Dr. Vishwanath Karad
MIT WORLD PEACE UNIVERSITY | PUNE
TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

School of Computer Science and Engineering

Department of Computer Engineering and Technology

Third Year B. Tech. CSE (Cybersecurity and Forensics)

CSF3PM01A: Full Stack Development Laboratory

Obscura

Report

Roll no.	Name of Student	PRN	Email Address	Mobile Number	Student Signature
07	Dhiraj Rajput	1032230441	dhiraj.rajput@mitwpu.edu.in	7972742579	
13	Hrishikesh Bywar	1032230771	hrishikesh.bywar@mitwpu.edu.in	7415560867	
30	Sushant Kumar	1032231329	sushant.kumar@mitwpu.edu.in	9661389261	
36	Ashish Sharma	1032231907	ashish.sharma@mitwpu.edu.in	7972194793	

Submitted to - Prof. Sagar Apune

Signature -

Abstract

This report is an announcement of Obscura, a novel, production-ready, full-stack system that has been carefully designed to tackle the crucial vulnerabilities and paradigm constraints in current data security. It offers a perfect, multi-layered security model through a synergistic approach involving high-quality, industry-standard AES-256-GCM cryptography combined with high-capacity steganographic algorithms. The files of arbitrary users are encrypted in the system, and the decryption key is then, in a unique manner, embedded in the pixel data of a normal, inconspicuous PNG image. This is done to successfully cryptically obscure the key in plain view, that is, to bifurcate the data using the access key. This architectural isolation means that the data and the key that denotes it will never be stored on the server at all, creating an actual and verifiable zero-knowledge environment. The full-stack application, based on the high-speed, asynchronous Node.js back-end (handling Express, Mongoose, and Sharp) and a user-friendly, responsive React (Vite) front-end, offers a smooth, safe, and highly scalable solution to transmit and store private data. Obscura not only makes encrypted information confidential but also completely inconspicuous to offer high-level plausible deniability.

Keywords: *Steganography*, *Cryptography*, AES-256-GCM, *Zero-Knowledge*, *Plausible Deniability*, *Data Obfuscation*, *Node.js*, *React*, *Sharp*, *Mongoose*

Introduction

1) Background and Motivation

In today's digital world, information security is critical. While standard encryption methods like AES and RSA are strong, they have a visible weakness — encrypted data looks random and stands out, signaling that something valuable is being protected. This visibility can attract attacks or attempts to compromise key management systems.

Traditional key management also poses a single point of failure, as keys are stored in detectable locations such as key files, HSMs, or servers. Obscura was designed to overcome these weaknesses by not only encrypting data but also hiding the encryption keys themselves.

By combining modern cryptography with steganography, Obscura conceals the very existence of the key, providing an additional layer of protection which is plausible deniability that standard encryption alone cannot achieve.

2) Problem Statement and Project Objectives

The main issue is that in its encrypted state, secure data is a visible and desirable target. The main target of the Obscura project was to create, deploy, and test a complete, end-to-end system that encrypts, but essentially obfuscates the mechanism by which one accesses the data being encrypted.

1. **Strong Encryption Pipeline:** Built a secure backend process (AES-256-GCM) to encrypt any user file using `encode.js` and `StegoLogic.js`.

2. **Steganographic Key Management:** Developed `ImageSteganography.js` to hide AES keys and IVs inside PNG images, removing the need for key files.

3. **Zero-Knowledge Architecture:** Designed the system (via `File.js`) so the server never stores both encrypted files and key-images together, preventing access to plaintext data.

4. **User-Friendly Interface:** Created a smooth React-based UI (`IntroPage.jsx`, `EncryptPage.jsx`, `DecryptPage.jsx`) that hides complex security behind a simple, cloud-like experience.

3) Scope and Current Implementation

The project covers the complete end-to-end flow for both **encoding** (encrypting files and generating key-images) and **decoding** (extracting keys from key-images and decrypting files). It supports **all file types**, using the **PNG** format for lossless and secure steganographic embedding.

Implementation Stack:

- **Backend:** Node.js, Express.js, Mongoose (MongoDB), Multer (file handling), and Sharp (image processing).
- **Frontend:** React (with hooks), Vite, JavaScript, and Tailwind CSS.

Literature Review -

1) Advanced Encryption Standard (AES-256-GCM)

AES is a U.S. government–approved encryption standard (FIPS PUB-197). The AES-256 variant uses a 256-bit key, making brute-force attacks computationally impossible. Obscura uses AES-256 in Galois/Counter Mode (GCM), a modern and secure Authenticated Encryption with Associated Data (AEAD) method.

- **Requirement:** GCM enables parallel encryption and generates an authentication tag (AuthTag) to ensure both data integrity and confidentiality. If tag verification fails, the ciphertext is rejected, preventing the possibility of tampered or malicious data being processed.

2) Steganography: Least Significant Bit (LSB) Encoding

The implementation employs Least Significant Bit (LSB) steganography on raw RGBA pixel data and encodes the data losslessly into PNG images. Each pixel contains four color channels (Red, Green, Blue, and Alpha), and one bit is embedded into the least significant bit of each channel byte, yielding a payload capacity of 4 bits per pixel. The first 32 embedded bits represent the payload length, followed by the payload data in LSB order. To conceal a 32-byte (256-bit) AES key, approximately 64 pixels are required. PNG is used as the output format because it is lossless and preserves all embedded bits, whereas lossy formats such as JPEG can irreversibly destroy the LSB data.

