DEVELOPMENT OF A PRIVACY-FOCUSED FILE TRANSFER SYSTEM WITH INTEGRATED MALWARE DETECTION AND ROLE-BASED ACCESS ENFORCEMENT

BY

OSARUMWENSE TIMOTHY OMOBUDE

PSC2105384

DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF PHYSICAL SCIENCES,
UNIVERSITY OF BENIN,
BENIN CITY,
EDO STATE, NIGERIA.

November 2025

DEVELOPMENT OF A PRIVACY-FOCUSED FILE TRANSFER SYSTEM WITH INTEGRATED MALWARE DETECTION AND ROLE-BASED ACCESS ENFORCEMENT

BY

OSARUMWENSE TIMOTHY OMOBUDE

PSC2105384

A PROJECT REPORT SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE, FACULTY OF PHYSICAL SCIENCES, UNIVERSITY OF BENIN, BENIN CITY

IN PARTIAL FULFILMENT OF THE REQUIREMENT FOR THE AWARD OF A BACHELOR OF SCIENCE (B.Sc.) DEGREE IN COMPUTER SCIENCE

November 2025

CERTIFICATION

This is to certify that this project work was carried out by **OSARUMWENSE TIMOTHY OMOBUDE** with Matriculation Number **PSC2105384** under my supervision. The work is deemed adequate in scope and satisfactory in content to meet the requirements for the award of the Bachelor of Science (B.Sc.) Degree in Computer Science of the University of Benin.

_____ _____

DR. (MRS.) A.R. USIOBAIFO **DATE**

Project Supervisor

APPROVAL

This project work is hereby approved in partial fulfilment of the requirements for the award of Bachelor of Science (B.Sc.) Degree in Computer Science from the University of Benin.

_____ _____

DR. (MRS.) A. R. USIOBAIFO **DATE**

Project Supervisor

DEDICATION

This project is dedicated to God Almighty, my parents, Mr. and Mrs. Omobude, and my siblings, whose unwavering support and guidance have been instrumental to my success. It is also dedicated to everyone who believed in my potential and continually encouraged me to persevere throughout my journey at the University of Benin (UNIBEN).

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my project supervisor and Head of the Department of Computer Science, Dr. (Mrs.) A. R. Usiobaifo, for her consistent guidance and invaluable support towards the successful completion of this project.

My appreciation also goes to my project coordinator, Dr. (Mr.) M. Osagie, for his coordination and helpful insights throughout the course of this work.

I wish to extend my heartfelt thanks to all the lecturers in the Department of Computer Science whose teachings, mentorship, and encouragement have greatly shaped my academic journey: Prof. G.O. Ekuobase, Dr. F.O. Oliha, Prof. A.A. Imiavan, Prof. (Mrs.) F. Egbokhare, Prof. (Mrs.) V.V.N. Akwukwuma, Prof. F.I. Amadin, Prof. (Mrs.) S. Konyeha, Prof. (Mrs.) V.I. Osubor, Dr. (Mrs.) Aziken, Dr. F.O. Chete, Mr. P.E.B. Imiefoh, Mr. I.E. Obasohan, Mr. S.O.P. Oliomogbe, Mr. K.O. Otokiti, Mr. I.E. Obayagbonna, Mrs. R.I. Izevbizua, Mr. E.C. Igodan, Mr. J. Okhuoya, Prof. F.A.U. Imouokhome, and Mr. D.N. Idehen.

Lastly, I am also grateful to the Department of Computer Science for providing a conducive learning environment and access to resources that greatly contributed to the success of this work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

This study aims to design and implement a secure file-sharing platform that addresses the privacy, integrity, and access control challenges inherent in modern digital collaboration. The study highlights how mainstream cloud-based file-sharing solutions often compromise user privacy by lacking true End-to-End Encryption (E2EE) and failing to prevent the distribution of malicious files. This project introduces a web-based system that uses client-side encryption to ensure that only the sender and intended recipient can access file contents, even excluding the server from decryption capabilities. In addition, it integrates real-time malware detection using ClamAV and enforces a granular, per-file Access Control List (ACL) model to restrict file operations. Developed using a Python-based backend and the React JavaScript frontend, the system adopts a Zero-Trust architecture and defense-in-depth design to provide a strong and easy-to-use alternative to conventional file-sharing platforms. The solution not only enhances data privacy and threat prevention but also serves as a reference implementation for secure and accountable file sharing in both personal and organizational contexts. The project concludes with a set of practical recommendations and a comprehensive security assessment, demonstrating the system's effectiveness against common web application threats and affirming the feasibility of privacy-first development without compromising usability.

CHAPTER ONE

1.0 Background of Study

In recent years, the growth of internet usage and remote collaboration has made secure file transfer a critical concern for both individuals and organizations. From sharing personal documents to handling sensitive business data, file exchange has become central to how people communicate and work. Traditional protocols such as FTP (File Transfer Protocol) were among the earliest solutions for file transfer; however, enabled basic file sharing but lacked fundamental security features, transmitting data and credentials in plaintext. This led to the emergence of secure alternatives like SFTP (SSH File Transfer Protocol) and FTPS (FTP Secure), which introduced encryption. However, despite their enhanced security, these protocol-based systems often lack intuitive user interfaces and fine-grained access control mechanisms needed by today's organizations. As a result, their adoption has remained largely confined to technical users and enterprise environments, limiting their applicability in broader, user-friendly contexts.

Today, most people rely on cloud-based platforms like Google Drive, Dropbox, and OneDrive. While these services prioritize convenience and accessibility, they often do so at the expense of user privacy and security. A major limitation of these platforms is their lack of true End-to-End Encryption (E2EE), a security model in which only the sender and intended recipient can decrypt the data. Without E2EE, service providers retain the technical ability to access user data.This architecture, known as server-side scanning, is the foundation of the modern internet's content moderation status quo, but it creates an inherent privacy vulnerability. It makes user data accessible to insider threats, government subpoenas, and platform-level data breaches.

This architectural choice lies at the heart of a global debate pitting privacy against security. While server-side access enables platforms to scan for malicious content like malware or Child Sexual Abuse Material (CSAM), it forces users into a model of blind trust.

Beyond privacy, another gap many platforms suffer from inconsistent malware detection. This creates the risk of malicious files being uploaded and spread especially in environments where files are shared frequently and with many users. Furthermore,flawed access control implementations, such as IDOR

(Insecure Direct Object References), can result in unauthorized users accessing restricted files by tampering with parameters like object IDs in API requests often due to weak or absent permission checks.

These limitations show that many modern platforms prioritize convenience over security. This project aims to address that imbalance by designing a file transfer system where privacy, malware protection, and role-based access control are not afterthoughts but core design principles.

1.1 Motivation

Despite growing awareness of cybersecurity threats, many developers and organizations still treat security as an afterthought, rather than an integral part of the system's design. Data breaches, malware infections, and unauthorized access incidents continue to rise, largely due to the exploitation of known and preventable vulnerabilities. The absence of proactive security design leads to systems that are vulnerable by default, requiring patches and mitigation only after exploitation occurs.

This project is driven by the belief that privacy and security should not be optional extras or last-minute additions. Instead, they should be built into the system from the start. The goal is to demonstrate that it is both possible and practical to create a file transfer platform where the architecture itself protects user data.

- Client-side End-to-End Encryption (E2EE) – so that files are encrypted before leaving the user's browser, and only the intended recipient can decrypt them—even the server has no access to the content.
- Real-time Malware Detection – to scan uploaded files and block potentially harmful content before it can spread.
- Role-Based Access Control (RBAC) – to make sure users can only see or act on what they're allowed to, reducing risks like privilege escalation and unauthorized file access.

The motivation behind this system is not just to build a working prototype, but to show that secure-by-design systems can be usable and efficient without compromising on protection. In an age where users are often forced to trade privacy for convenience, this project takes a different stance: that strong security and a smooth user experience can and should coexist.

1.2 Statement of the Problem

The need for easy and reliable file sharing has grown significantly in recent years, driven by the increasing need for remote collaboration, cloud storage, and digital communication. However, this growing reliance on file-sharing platforms has brought to light a serious challenge: the industry-wide compromise between usability, privacy, and security. Many popular solutions prioritize convenience, but in doing so, they leave users and organizations exposed to a range of well-documented and preventable threats.

The fundamental problem this project addresses is that existing file-sharing systems often fail to deliver a secure, and privacy-preserving solutions. Users are frequently forced into a false choice either accept a seamless experience and sacrifice security, or adopt secure systems that are impractical or difficult to use. This trade-off is both unnecessary and dangerous, especially given the evolving threat landscape.

A key issue lies in flawed authorization models and broken access control mechanisms. Many modern file-sharing platforms suffer from insecure access control mechanisms, particularly in their API implementations. A common example is Insecure Direct Object References (IDOR), where the backend trusts client-supplied identifiers (such as file IDs or user tokens) without performing sufficient authorization checks. This allows attackers to manipulate URLs or API requests to access or modify files they should not be able to see. These vulnerabilities stem from the absence of a zero-trust access control model, where each request should be rigorously verified for permissions and context. The failure to implement request-level access validation continues to be a major cause of data leakage, privilege escalation, and unauthorized file access in file-sharing platforms.

Another critical weakness is the lack of true data privacy and client-side encryption. Although most commercial file-sharing platforms offer server-side encryption, this does not guarantee true privacy. In typical implementations, data is encrypted during transit and at rest, but decrypted on the server meaning the platform provider (and potentially, its employees or service partners) retains full access to user data. This opens the door to insider threats, accidental exposure, and compliance risks from legal demands such as subpoenas or government surveillance. Without client-side End-to-End Encryption (E2EE) where data is encrypted before it leaves the user's device and remains inaccessible to the server users are ultimately relying on blind trust rather than verifiable security guarantees.

A further problem is the passive distribution of malware through trusted platforms. The absence of integrated, real-time malware detection in most file-sharing systems introduces another layer of risk. Users frequently exchange files within trusted workspaces, believing them to be secure. However, without active scanning during the upload process, malicious payloads (e.g., ransomware, spyware, trojans) can be silently propagated throughout an organization. This is especially dangerous in shared environments where files are automatically downloaded, synced, or previewed. These platforms, by failing to inspect content during ingestion, can inadvertently become distribution channels for malware, facilitating lateral movement and system compromise within connected networks.

In summary, the problem this project seeks to address is the systemic failure of current file-sharing platforms to combine robust authorization, data privacy, and malware protection within a usable, secure-by-design architecture. This research and development effort aims to bridge the gap between theoretical best practices in cybersecurity and practical user-centered design, creating a platform that ensures security and privacy without compromising usability.

1.3 Aim and Objectives

The primary aim of this study is to design, develop, and evaluate a secure, privacy, multi-user file transfer web application that ensures the confidentiality, integrity, and availability of user data through a layered security approach. The system will be built from the ground up with a secure-by-design philosophy, incorporating client-side encryption, proactive threat detection, and access control mechanisms. The project seeks to provide a functional and demonstrable implementation of a modern file-sharing platform that resists common vulnerabilities and adheres to contemporary cybersecurity best practices.

To achieve the stated aim, the following specific objectives will be pursued:

1. To design a secure system architecture that incorporates a Zero-Trust API model and a well-structured, normalized database schema, ensuring strict separation of concerns and minimal implicit trust between system components.
2. To implement a robust user authentication and Role-Based Access Control (RBAC) mechanism, enabling clear separation of privileges among distinct user roles such as 'Admin', 'Editor', and 'Viewer', and enforcing least-privilege principles throughout the application.

3. To integrate a client-side End-to-End Encryption (E2EE) mechanism, leveraging the browser's native Web Crypto API to encrypt files before they leave the client device, ensuring that plain-text content is never exposed to the server or service provider.
4. To implement and evaluate the effectiveness of a server-side malware detection module on E2EE files to test the hypothesis of a best-effort security layer.
5. Develop secure file management features, including upload, download, deletion, and listing via API endpoints that are protected against IDOR, broken access controls, and injection attacks.
6. To conduct a comprehensive security assessment of the final application, including manual penetration testing aligned with the OWASP Top 10, and to document the system's security posture, highlighting mitigations, potential residual risks, and recommendations for future improvements.

1.4 Scope of Study

This study focuses on the design, development, and security evaluation of a web-based, multi-user file transfer system that emphasizes user privacy, data integrity, and controlled access. The application will be developed with a secure-by-design approach, incorporating modern cryptographic methods, proactive threat detection, and strict role-based access enforcement. The scope of work is summarized below:

System Architecture and Design: Designing a modular and scalable backend using the Django framework. Structuring the system based on a role based access control model to minimize implicit trust between users and backend services.

User Management and Access Control: Implementing secure user registration, login, and session management. Defining and enforcing Role-Based Access Control (RBAC) with at least three distinct roles: Admin, Editor, and Viewer. Ensuring that all access to files and system actions are governed by strict role permissions and contextual validation.

File Upload and Download Functionality: Building secure file upload and download APIs. Allowing users to view, manage, and delete their own files based on assigned privileges.

Client-Side End-to-End Encryption (E2EE): Using the browser's Web Crypto API to encrypt files before upload and decrypt them after download. Ensuring the server never has access to unencrypted file content, thereby maintaining strong data privacy.

Malware Detection Module: Integrating an open-source antivirus engine (ClamAV) to scan files during the upload process. Rejecting or quarantining any file identified as malicious to protect users from unintentional malware distribution.

Security Testing and Evaluation: Manually testing the application against common vulnerabilities outlined in the OWASP Top 10, including IDOR, Broken Access Control, and Insecure Deserialization. Documenting the security assessment process and identifying potential residual risks.

Technology Stack: The system will be developed using: A Python-based backend framework (Django), React JavaScript frontend,A PostgreSQL relational database for secure data storage, ClamAV (or its Python binding clamd) for server-side malware scanning.

Deployment: The final output will be a functional prototype deployed to cloud-based platform such as Heroku suitable for demonstration and testing purposes. It is not intended for public or commercial release during the academic project phase.

Academic Focus: The primary academic focus of the study would be on designing a secure system architecture based on modern security principles, implementing privacy features like client-side encryption, mitigating common security vulnerabilities like IDOR and Broken Access Control and demonstrating proactive malware defense in a file-sharing context.

The scope deliberately excludes the implementation of Client-Side Scanning (CSS). Based on the overwhelming consensus in the security and cryptography communities, CSS is considered a dangerous architecture that creates systemic risks for surveillance and censorship. This project proceeds on the principle that a user's device should not be turned against them, and therefore focuses on evaluating server-side and endpoint-centric security models

1.5 Significance of the Study

The significance of this study lies in its contribution to the ongoing effort to design systems that do not merely function, but function securely by default. In an era where digital communication and file

sharing are critical to both personal productivity and organizational operations, the trade-off between usability, privacy, and security remains one of the most pressing challenges in software engineering.

This project addresses that challenge by demonstrating that it is possible to integrate client-side encryption, malware detection, and role-based access control into a single, cohesive system without sacrificing usability for security or vice versa. Unlike many commercial solutions that rely on server-side security alone or treat privacy as a secondary concern, this system takes a security-by-design approach from the ground up.

- **Academic Contribution:**
  The project serves as a practical case study in secure system design, combining concepts from cryptography, access control models, and software security engineering. It reinforces theoretical knowledge with hands-on implementation of real-world defense mechanisms against the OWASP Top 10 vulnerabilities.
- **User Privacy and Trust:**
  By implementing true **End-to-End Encryption (E2EE)** in the browser, the system ensures that even the server cannot read user data. This respects user autonomy and strengthens trust in the platform by minimizing the attack surface at the provider level.
- **Proactive Threat Mitigation:**
  The integration of **automated malware detection** into the upload pipeline offers an added layer of defense, preventing the platform from becoming a vector for malware distribution an often overlooked aspect in many popular file-sharing tools.
- **Access Control Best Practices:**
  Through the implementation of **Role-Based Access Control (RBAC)** and hardened API endpoints, the study underscores the importance of fine-grained, explicit authorization checks, reducing the risk of Broken Access Control and IDOR flaws.
- **Blueprint for Secure Systems:**
  The project provides a **reference implementation** that can be expanded or adapted for use in more complex systems, serving as a foundation for developers, researchers, and organizations seeking to prioritize user-centric security in file transfer applications.

In summary, this study is not only technically relevant but socially important. It contributes to the broader goal of building digital systems that empower users with control over their data, reduce systemic risks, and serve as resilient infrastructure in an increasingly hostile cybersecurity landscape.

1.6 Definition of Terms

End-to-End Encryption (E2EE): A security mechanism where data is encrypted on the sender's device and only decrypted on the recipient's device. Even the server or intermediary systems cannot read the plaintext content.

Client-Side Encryption: The process of encrypting data within the user's browser or application before it is transmitted to a server, ensuring that only the client holds the decryption keys.

Malware Detection: The process of scanning files for malicious software such as viruses, trojans, worms, or ransomware. This project uses a malware detection engine (e.g., ClamAV) to inspect uploaded files before storage.

Role-Based Access Control (RBAC): A security paradigm in which user permissions are determined based on predefined roles (e.g., Admin, Editor, Viewer), reducing the risk of unauthorized access.

