Cryptography - Encryption Model Research and Selection

In designing a secure and efficient encryption system for file transfer between a Next.js frontend and a FastAPI backend, I conducted extensive research and experimentation with multiple cryptographic approaches. My primary goals were to ensure data confidentiality, maintain end-to-end security, and avoid server-side exposure to plaintext content.

Initial Methods Explored

1. **Backend-side Encryption (FastAPI using xor keys)**

My first attempt involved sending files from the client to the server in plaintext and performing encryption using Python's cryptography library. While this simplified key management, it introduced a major security flaw: the backend had full access to unencrypted files and user-supplied passwords. This violated my end-to-end security requirements and exposed sensitive data to server compromise.

2. **Symmetric Encryption with Pre-shared Key**

Next, I explored encrypting files using AES on the client side, with a hardcoded or pre-shared key. While this reduced server access to unencrypted data, hardcoding keys or managing them manually across clients posed severe risks:

- Keys could be extracted from the frontend.
- All users would share the same encryption key.
- Secure key rotation and storage were unmanageable in this setup.
- This method was rejected due to poor key security and lack of user-specific encryption.

3. **RSA (Asymmetric Encryption)**

I then experimented with using RSA encryption, generating a public/private keypair and encrypting the file using the server's public key. However, RSA has practical limitations:

- It is inefficient for encrypting large files (limited to small data sizes).
- Requires careful key management and secure server-side private key storage.
- Decryption could only happen on the server, which reintroduced plaintext exposure.
- Thus, RSA was ruled out for this use case due to performance limitations and server-dependency for decryption.

Chosen Model: AES-GCM with PBKDF2 Key Derivation (Client-side)

After evaluating the limitations above, I adopted a client-side encryption model using the WebCrypto API:

- AES-GCM: Chosen for its speed and built-in integrity protection. It ensures that tampered ciphertext will fail to decrypt, preventing silent corruption.
- PBKDF2: Used to derive a strong symmetric key from a user-supplied password. It incorporates a salt and thousands of iterations to prevent brute-force and rainbow table attacks.
- IV (Initialization Vector): A fresh 12-byte IV is randomly generated for every encryption operation, guaranteeing uniqueness and resisting ciphertext pattern analysis.
- Blob Construction: The IV is prepended to the encrypted file so it can be reused for decryption.

This model allows files to be encrypted in the browser before upload, and decrypted only when downloaded with the correct password. The backend stores only encrypted data and never sees the password or original file.