A SYNOPSIS ON

Role-Based File Access System with AES-128 Encryption

Submitted in partial fulfilment of the requirement for the award of the degree of

BACHELOR OF TECHNOLOGY

In

Computer Science & Engineering

Submitted by:

Manan Sah	**2261643**
Himanshu Kapri	**2261269**
Vivek Rawat	**2261621**
Nitin Malhotra	**2261402**

Under the Guidance of

Mr.Ansh Dhingra

Lecturer

Project Team ID: 75

Department of Computer Science & Engineering

Graphic Era Hill University, Bhimtal, Uttarakhand

March-2025

We hereby certify that the work which is being presented in the Synopsis entitled **“Role-Based File Access System with AES-128 Encryption”** in partial fulfilment of the requirements for the award of the Degree of Bachelor of Technology in Computer Science & Engineering of the Graphic Era Hill University, Bhimtal campus and shall be carried out by the undersigned under the supervision of **Mr. Ansh Dhingra, Lecturer**, Department of Computer Science & Engineering, Graphic Era Hill University, Bhimtal.

Manan Sah	2261643
Himanshu Kapri	2261269
Vivek Rawat	2261621
Nitin Malhotra	2261402

The above mentioned students shall be working under the supervision of the undersigned on the **“Role-Based File Access System with AES-128 Encryption”**

Signature	Signature
Mr. Ansh Dhingra	**Mr. Ankur Singh Bisht**

<u>Internal Evaluation (By DPRC Committee)</u>

Status of the Synopsis: Accepted / Rejected

Any Comments:

Name of the Committee Members:	**Signature with Date**
1.	
2.	

Table of Contents:

Chapter 1:

Introduction and Problem Statement

1.1 Introduction:

In today's digital age, data security has become a critical concern, especially in environments where multiple users collaborate and access shared resources. Traditional file storage systems often rely on basic access control models that lack the flexibility and security needed to protect sensitive information. As organizations grow and data becomes more distributed, the risk of unauthorized access, data breaches, and internal misuse increases significantly.

To overcome these challenges, the proposed system combines Role-Based Access Control (RBAC) with AES-128 encryption to ensure secure file access and storage. RBAC allows the system to define clear access policies based on user roles, such as admin, manager, or employee, thus limiting access to files strictly to authorized users. This role-based approach simplifies permission management while aligning access privileges with organizational structure.

To further enhance security, the system employs AES-128, a highly trusted symmetric encryption algorithm used widely in both commercial and governmental applications. This ensures that files remain protected not only through access restrictions but also through strong encryption—both at rest and during transmission.

Together, RBAC and AES-128 encryption offer a robust, scalable, and secure solution for managing file access in collaborative digital environments, helping organizations safeguard their data against unauthorized usage and cyber threats.

1.2 Problem Statement:

As digital collaboration and remote data access become increasingly prevalent, traditional storage systems are struggling to meet the security demands of modern organizations. Many existing file storage solutions provide only basic access control mechanisms, typically user-based or permission-driven, which often lack the granularity and scalability needed to manage complex access requirements in large or distributed teams.

In these systems, sensitive files may be accessible to users beyond their intended scope due to misconfigured permissions, shared credentials, or insufficient access segregation. Furthermore, while some platforms may offer encryption features, they are either not applied uniformly across all files or rely on weak encryption methods that can be easily compromised. This leads to serious vulnerabilities where sensitive data—such as personal records, financial documents, or proprietary business files—can be exposed, intercepted, or tampered with.

There is also a growing concern over internal threats, where users with legitimate access can misuse or leak data either intentionally or due to negligence. In such cases, it becomes vital to not only control who can access files, but also what actions they can perform based on their role in the organization.

To address these critical security gaps, there is a need for a system that:

- Clearly differentiates access rights based on user roles (e.g., Admin, Manager, Staff)
- Ensures robust encryption of files both at rest (while stored) and in transit (during file transfers)
- Provides secure and reliable authentication mechanisms to prevent unauthorized access
- Maintains an auditable trail of user activity for accountability and compliance
- Offers scalability and ease of administration for growing teams and organizations

The proposed solution aims to solve these issues by integrating Role-Based Access Control (RBAC) with AES-128 encryption, delivering a secure, efficient, and manageable framework for protecting sensitive digital assets.