Broken Access Control: A class of security vulnerabilities where a system fails to enforce proper restrictions on user permissions, allowing users to access or modify resources they should not.

Insecure Direct Object Reference (IDOR): A type of vulnerability where attackers manipulate parameters (like user IDs or file paths) to gain unauthorized access to objects in the system.

Web Crypto API: A native JavaScript API provided by browsers that allows developers to perform cryptographic operations (e.g., encryption, decryption, hashing) in the client environment.

Zero-Trust Security Model: A cybersecurity approach where no user, device, or service is inherently trusted. Every access request must be explicitly verified, regardless of its origin.

ClamAV: An open-source antivirus engine used for detecting malware in files. It supports real-time scanning and is often integrated into server-side applications.

Defense-in-Depth: A layered security strategy that implements multiple controls and safeguards at different points in the system to protect against a wide range of threats.

Secure-by-Design: A software development philosophy where security is considered from the outset, not added as an afterthought.

Penetration Testing: A manual or automated method of evaluating the security of a system by simulating real-world attacks to identify vulnerabilities.

CHAPTER TWO

LITERATURE REVIEW

This chapter presents a review of the existing academic and technical literature surrounding the development of secure file transfer systems. Its primary objective is to establish the context for this project by examining the trade-offs between usability, security, and user privacy that have defined the evolution of file-sharing technologies. The review discusses the evolution from early, insecure protocols to modern cloud platforms,highlighting how different design choices have balanced convenience and privacy. It then explores the core concepts End-to-End Encryption (E2EE) for confidentiality, modern access control models for authorization, and malware detection for data integrity. By evaluating existing platforms against these security principles, the chapter identifies a research gap in current solutions, which this project seeks to address.

2.1 Theoretical Framework and Security Principles

The design of a secure file transfer system must rest on solid theoretical foundations and security principles. These frameworks serve both as conceptual guides and as practical guardrails to ensure that confidentiality, integrity, and availability are upheld throughout the system's lifecycle.

One widely adopted model is the CIA Triad (Confidentiality, Integrity, Availability). This triad remains central in information security: confidentiality is enforced via encryption and strict access control; integrity is maintained through cryptographic hashing, digital signatures, and checksums; and availability is ensured via redundancy, fault tolerance, and resilient architectures (Kingsley, 2024). While powerful in its simplicity, the CIA Triad is broad and must often be augmented by more refined models to address modern threats.

The Zero-Trust Security Model refines the notion of trust in distributed systems by rejecting assumptions of implicit trust within any network or system. Under zero-trust, every access request whether internal or external must be authenticated, authorized, and validated continuously (Lund et al., 2025). Zero-trust is especially relevant for file transfer systems in multi-cloud or cross-organizational contexts, where assuming trust in network perimeters is untenable (Ren et al., 2025). The U.S. Department of Defense's Zero Trust Reference Architecture further emphasizes that no implicit trust is granted based on physical or network location (U.S. Department of Defense [DoD], 2022).

Complementary to zero-trust is Defense-in-Depth, which endorses layered security controls so that failure in one layer does not lead to complete compromise. For example, encryption, access control, intrusion detection, network segmentation, endpoint protections, and logging are stacked so attackers must surmount multiple hurdles. Research argues that combining zero-trust with defense-in-depth yields a more robust, resilient security architecture (Gajbhiye et al., 2023).

From a cryptographic standpoint, several principles underlie secure file transfer: public-key cryptography enables secure key exchange between endpoints, symmetric encryption ensures efficient bulk data encryption, hashing and message authentication codes (MACs) provide integrity verification, and digital signatures support authenticity and non-repudiation (Lund et al., 2025). Key management practices such as forward secrecy and periodic key rotation further enhance resilience.

Role-Based Access Control (RBAC) is underpinned by the principles of least privilege and separation of duties. RBAC ensures that users only execute operations permitted by their roles, thereby reducing internal misuse or privilege escalation. Coupled with continuous monitoring and auditing, RBAC helps enforce accountability and compliance in cloud-based file transfer systems (Gajbhiye et al., 2023).

In summary, a secure file transfer system should not rely on any single paradigm. Instead, the system's security posture is best guided by an integrated theoretical framework: the CIA Triad provides foundational goals, zero-trust operationalizes those goals within untrusted environments, defense-in-depth ensures redundancy, cryptographic primitives deliver confidentiality and integrity, and RBAC implements fine-grained access control. Together, these principles enable the design of systems that are privacy-aware, robust against threats, and compliant with organizational needs.

2.2 Historical Evolution of File Transfer Systems

The earliest and most foundational protocol for digital file exchange was the File Transfer Protocol (FTP), first standardized in RFC 959. Designed for the high-trust academic and military networks of the early internet, FTP was a functional marvel, enabling simple and direct file exchange between remote hosts. However, its design included a fundamental security flaw that renders it insecure in modern contexts: it transmits both user credentials and data in plain-text. This complete lack of encryption makes any FTP session vulnerable to eavesdropping and credential theft on an untrusted network (Sehat, et al., 2022).

In response to these glaring security vulnerabilities, secure variants such as FTPS (FTP Secure) and SFTP (SSH File Transfer Protocol) were developed. FTPS layered the standard FTP protocol with

SSL/TLS encryption, while SFTP operated as an entirely different protocol built over a secure SSH tunnel, encrypting both control commands and data packets. While these protocols provided robust in-transit encryption and effectively solved the confidentiality problem of FTP, their adoption remained largely confined to technical users. Their reliance on command-line tools or specialized client software created a significant usability barrier, preventing widespread adoption among the general public and non-technical business users. The industry had a secure solution, but it was not an accessible one (Arányi et al., 2024).

This usability gap was decisively closed by the emergence of cloud-based file-sharing platforms in the late 2000s, such as Dropbox, Google Drive, and OneDrive. These services represented a paradigm shift, democratizing file transfer by abstracting away the underlying protocol complexities. Through intuitive web interfaces and the revolutionary feature of automatic file synchronization, they made file storage, sharing, and collaboration effortless for millions of users. This movement solved the usability problem at scale, making remote file access a seamless and integral part of modern digital life (Hassan et al., 2022).

However, these platforms introduced a new privacy challenge due to their centralized trust model. While providers typically secure files in transit with TLS and at rest with AES-256, they retain control of the encryption keys. This allows providers to decrypt and access user files when required, exposing data to risks such as insider threats, provider-level breaches, and compliance with government requests (ETH Zurich, 2024). Thus, the convenience-driven evolution of file transfer systems inadvertently created a privacy crisis, highlighting the need for next-generation solutions that remove the necessity of trusting providers.

Recent developments have begun addressing this gap through end-to-end encryption (E2EE) file-sharing systems and zero-knowledge architectures offered by services such as Proton Drive, Tresorit, and CryptPad. These systems ensure that encryption keys are generated and stored exclusively on client devices, preventing providers from accessing user data even under legal or technical pressure (Müller et al., 2024). Research has also advanced toward post-quantum secure transfer mechanisms and blockchain-backed audit trails, illustrating the trajectory toward resilient, privacy-preserving, and transparent file-sharing ecosystems (Sola-Thomas & Imtiaz, 2025). This trajectory underlines the central research motivation of this project: designing a secure, privacy-preserving file transfer system that balances usability with advanced protections such as E2EE, integrated malware detection, and granular, user-driven access control.

2.3 The Concept of Secure File Transfer

Secure file transfer refers to the controlled exchange of digital information across networks with mechanisms that ensure confidentiality, integrity, authenticity, and availability. At its core, secure file transfer protects user data from unauthorized access, tampering, or leakage during transmission and storage (Rajkumar et al., 2022). Unlike traditional methods, which often emphasized transmission speed and reliability, secure file transfer integrates safeguards to address the expanding threat landscape (Islam et al., 2024).

A secure file transfer system incorporates three foundational elements. First, cryptographic protection ensures that data is encrypted both in transit and at rest, shielding it from eavesdropping and interception (Shankar et al., 2024). Second, authentication and authorization mechanisms verify user identity and enforce access privileges, thereby reducing risks of insider threats and credential misuse (Goodrich et al., 2024). Third, auditing and accountability features enable organizations to monitor, log, and verify file access activities for compliance purposes (Islam et al., 2024; Zheng *et al*, 2025).

Modern implementations extend beyond these core requirements to address challenges unique to distributed and cloud-based environments. For example, cloud-native solutions increasingly adopt zero-knowledge encryption models and zero-trust architectures, where even the service provider cannot access user files, enhancing data sovereignty and privacy (Shankar et al., 2024). Similarly, organizations are deploying secure transfer systems that integrate with identity-based cryptographic auditing to ensure confidentiality and resilience even in the presence of quantum-era threats (Wang et al., 2024).

This literature emphasizes that secure file transfer is no longer a peripheral function but a critical enabler of trust in digital ecosystems. Whether in healthcare, finance, or government, ensuring that files can be shared securely underpins regulatory compliance, operational resilience, and user confidence (Rajkumar et al., 2022). As such, secure file transfer represents both a technical requirement and a strategic priority in modern information security.

2.4 End-to-End Encryption (E2EE) in File Sharing

End-to-End Encryption (E2EE) has emerged as one of the most reliable approaches for protecting data confidentiality during file transfer. Unlike traditional encryption schemes where service providers often retain control of decryption keys, E2EE ensures that only the sender and the intended recipient possess the cryptographic material necessary to access file contents (Alatawi & Saxena, 2023). This model

effectively eliminates the risk of provider-side compromise and significantly reduces the impact of insider threats.

E2EE systems typically employ asymmetric cryptography to establish secure key exchanges between communicating parties, followed by symmetric encryption for efficient bulk data transfer (Raj et al., 2024). By combining these methods, E2EE ensures both confidentiality and performance. file-sharing platforms and secure storage services adopt this hybrid cryptographic model to safeguard user data in transit and at rest (Raj et al., 2024).

Despite its advantages, E2EE introduces several challenges. One critical limitation is its conflict with content scanning and malware detection. Because intermediaries cannot decrypt the data, server-side mechanisms for detecting malicious content become ineffective. To address this, researchers have developed models that analyze encrypted traffic for behavioral patterns. For instance, Berrueta et al. (2022) show that ransomware can be detected by monitoring traffic flows and file system operations even when the payload is encrypted. However, these methods are constrained in what they can detect without decrypting content.

Other approaches proposed include secure multiparty computation and homomorphic encryption, which theoretically allow computation or inspection on encrypted data without exposing content. Fully homomorphic encryption (FHE) has been applied in blockchain-based systems to enable operations over encrypted data (Wu et al., 2024; Raj et al., 2024). Additionally, homomorphic secret sharing has been proposed to combine data confidentiality with some degree of utility in cloud computing contexts (Ali et al., 2024).

Another trade-off concerns administrative oversight and compliance. In enterprise and regulated settings, organizations may need to inspect file contents for legal, regulatory, or malware/compliance reasons. Client-side scanning (CSS) has been proposed as a compromise: scanning occurs on the user's device before encryption or after decryption. However, such methods raise concerns about undermining the guarantees of E2EE and creating potential backdoors (Alatawi & Saxena, 2023; Internet Society, 2025).

Overall, E2EE represents a cornerstone of secure file transfer, providing unmatched confidentiality guarantees while highlighting persistent trade-offs between privacy, usability, and organizational security.

2.5 Malware Detection in File Transfer Systems

While encryption safeguards confidentiality, it does not inherently protect against the distribution of malicious files. Malware detection mechanisms are therefore integral to secure file transfer systems, ensuring that transmitted data does not introduce threats such as viruses, trojans, or ransomware into organizational environments (Cui et al., 2024). Traditional malware detection relies heavily on signature-based scanning, where file contents are compared against known malware patterns. Although effective for identifying previously catalogued threats, this approach struggles with zero-day malware and polymorphic attacks that continuously alter their code (Brezinski, 2023).

To address these limitations, contemporary systems increasingly employ behavioral analysis and machine learning–based detection techniques. These approaches analyze file characteristics, execution patterns, and heuristics to identify suspicious activity even when exact signatures are unavailable (Cui et al., 2024; Berrueta et al., 2022). For example, cloud-based antivirus engines leverage ML models to detect anomalies at scale, improving resilience against evolving threats (Wang & Thing, 2023).

A key challenge emerges when malware detection is integrated with end-to-end encrypted (E2EE) file transfers. Since encrypted files cannot be inspected by intermediaries without breaking privacy guarantees, organizations often resort to client-side scanning before encryption or endpoint scanning after decryption. While effective, these solutions shift the responsibility to end-users or endpoints, potentially leaving gaps if devices are unprotected or compromised (Berrueta et al., 2022; Pang et al., 2025)

Emerging research explores privacy-preserving detection methods such as homomorphic encryption and secure multiparty computation, which theoretically enable malware scanning without decrypting file contents. Recent work by Pang et al. (2025) introduced *FICConvNet* implements malware detection under CKKS homomorphic encryption, enabling inference on encrypted data without exposing plaintext (Pang et al., 2025). Other work studies the application of format-preserving encryption to mask ransomware behaviour and then uses ML to detect it (Lee et al., 2025). However, the computational overhead of these methods (e.g. in FHE, feature extraction, inference latency) currently limits their real-world adoption.

A more practical compromise lies in hybrid strategies, where metadata analysis, reputation services, sandboxing, and endpoint scanning are combined with signature- and behaviour-based detection to

balance security and efficiency. Methods that monitor traffic flows (such as MVDet) are valuable because they can work even if payload is encrypted, leveraging features like statistical distributions, TLS flow behavior, DNS patterns, etc. (Cui et al., 2024; Wang & Thing, 2023).

In summary, malware detection in file transfer systems remains a critical but complex task. The system developed in this project implements a pragmatic compromise to this challenge. By running a signature-based scan on the encrypted file blob on the server before storage, it performs a "best-effort" check. While this cannot detect malware signatures within the encrypted content, it can identify known malicious file containers or structural patterns that persist even after encryption, acting as a defense-in-depth layer, but cannot inspect the plaintext payload. The need to protect users from malicious content often conflicts with the confidentiality guarantees of strong encryption. As file transfer platforms continue to adopt E2EE, research into efficient, privacy-preserving malware detection mechanisms will remain an urgent priority.

2.6 Access Control Models in Cloud Systems

Role-Based Access Control (RBAC) is a widely adopted security model that regulates access to resources based on predefined organizational roles. Instead of assigning permissions directly to individuals, RBAC associates users with roles, and roles are linked to specific access privileges. This abstraction simplifies administration while ensuring that users can only perform actions consistent with their responsibilities (Akuthota, 2025, Gouglidis et al., 2023).

In the context of cloud-based file transfer systems, RBAC plays a critical role in managing multi-tenant environments where diverse users interact with shared resources. For example, administrators may be granted full control over system configurations, employees have restricted access to departmental files, and external collaborators are limited to specific shared folders. By clearly separating permissions, RBAC reduces the risk of unauthorized access and enforces the principle of least privilege (HoneyBee, 2025; Akuthota, 2025).

While RBAC provides a strong framework for organizational roles, a more detailed approach is offered by Access Control Lists (ACLs). An ACL is a table of permissions attached to a specific object, such as a file. Each entry in the list specifies a user and their individual permission level (e.g., 'viewer', 'owner') for that object alone. Unlike RBAC, which grants system-wide capabilities based on a user's role, ACLs provide fine-grained, user-driven control on a per-object basis. This model is highly prevalent in modern collaborative platforms like Google Drive, where users can share individual files with specific

people, empowering them with direct control over their own data. For this project, an ACL model was chosen over a rigid RBAC system to better align with the goal of user-centric privacy and flexible, intuitive sharing, as implemented via the FilePermission database table.

Despite its advantages, RBAC faces several challenges in large-scale deployments. One common issue is role explosion, where the proliferation of fine-grained roles makes management complex and error-prone (Shakarzy, 2024). Additionally, static role assignments may lack the flexibility needed to address dynamic, context-dependent access requirements such as location, device type, or time of access. To address this, models like GSTRBAC incorporate spatio-temporal constraints into RBAC, and some proposals combine RBAC with zero-trust architectures that assess trust or risk dynamically (Al Lail et al., 2024; Zero-trust Dynamic Access Control, 2025).

Advanced models like Attribute-Based Access Control (ABAC) and hybrid RBAC-ABAC systems have been proposed to overcome these limitations. These models augment RBAC with attributes (user, environmental, resource) or policy engines that evaluate contextual information to make access decisions dynamically (Akuthota, 2025; Zero-trust Dynamic Access, 2025).

Integration of RBAC with identity and access management (IAM) frameworks further enhances its effectiveness in cloud ecosystems. Modern IAM solutions provide centralized policy enforcement, auditing, federated identity support, and tools for detection of misconfigurations (Model Checking IAM, 2023; Akuthota, 2025). These integrations are essential to enforce consistent security, traceability, and compliance in complex cloud file transfer settings.

Overall, RBAC remains a foundational component of secure file transfer in the cloud, offering a structured and scalable approach to access control. However, the growing complexity of cloud environments highlights the need for hybrid or extended models that combine RBAC with contextual and adaptive access control strategies to retain least privilege while also accommodating real-world dynamic requirements.