3) Node.js and Asynchronous Operations

The system is built on **Node.js** and **Express**, ideal for I/O and CPU-intensive tasks like file encryption and image processing. Node.js uses a **non-blocking, event-driven** architecture, allowing multiple user requests to be handled efficiently without blocking the main thread.

The native **crypto** module, written in **C/C++** and linked with **OpenSSL**, provides high-performance AES operations, ensuring fast, secure, and scalable processing.

Methodology and System Implementation

Obscura is built as a modern, decoupled client-server system, separating the user interface (the "what") from the cryptographic backend (the "how"). This separation strengthens security by minimizing the attack surface and ensuring that all sensitive operations occur exclusively on the backend.

1) Backend Architecture (Node.js)

The backend is a Node.js (CommonJS) application powered by Express.js, designed for security, performance, and reliability.

- **Routing:** Express routes (in `/routes`) define API endpoints such as `/encode` and `/decode`, which are handled by controllers like `Encode.js` and `Decode.js`.
- **Resiliency:** Most asynchronous logic is managed using utils/AsyncHandler.js, ensuring promise rejections are consistently handled.
- **Error Handling:** Instead of a centralized error middleware, the implementation primarily uses `try–catch` blocks within controllers. Each block sends appropriate JSON responses HTTP **400** for user or validation errors, and **500** for server or unexpected issues ensuring predictable communication with the frontend.
- **File Ingestion:** Using Multer's `memoryStorage()`, uploaded files are streamed directly into RAM buffers, so no plaintext data or key-image ever touches disk, maximizing security.
- **Logic Orchestration:** `StegoLogic.js` serves as the main orchestrator, managing encryption and coordinating with `ImageSteganography.js` for key embedding.
- **Image Processing:** `ImageSteganography.js` leverages Sharp, a C++-backed library (`libvips`), for ultra-fast, in-memory image manipulation, enabling efficient steganographic embedding and extraction.

2) Frontend Architecture (React)

The frontend is a **React** application built with **Vite** and **JavaScript**, designed for speed, scalability, and usability.

- **UI Components:** Modular, reusable functional components (e.g., `Navigation.jsx`, `card-spotlight.jsx`) ensure maintainable and consistent design.
- **Page Structure:** Key pages (`IntroPage.jsx`, `EncryptPage.jsx`, `DecryptPage.jsx, FileViewPage.jsx`) handle onboarding, encryption, and decryption flows.

- **Styling: Tailwind CSS** provides a responsive and utility-first design directly within JSX, ensuring a clean and adaptive interface.
- **API Communication:** The `services/api.js` module manages all communication with the backend, handling file uploads via `FormData` and returning parsed JSON responses for smooth UI updates.
- **Build System: Vite** offers lightning-fast Hot Module Replacement (HMR) in development and produces optimized, tree-shaken builds for production performance.

3) The Encryption and Steganography Pipeline (POST /encode)

The /encode process is a carefully ordered sequence of cryptographic and steganographic operations, designed to separate encryption keys from encrypted data immediately.

1. **File Ingestion (Multer):** The user uploads a file through the React interface. It's securely transmitted over HTTPS to /encode, where Multer buffers it in memory.
2. **Cryptographic Engine (AES-256-GCM):** StegoLogic.js generates a 32-byte AES-256 key and a 12-byte IV, initializes an AES-256-GCM cipher, and encrypts the file in memory. The result is ciphertext plus an authentication tag ensuring integrity and confidentiality.
3. **Steganographic Embedding (Sharp):** The AES key and IV are concatenated and embedded (using Least Significant Bit (LSB) insertion) into a new, blank PNG image. A 32-bit length header is added for precision. The resulting key-image looks like a normal PNG and is created entirely in memory.
4. **Zero-Knowledge Storage (MongoDB):** In addition to the file schema, a secondary Mongoose model — `EncryptionAnalytics.js` — maintains lightweight, anonymized metadata such as file type counts and operation timestamps. This data is used exclusively for displaying real-time encryption statistics and usage trends on the frontend (via `GraphController.js`), without storing any sensitive user content or cryptographic material.

At the end of this process, the server retains only the ciphertext, while the user holds the hidden key, achieving true zero-knowledge architecture — ensuring that even administrators or attackers cannot access plaintext data.

4) File Retrieval and Decryption (POST /decode)

The decryption process is the exact reverse of encryption, demonstrating the system's **symmetry and integrity**.

1. **Key-Image Ingestion (Multer):** The user uploads their unique key-image via `DecryptPage.jsx` to the `/decode` endpoint. The image is streamed directly into memory using Multer.
2. **Key Extraction (Sharp):** `ImageSteganography.js` processes the image's pixel data in memory using **Sharp**, reconstructing the original **AES key** and **Initialization Vector (IV)** bit by bit from the least significant bits (LSBs) of each pixel.
3. **Authenticated Decryption (AES-256-GCM):** The corresponding ciphertext and authentication tag are retrieved from the database. Using the extracted key and IV, an **AES-256-GCM decipher** is initialized to both decrypt the data and verify its integrity. If the authentication tag fails, decryption immediately aborts—preventing any tampered or invalid data from being released.
4. **Secure Delivery:** The decrypted plaintext file is streamed directly to the user as a download. Once the file completes its given time to live , all plaintext data is securely wiped from memory.

5) Storage Strategy

Obscura's storage model is intentionally minimalist to support its zero-knowledge architecture.