Chapter 2:

Background / Literature Survey

In the field of information security, the need to manage user access and ensure data confidentiality has led to the development of several technologies and frameworks. Among these, Role-Based Access Control (RBAC) and Advanced Encryption Standard (AES-128) stand out as widely adopted and reliable methods for securing data in both academic and industrial domains.

Role-Based Access Control (RBAC) is a policy-neutral access control mechanism that restricts system access based on a user's role within an organization. Instead of assigning permissions to individual users, RBAC assigns permissions to roles, and users are then assigned to these roles. This abstraction simplifies the management of permissions in large systems and enhances security by ensuring users only access what they are authorized to. RBAC is recognized and recommended by the National Institute of Standards and Technology (NIST) and has been adopted in various sectors including healthcare, banking, and government systems.

AES-128 (Advanced Encryption Standard with 128-bit keys) is a symmetric encryption algorithm that provides a high level of data confidentiality. It was established as a standard by NIST in 2001 and is widely used due to its balance between security and computational efficiency. AES-128 ensures that files are strongly encrypted, making unauthorized access virtually impossible without the corresponding decryption key. Its lightweight footprint and fast processing capabilities make it suitable for real-time applications, especially those implemented in web environments.

From a development standpoint, PyCryptodome is a robust Python library that implements cryptographic primitives, including AES. It provides a simple API for encrypting and decrypting data and is well-suited for integration into custom applications requiring file-level encryption.

Web frameworks like Flask and Django enable developers to rapidly build and deploy secure web applications. Flask, being lightweight and modular, is ideal for small to medium-sized systems where flexibility is important. Django, on the other hand, offers a more structured and full-stack approach with built-in features like user authentication, ORM, and admin panels. Both frameworks support integration with third-party libraries like PyCryptodome, making them suitable choices for implementing encrypted file access systems.

Existing systems, such as Google Drive, Dropbox, and OneDrive, implement some form of access control, allowing file owners to share data selectively. However, these services are proprietary and do not provide source code or APIs that expose the underlying implementation of their access control or encryption mechanisms. This limits their usefulness for academic projects and research purposes, where transparency and customization are critical.

Several research papers and case studies have demonstrated the effectiveness of combining RBAC with encryption to enhance data security. For instance, studies on cloud-based storage models have shown that the hybrid approach significantly reduces the risk of data leakage while simplifying permission management.

In conclusion, the integration of RBAC and AES-128 within a Python-based web application offers a practical and secure solution for controlled file access. This project builds on proven methodologies and modern tools to address current gaps in data security and access management.

Chapter 3:

Objectives

The primary aim of this project is to design and implement a secure file access system that leverages both **Role-Based Access Control (RBAC)** and **AES-128 encryption** to protect sensitive data in multi-user environments. The specific objectives of the project are as follows:

- **To implement Role-Based Access Control (RBAC):** The system will categorize users into different roles (e.g., Admin, Manager, Employee) and define specific permissions for each role. This will ensure that users can only perform actions and access files relevant to their roles, reducing the risk of accidental or malicious data access.

- **To integrate AES-128 encryption for secure file storage and retrieval:** Every file stored in the system will be encrypted using the AES-128 standard, ensuring data confidentiality. Files will remain encrypted both at rest (in storage) and in transit (during download/upload), making unauthorized access or data leaks nearly impossible without proper credentials and keys.

- **To develop a user-friendly and responsive web interface using Flask or Django:** The project aims to build a clean and intuitive web-based platform that allows users to log in, upload and download files, and manage access—all within a streamlined interface. Flask will be used for lightweight implementation, while Django may be considered for projects requiring more robust feature integration.

- **To ensure strong authentication and secure session handling:** The system will implement user authentication using hashed credentials and session tokens to validate users before granting access. Additional security mechanisms like password hashing (using bcrypt or Argon2) and input validation will be used to guard against common web vulnerabilities such as SQL injection and cross-site scripting (XSS).

- **To maintain audit trails and access records using SQLite:** All file access activities, such as uploads, downloads, role changes, and permission updates, will be logged into a lightweight SQLite database. This enables transparency, traceability, and accountability, which are critical for auditing and maintaining secure operations.