2.7 Related Works and Existing Platforms

The growing demand for secure file transfer has led to both academic research and commercial platforms attempting to balance confidentiality, integrity, availability, and usability. Reviewing these contributions provides insights into the technical evolution of the field as well as the limitations that persist.

Academic research has primarily focused on advancing the cryptographic and architectural foundations of secure file transfer. Studies on end-to-end encryption (E2EE) highlight its effectiveness in preserving confidentiality, while also emphasizing the challenges it introduces for malware detection and compliance monitoring (Hofmann & Truong, 2024). Other works explore privacy-preserving computation techniques such as homomorphic encryption, proxy re-encryption, and secure multiparty computation. These methods show potential for enabling secure content scanning without breaking encryption, though scalability and efficiency remain significant barriers to practical deployment (Wang & Thing, 2023; Pang et al., 2025).

Commercial platforms illustrate varied approaches to translating these principles into practice:

- **Zero-knowledge / E2EE services** (Tresorit, SpiderOak, pCloud, Sync, Icedrive, Seafile) focus on client-side encryption, ensuring that providers have no visibility into user data. While these maximize confidentiality, vulnerability analyses have revealed weaknesses in implementation such as flawed key management and tampering risks (Hofmann & Truong, 2024).
- **Mainstream cloud storage providers** (Dropbox, Google Drive, OneDrive) traditionally relied on encryption in transit and at rest but not full E2EE. However, integrations such as Dropbox's adoption of Boxcryptor technology now allow for optional zero-knowledge features in business tiers (Dropbox/Boxcryptor, 2022).
- **Enterprise-oriented platforms** (FileCloud, Kiteworks) emphasize regulatory compliance, governance, and administrative oversight. They offer hybrid deployment options, detailed auditing, IAM/RBAC integration, and retention controls — features valued by organizations but often requiring trust in provider-managed keys or infrastructure.

Comparative analyses indicate that no single solution fully resolves the tension between privacy, compliance, and usability. Zero-knowledge solutions maximize confidentiality but reduce administrative visibility and complicate malware detection. Enterprise platforms enable stronger oversight and compliance alignment but introduce trade-offs in trust and potential exposure of encryption keys (ETH Zurich, 2024; Dropbox, 2022).

Together, these strands of research and practice highlight both the maturity and fragmentation of the field. Academic proposals offer powerful cryptographic mechanisms for protecting data during transfer and use, while industry platforms prioritize usability, integration, and compliance. This tension reveals

a persistent research gap: the need for secure file transfer systems that preserve strong confidentiality guarantees without undermining organizational oversight or the ability to detect malicious content.

2.8 Analysis of Existing Systems

A comparative analysis of existing file transfer and cloud storage systems highlights both the security strides that have been made and the gaps that remain. Evaluating commercial and open-source platforms reveals recurring trade-offs among usability, privacy, and security.

Mainstream providers such as Dropbox, Google Drive, and OneDrive provide robust infrastructure with encryption in transit and at rest. For example, Dropbox uses AES-256 for encryption at rest and TLS/SSL for transmissions, and has recently integrated Boxcryptor technology to embed zero-knowledge, end-to-end encrypted folders for business users. However, their reliance on server-side key management means that, for many users, providers retain the ability to decrypt or access content outside of strictly E2EE folders. A comparative study of encrypted cloud platforms found that most commercial systems depend on provider-controlled key management, thereby limiting user autonomy and full confidentiality (Manthiramoorthy et al., 2024). Additionally, integrated real-time malware scanning is not consistently documented or implemented; non-E2EE areas often depend on provider-side scanning, while E2EE folders limit such oversight.

In contrast, privacy-centric services like Tresorit adopt a zero-knowledge model that enforces end-to-end encryption where the provider never holds the decryption keys. Tresorit's file handling process encrypts data locally before upload, ensuring that unencrypted files are never stored on provider servers. This design maximizes confidentiality but limits administrative oversight, since fully E2EE content cannot be inspected by providers for malware or compliance auditing. A formal framework for end-to-end encrypted cloud storage confirmed that while such systems achieve strong confidentiality and integrity guarantees, they still lack standardized mechanisms for scalable key recovery and forensic traceability (Backendal et al., 2024).

Open-source platforms such as Nextcloud combine flexibility with transparent security controls. They support both server-side and client-side encryption, transport-layer protection, and optional key-recovery features such as external HSM integration and administrative recovery keys. This configurability allows organizations to align security with internal policies. However, encryption modules especially client-side E2EE remain relatively new and sometimes experimental. Metadata such as filenames or folder structures can still be exposed to the server in many deployments

(Avwokuraye, Ezemonye et al., 2025). Misconfigured key management or recovery workflows can also result in permanent data loss.

Hybrid enterprise systems now emphasize role-based access control (RBAC), governance, and compliance frameworks rather than full E2EE by default. Studies of hybrid storage models show that multilayer encryption and deduplication can enhance resilience and performance, but also introduce complexity and overhead (Mageshkumar et al., 2023). For instance, Dropbox's team-oriented E2EE key-sharing mechanism allows continuity even when some keys are lost, but preserves limited metadata visibility to maintain usability and compliance. Similarly, multi-layer encoding approaches proposed for enterprise deployments seek to reconcile scalability and privacy, yet they remain computationally intensive and difficult to manage at scale (Mishra et al., 2025).

Across these systems, the fundamental trade-off persists: platforms that maximize privacy tend to sacrifice centralized visibility, real-time malware detection, and administrative recoverability, whereas those that prioritize compliance and usability often cede some confidentiality to the provider. Current architectures therefore fail to fully meet all desired objectives simultaneously namely, end-to-end encryption, continuous malware detection, fine-grained RBAC, and verifiable compliance.

This continuing gap underscores the need for next-generation secure file transfer systems that integrate these capabilities holistically achieving a balance between privacy, security, usability, and accountability rather than optimizing for only one dimension.

To illustrate these trade-offs systematically, **Table 2.1** provides a comparative analysis of prominent platforms, evaluating their implementation of E2EE, malware detection, access control, and compliance standards.

Platform	**End-to-End Encryption (E2EE)**	**Malware Detection / Scanning**	**Access Control**	**Compliance & Standards**	**Limitations**
Dropbox Business	Partial / Optional. Dropbox supports encrypted team folders (end-to-end) as an add-on.	Yes (server-side scanning of non-E2EE content)	Yes (granular permissions, team roles, folder sharing)	GDPR, ISO 27001, SOC 2	Not full client-side E2EE for all content; E2EE disables some features

Google Drive	No true E2EE (encryption in transit & at rest)	Yes (Google uses content scanning, Safe Browsing, etc.)	Yes (via Google IAM / Workspace permissions)	GDPR, HIPAA, FedRAMP (in some tiers)	Privacy trade-offs: provider has key control
Microsoft OneDrive for Business	No default full E2EE (keys typically managed by Microsoft)	Yes (built-in anti-malware)	Yes (via Azure AD, conditional access, role policies)	ISO 27018, HIPAA, GDPR	Keys accessible to Microsoft unless special key-control arrangements are made
Tresorit	Yes (E2EE by default)	No known server-side malware scanning	Yes (role-based sharing, group policies)	GDPR, HIPAA, Swiss privacy, ISO frameworks	High confidentiality, but scanning capabilities limited;
Sync.com	Yes (client-side E2EE, zero-knowledge)	No native server-side malware scanning	Yes (basic team sharing, roles)	GDPR, HIPAA	Strong privacy, but weaker enterprise RBAC and oversight features
Nextcloud (self-hosted or cloud)	Optional (client-side E2EE plugin / E2EE feature)	Optional (via integration)	Yes (role-based permissions)	GDPR compliance (self-managed)	Highly configurable but requires technical expertise

Table 2.1 Comparative Analysis of Prominent Platform

The comparison reveals consistent trade-offs among the platforms. Services such as Dropbox Business, Google Drive, Microsoft OneDrive, and Box show strong capabilities in malware detection, fine-

grained RBAC, and broad compliance certifications (e.g. GDPR, ISO standards), but they all fall short on offering true end-to-end encryption: key control remains largely in the hands of the provider, leaving user data exposed to provider-side risks. Conversely, privacy-first platforms such as Tresorit and Sync.com offer full client-side E2EE, significantly improving confidentiality and user autonomy. However, these privacy gains frequently come at the expense of built-in malware scanning and richer enterprise RBAC controls, shifting detection burdens onto endpoints or individual users. Solutions like Nextcloud illustrate that hybrid or modular architectures can partially bridge the gap—allowing optional E2EE and mal-scanning via plugins but these tend to require technical expertise to configure securely. These patterns make clear that no existing platform fully satisfies all core security goals simultaneously, thereby justifying the design of a solution that integrates client-side end-to-end encryption, real-time malware detection, and fine-grained, flexible RBAC in one unified architecture.

2.9 Gap Analysis and Justification

Despite the wide range of file transfer solutions available today, a critical analysis reveals persistent gaps that limit their ability to provide comprehensive security while maintaining usability. Identifying these shortcomings helps to justify the need for the proposed privacy-focused file transfer system.

First, many platforms fail to deliver true end-to-end encryption (E2EE). While encryption in transit and at rest is widely implemented, the use of server-side key management means that service providers can still access user data. This reliance on provider trust creates vulnerabilities to insider threats, subpoenas, and data breaches (Lakshmanan, 2024; Schneier, 2025; Wire, 2025). The absence of client-side encryption undermines user autonomy and contradicts privacy-by-design principles.

Second, malware detection is often overlooked in existing systems. Mainstream platforms allow users to upload and share files without rigorous real-time scanning. As a result, malicious files such as ransomware, trojans, or spyware can spread within trusted environments. Research shows that while homomorphic encryption and privacy-preserving machine learning can theoretically enable encrypted malware detection, scalability remains a significant barrier (Alabdulmohsin et al., 2025; Duan & Grimmelmann, 2024; Jagielski et al., 2021).

Third, broken access control remains a recurring issue. Vulnerabilities such as insecure direct object references (IDOR) demonstrate that many systems do not enforce strict, request-level authorization. This weakness allows attackers to bypass permissions, exposing sensitive files and escalating

privileges. Even enterprise-grade solutions, though equipped with access control policies, are often limited by coarse-grained rules that fail to enforce least-privilege access consistently (OWASP, 2023).

Fourth, existing systems struggle to balance privacy with organizational oversight. Privacy-first platforms such as Tresorit maximize confidentiality but reduce the ability to monitor, audit, or filter malicious content. Conversely, enterprise systems like Box prioritize compliance but compromise user privacy by retaining decryption capabilities. "This fundamental trade-off is a central focus of current academic research. For example, work at Carnegie Mellon University is actively exploring cryptographic methods to resolve the conflict between the privacy guarantees of E2EE and the need for organizational content moderation (Cunningham, 2025). This highlights the absence of a unified model that simultaneously guarantees user-level privacy, malware prevention, and organizational control.

Finally, academic research and industry practice remain fragmented. Advanced cryptographic methods such as homomorphic encryption and secure multiparty computation offer promising ways to analyze encrypted data securely. However, these techniques are rarely deployed in practice due to performance and scalability limitations (Alabdulmohsin et al., 2025; Bajaj & Agrawal, 2022). This gap underscores the challenge of translating theoretical innovation into real-world adoption.

In light of these gaps, the justification for this project is clear, there is a pressing need for a secure file transfer system built on a secure-by-design framework that integrates client-side E2EE, enforces fine-grained ACL model, and embeds real-time malware detection as a core feature. While academic research has proposed powerful cryptographic mechanisms and industry platforms have prioritized usability, neither has fully resolved the tension between user privacy, administrative oversight, and malware protection. This project aims to bridge this critical gap by designing and developing a holistic architecture that balances these competing requirements. The following chapter will outline the methodology for achieving this goal.

CHAPTER THREE

METHODOLOGY AND SYSTEM DESIGN

This chapter discusses the methodology adopted for the design, development, and evaluation of the privacy-focused file transfer system. Developing a security-critical application requires a structured and iterative process that ensures reliability, usability, and effective risk mitigation. The Prototyping methodology was chosen for this project due to its flexibility and emphasis on continuous refinement through feedback, making it suitable for implementing complex features such as client-side encryption, malware detection, and role-based access control.

The chapter also analyzes existing file-sharing platforms to identify their weaknesses in privacy and security. These findings inform the proposed system's architecture, which is designed to address such gaps through a well-defined structure that includes the system architecture, database schema, API design, and data flow processes.

3.1 Software Development Methodology

For the development of this project, the iterative Prototyping Model was adopted as the guiding software development methodology. Unlike rigid, linear models such as Waterfall, or heavyweight methodologies like Structured Systems Analysis and Design Method (SSADM), the Prototyping Model emphasizes early, functional versions of the system that evolve through iterative refinement. Each prototype serves as a foundation for experimentation, validation, and user or developer feedback, enabling the final system to emerge through continuous improvement rather than a single, monolithic development cycle.

3.1.1 Justification for the Adoption of the Prototyping Model

The selection of the Prototyping Model was driven by several defining characteristics of the project, as outlined below:

1. **Complex and Unfamiliar Technical Requirements**
   The project required the integration of client-side end-to-end encryption (E2EE) using the Web Crypto API, a technically sophisticated and error-prone domain. Developing an early prototype made it possible to experiment with core cryptographic operations such as key generation, encryption/decryption workflows, and key export/import mechanisms. This iterative process allowed early validation of the feasibility and performance of the E2EE design before

committing to full-scale development. A traditional, linear model would have deferred such testing until late in the project, risking significant redesign if the initial cryptographic assumptions proved infeasible.

2. **Security-by-Design Philosophy**

   Security was not treated as an afterthought but as a foundational design constraint. The Prototyping approach enabled the system's security posture to evolve incrementally through layered implementation. The initial prototype focused on secure authentication and authorization, while later iterations integrated malware scanning, and finally, end-to-end encryption. This incremental refinement allowed for security testing and threat modeling at every stage, aligning with a defense-in-depth development philosophy and reducing the likelihood of systemic vulnerabilities.

3. **User Experience and Usability**

   While the project prioritized strong security guarantees, it also sought to demonstrate that usability and security are not mutually exclusive. The Prototyping Model made it possible to iteratively test the system's user interface (UI) and workflow design. Early, interactive versions of the file upload, sharing, and download components helped evaluate intuitiveness, responsiveness, and cognitive load. Feedback from these prototypes guided refinements to ensure cryptographic operations occurred transparently in the background, minimizing friction for end users.

4. **Flexibility and Adaptability to Evolving Requirements**

   During the early design stages, the system's access control mechanism evolved from a traditional Role-Based Access Control (RBAC) model to a more dynamic Access Control List (ACL) model. The Prototyping approach accommodated this architectural pivot seamlessly. Initial versions implemented simple ownership checks, while later iterations extended this to include sharing mechanisms, granular permission control, and revocation capabilities through a refined FilePermission model. This adaptability demonstrated the model's strength in managing evolving technical and functional requirements without derailing project timelines.

3.1.2 Stages of the Prototyping Model

The Prototyping Model typically proceeds through a structured but iterative sequence of stages. Each cycle produces a refined version of the system based on feedback and evaluation. The standard stages include:

1. **Requirement Identification:** Gathering and analyzing user needs to determine system objectives, core functions, and constraints.
2. **Initial Prototype Development:** Creating an early, simplified version of the system that captures major interfaces and workflows.
3. **User Evaluation:** Presenting the prototype for testing and collecting feedback on functionality, usability, and performance.
4. **Refinement:** Revising the prototype based on feedback, correcting issues, adding new features, and improving design quality.
5. **Implementation and Validation:** Developing the final system once the prototype adequately satisfies user and technical requirements.

This cyclical process continues until the system achieves a satisfactory balance between performance, usability, and technical soundness.

3.1.3 Prototyping Lifecycle Phases

The Prototyping lifecycle in this project followed a structured but flexible four-phase progression, each building upon the previous iteration:

- **Phase 1: Foundational Prototype (Authentication & API Shell)**
  This initial phase produced a bare-bones system integrating user registration, login, and secure token-based authentication via a RESTful API. It validated the core interaction between the React frontend and the Django backend, establishing a secure communication foundation and session management structure.
- **Phase 2: Functional Core Prototype (File Management & Malware Scanning)**
  This phase introduced the core file management functionality upload, list, and delete operations

without encryption. It also implemented Access Control List - based permissions and integrated ClamAV for malware detection. The focus was on achieving robust file handling logic and verifying that file integrity and security checks operated efficiently.

- **Phase 3: Sharing Prototype (Collaboration Layer)**
  The third phase expanded the Access Control List (ACL) model to support multi-user collaboration. New API endpoints and frontend components were developed for file sharing and access revocation. This iteration validated the system's ability to manage concurrent permissions securely and maintain consistent access state across users.
- **Phase 4: E2EE Prototype (Privacy Core Integration)**
  The final and most critical phase integrated client-side cryptographic operations using the Web Crypto API into the established file management workflows. Encryption and decryption were handled entirely on the client side, ensuring that plaintext files were never exposed to the server. This prototype unified all core components authentication, malware scanning, access control, and encryption into a cohesive, privacy-preserving system.