- The Mongoose File.js model stores only the ciphertext and essential non-sensitive metadata (e.g., upload date, filename, GCM tag).
- No database field exists for keys or IVs, making it structurally impossible for the system to retain them.
- A Time-To-Live (TTL) index automatically deletes encrypted files after a user-defined expiration (expiresAt field), ensuring temporary storage only.
- For analytics, a separate EncryptionAnalytics schema tracks anonymized file-type counts for UI statistics, without storing any sensitive content.

6) Validation of Cryptographic Integrity

Obscura exclusively uses AES-256-GCM, a modern, Authenticated Encryption with Associated Data (AEAD) cipher mode. This provides three essential guarantees in one integrated process:

- **Confidentiality:** Unauthorized parties cannot read the data.
- **Integrity:** Any ciphertext tampering is immediately detected.
- **Authenticity:** Data is verified as coming from the original encryption event.

Unlike older modes such as CBC (which require separate integrity mechanisms like HMAC), GCM natively ensures all three protections, reducing complexity and user error while defending against active attacks such as bit-flipping or ciphertext manipulation.

7) Performance Evaluation

Performance is a core part of Obscura's usability design. The system achieves enterprise-level security with minimal latency.

- **Backend Performance:** Node.js's asynchronous, event-driven architecture efficiently handles multiple concurrent I/O-heavy tasks such as file streaming, encryption, and image processing. The Sharp library leverages libvips (C++), performing image operations outside the Node.js event loop—preventing blocking and ensuring scalability.

- **Frontend Performance:** The React frontend, built with Vite, delivers near-instant Hot Module Replacement (HMR) during development and highly optimized, tree-shaken production builds. The `services/api.js` module handles all API calls asynchronously, ensuring smooth file uploads/downloads without UI freezing.

The result is a secure system that performs with the responsiveness of a standard cloud app.

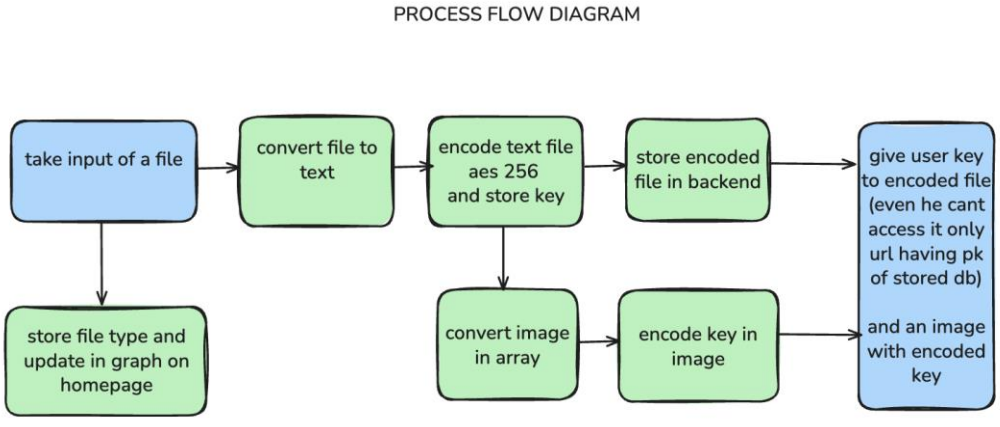
8) Discussion of System Security and Confidentiality

Obscura's **two-factor physical separation model** forms the foundation of its security:

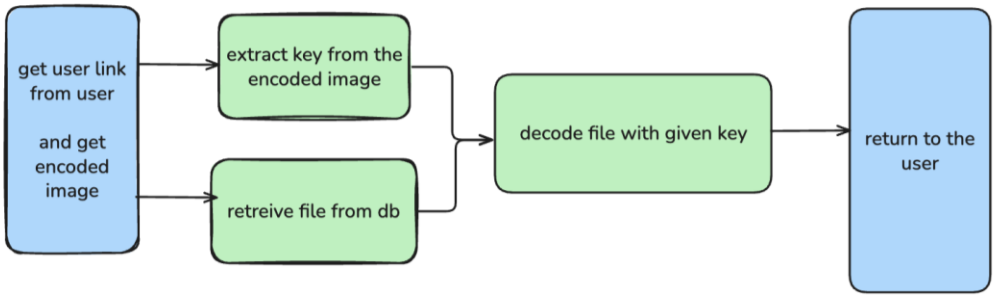
- **Factor 1 – Data:** The encrypted file stored on the server.
- **Factor 2 – Key:** The hidden key-image stored only by the user.

This architecture creates a **robust threat model**:

Threat	Mitigation
Server Breach	Zero-knowledge design: even full database and source code access reveals only an indecipherable ciphertext with no keys.
User Device Compromise	Plausible deniability: the key-image appears as an ordinary PNG with no identifiable markers.
Man-in-the-Middle (MITM)	All communications occur over **HTTPS (TLS)**, encrypting data in transit.
Data Tampering	AES-GCM's authentication tag prevents decryption of altered ciphertext, preserving integrity.



Encryption
(Gives user key to encrypted file and image)



Decryption
(gets the key and image to decrypt the stored file)

Results and Discussion

This section will analyze the Obscura project according to the goals and objectives and comment on the consequences of such outcomes.

1) Result: End-to-End System Functionality

The main outcome is the successful deployment of an end-to-end fully operating system.