- **To offer modularity and scalability:** The system will be designed to allow easy extension of roles, permissions, and features. Future upgrades such as integration with cloud storage or biometric login will be made possible by following modular and clean code practices.

By achieving these objectives, the project will deliver a practical and effective solution for managing secure file access in collaborative digital environments, suitable for academic institutions, businesses, and organizations dealing with sensitive information.

Chapter 4:

Hardware and Software Requirements

4.1 Hardware Requirements

Sl. No	**Name of the Hardware**	**Specification**
1	Computer/Laptop	i5 Processor or better, 8GB RAM, 512GB Storage
2	Network	Internet connectivity for testing cloud interface

4.2 Software Requirements

Sl. No	**Name of the Software**	**Specification**
1	Python	Version 3.10+
2	Flask/Django	Web framework
3	PyCryptodome	For AES-128 Encryption
4	SQLite	Lightweight database for roles
5	Windows OS	With NTFS permissions support

Chapter 5:

Proposed System Architecture

The proposed system is designed to provide a secure, scalable, and user-friendly platform for file access and management using Role-Based Access Control (RBAC) combined with AES-128 encryption. The architecture adopts a modular approach to simplify development, testing, and future enhancement.

5.1 System Components

1. **Login Module:** This is the entry point of the system. It manages user authentication using secure credentials and session tokens. Upon successful login, the system identifies the user's role (e.g., Admin, Editor, Viewer) and stores this information for session handling. Passwords are securely stored using hashing algorithms like bcrypt or Argon2.

2. **Access Control Module:** This module handles role-based access decisions. Based on the user's assigned role, it checks whether the requested action (e.g., upload, view, delete) is permitted. It ensures that users can only interact with files and features they are authorized for. Role policies are dynamically enforced at runtime.

3. **Encryption Module:** Critical for data security, this module uses the AES-128 algorithm to encrypt files before they are stored and decrypt them upon authorized access. The encryption keys are securely generated and managed to prevent unauthorized decryption. Each file is encrypted uniquely, preventing exposure in case of data leakage.

4. **Database Module:** This module uses SQLite to maintain records of users, roles, permissions, encrypted file metadata, and access logs. It enables easy tracking of who accessed what and when. The database also helps manage audit trails and is designed for future expansion to more powerful DBMS like PostgreSQL or MySQL.

5. **File Management Module:** This module handles uploading, downloading, organizing, and storing encrypted files in the local or cloud-based storage system. It integrates closely with the encryption module to ensure data is never stored in plain text.

5.2 System Workflow

The end-to-end system workflow is described below:

1. **User Login:** The user accesses the platform and enters login credentials. The system authenticates the user and retrieves the assigned role.

2. **Role Verification:** Once authenticated, the system loads permissions associated with the user's role from the database.

3. **Action Request:** The user performs an action such as uploading or downloading a file. The system checks if the role allows the requested action.

4. **Encryption/Decryption:**

- If uploading, the file is encrypted using AES-128 before storage.
- If downloading, the file is decrypted on-the-fly only if the user is authorized.

5. **Storage and Access Logging:** All actions are logged in the database with timestamps and user details for audit and monitoring purposes.
6. **Response to User:** Based on the authorization decision, the system either allows the action and returns the encrypted/decrypted file, or it denies the request with a notification.

Chapter 6:

Encryption and Security Model

The security of any file access system heavily relies on how effectively data is encrypted, stored, and managed across different user roles. The proposed system integrates **Advanced Encryption Standard (AES-128)** encryption with **Role-Based Access Control (RBAC)** to ensure confidentiality, integrity, and controlled access to data.

6.1 AES-128 Encryption

AES (Advanced Encryption Standard) is a symmetric encryption algorithm widely recognized for its speed, efficiency, and resistance to cryptographic attacks. AES-128 uses a 128-bit key length which provides a strong level of security suitable for both commercial and governmental applications.

In the system:

- All files are encrypted using AES-128 before being stored.
- The encryption process ensures that files cannot be accessed or understood without the appropriate key, even if storage is compromised.
- Encryption is applied both **at rest** (while stored on disk) and **in transit** (when being transferred between client and server).