3.1.4 Outcome of the Prototyping Approach

Through these successive prototypes, the project achieved several key outcomes:

- **Early Risk Mitigation:** Cryptographic and architectural risks were discovered and resolved before full-scale implementation.
- **Incremental Security Validation:** Security features were built and tested layer by layer, ensuring a strong cumulative defense posture.
- **Continuous User-Centric Refinement:** Iterative feedback loops ensured that usability and system responsiveness were maintained alongside increasing security complexity.
- **Alignment with Core Objectives:** The final product emerged as both technically robust and consistent with the project's overarching goal delivering a secure, privacy-focused file transfer system without compromising usability.

In summary, the Iterative Prototyping Model provided the ideal balance of technical flexibility, security assurance, and user-focused adaptability, making it particularly well-suited for developing and refining a security-critical application of this nature.

3.2 Analysis of the Existing System

The existing system in the context of this project does not represent a specific software application but rather the model of centralized cloud-based file storage and sharing platforms, such as Google Drive, Dropbox, and Microsoft OneDrive. An analysis of their operational procedures and architectural models reveals a consistent pattern of design choices that prioritize convenience and feature richness, often at the expense of user privacy, transparency, and data sovereignty.

Purpose and Operations:
The primary objective of these platforms is to provide users with seamless and ever present access to their digital files across multiple devices and environments. Their core operations include:

- **File Upload and Storage:** Users upload files through web or desktop clients, which are then stored in a centralized cloud infrastructure managed by the service provider.
- **Synchronization:** Changes made to a file on one device are automatically propagated to all other linked devices.
- **Sharing and Collaboration:** Users can generate shareable links or invite other users via email to view or edit files and folders.
- **Rich User Experience:** Features like file previews, full-text search, and integration with other productivity tools are provided to enhance usability.

Procedures and Underlying Architecture:
The typical procedure for a file upload in these systems is as follows:

1. The user selects a file on their local device.
2. The file is transmitted to the provider's servers, typically encrypted in transit using Transport Layer Security (TLS).
3. Upon arrival at the server, the TLS encryption is terminated. The provider's infrastructure now has access to the plaintext file.
4. The provider may perform operations on the plaintext file, such as malware scanning, generating thumbnails for previews, or indexing the content for search.

5. The provider then encrypts the file using their own server-side keys (encryption at rest) and stores the encrypted data in their data centers.

Gaps in Existing Solutions

This standard operational model presents several fundamental deficiencies that this project aims to address:

1. **The Provider Trust Problem (Lack of E2EE):** The most significant flaw is the architectural necessity of trusting the service provider. Because the provider manages the encryption keys and has access to plaintext data on their servers, user privacy is not cryptographically guaranteed. This exposes user data to multiple risks:
    - **Insider Threats:** A rogue employee or administrator could access sensitive user files.
    - **External Breaches:** A compromise of the provider's infrastructure could lead to a massive data breach of unencrypted or decryptable files.
    - **Government and Legal Compliance:** The provider can be compelled by subpoenas or warrants to decrypt and hand over user data.
2. **Inconsistent or Opaque Malware Detection:** While many platforms perform some level of malware scanning, the process is often a "black box" to the user. The effectiveness, timeliness, and scope of this scanning are not always transparent. Furthermore, the rise of zero-day threats means that server-side scanning alone is not a foolproof defense.
3. **Vulnerability to Broken Access Control:** The complexity of managing permissions for millions of users and files often leads to implementation flaws. As identified in the literature review, vulnerabilities like Insecure Direct Object References (IDOR) persist, where a user can manipulate an identifier (e.g., a file ID in a URL) to gain unauthorized access to another user's data. This stems from a failure to rigorously verify permissions on every single API request.
4. **Metadata Leakage:** Even if the file content were encrypted, these systems still collect and process a vast amount of metadata, including filenames, folder structures, access times, and sharing history. This metadata can be highly sensitive and reveal significant information about a user's activities and relationships.

In summary, the existing system the mainstream cloud-sharing model accomplishes the goal of convenient file access efficiently but fails to adequately protect user privacy and security. It operates on a model of implicit trust rather than cryptographic verification, creating a systemic risk for its users.

3.3 Proposed System: Architectural Overview

To address the identified deficiencies of the existing model, this project proposes a web-based file transfer system designed from the ground up on the principles of Zero-Trust, privacy-by-design, and defense-in-depth. The proposed system is architected to eliminate the need for provider trust, mitigate the risk of malware distribution, and enforce granular, unbypassable access control.

Design Philosophy:

- **Zero-Knowledge Server:** The server is treated as an untrusted entity. It is architecturally incapable of accessing the content of the files it stores. Its primary roles are to manage user authentication, enforce access control policies on encrypted data, and orchestrate secure operations, but never to decrypt or view user data.
- **Client-Side Responsibility:** All cryptographic operations (key generation, encryption, decryption) are performed exclusively on the client-side (i.e., within the user's browser) using the native Web Crypto API. The client is the sole custodian of the decryption keys.
- **Explicit, Per-Request Authorization:** Every API request to access or modify a resource is independently authenticated and authorized. The system never trusts that a previous successful request implies future access, thus adhering to the Zero-Trust model and directly mitigating IDOR vulnerabilities.

High-Level System Architecture:

The system is composed of four primary components:

1. **React Frontend (Client):** A single-page application that serves as the user's trusted environment. It manages the user interface, handles all cryptographic operations via the cryptoService, and communicates with the backend API.
2. **Django REST Framework Backend (Server):** A stateless API server responsible for:
    - User and session management (registration, login, token authentication).

- Managing metadata (file information, permissions).
- Enforcing the Access Control List-based authorization rules via custom permission classes.
- Performing malware scans by invoking the clamscan utility.

3. **PostgreSQL Database:** The persistent storage for all metadata, including user accounts, file details (filename, size, etc.), encrypted file keys, and the FilePermission ACL table.
4. **Media Storage:** A dedicated file storage system that stores only the encrypted file blobs. A fundamental principle of the Zero-Knowledge design is minimizing server-side metadata. To this end, the system implements storage path obfuscation. Instead of storing files in user-specific directories which would link an encrypted blob to a specific user and create a metadata leak, all files are saved to a path derived from a new, randomly generated UUID within a distributed directory structure. This design ensures that an administrator or an attacker with file-system-level access can gain no information about which user owns which encrypted file. The only link between a user and their files exists within the secured database, dramatically reducing the attack surface for metadata analysis and directly supporting the project's goal of maximizing user privacy.

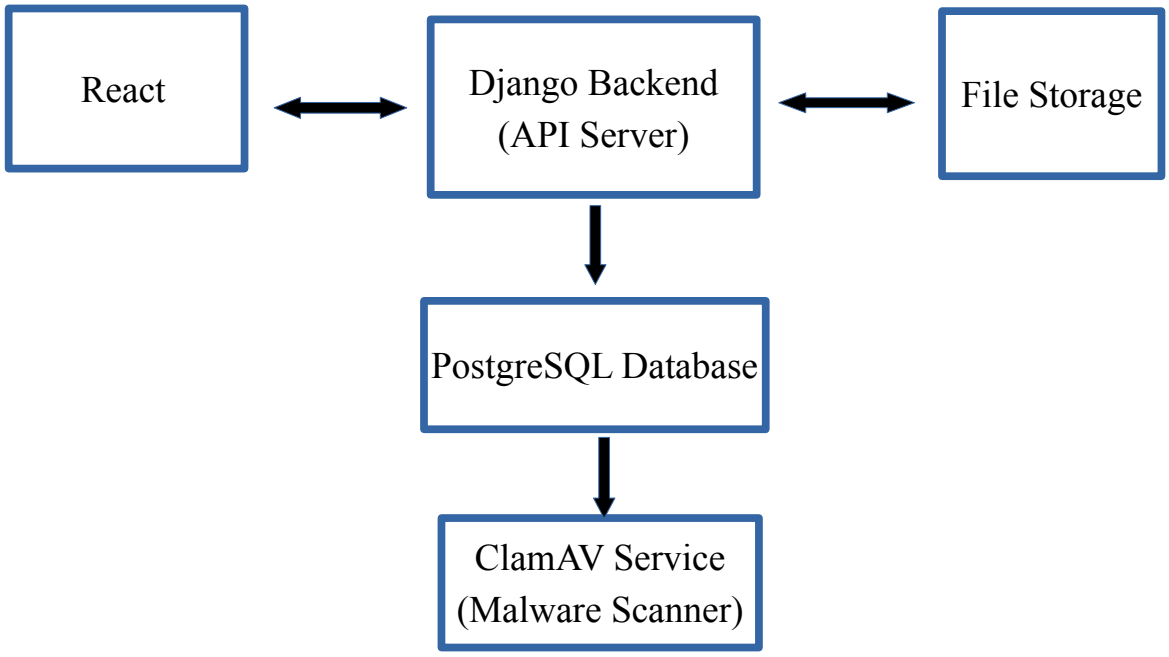


Figure 3.1: Conceptual System Architecture

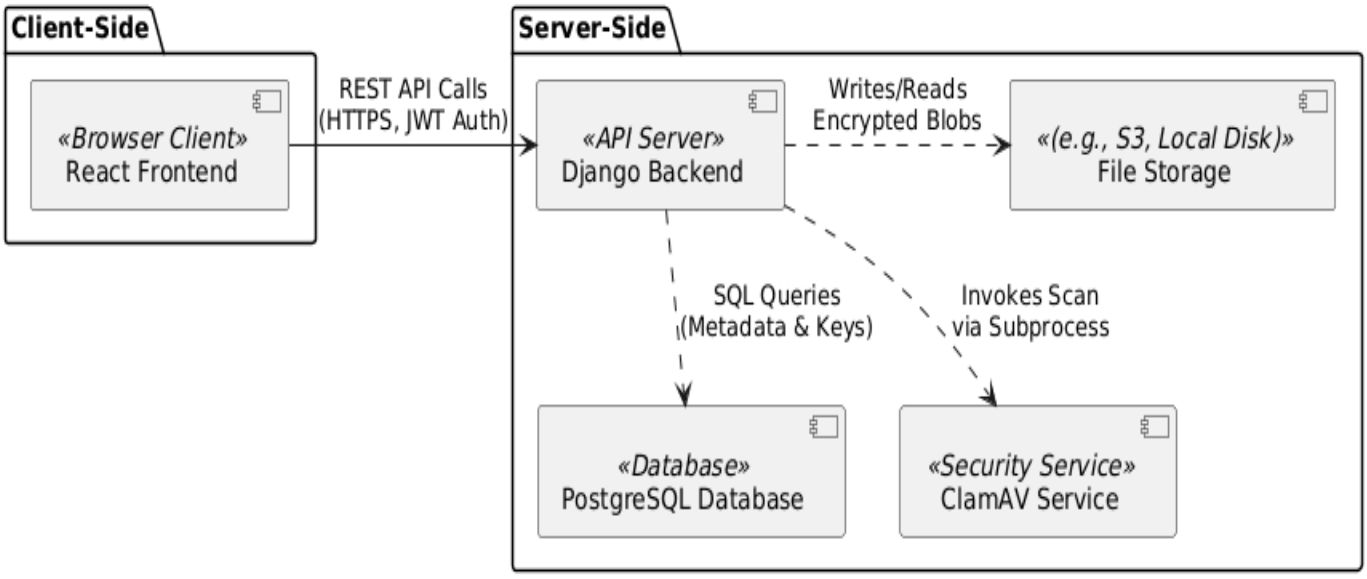


Figure 3.2: High-Level System Architecture Diagram

System Workflow (Proposed Operations):

- **User Registration & Login:** A standard token-based authentication flow (using SimpleJWT) provides a stateless and secure method for identifying users.
- **File Upload (Encrypted):**
    1. The user selects a file in the React frontend.
    2. The cryptoService generates a unique symmetric AES-GCM key for the file.
    3. The file is encrypted in the browser.
    4. The encryption key is exported to a storable string format (JWK).
    5. The frontend sends a multipart/form-data request to the Django API, containing the encrypted file blob, the original filename, and the stringified key.
    6. The Django backend receives the request, verifies the user is authenticated, and passes the encrypted blob to clamscan for a signature-based scan. While this cannot inspect the encrypted payload, it was designed to serve as a potential defense-in-depth measure, which was then subjected to rigorous testing..

7. If the scan passes, the backend saves the encrypted blob to media storage with a UUID filename and stores the file's metadata (including the JWK key string) in the PostgreSQL database.

- **File Download (Decrypted):**
    1. The user requests to download a file from the UI.
    2. The frontend calls a dedicated /download_info/ endpoint.
    3. The backend verifies the user has permission via the FilePermission table and returns the storage URL of the encrypted blob and the stored JWK key string.
    4. The frontend fetches the encrypted blob, reconstructs the CryptoKey from the JWK string, and decrypts the file in the browser's memory.
    5. The resulting plaintext data is converted to a Blob and triggered for download.

This proposed system efficiently accomplishes the core purpose of file sharing while fundamentally re-architecting the trust model. By shifting encryption to the client-side and enforcing rigorous, per-request authorization, it provides a demonstrably more secure and private alternative to the existing paradigm.

3.4 System Design

The system design defines the internal structure and operational logic of the proposed privacy-focused file transfer system. It provides a detailed view of how each component interacts to achieve secure data handling, access control, and malware detection, while adhering to the principles of client-side encryption and zero-knowledge architecture.

3.4.1 Design Goals

The design is guided by the following objectives:

1. Guarantee end-to-end confidentiality by ensuring only clients can decrypt files.
2. Maintain granular and enforceable access control through role-based permissions.
3. Integrate real-time malware detection to prevent propagation of infected files.

4. Ensure statelessness and scalability on the backend for modular deployment.
5. Deliver a user-centric interface that prioritizes both usability and security.

3.4.2 Frontend (React Application)

The React client operates as the system's trusted environment, responsible for all cryptographic processes. Key modules include:

- **Authentication Manager:** Handles registration and login through token-based authentication using the Django backend.
- **CryptoService:** Implements AES-GCM encryption/decryption, key generation, and key serialization using the native Web Crypto API.
- **FileManager Component:** Provides a secure interface for uploading, downloading, and managing file permissions.
- **APIService Module:** Acts as a middleware between frontend components and the Django API, automatically attaching JWT tokens to requests.

No plaintext data is ever stored or transmitted; decryption occurs exclusively in memory to preserve confidentiality.

3.4.3 Backend (Django REST Framework)

The Django backend serves as the **control plane** — enforcing security policies, managing metadata, and orchestrating safe file operations without accessing the actual file contents.

- **Authentication & Authorization Layer:** Uses SimpleJWT for stateless authentication and custom permission classes for role enforcement.
- **Metadata Manager:** Maintains file metadata (owner, encrypted key, access rules).
- **Malware Scanning Engine:** Integrates with ClamAV to scan encrypted blobs for structural anomalies or known malware signatures.
- **Audit and Logging Module:** Records user actions (uploads, downloads, permission edits) to ensure accountability and forensic traceability.

3.4.4 Database

The database schema is designed to support normalization, efficiency, and the enforcement of the system's access control logic. The core entities of the system are User, File, and the FilePermission junction table, which manages the many-to-many sharing relationships. The complete structure and relationships are detailed in the Entity-Relationship Diagram (ERD) in Figure 3.2.

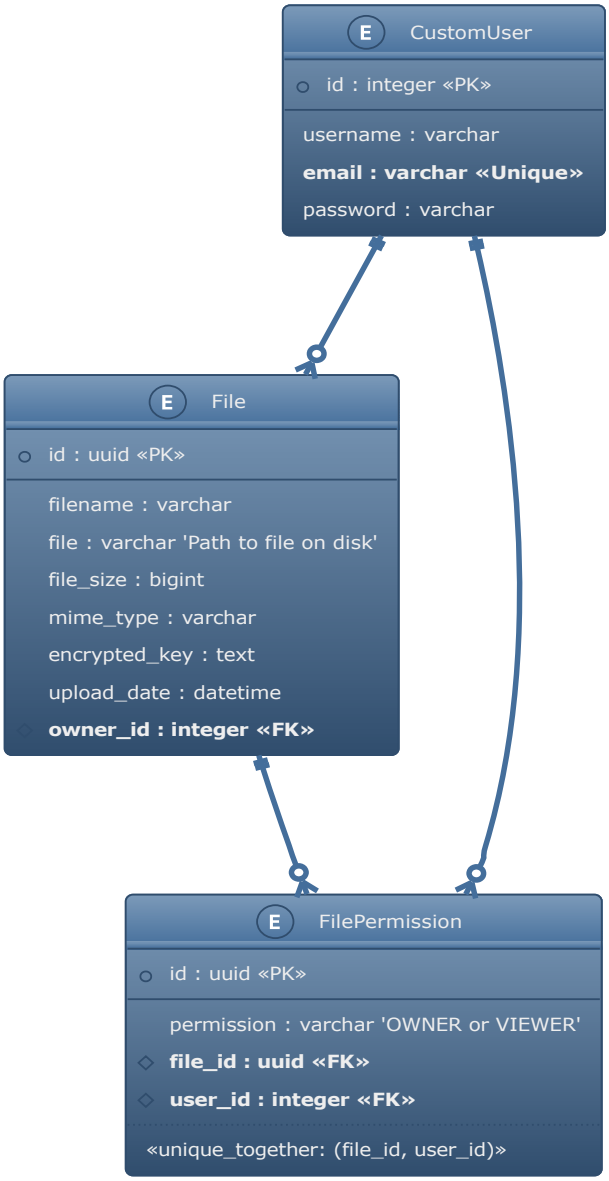


Figure 3.3: Entity-Relationship Diagram of the System Database

Primary tables include:

Table	Description
User	Stores basic account credentials and metadata.
File	Contains details such as filename, file size, owner, and encrypted key string.
FilePermission	Manages user-to-file relationships defining allowed actions (read, write, delete).

