C. ((2017).). *Web Cryptography API. W3C Recommendation.* Retrieved from World Wide Web Consortium.: https://www.google.com/search?q=https://www.w3.org/TR/WebCryptoAPI/

Whitten, A. &. (1999). Why Johnny can't encrypt: A usability evaluation of PGP 5.0. *In Proceedings of the 8th USENIX Security Symposium.*

Yang, G. C. (2025). Implementing zero-knowledge proofs for enhanced security in cloud authentication systems. *Journal of Cloud Computing, 14(1), 145-158.*

Zissis, D. &. (2012). *Addressing cloud computing security issues*. Retrieved from Future Generation Computer Systems, 28(3), 583–592.: https://doi.org/10.1016/j.future.2010.12.006

Appendix – A



making cloud storage truly private/Enhancing data confidentiality and integrity in public cloud storage through client-side cryptographic overlay. P187477

Coventry University

Certificate of Ethical Approval

Applicant:	Khushal Patel
Project Title:	making cloud storage truly private/Enhancing data confidentiality and integrity in public cloud storage through client-side cryptographic overlay.

This is to certify that the above named applicant has completed the Coventry University Ethical Approval process and their project has been confirmed and approved as Low Risk

Date of approval:	21 Jun 2025
Project Reference Number:	P187477