Table 6.1 Pseudo codes of the AES-128 Encryption Algorithm

```
FUNCTION encrypt_file(file_path, key):
    data = READ file from file_path
    iv = GENERATE random 16-byte IV
    cipher = CREATE AES cipher with key and iv in CBC mode
    encrypted_data = ENCRYPT data with cipher
    STORE iv + encrypted_data in encrypted_file
    RETURN encrypted_file
```

```
FUNCTION decrypt_file(encrypted_file, key):
    iv = EXTRACT first 16 bytes from encrypted_file
    cipher = CREATE AES cipher with key and iv in CBC mode
    encrypted_data = REMAINING bytes from encrypted_file
    data = DECRYPT encrypted_data with cipher
    RETURN data
```

6.2 Key Derivation and Management

Proper key management is critical to prevent unauthorized access:

- User passwords are not used directly as encryption keys. Instead, secure key derivation functions such as **PBKDF2** or **Argon2** are used to generate cryptographic keys from passwords.
- These key derivation functions add **salt** and use multiple iterations to produce secure keys resistant to brute-force and rainbow table attacks.
- Keys are securely stored in memory only during the session and cleared after logout to reduce the risk of key leakage.

Table 6.2 Pseudo codes of the Key Derivation and Management Algorithm

```
FUNCTION derive_key(password, salt):
key = APPLY PBKDF2 or Argon2 on (password, salt, iterations=100000)
RETURN key
```

```
FUNCTION store_key_in_session(key):
session["key"] = key
SET timer to auto-delete key after logout or timeout
```

6.3 Role-Based Key Access

To align encryption with RBAC:

- Each user role (e.g., Admin, Editor, Viewer) is assigned different levels of decryption key access.
- Higher privilege roles can decrypt a broader range of files, while lower-level roles have restricted access.
- The mapping between roles and keys is maintained in a secure, encrypted database.

Table 6.3 Pseudo codes of the Key Derivation and Management Algorithm

```
FUNCTION get_user_role(user_id):
    RETURN DATABASE_QUERY("SELECT role FROM users WHERE id = user_id")
FUNCTION get_allowed_keys(role):
    RETURN DATABASE_QUERY("SELECT key_id FROM role_key_map WHERE role = role")
FUNCTION authorize_decryption(user_id, file_id):
    user_role = get_user_role(user_id)
    allowed_keys = get_allowed_keys(user_role)
    file_key_id = DATABASE_QUERY("SELECT key_id FROM files WHERE id = file_id")
    IF file_key_id IN allowed_keys:
        RETURN True
    ELSE:
            RETURN False
```

6.4 OS-Level File Security

In addition to application-level encryption, the system also utilizes **NTFS permissions** provided by the Windows operating system:

- These permissions restrict file access based on user accounts and roles, adding another layer of protection.
- Even if someone bypasses the application interface, NTFS permissions help prevent unauthorized access from the operating system level.

Table 6.4 Pseudo codes of the OS-Level File Security (NTFS Permissions - Simulated)

```
FUNCTION apply_ntfs_permissions(file_path, user_role):
    IF user_role == "Admin":
        SET full_control(file_path)
    ELSE IF user_role == "Editor":
        SET read_write_permissions(file_path)
    ELSE:
        SET read_only(file_path)
```

6.5 Secure Communication and Storage

- HTTPS protocols are used to ensure that all data in transit between users and the server is encrypted.
- Encrypted files are stored in a secure directory structure with restricted access.
- Logs and audit trails are also encrypted or obfuscated to prevent tampering or leakage of metadata.

Table 6.5 Pseudo codes for Secure Communication and Storage

```
SERVER CONFIGURATION:
    ENABLE HTTPS with SSL certificate
    REDIRECT all HTTP traffic to HTTPS

FUNCTION store_encrypted_file(file_data, path):
    encrypted_data = encrypt_file(file_data)
    SAVE encrypted_data TO secure_directory + path
    SET directory permissions = 700
```

6.6 Integrity Checks

To ensure the data has not been tampered with:

- Encrypted files are appended with cryptographic hash values (e.g., SHA-256) for integrity verification.
- Upon decryption, the file's hash is recalculated and matched to the original hash before allowing access.