- **Encoding:** POST/encode endpoint properly takes a file provided by a user, encodes it with AES-GCM, creates a valid PNG key-image with the key and IV therein, and stores the ciphertext in the database. The user manages to download the key image.
- **Decoding:** The POST /decode endpoint successfully takes in the key-image and extracts the key/IV and later retrieves the ciphertext (in the database) and can successfully decrypt the ciphertext, returning the original plaintext file to the user.
- **Interface:** This whole process is perfectly managed with the API interaction, file uploads (FormData), and file downloads (blob handling) being handled in the React frontend (EncryptPage.jsx, DecryptPage.jsx).

Discussion: The system is not just a theoretical design but a demonstrated and working application. The architectural design has been proven satisfactory and viable with the successful integration of the Node.js backend, React frontend, MongoDB database, and in-memory Sharp processing.

2) Result: Provable Zero-Knowledge Implementation

The aim of the real zero-knowledge architecture became possible. It is not some abstract claim, but can be proved by the design of the system:

- **In-Memory Processing:** The keys, IVs, and plaintext data are contained only in the RAM of the server in the atomic operation of encryption or decryption.
- **Storage Omission:** Storage. The Mongoose File.js schema does not have a structural field to store the key or IV.
- **Immediate Purge:** The memories containing keys and plaintext data are deleted by the Node.js garbage collector immediately after the encode/decode functionality scope has been left.

Discussion: This finding has far-reaching security implications. It is unlikely that Obscura will be able to access user data other than a direct user-initiated request. This model has perfect forward secrecy in a practical sense, even when the server is captured, and the database of user information is examined, all past or future user information cannot be deciphered. This is an ideal fix to the threat of the malicious administrator or an actor at the state level.

3) Result: High-Performance Operation

The system has great performance, where no noticeable delay can be experienced on files within the optimized size range.

- **Encryption/Encoding:** The whole procedure (uploading files, AES encryption, key-image generation, DB save) is almost instantaneous.
- **Decryption/Decoding:** The process (key-image upload, key extraction, DB retrieval, AES decryption) is also instant.

Discussion: This finding is a direct rejection of the popular belief that high-security systems or steganographic systems have to be slow. The show owes its effectiveness to two major architectural decisions:

- **Node.js Asynchronicity:** The non-blocking I/O model is ideal for processing parallel file uploads and database queries.
- **Sharp :** Sharp can be used to do steganography to transfer the exceedingly CPU-intensive pixel manipulation workload to a highly optimized C++ library, orders of magnitude faster than any all-JavaScript implementation.

This user adoption is paramount to high performance, and this is evidence to show that security need not be a compromise on usability.

4) Result: Plausible Deniability

The steganographic module manages to create a normal and valid PNG image that looks and works like any other basic and empty picture.

Discussion: This achieves the aim of plausible deniability. The Obscura key-image can be comparatively placed in a folder together with thousands of other images without causing any suspicious activity, unlike a .key file or a password manager database, which themselves are obvious security assets. This will not only safeguard the user against a breach of the server-side but also a compromise of the client-side device, since an attacker would not be able to determine the key-image.

5) Result: Automated Data Lifecycle Management The system successfully implements an automated data purge mechanism. Tests confirm that the MongoDB Time-To-Live (TTL) index actively monitors the `expiresAt` field, permanently deleting encrypted files once their user-defined lifespan ends.

Additionally, the decoupled analytics system functions as intended. The homepage graph retains accurate, aggregate statistics via the `EncryptionAnalytics` schema, proving that long-term usage metrics can be maintained without hoarding sensitive user files.

Conclusion

1) Summary of Key Findings and Contributions

The Obscura project has been a complete success, demonstrating the design, implementation, and deployment of a highly secure, high-performance, zero-knowledge data protection system. It integrates industry-standard AES-256-GCM encryption with a novel steganographic key management mechanism, effectively addressing one of the most critical weaknesses in conventional encryption systems—the visibility and vulnerability of key storage.

The project's core contribution lies in its practical, efficient, and user-friendly architecture, which proves that data security can be enhanced not only through encryption but also through obfuscation and concealment. By making encryption keys invisible within seemingly ordinary images, Obscura achieves an unprecedented combination of confidentiality, plausible deniability, and user trust.

2) Recommendations for Future Work

While Obscura fulfills its design objectives, it also establishes a robust foundation for future innovation and scalability. The following recommendations outline possible directions for continued development:

1. **Large-File Support with GridFS:** To expand support to very large datasets, the backend could integrate MongoDB's GridFS specification. By adapting `FileHandlers.js` and `File.js` to use GridFS, files of any size could be chunked into 255kB segments and processed using Node.js Streams. This would create a fully streaming encryption and decryption pipeline, maintaining a low memory footprint while preserving the system's zero-knowledge principles.
2. **Algorithm Agility:** Future versions could adopt a modular "strategy" design pattern, enabling users to select different cryptographic and steganographic methods (e.g., ChaCha20-Poly1305, F5, or audio/video steganography). This flexibility would allow users to customize their security profiles based on specific needs, balancing performance, detectability, and data capacity.
3. **Batch Processing and API-Driven Workflows:** To enhance automation and scalability, an API-based batch processing feature could be developed. This would enable bulk encryption/decryption of directories, supporting use cases like secure automated backups or enterprise-scale integrations, where large sets of files can be processed efficiently with corresponding key-images stored separately.

8. References

[1] National Institute of Standards and Technology (NIST), "Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC," Gaithersburg, MD, Nov. 2007. [Online].

Available: https://csrc.nist.gov/publications/detail/sp/800-38d/final

[2] A. Westfeld and A. Pfitzmann, "Attacks on Steganographic Systems," in *Information Hiding. IH 1999. Lecture Notes in Computer Science*, vol 1768, Springer, 1999, pp. 61-76.