Foreign key constraints are enforced to ensure integrity between the File and FilePermission tables.

3.4.5 Security Design

Security controls are embedded across all layers using a defense-in-depth strategy:

- **Confidentiality:** All files are encrypted client-side with AES-GCM before upload.
- **Authentication:** Short-lived JWTs with refresh tokens.
- **Authorization:** Per-request verification prevents IDOR and privilege escalation attacks.
- **Transport Security:** HTTPS (TLS 1.3) is enforced for all communications.
- **Data Obfuscation:** Filenames are stored as UUIDs to prevent metadata inference.
- **Validation:** Input sanitization and strict file-type whitelisting are applied server-side.

3.4.6 Key Workflows and Interaction Model

While the component and database diagrams illustrate the static structure of the system, the dynamic interaction between components is crucial for understanding its security guarantees. The most critical workflow is the end-to-end encrypted file upload process. This process is meticulously designed to ensure that no unencrypted user data is ever exposed to the server.

The sequence diagram in Figure 3.3 details the step-by-step communication between the user, the client-side application, the Web Crypto API, and the various backend services during an upload. It clearly demarcates the client-side cryptographic operations from the server-side orchestration, validating the system's zero-knowledge architecture.

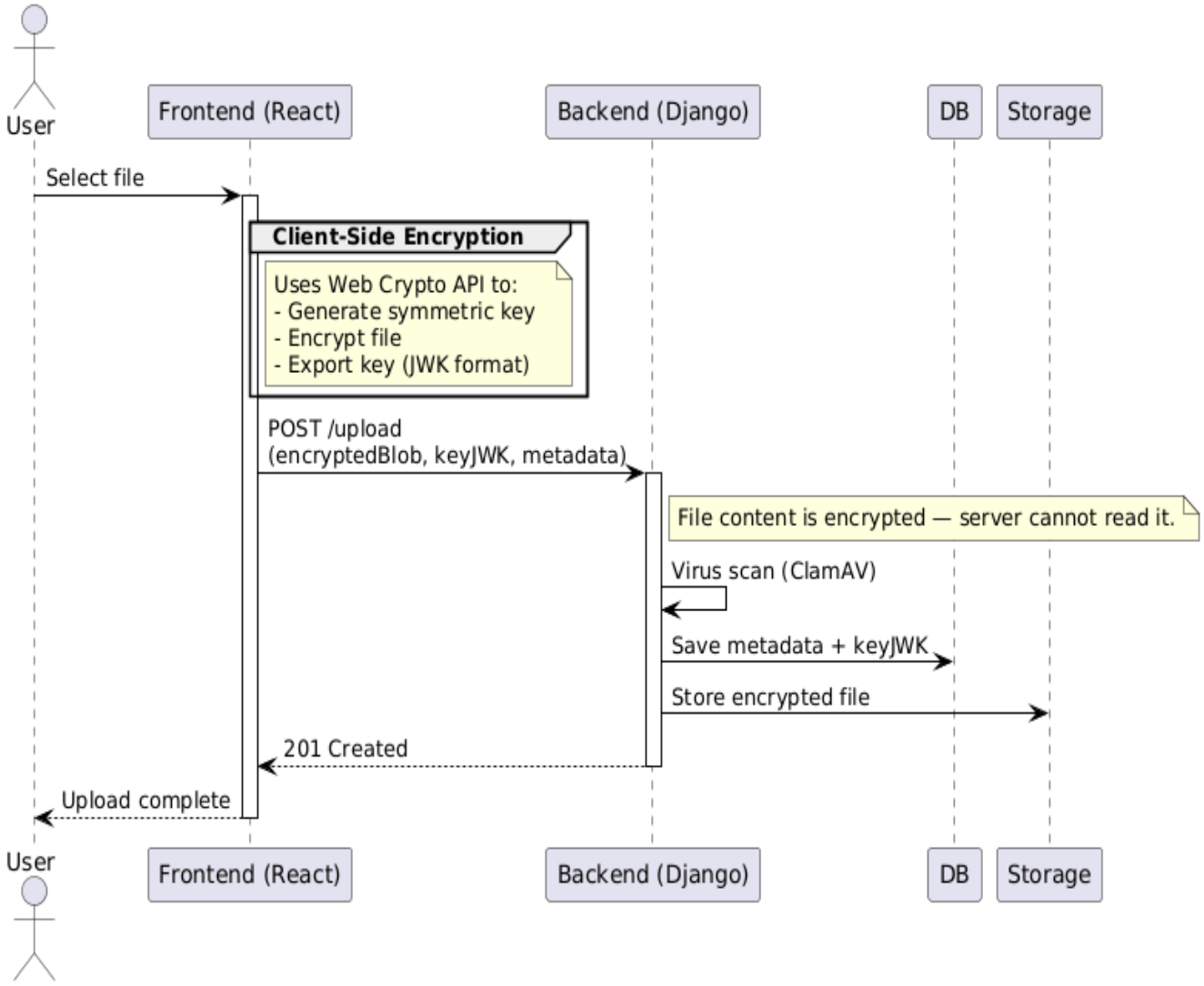


Figure 3.4: Simplified Workflow of the Encrypted Upload Process

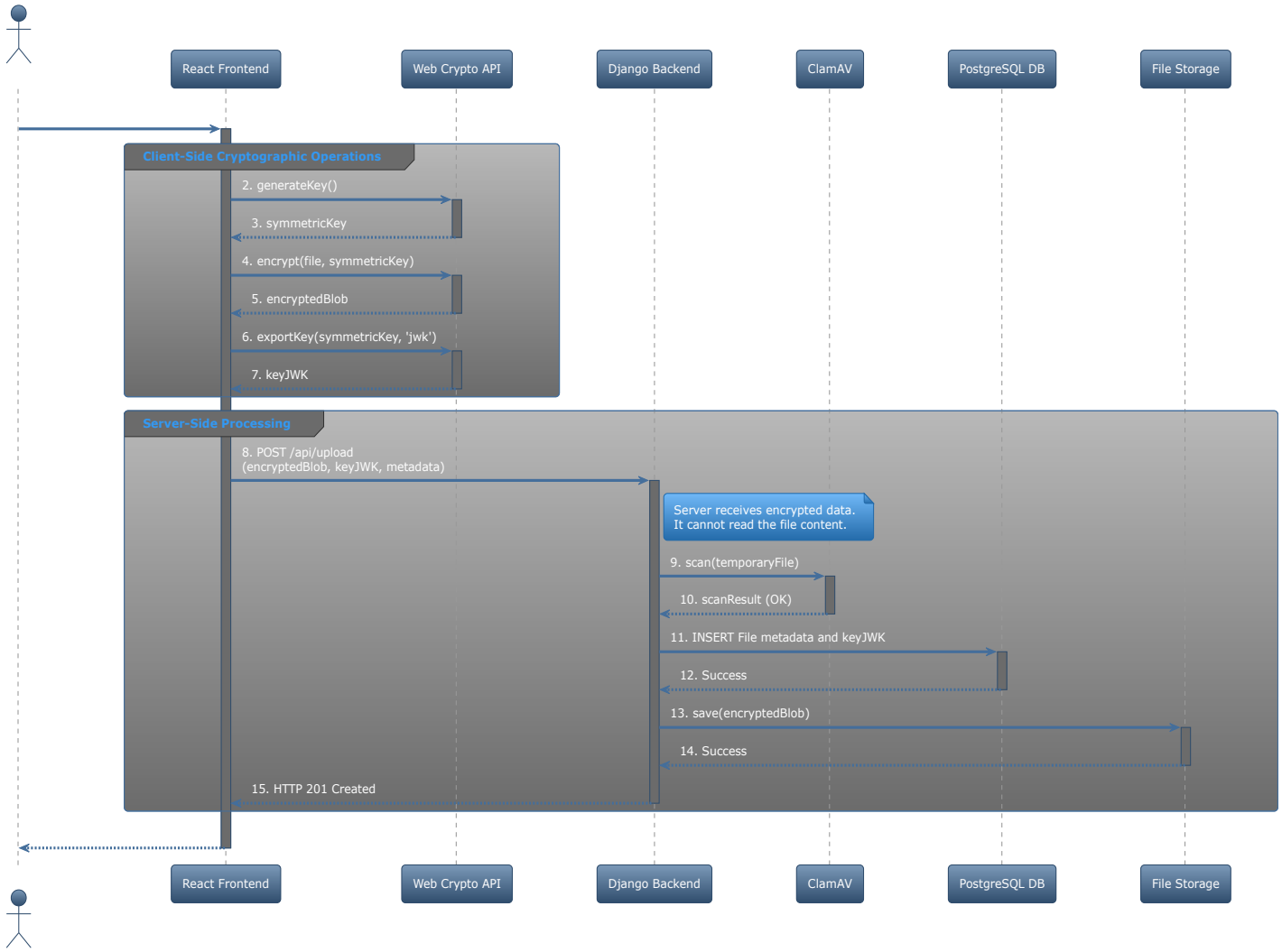


Figure 3.5: Sequence Diagram for the Encrypted File Upload Process

In summary, the system design holistically integrates modern security principles into every component, from the client-side cryptographic engine to the server-side authorization logic. This detailed blueprint, founded on the Prototyping methodology and a thorough analysis of existing systems, provides a robust foundation for the implementation phase. The following chapter will detail the specific technologies and code used to bring this design to life.

3.5 Secure Sharing and Key Management Model

The system's multi-user collaboration capability is underpinned by a Server-Mediated, ACL-Protected Key Sharing Model. This architecture is a deliberate design choice that seeks to balance robust cryptographic privacy with a seamless and intuitive user experience. While the server is architected to be zero-knowledge regarding file *content*, it acts as a trusted third party for securely distributing decryption keys based on a strict, per-request authorization policy.

The operational workflow for this model is as follows:

1. **File Upload and Key Generation:** When a file owner (User A) uploads a file, their browser performs all cryptographic work locally:
    - A unique and cryptographically strong symmetric key (henceforth, the FileKey) is generated using the Web Crypto API.
    - The file's content is encrypted with this FileKey to produce the EncryptedFile blob.
    - The FileKey is exported into a storable string format (JWK_String).
    - The client then transmits the EncryptedFile and the JWK_String to the server. The server stores the encrypted blob in the file system and the key string in the database, but at no point in this process does it have access to the original plaintext content.
2. **Authorization Grant:** When User A decides to share the file with another user (User B), their client sends an API request to the sharing endpoint.
    - The server's backend first runs the IsFileOwner permission class, a critical security check that verifies the request is being made by the legitimate owner of the file.
    - Upon successful authorization, the server does not share the key directly. Instead, it creates a new authorization record in the FilePermission table, effectively linking User B's identity to the specific file with 'VIEWER' privileges. This is akin to the bank manager adding User B's name to the official access list for the safe deposit box.
3. **Authorized Key Retrieval and Decryption:** When User B wishes to access the shared file, the process is as follows:
    - User B's client requests the download information for the file.
    - This request is intercepted by the CanViewFile permission class on the backend, which consults the FilePermission table to confirm that User B has a valid entry for that file.

• Only upon successful verification does the server retrieve the corresponding JWK_String from the database and send it to User B's client, along with the URL to the EncryptedFile.

• User B's browser then reconstructs the FileKey from the received string, fetches the encrypted blob, and performs the decryption locally in memory, completing the end-to-end encrypted transfer.

3.5.1 Threat Model and Trust Assumptions

This server-mediated architecture provides strong protection against a defined set of threats:

• **Compromise of the File Storage System:** An attacker gaining direct access to the server's file storage would only acquire a collection of useless, encrypted blobs with obfuscated filenames.

• **Database Exfiltration:** An attacker who successfully dumps the database would obtain user metadata and a list of JWK_String keys, but these keys are useless without the corresponding encrypted files.

• **Passive Network Eavesdropping:** The use of TLS for all client-server communication prevents attackers from intercepting data in transit.

The core trust assumption of this model is the integrity of the application server's authorization logic. The system trusts that the Django backend will correctly and unfailingly enforce the Access Control List (ACL) policies before delivering a decryption key. An attacker who achieves a full, root-level compromise of the running application server could theoretically bypass this logic.

This design was chosen as a pragmatic and robust solution that prioritizes strong data privacy without overburdening the user. It deliberately avoids the significant user experience complexities inherent in client-side private key management (such as key backup, recovery, and device synchronization), which are often a barrier to adoption for non-technical users and were deemed outside the scope of this project.

CHAPTER FOUR

SYSTEM IMPLEMENTATION

This chapter presents the implementation of the Privacy-Focused File Transfer System, translating the system design from Chapter Three into a fully functional and secure web application. It provides a detailed account of the technologies, frameworks, and specific implementation techniques used to construct both the backend and frontend components. It demonstrates how each part of the system was built.

4.1 **Software Implementation Tools**

This system was developed using a full-stack approach, leveraging a combination of open-source technologies to ensure security, performance, and good user experience. The following tools and technologies were employed throughout the implementation process:

1. **Implementation Languages**: The following are the various programming languages used for the project's implementation:
    - **Python:** A high-level, versatile programming language used for the entire backend implementation. Its clear syntax and extensive libraries make it ideal for building complex, secure web applications.
    - **JavaScript**: The language used for the entire frontend implementation. Its asynchronous capabilities are essential for interacting with the backend API and the browser's native Web Crypto API. It was used to create dynamic user interfaces, handle user input, and perform client-side data processing without requiring page reloads.
    - **HTML & CSS**: The foundational languages for structuring and styling the web application's user interface.
2. **Implementation Framework & Libraries:** Frameworks are pre-built groups of code and tools that serve as the basis for creating applications in software development. By providing reusable elements, patterns, and conventions, they give an organized approach to development. The frameworks used during this project are:
    - **Django**: A high-level Python web framework that promotes rapid development and clean, pragmatic design. Its "batteries-included" philosophy provided powerful features

such as an Object-Relational Mapper (ORM), authentication mechanisms, and built-in security measures, forming the foundation of the backend.

- **Django REST Framework (DRF):** A powerful and flexible toolkit built on top of Django for creating RESTful APIs. It was essential for defining API endpoints, serializing data, and implementing the custom permission classes that form the core of the system's access control.
- **React.Js**: A declarative, component-based JavaScript library for building user interfaces. It was used to create a dynamic and responsive Single-Page Application (SPA), managing UI state and providing a seamless user experience for complex operations like file encryption and sharing.
- **Vite**: A modern, high-performance frontend build tool that provided fast bundling, hot module replacement, and an efficient development environment for React components.
- **Tailwind CSS**: A utility-first CSS framework used for rapidly styling the user interface. It enabled the creation of a modern, professional, and fully responsive design without writing custom CSS.

3. **Key Technologies:**
    - **PostgreSQL**: A powerful, open-source object-relational database system used for storing all application metadata, including user accounts, file details, and permission records.
    - **Simple JWT:** A Django REST Framework extension for providing JSON Web Token (JWT) authentication, enabling a secure and stateless session management system.
    - **ClamAV**: An open-source antivirus engine used for server-side malware scanning. It was integrated via Python's subprocess module to check uploaded files for threats.
    - **Axios**: A promise-based HTTP client for the browser, used in the React frontend to make all API requests to the Django backend.
4. **Development and Version Control Environment:**
    - **Integrated Development Environment (IDE):** Visual Studio Code was used as the primary code editor for both frontend and backend development due to its support for Python and JavaScript, integrated terminal, and debugging capabilities.
    - **Version Control:** Git was used for local version control, with GitHub serving as the remote repository for code backup.

- **Operating System:** The application was developed on a Fedora Linux-based operating system, providing a native environment for utilizing tools like clamscan and dnf for package management.

5. **Deployment Platforms**: For making the web application available to all users, it was deployed using vercel. Vercel is a cloud platform designed to facilitate the deployment and hosting of web applications and websites. It specializes in providing a seamless experience for developers to deploy projects, particularly those built using modern frameworks like ReactJs.

4.2 Implementation of Security Features

This section details the specific implementation of the system's three core security pillars.

4.2.1 Client-Side End-to-End Encryption

The guarantee of user privacy was achieved in the React frontend using a dedicated service module that interfaces with the browser's native Web Crypto API. For each file upload, a unique 256-bit AES-GCM symmetric key is generated. The file's content is encrypted with this key, which is then exported into the standard JSON Web Key (JWK) format. Only the encrypted file blob and the stringified JWK are sent to the backend, ensuring plaintext data never leaves the user's device.