Table 6.6 Pseudo codes for Integrity Checks

```
FUNCTION append_hash_to_file(data):
    hash = CALCULATE SHA-256 hash of data
    return data + hash

FUNCTION verify_file_integrity(encrypted_file_with_hash):
    data, original_hash = SPLIT file and hash
    new_hash = SHA-256(data)
    IF new_hash == original_hash:
        RETURN True
    ELSE:
        RETURN False
```

Chapter 7:

System Implementation

The implementation of the **Role-Based File Access System with AES-128 Encryption** follows a modular and scalable architecture using Python as the core language. The backend is powered by Flask, which handles routing, request handling, and integration with the encryption and authentication modules.

7.1 Technologies Used

- **Python**: The primary programming language for backend logic, cryptography, and server management.
- **Flask**: A lightweight WSGI web application framework used for handling routing, session management, and RESTful APIs.
- **SQLite**: A serverless, lightweight database used for managing user roles, permissions, and login data.
- **PyCryptodome**: A self-contained cryptographic library used for implementing AES-128 encryption and decryption.
- **HTML/CSS/JavaScript**: Used for building the frontend user interface (UI) of the application.

7.2 Key Functional Components

- **Authentication Module**: Manages user login, session control, and password hashing using PBKDF2.
- **Role-Based Access Control (RBAC) Module**: Assigns roles and manages access policies for different file categories.
- **Encryption Module**: Handles AES-128 encryption/decryption, key generation, and secure file storage.
- **Database Handler**: Connects with SQLite to query user data, roles, and file access logs.
- **Configuration Module**: Loads system environment settings like encryption keys, session limits, and debug modes.

7.3 Application Workflow

1. User registers and is assigned a role (e.g., Admin, Editor, Viewer).
2. During login, the credentials are validated and session initialized.
3. When a file is uploaded, it is encrypted using AES-128 and stored
4. During retrieval, the system checks user roles and decrypts files accordingly.

7.4 Directory Structure

```
project-root/
├── app/
│   ├── __init__.py
│   ├── main.py                # Main Flask application entry point
│   ├── config/
│   │   ├── settings.py        # App configurations and constants
│   │   └── encryption_keys.json
│   ├── authentication/
│   │   ├── login.py           # User login and authentication logic
│   │   ├── register.py        # User registration and hashing
│   │   └── session.py         # Session control and logout
│   ├── encryption/
│   │   ├── aes_cipher.py      # AES-128 encryption/decryption logic
│   │   └── key_handler.py     # Key generation and PBKDF2-based derivation
│   ├── rbac/
│   │   ├── role_check.py      # Access control logic
│   │   └── role_assign.py     # Assign roles to users
│   ├── db/
│   │   ├── database.py        # SQLite connection and schema creation
│   │   └── models.py          # ORM or SQL abstraction for tables
│   ├── static/
│   │   └── styles.css         # CSS styles for the frontend
│   └── templates/
│      ├── login.html          # Login page
│      ├── dashboard.html      # User dashboard
│      └── upload.html         # File upload interface
└── README.txt                 # Project documentation
```

Chapter 8:

Testing and Validation

Rigorous testing is crucial for ensuring the security, functionality, and reliability of the Role-Based File Access System. The system underwent multiple stages of testing—both automated and manual—to validate its robustness across diverse scenarios

.

8.1 Unit Testing

Each module was individually tested to ensure isolated functionalities work as intended:

- **Login Module**: Verified user authentication with correct and incorrect credentials.
- **Encryption Module**: Tested encryption and decryption functions with various file types and sizes.
- **RBAC Module**: Ensured accurate role recognition and mapping during user interaction.
- **Database Module**: Confirmed successful CRUD operations on user, role, and file metadata tables.
-

8.2 Integration Testing

Modules were integrated incrementally, and test cases were executed to ensure the interaction between them was seamless:

- Ensured that encrypted files can be decrypted correctly *only* after passing authentication and role validation.
- Validated the database response when queried simultaneously by multiple modules (e.g., during login and file fetch).

8.3 Functional and Role-Based Testing

The RBAC mechanism was tested to verify that:

- **Admins** have full access, including file upload, view, delete, and manage user roles.
- **Editors** can upload and view files but cannot delete or manage roles.
- **Viewers** can only access specific files based on assigned permissions.

Unauthorized access attempts triggered appropriate error messages and logs in the audit trail.

8.4 Encryption Validation

The following checks were performed:

- Manually verified that encrypted files are unreadable without the decryption process.
- Verified that a mismatch in keys (incorrect user access) fails gracefully and denies access without data leakage.
- Confirmed consistent behavior across file types (text, PDF, images).