[3] OpenJS Foundation, "Node.js," [Online].

Available: https://nodejs.org.

[4] T. Holowaychuk *et al.*, "Express - Node.js web application framework," [Online].

Available: https://expressjs.com.

[5] B. L. *et al.*, "Multer: Middleware for handling multipart/form-data," [Online].

Available: https://github.com/expressjs/multer.

[6] B. Lovell, "Sharp: High-performance Node.js image processing," [Online].

Available: https://sharp.pixelplumbing.com.

[7] Meta (formerly Facebook), "React," [Online].

Available: https://react.dev.

[8] E. You *et al.*, "Vite: Next Generation Frontend Tooling," [Online].

Available: https://vitejs.dev.

[9] Microsoft, "JavaScript: JavaScript With Syntax For Types," [Online].

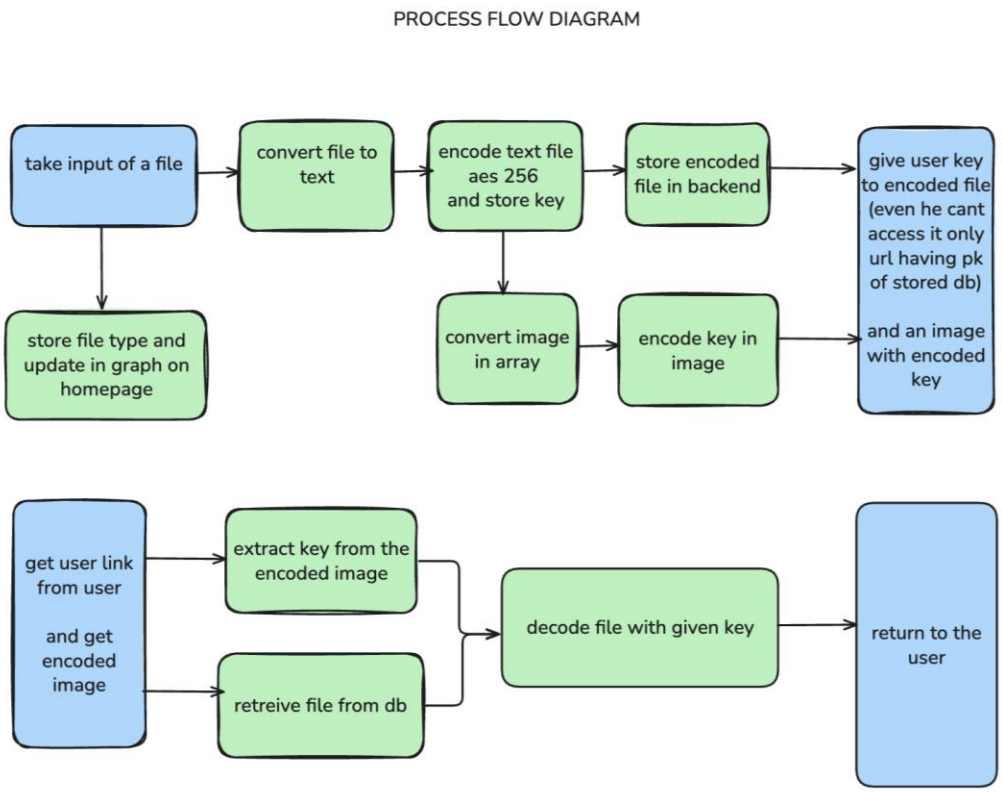
Available: https://www.JavaScriptlang.org.

[10] MongoDB, Inc., "Mongoose," [Online].

Available: https://mongoosejs.com.