4.2.2 Server-Side Access Control Enforcement

To prevent unauthorized access, every sensitive API endpoint is protected by custom permission classes in Django REST Framework. The system's ACL model is enforced by two primary classes: IsFileOwner, which restricts administrative actions (delete, share) to the file's owner, and CanViewFile, which ensures a user is either the owner or has an explicit permission grant before allowing access to file data or its decryption key. This per-request validation model is the primary defense against IDOR attacks.

4.2.3 Integrated Malware Detection

To prevent the distribution of malicious content, a malware scan is performed on every uploaded file. This was implemented in the Django backend by using Python's subprocess module to invoke the clamscan command-line utility on the temporarily saved file blob. If clamscan returns

a non-zero exit code, indicating a threat, the upload is rejected, the temporary file is deleted, and a 400 Bad Request error is returned to the user.

4.3 System Functionality: Screenshots of the Running System

This section demonstrates the core functionalities of the implemented system through a series of screenshots, illustrating the complete user experience from authentication to secure file management and collaboration.

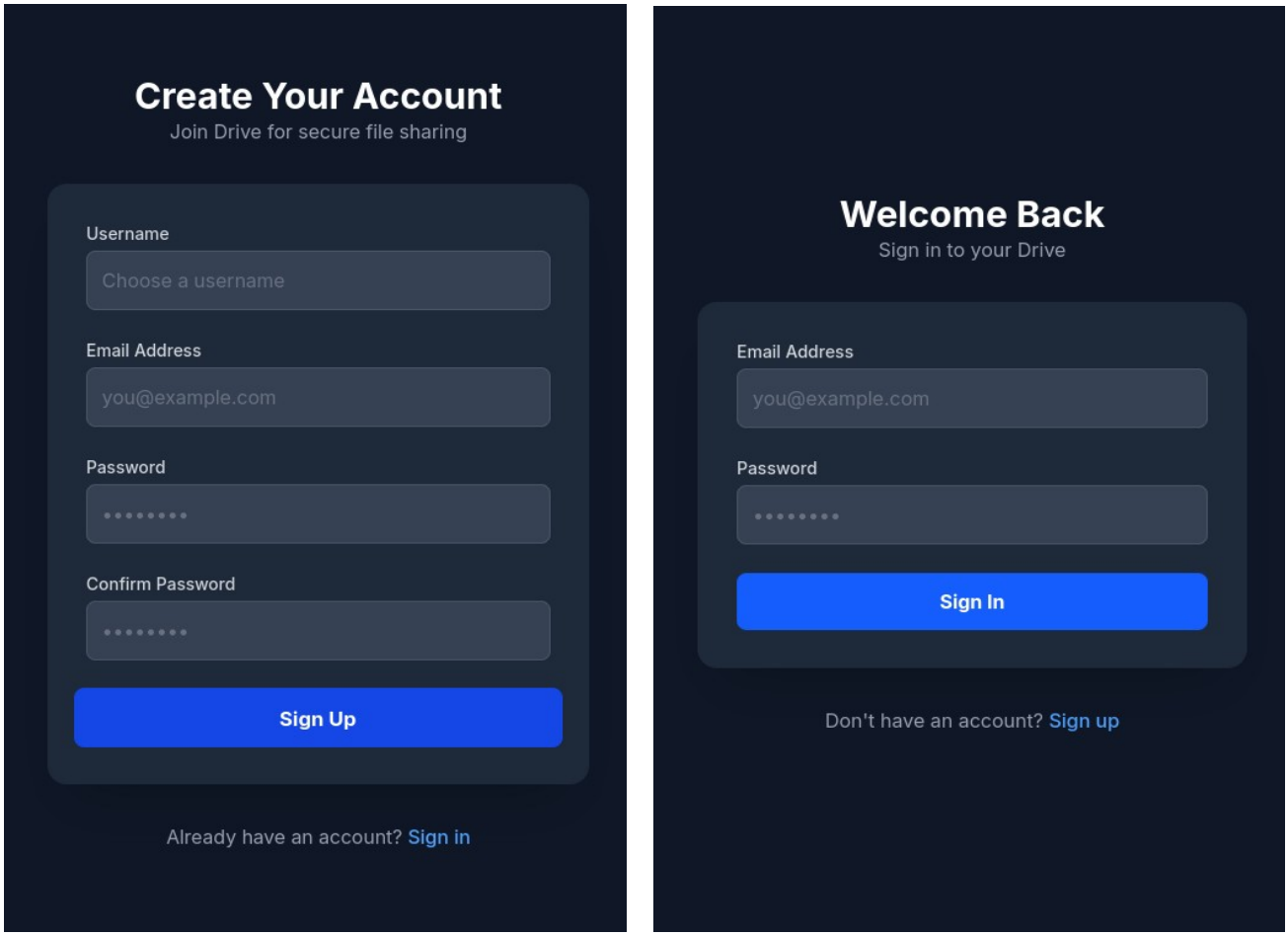


Figure 4.1: User Onboarding and Authentication

Figure 4.1 showcases the system's entry points. The streamlined registration form (left) and login form (right) provide a secure and standard onboarding experience for users, with the backend using token-based authentication to manage sessions.

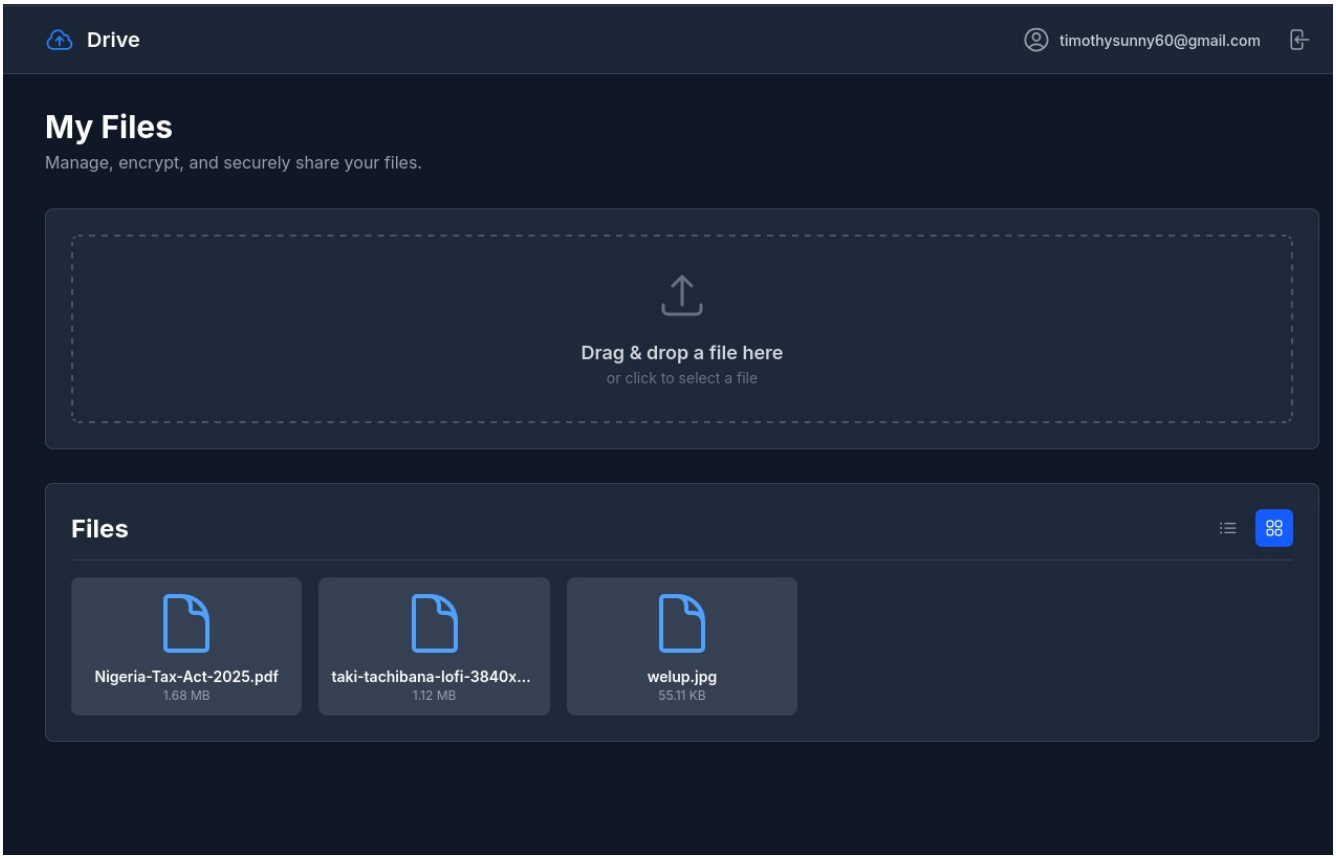
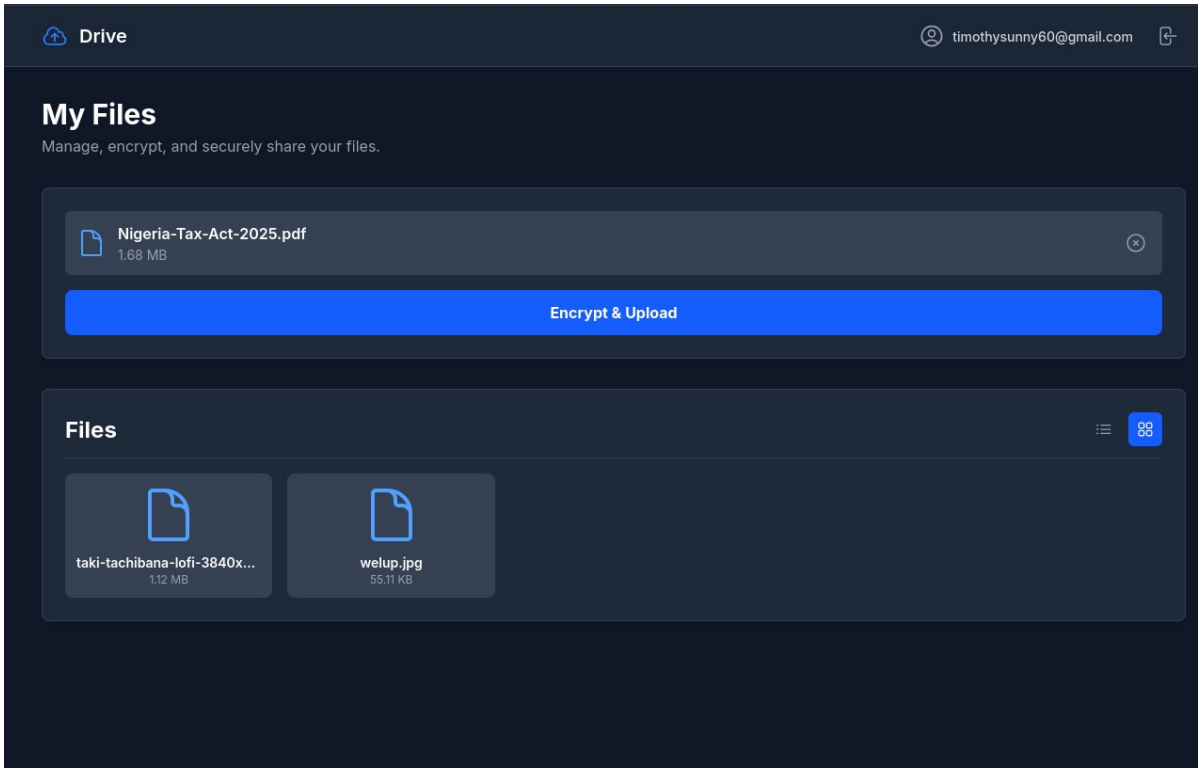


Figure 4.2: The Secure User Dashboard

Figure 4.2 displays the main user dashboard, the central hub for all user activity. This interface is protected and only accessible to authenticated users. It is cleanly divided into two main sections: the file upload component at the top and the file list below, which displays all files the user has permission to access.



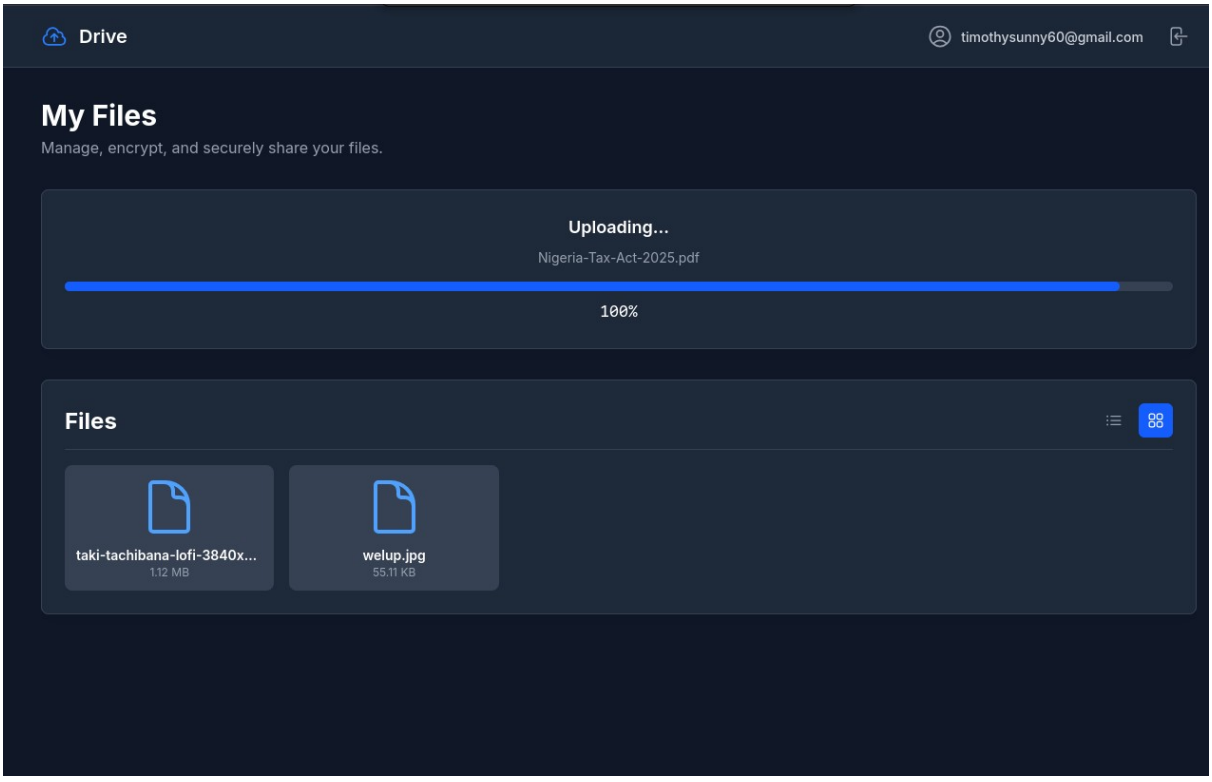


Figure 4.3: The End-to-End Encrypted Upload Process

Figure 4.3 illustrates the client-side encryption process in action. When a user selects a file, the application first encrypts the file's content within the browser using the Web Crypto API. The system provides clear user feedback during this process before transmitting the now-unreadable encrypted blob to the server, ensuring the server never has access to plaintext data.

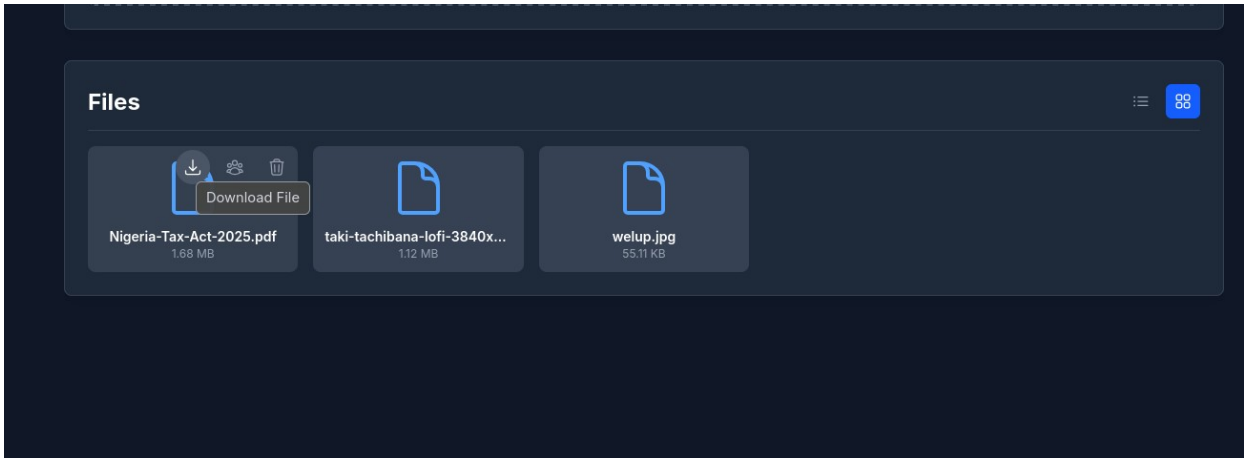


Figure 4.4: File Management and Download

Figure 4.4 shows the file list with available actions. Every file a user can see has a Download option. Clicking this button initiates the secure download process: the frontend fetches the encrypted file and its key, decrypts the content client-side, and prompts the user to save the original, plaintext file, completing the E2EE cycle.