8.5 Performance Testing

- Stress-tested with concurrent logins and file uploads to ensure the system does not crash under load.
- Measured encryption/decryption speed for large files (>10MB) to ensure minimal latency.

Chapter 9:

Conclusion

The **Role-Based File Access System with AES-128 Encryption** provides a robust, scalable, and secure solution to the problem of managing sensitive data in multi-user environments. By leveraging **Role-Based Access Control (RBAC)** for dynamic and granular access permissions, and combining it with **AES-128 encryption** for data confidentiality, the system ensures a high level of information security while maintaining ease of use and accessibility.

This project achieves the following key milestones:

- **Security**: It significantly enhances the security posture of file storage systems by ensuring that only authorized individuals can access or modify specific files.
- **Flexibility**: Through the role management interface, administrators can easily assign, modify, or revoke access based on evolving organizational needs.
- **Transparency**: Every access or modification is logged, enabling accountability and traceability, which are critical in enterprise and academic environments.
- **User Experience**: A clean, responsive interface ensures that even non-technical users can manage and access their files with minimal training or support.
- **Portability**: The use of Python, Flask, SQLite, and PyCryptodome makes the system lightweight and portable across different operating systems and hardware configurations.

This system lays the groundwork for further enhancements like cloud integration, biometric access, real-time intrusion detection, and more sophisticated audit mechanisms. It demonstrates the practical application of theoretical security principles and shows how modern technologies can be synergized to solve real-world problems.

Ultimately, this project is not just a proof of concept, but a deployable prototype that can be adapted to educational institutions, small businesses, and even larger organizations with moderate customization.

Chapter 10:

Future Enhancements

While the current system provides a solid foundation for secure and role-based file access, there are several opportunities to improve its functionality, scalability, and intelligence through future enhancements:

- **Integration of Machine Learning for Anomaly Detection:** Machine Learning algorithms can be used to analyze access patterns and detect unusual or suspicious activities. For instance, the system could automatically flag or restrict access if a user attempts to access files outside their usual working hours or from an unfamiliar IP address.

- **Blockchain for Tamper-Proof Audit Logs**: Incorporating blockchain technology can ensure that access logs are immutable and tamper-proof. This is particularly valuable in industries where compliance and traceability are crucial, such as healthcare, finance, and legal services.

- **Cloud-Based File Access (AWS, Google Cloud, Azure)**: To support modern workflows and remote access, the system could be expanded to integrate with cloud storage platforms. This would enable users to securely access encrypted files from anywhere while maintaining strict role-based permissions.

- **Biometric Authentication**: Adding biometric security mechanisms such as fingerprint or facial recognition would provide an extra layer of identity verification, reducing the risk of unauthorized access due to password compromise.

- **Role Hierarchies and Delegation**: Advanced role structures could be implemented, where higher-level roles can delegate specific permissions temporarily to subordinates—useful in dynamic teams or during role transitions.

- **End-to-End Encryption with Key Rotation**: Introducing automated key rotation and end-to-end encryption ensures even higher levels of confidentiality, especially for long-term file storage.

- **Multi-Factor Authentication (MFA)**: Enhancing login security with MFA using email, OTPs, or authenticator apps would further harden the authentication process.

- **Responsive Mobile Interface or App**: A mobile-friendly web interface or native mobile app would allow users to access files and manage roles securely on-the-go.

References:

[1] National Institute of Standards and Technology. "Advanced Encryption Standard (AES) Specification."

[2] Sandhu, R., Coyne, E. J., Feinstein, H. L., & Youman, C. E. (1996). "Role-Based Access Control Models."

[3] PyCryptodome Documentation. Available at: https://pycryptodome.readthedocs.io

[4] Django Project Documentation. Available at: https://www.djangoproject.com/

[5] SQLite Official Documentation. Available at: https://www.sqlite.org/index.html

[6] "Network Security: Private Communication in a Public World." Prentice Hall.

[7] Bishop, M. (2003). "Computer Security: Art and Science." Addison-Wesley.

[8] Kizza, J. M. (2013). "Guide to Computer Network Security." Springer.

[9] Argon2 Password Hashing Algorithm. Available at: https://password-hashing.net/argon2-specs.pdf