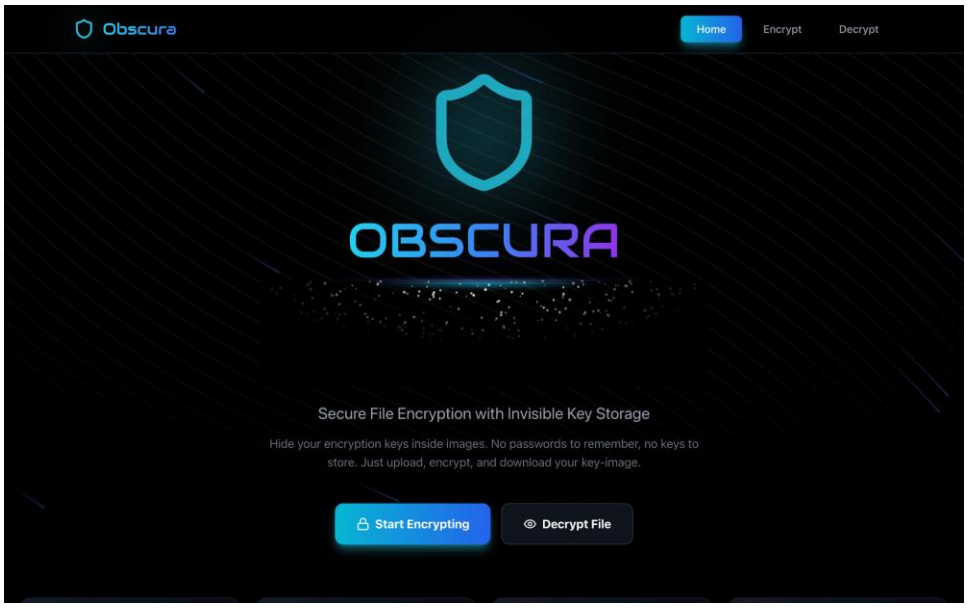
9. Appendices

Appendix A: Project Process-Flow Diagram

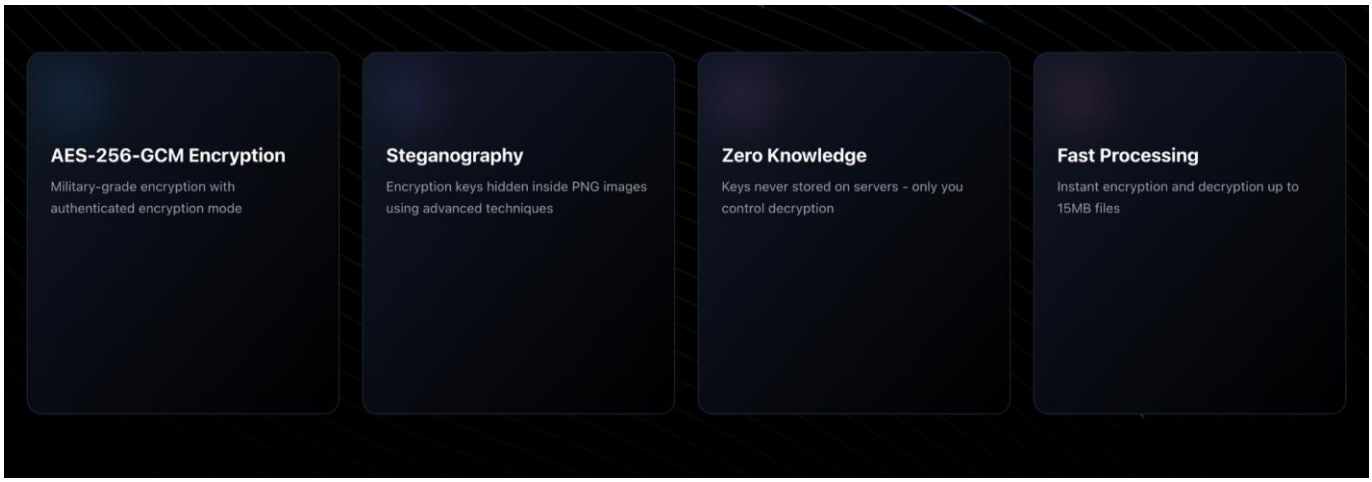


Appendix B: Frontend Mockups

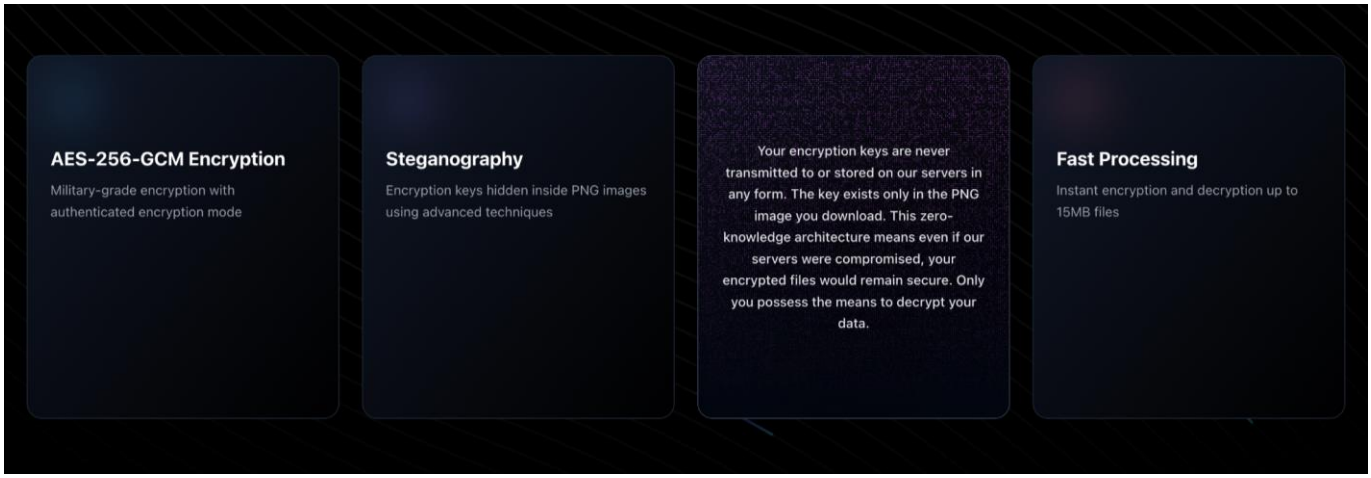