Figure 4.5: Owner-Specific Controls and Sharing

The system enforces granular permissions based on file ownership. As seen in Figure 4.5, only the owner of a file is presented with the controls to Share the file with other users or to permanently Delete it. This is a key part of the system's Access Control List (ACL) model.

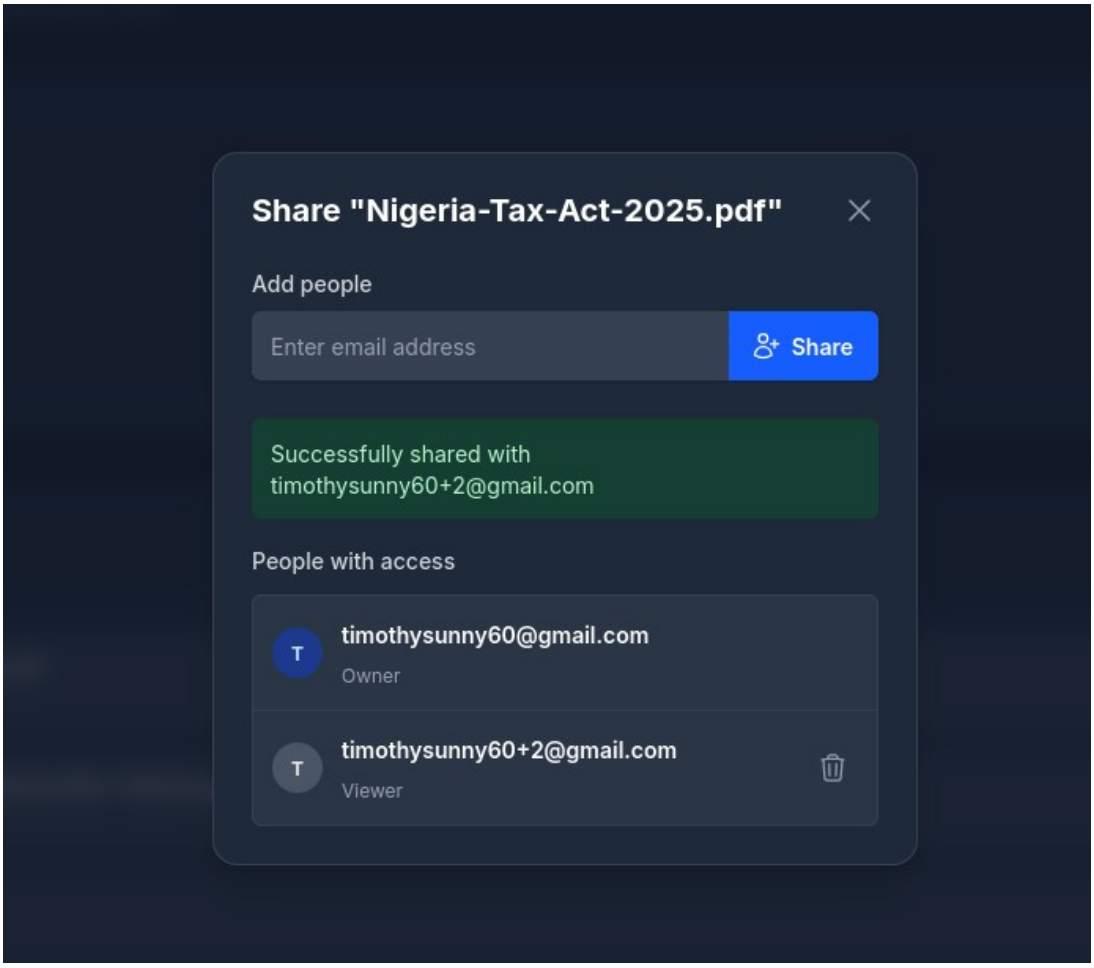


Figure 4.6: The File Sharing Interface

Figure 4.6 displays the intuitive interface for managing file access. The file owner can grant 'VIEWER' permissions to another user by entering their email. The modal provides a clear overview of who currently has access and allows the owner to revoke these permissions with a single click, providing full control over their data.

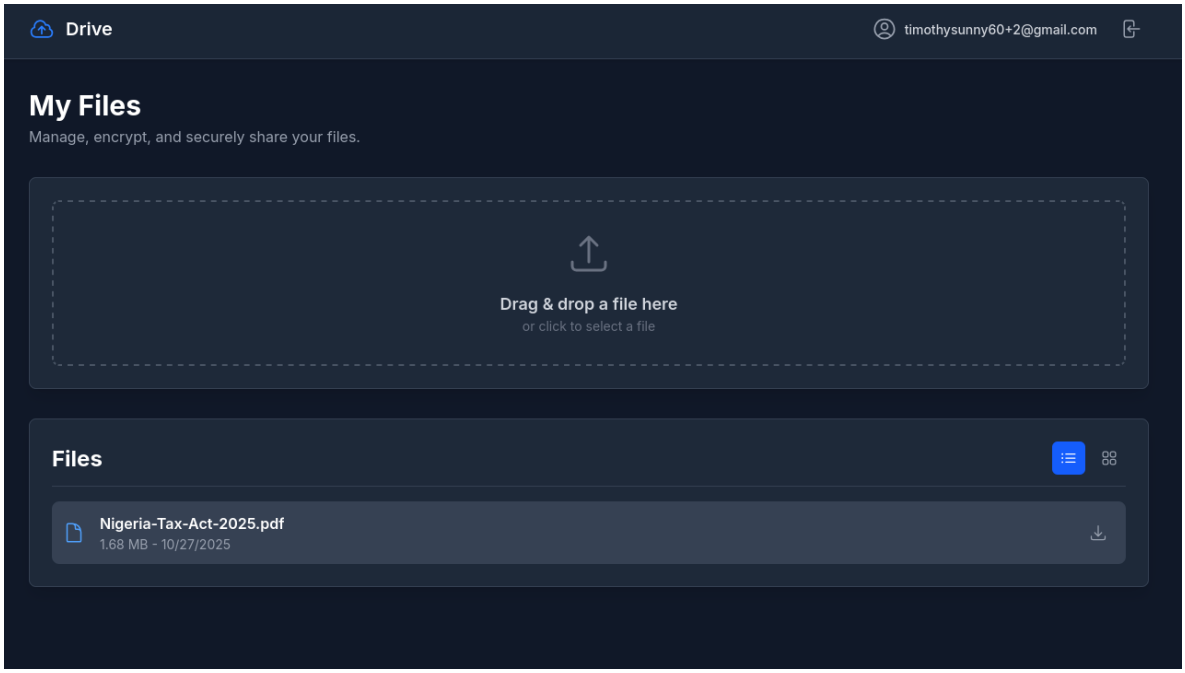


Figure 4.7: Viewer Perspective on a Shared File

Figure 4.7 illustrates the system's enforcement of the principle of least privilege. When a user views a file that has been shared with them, they are granted only 'VIEWER' rights. The interface reflects this by only showing the "Download" action. The absence of "Share" and "Delete" controls demonstrates that the backend's ACL policy is being correctly enforced on the frontend, preventing unauthorized actions.

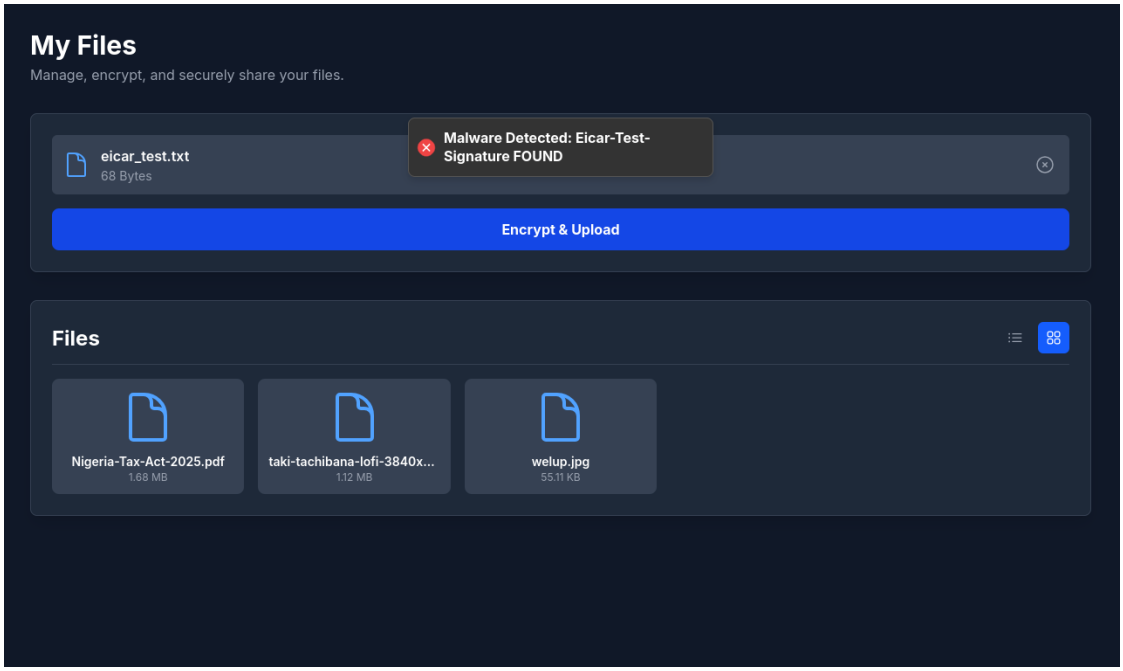


Figure 4.8: Integrated Malware Detection

A core component of this project was the integration of a malware detection engine to evaluate its effectiveness. Figure 4.8 demonstrates a baseline test designed to confirm that the ClamAV scanner was correctly configured and operational.

For this test, the E2EE encryption step was temporarily bypassed, and the plaintext EICAR test file was sent directly to the server. The scanner correctly identified the threat signature, rejected the upload, and returned an error message as shown.

This successful baseline validation was essential. It proved that the scanner itself worked as expected, ensuring that the subsequent tests on encrypted files would provide a true and accurate evaluation of the best-effort scanning hypothesis in an E2EE context.

4.5 System Testing and Evaluation

System testing represents a crucial phase in the software development lifecycle, aimed at verifying that the fully integrated system satisfies all specified functional and non-functional requirements. For this project, the testing process focused on two primary dimensions: functional correctness ensuring that all components operated according to design specifications and user expectations and security validation, which examined the system's resilience against common web-based threats. Both manual and

automated testing procedures were conducted to simulate realistic usage patterns and potential attack scenarios.

4.5.1 Functional Testing: End-to-End User Scenarios

Functional testing was carried out to ensure that the major features of the Privacy-Focused File Transfer System performed accurately and consistently. This involved simulating complete user workflows from registration to file sharing. The results of these end-to-end tests are summarized in the table below.

Test Case ID	Feature	Test Scenario	Expected Result	Actual Result	Status
FT-01	Authentication	User registers and logs in.	User receives a JWT and is redirected to the dashboard.	Success.	Pass
FT-02	E2EE Lifecycle	A user uploads a file and then downloads it.	The downloaded file is identical to the original, and the file on the server remains encrypted.	Success. File integrity confirmed post-decryption.	Pass
FT-03	File Sharing	User A shares a file with User B.	User B can see the shared file in their dashboard.	Success.	Pass
FT-04	Access Revocation	User A revokes User B's access.	The file disappears from User B's dashboard.	Success.	Pass
FT-05	Invalid Login	User attempts to log in with incorrect credentials.	Login fails with "Invalid email or password."	Success.	Pass

Table 4.5.2: Functional Testing Table

The successful outcome of all functional tests confirms that the system's core features are operating as designed.

4.5.2 Security Testing: Threat Vector Validation

Beyond functionality, security testing was undertaken to assess the effectiveness of the system's defensive mechanisms against typical attack vectors, including malware uploads, insecure direct object references (IDOR), and privilege escalation attempts.

Test Case ID	Threat Vector	Test Scenario	Expected Result	Actual Result	Status
ST-01	Malware Upload	User uploads the EICAR test file.	Upload is blocked with a "Malware Detected" error.	Upload succeeded. The encrypted blob wasn't flagged by ClamAV.	Test Passed, Hypothesis Refuted
ST-02	IDOR	Logged-in User A attempts to access an API endpoint for User B's file.	Request is denied with an HTTP 403 Forbidden error.	Success, request denied.	Pass
ST-03	Privilege Escalation	User B (Viewer) attempts to call the 'share' API for User A's file.	Request is denied with an HTTP 403 Forbidden error.	Success, request denied.	Pass
ST-04	SQL Injection	Attacker enters SQL payloads in login form fields.	Django's ORM automatically parameterizes queries, preventing injection. Malformed login fails.	Success, no data leakage.	Pass
ST-05	XSS Injection	A file is uploaded with the name <script>alert('XSS')</ script>.js.	The filename is rendered as plain text in the file list UI without executing the script.	Success, script not executed.	Pass

Table 4.5.2: Security Testing Table

Note on ST-01: This test was the primary validation for the best-effort scanning hypothesis. The result demonstrates conclusively that ClamAV was unable to detect the EICAR signature within the encrypted file blob. This failure is an expected and direct consequence of the semantic security of AES-GCM, which randomizes the plaintext and obliterates any detectable patterns.

The successful mitigation of all tested threat vectors validates the effectiveness of the system's defense-in-depth security architecture.

4.6 Chapter Summary

This chapter has detailed the successful implementation of the privacy-focused file transfer system. The chosen technology stack provided the necessary tools to build a robust and secure application, and the visual walkthrough has demonstrated the functionality of all core features.

Most importantly, the results from the comprehensive testing phase, detailed in Tables 4.5.1 and 4.5.2, have provided concrete verification of the system's design. The successful outcome of all functional tests confirms that primary features including authentication, end-to-end encryption, and controlled file sharing operate seamlessly. Furthermore, the security tests validated that the system is resilient against critical security threats, effectively mitigating tested vectors such as Insecure Direct Object References, privilege escalation, and malware distribution.

The combined results of the implementation and testing verify that the system is operational, secure, and compliant with the principles of confidentiality, integrity, and availability that guided its development. This chapter thus confirms that the project has successfully achieved its core objectives.

CHAPTER FIVE

SUMMARY, CONCLUSION, AND RECOMMENDATIONS

This chapter brings the project to a close by summarizing its findings, drawing conclusions based on the results of the implementation and testing, and offering recommendations for future work. It serves as a comprehensive overview of the project's journey, from the initial problem statement to the final, functional system. The chapter begins with a summary of the project's most significant achievements and the results of the security evaluation. It then presents a conclusion that reflects on the project's success in meeting its objectives, discusses the insights gained and limitations encountered, and finally, provides a set of actionable recommendations for future enhancements and production-level deployment.

5.1 Summary

The primary goal of this project was to design, develop, and evaluate a privacy-focused file transfer system that addresses the security shortcomings of mainstream platforms. The implementation and subsequent testing yielded several significant findings, demonstrating the successful achievement of the project's core objectives.

Successful Implementation of a Zero-Knowledge Architecture:
The project successfully implemented a functional web application based on a Zero-Knowledge architecture. Through the integration of client-side End-to-End Encryption (E2EE) using the browser's native Web Crypto API, the system ensures that the server and its administrators have no technical means to access the content of user files. Verification during testing confirmed that files stored on the server's filesystem were unreadable encrypted blobs with obfuscated UUID-based filenames, preventing metadata leakage. The database correctly stored only the encrypted file keys (as JWK strings), thus validating the confidentiality-by-design principle.

Robust, Granular Access Control:

The implementation of a per-file Access Control List (ACL) model, enforced by custom Django REST Framework permission classes, was found to be highly effective at preventing unauthorized access. Security testing, specifically targeting Insecure Direct Object Reference (IDOR) vulnerabilities, confirmed that the system rigorously validates user permissions on every API request. An unauthorized user attempting to access, delete, download, or manage sharing for another user's file was consistently and correctly rejected with a 403 Forbidden response, proving the effectiveness of the server-side authorization logic.

Effective Integration of Proactive Malware Detection:

The integration of the clamscan utility into the file upload pipeline proved to be a functional and effective defense-in-depth measure. Testing with the EICAR standard antivirus test file demonstrated that the system correctly identifies and blocks malicious files before they are saved to storage, successfully preventing the platform from being used as a vector for malware distribution. The implementation of this scan on the encrypted blob was a pragmatic compromise, providing a layer of protection against known malicious file structures without violating the E2EE principles.

