Project Approach Document

AlgosQuest 2025: Compressed & Encrypted Data Transfer Protocol

Project Name: Secure Data Transfer Protocol

Repository: https://github.com/Nandukumar-koribilli/AlgosQuest_TECOS.git

1. Team Composition (Fearless Coders)

- **Team Lead:**
  - K. Nandu Kumar - Project Architecture & Core Logic Implementation
- **Team Members:**
  - K. Koshik - Backend API Development & Database Management
  - Y. Sai Amrutha - Frontend UI/UX Design & React Implementation
  - K. Sai Sandeep - Security Integration (Encryption/Decryption Algorithms)
  - N. Lakshmi Surekha - Testing, Documentation & Quality Assurance

2. Problem Statement

- **Challenge:** Design a lightweight, highly compressed, end-to-end encrypted protocol to securely transfer large data files over constrained bandwidth.

Core Requirements:

- **Security:** Data must be encrypted before transmission to ensure privacy.
- **Efficiency:** High compression ratios to optimize bandwidth usage.
- **Scalability:** Ability to handle large files via chunking or streaming.
- **Technology:** Node.js, Database integration, Encryption, and Compression libraries.

3. Proposed Solution

Our solution is a Zero-Knowledge File Transfer System. Unlike traditional storage services where the server holds the decryption keys, our approach ensures that files are encrypted client-side before they ever touch the network. The server acts purely as a storage provider and has no ability to read the file contents.

Core Workflow:

1. **Compression (Client-Side):** Input files are compressed using Gzip/Brotli algorithms to reduce size by up to 90%, minimizing bandwidth usage.
2. **Encryption (Client-Side):** The compressed binary data is encrypted using AES-256-GCM (Military-grade standard). A unique key is generated in the browser.
3. **Transmission:** The encrypted blob is uploaded to the Node.js server. Large files are split into chunks to ensure reliability over poor connections.
4. **Retrieval:** The recipient uses a link containing the decryption key (encoded in the URL

anchor/fragment). Browsers automatically strip fragments before sending requests, ensuring the server never receives the decryption key to download, decrypt, and decompress the file locally.

4. Technical Architecture

A. Frontend (Client-Side)

- **Framework:** React 19 with Vite (High performance).
- **State Management:** React Hooks alongside the Web Streams API for efficient, non-blocking processing of large file chunks.
- **Real-time Feedback:** Socket.io for upload/download progress bars.
- **Cryptography:** Web Crypto API for in-browser encryption (keys never leave the user's device).

B. Backend (Server-Side)

- **Runtime:** Node.js with Express.
- **Storage Logic:** Stream-based architecture using fs.createWriteStream to pipe incoming data directly to disk, ensuring constant $O(1)$ memory usage regardless of file size.
- **Database:** SQLite (via better-sqlite3) for tracking transfer metadata (expiry, download counts, logs).
- **Compression:** Node.js zlib module for server-side stream handling if fallback is needed.

C. Security Stack

- **Algorithm:** AES-256-GCM (Authenticated Encryption).
- **Integrity:** GCM mode ensures files haven't been tampered with during transit.
- **Key Management:** Keys are transferred via URL Hashes (Client-to-Client), bypassing the server entirely.

5. Key Features & Innovations

1. **"Trust-No-One" Architecture:** The server stores only encrypted blobs. Even if the server is compromised, the data remains unreadable without the specific client-link.
2. **Adaptive Compression:** The system detects file types; text-heavy files (logs, code) use Brotli for maximum compression, while media files use standard Gzip.
3. **Resumable Uploads:** Large files are chunked. If the connection drops, the transfer can resume from the last successful chunk.
4. **Ephemeral Storage:** Files can be set to auto-expire after a specific time or number of downloads, handled by a background cron job on the server.

6. Implementation Roadmap

- **Phase 1: Core Setup** - Initialize React/Node.js environment and set up SQLite schema.
- **Phase 2: Logic Implementation** - Implement the client-side pipeline using CompressionStream and SubtleCrypto to process data chunks in memory without

freezing the UI.

- **Phase 3: API Integration** - Develop POST /upload and GET /download streaming endpoints.
- **Phase 4: UI/UX** - Design the dark-themed interface with drag-and-drop support.
- **Phase 5: Testing & Deployment** - Verify encryption integrity and deploy for demo.

7. Conclusion

This project demonstrates a production-ready approach to secure data transfer. By combining Brotli compression with Client-side AES encryption, we solve the twin problems of limited bandwidth and data privacy simultaneously, delivering a solution that is both fast and mathematically secure.