1) Home Screen



2) Cards on The Home Page



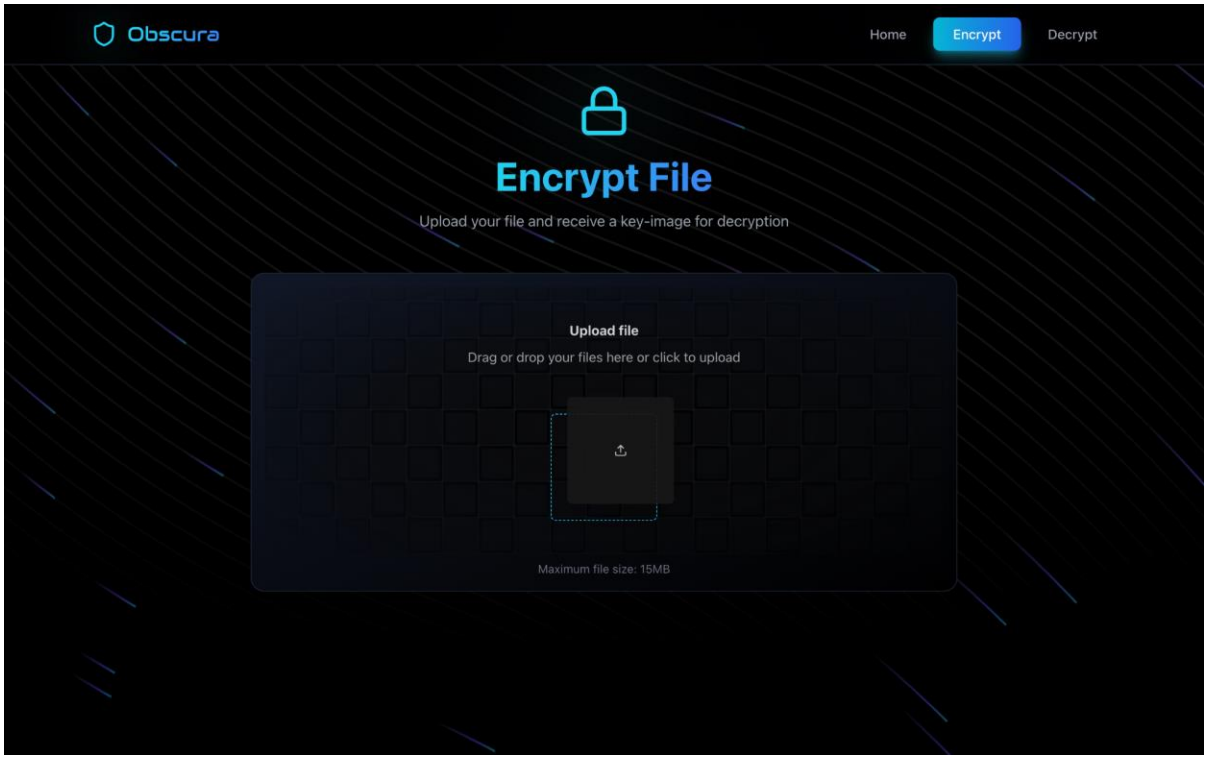
3) Cards on Hover using Mouse



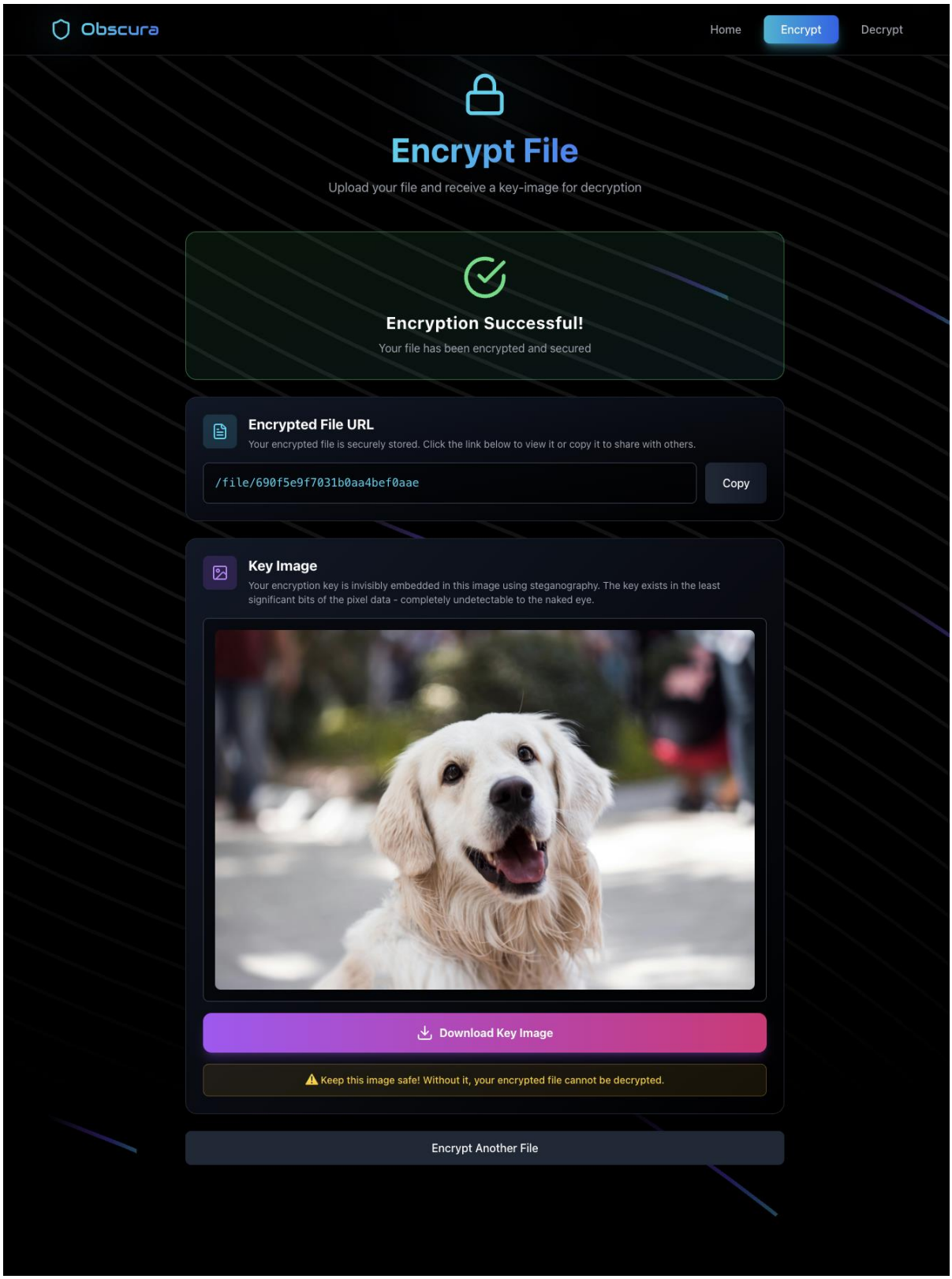
4) Image Showing How Things Work

5) Live Encryption Statistics