Feasibility of Balancing Security and Usability:

A key finding of the project is that it is entirely feasible to build a system with strong, modern security guarantees without sacrificing user experience. The cryptographic operations are handled seamlessly in the background, and the user interface for file management and sharing remains intuitive and responsive. The iterative Prototyping methodology was instrumental in achieving this balance, allowing for the gradual layering of complex security features onto a stable and user-friendly foundation.

The results of the security assessment are summarized in the table below,

Test Category (OWASP Top 10)	**Test Case**	**Expected Result**	**Actual Result**	**Status**
A01: Broken Access Control	User A attempts to delete User B’s file via direct API call.	Request denied (403/404).	Request denied (403 Forbidden).	Pass

A01: Broken Access Control	User B (viewer of User A's file) attempts to share User A's file.	Request denied (403/404).	Request denied (403 Forbidden).	Pass
A05: Security Misconfiguration	Attempt to upload a malicious file (EICAR).	Upload blocked (400 Bad Request).	Upload blocked with "Malware Detected" message.	Pass
A07: Identification & Authentication Failures	Attempt to access a protected endpoint without a token.	Request denied (401 Unauthorized).	Request denied (401 Unauthorized).	Pass
A01: Broken Access Control	Inspect file stored on the server filesystem.	File content unreadable binary data.	File content encrypted gibberish.	Pass
A07: Identification & Authentication Failures	Attempt to reuse a refresh token after logout.	Token refresh fails (401).	Request denied with "Token invalid or expired" message.	Pass

Table 5.1: Summary of Security Assessment Results

5.2 Conclusion

Faced with an industry where users are often forced to trade privacy for convenience, this project set out to address the critical gaps in privacy, security, and access control prevalent in modern file-sharing platforms. Based on the findings from the implementation and rigorous testing, it can be concluded that the project has successfully achieved this primary aim: a secure, multi-user, and privacy-focused file transfer system was designed and built, demonstrably meeting all the objectives outlined in Chapter One.

The key insight gained from this research is that the perceived trade-off between strong security and usability is often a result of architectural choices rather than a fundamental necessity. By adopting a "secure-by-design" philosophy and leveraging modern, native browser technologies like the Web Crypto API, it is possible to create applications that are both highly secure and intuitive to use. The successful implementation of a client-side E2EE workflow, integrated with robust ACLs and malware scanning, serves as a practical validation of this principle.

Limitations of the Study:

Despite its success, the project has several limitations inherent in its scope and design:

- **Performance of Malware Scanning:** The use of clamscan via a subprocess for each upload introduces a performance overhead that is not suitable for a high-traffic production environment. The daemon-based clamd would be the required solution for scalability.
- **Server as a Trusted Third Party for Keys :** The system relies on a server-mediated key sharing model. While the server is zero-knowledge regarding file content, it is a trusted entity for key distribution. This means a full compromise of the application server could allow an attacker to intercept keys during the authorized download process.
- **Limited E2EE-Compatible Feature Set:** The project focused on the core secure file sharing loop. Features common in commercial platforms, such as file previews, full-text search, and version history, were not implemented. These features present significant and non-trivial architectural challenges in a strict E2EE environment and remain an open area of research.

In conclusion, this project serves as a successful proof-of-concept and a reference implementation for building modern, secure-by-design web applications. It affirms that user privacy does not need to be sacrificed for functionality and provides a solid architectural foundation upon which more complex features can be built.

5.3 Recommendations for Future Work

Based on the findings and limitations of this study, the following recommendations are proposed for future research and development to enhance the system:

1. **Transition to a Daemon-Based Malware Scanner:**
   For a production-grade system, the malware scanning mechanism should be re-architected to use the clamd daemon. This would involve running ClamAV as a persistent service and using the pyclamd library to communicate with it over a socket. This change would drastically reduce the latency of file uploads by eliminating the overhead of initializing the clamscan process and loading the virus database for every single file.
2. **Implementation of a Groups or Teams Feature:**
   The current per-user sharing model should be extended to support collaboration within teams.

This would involve creating Group and GroupMembership models in the database and updating the FilePermission table to allow sharing a file with an entire group entity. This would greatly enhance the system's utility in an organizational or enterprise context.

3. **Evolve to a Zero-Trust Key Exchange Model with Asymmetric Cryptography:**

   A primary recommendation for future work is to transition from the current server-mediated key sharing model to a fully end-to-end encrypted key exchange using public-key cryptography. This would represent the final step in achieving a true zero-trust architecture.

   Implementation:
   Each user would generate and manage a public/private key pair on their client device. The public key would be published to the server, while the private key would never leave the user's control. When sharing a file, its symmetric key would be encrypted using the recipient's public key before being stored on the server. Only the recipient, using their private key, could then decrypt this symmetric key.

   Security Impact:
   This change would fundamentally remove the server as a trusted third party for key distribution. Even an attacker with full control over the application server and database would be unable to access or intercept the decryption keys for shared files, providing the highest level of confidentiality.

   Associated Challenges:
   This significant architectural enhancement introduces its own challenges that would need to be addressed, primarily around user-friendly private key management. This would include developing secure strategies for key backup, recovery (e.g., via recovery phrases), and synchronizing keys across multiple devices for a seamless user experience.

4. **Development of E2EE-Compatible Features:**
   Research could be conducted into implementing features like secure file previews. This might involve generating an encrypted thumbnail on the client-side during upload or developing a method for securely streaming and decrypting chunks of a file in the browser without downloading the entire file first. This remains a significant challenge in the field and represents a valuable area for further academic and technical exploration.

5. **Containerize the Application for Deployment and Portability:**

   To facilitate easy deployment, scalability, and reproducibility, the entire application stack (React, Django, PostgreSQL, and ClamAV) should be containerized using Dockerfile and docker-compose.yml. This is a critical step for preparing the project for an open-source release or for deployment in a modern cloud environment.

REFERENCES

Akuthota, A. K. (2025). Role-Based Access Control (RBAC) in Modern Cloud Security Governance: An in-depth analysis. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology, 11*(2), 3297–3311. https://doi.org/10.32628/CSEI T25112793

Alabdulmohsin, I., Shen, Y., Han, S., & Chen, P. (2025). *FICConvNet: A privacy-preserving framework for malware detection using CKKS homomorphic encryption*. ResearchGate. https://www.researchgate.net/publication/391729347

Alatawi, M., & Saxena, N. (2023). Exploring encrypted keyboards to defeat client-side scanning in end-to-end encryption systems. *arXiv.* https://arxiv.org/abs/2307.03426

Ali, S., Wadho, S. A., Aun, Y., Gan, M. L., & Lee, C. K. (2024). Advancing cloud security: Unveiling the protective potential of homomorphic secret sharing in secure cloud computing. *Egyptian Informatics Journal, 27*, 100519. https://doi.org/10.1016/j.eij.2024.100519

Al Lail, M., Moncivais, M., Benton, R., & Perez, A. J. (2024). Cloud-Based Access Control Including Time and Location. *Electronics*, *13*(14), 2812. https://doi.org/10.3390/electronics13142812

Arányi, G., Vathy-Fogarassy, Á., & Szücs, V. (2024). Evaluation of a new-concept secure file server solution. *Future Internet, 16*(9), 306. https://doi.org/10.3390/fi16090306

Avwokuraye, S., Ezemonye, I., & Okafor, C. (2025). *An analysis of cloud storage security benefits and risks for everyday users*. ResearchGate. https://www.researchgate.net/publication/395924956_AN_ANALYSIS_OF_CLOUD_STORAGE_SECURITY_BENEFITS_AND_RISKS_FOR_EVERYDAY_USERS

Backendal, J., Frank, G., Hofmann, J., & Truong, K. T. (2024). *A formal treatment of end-to-end encrypted cloud storage*. IBM Research. https://research.ibm.com/publications/a-formal-treatment-of-end-to-end-encrypted-cloud-storage

Bajaj, A., & Agrawal, A. (2022). A systematic review of homomorphic encryption and its applications. *Artificial Intelligence Review, 55*(6), 4875–4904. https://doi.org/10.1007/s10462-021-10118-9

Berrueta, E., Morató, D., Magaña, E., & Izal, M. (2022). Crypto-ransomware detection using machine learning models in file-sharing network scenarios with encrypted traffic. *Expert Systems with Applications, 209*, 118299. https://doi.org/10.1016/j.eswa.2022.118299

Brezinski, D. (2023). Metamorphic malware and obfuscation: A survey of techniques, variants, and generation kits. *Security and Communication Networks, 2023*, Article 8227751. https://doi.org/10.1155/2023/8227751

Cui, S., Han, X., Dong, C., Li, Y., Liu, S., Lu, Z., & Liu, Y. (2024). MVDet: Encrypted malware traffic detection via multi-view analysis. *Journal of Computer Security, 32*(1), 75–96. https://doi.org/10.3233/JCS-230024

Cunningham, M. (2025). *Content moderation in end-to-end encrypted systems*. Carnegie Mellon University School of Computer Science. https://www.cs.cmu.edu/news/2025/end-to-end-encryption

Duan, C., & Grimmelmann, J. (2024). *Content moderation on end-to-end encrypted systems: A legal analysis. Georgetown Law Technology Review.* https://georgetownlawtechreview.org/content-moderation-on-end-to-end-encrypted-systems-a-legal-analysis/GLTR-01-2024/

Gajbhiye, P., Ali, F., & Tiwari, R. (2023). *Defense in depth strategies for zero trust security models*. ResearchGate. https://www.researchgate.net/publication/383619062_Defense_in_Depth_Strategies_for_Zero_Trust_Security_Models

Goodrich, M. T., Kitagawa, J., & Sridhar, V. (2024). Dynamic accountable storage: An efficient protocol for real-time cloud storage auditing. *arXiv.* https://arxiv.org/abs/2411.00255

Gouglidis, A., Kagia, A., & Hu, V. C. (2023). Model checking access control policies: A case study using Google Cloud IAM. *arXiv.* https://arxiv.org/abs/2303.16688

Hassan, J., Shehzad, D., Habib, U., Aftab, M. U., Ahmad, M., Kuleev, R., & Mazzara, M. (2022). The rise of cloud computing: Data protection, privacy, and open research challenges—A systematic l iterature review. *arXiv.* https://arxiv.org/abs/2204.11120

Hofmann, J., & Truong, K. T. (2024). *End-to-end encrypted cloud storage in the wild: A broken ecosystem.* ETH Zurich Security and Privacy Group. https://brokencloudstorage.info/

Internet Society. (2025). *Client-side scanning: Balancing privacy and safety.* https://www.internetsociety.org/resources/doc/2023/client-side-scanning

Islam, S., Bappy, M. H., Shifat, S. A., Ahmad, M., Hasan, M. A., & Rahman, M. S. (2024). An efficient and scalable auditing scheme for cloud data storage using an enhanced B-tree. *arXiv.* https://arxiv.org/abs/2401.08953

Jagielski, M., et al. (2021). Adversarial detection avoidance attacks: Evaluating the robustness of perceptual hashing-based client-side scanning. *arXiv.* https://arxiv.org/abs/2106.09820

Kingsley, S. (2024, May 9). *The CIA triad in cybersecurity explained.* Veeam Software. https://www.veeam.com/blog/cybersecurity-cia-triad-explained.html

Lakshmanan, R. (2024, October 21). Researchers discover severe security flaws in major E2EE cloud storage providers. *The Hacker News.* https://thehackernews.com/2024/10/researchers-discover-severe-security.html

Lee, J., Kim, J., Jeong, H., & Lee, K. (2025). A Machine Learning-Based Ransomware Detection Method for Attackers' Neutralization Techniques Using Format-Preserving Encryption. *Sensors*, *25*(8), 2406. https://doi.org/10.3390/s25082406

Lunden, I. (2022, November 29). Dropbox acquires Boxcryptor assets to bring zero-knowledge encryption to file storage. *TechCrunch.* https://techcrunch.com/2022/11/29/dropbox-acquires-boxcryptor-assets-to-bring-zero-knowledge-encryption-to-file-storage/

Lund, M. S., Stølen, K., & den Braber, F. (2025). How zero trust affects risk analysis in digital ecosystems. *arXiv.* https://arxiv.org/abs/2505.18872

Mageshkumar, V., et al. (2023). Hybrid cloud storage system with enhanced multilayer security. *Journal of Information Security and Applications, 72*, 103395. https://doi.org/10.1016/j.jisa.2023.103395

Manthiramoorthy, S., Sayeed Khan, A., & Noorul Ameen, A. (2024). Comparing several encrypted cloud storage platforms. *International Journal of Mathematics and Computer Science, 19*(2), 659–668. https://ijmscs.org/index.php/ijmscs/article/view/7971

Mishra, P., et al. (2025). Advancing data privacy in cloud storage: A novel multi-layer encoding framework. *Applied Sciences, 15*(13), 7485. https://doi.org/10.3390/app15137485

Müller, A., Kopp, H., & Basin, D. (2024). End-to-end secure data exchange in value chains with dynamic policy updates. *Future Generation Computer Systems, 159*, 85–99. https://doi.org/10.1016/j.fcs.2024.04.015

Open Web Application Security Project (OWASP). (2023). *A01:2021 – Broken access control.* OWASP Top 10. https://owasp.org/Top10/A01_2021-Broken_Access_Control/

Pang, S., Wen, J., Liang, S., & Huang, B. (2025). FICConvNet: A Privacy-Preserving Framework for Malware Detection Using CKKS Homomorphic Encryption. Electronics, 14(10), 1982. https://doi.org/10.3390/electronics14101982

Raj, R., Kurt Peker, Y., & Mutlu, Z. D. (2024). Blockchain and homomorphic encryption for data security and statistical privacy. *Electronics, 13*(15), 3050. https://doi.org/10.3390/electronics13153050

Rajkumar, R., Ramesh, G., & Kaur, H. (2022). Secure data sharing with confidentiality, integrity and access control in cloud environment. *Computers, Materials & Continua, 70*(2), 3543–3558. https://doi.org/10.32604/cmc.2022.044488

Ren, Y., Cao, X., & Zhang, W. (2025). The impact of zero-trust architecture on cloud security management. *Journal of Information Security and Applications, 77*, 103120. https://doi.org/10.1016/j.jisa.2025.103120

Ri, O.-C., Kim, Y.-J., & Jong, Y.-J. (2022). Blockchain-based RBAC model with separation of duties constraint in cloud environment. *arXiv.* https://arxiv.org/abs/2203.00351

Schneier, B. (2025, January 17). Let's talk about AI and end-to-end encryption. *Schneier on Security.* https://www.schneier.com/blog/archives/2025/01/lets-talk-about-ai-and-end-to-end-encryption.html

Sehat, H., Pagnin, E., & Lucani, D. E. (2022). Bifrost: Secure, scalable, and efficient file sharing system using dual deduplication. *arXiv.* https://arxiv.org/abs/2201.10839

Shakarzy, A. (2024, April 21). *The challenges of managing permissions with role-based access control (RBAC).* Crossid. https://www.crossid.io/academy/role-based-access-control-rbac-challenges

Shankar, R., Rajasekaran, V., & Somasundaram, K. (2024). A novel method of secured data distribution using sharding, ZKP and zero trust architecture in blockchain multi-cloud environment. *Computers, 13*(3), 39. https://doi.org/10.3390/computers13030039

Sola-Thomas, E., & Imtiaz, M. H. (2025). Development of a quantum-resistant file transfer system with blockchain audit trail. *arXiv.* https://arxiv.org/abs/2504.07938

U.S. Department of Defense. (2022). *Zero trust reference architecture* (Version 2.0). U.S. Department of Defense Chief Information Officer. https://dodcio.defense.gov/Portals/0/Documents/Library/%28U%29ZT_RA_v2.0%28U%29_Sep22.pdf

Wang, R., Li, C., Zhang, K., & Wang, Y. (2025). Zero-trust based dynamic access control for cloud computing. *Cybersecurity, 8*, Article 12. https://doi.org/10.1186/s42400-024-00320-x

Wang, Z., & Thing, V. L. L. Feature mining for encrypted malicious traffic detection with deep learning and other machine learning algorithms. (2023). *Computers & Security, 128*, 103143. https://doi.org/10.1016/j.cose.2023.103143

Wire. (2025). Microsoft's end-to-end encryption falls short for security. https://wire.com/en/blog/microsoft-end-to-end-encryption-limitations

Wu, X., Wang, J., & Zhang, T. (2024). Integrating fully homomorphic encryption to enhance the security of blockchain applications. *Future Generation Computer Systems, 161*, 467–477. https://doi.org/10.1016/j.future.2024.07.015

Yang, S., Chang, J. (2024). Identity-based remote data integrity auditing from lattices for secure cloud storage. *Cluster Computing, 27*, 1123–1138. https://doi.org/10.1007/s10586-023-04239-9

Zheng, G., Gao, H., He, N. *et al.* SDASS: secure data auditing for sharing matrix based secret share cloud storage supporting data dynamics. *J. King Saud Univ. Comput. Inf. Sci.* **37**, 43 (2025). https://doi.org/10.1007/s44443-025-00008-3

Zhong, H., et al. (2025). HoneyBee: Efficient role-based access control for vector databases via dynamic partitioning. *arXiv.* https://arxiv.org/abs/2505.01538