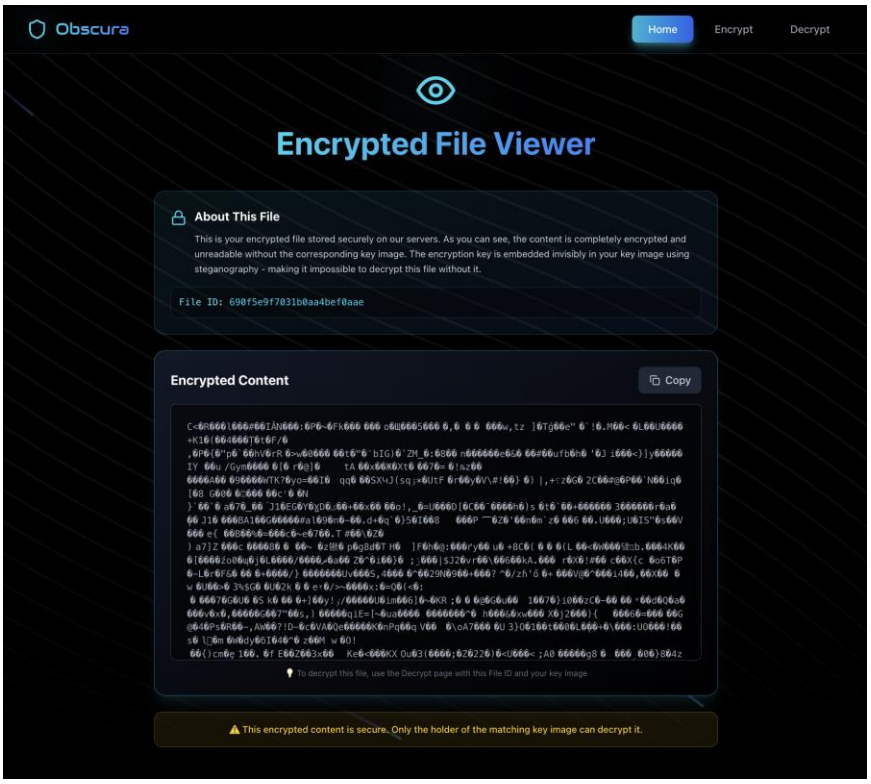
6) Encrypt Page Before



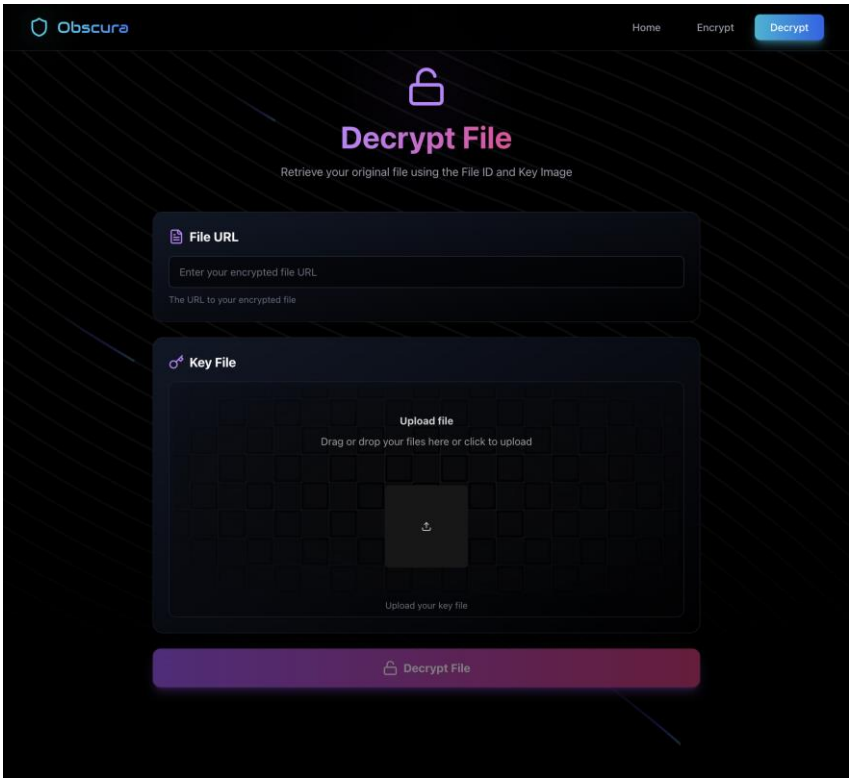
7) Encryption page After



8) Encrypted File View



9) Decryption Page Before



10) Decryption Page After