COP 4338 BONUS Assignment: XOR-Based File Encryption System

📑 Table of Contents

1. 📧 TA Contact Information
2. 📖 Assignment Overview
3. 🎯 Learning Objectives
4. 🔐 Understanding Encryption & XOR Cipher
5. ⚙️ Assignment Tasks
6. 🏗️ Project Framework
7. 🔧 Development & Testing
8. 📤 Assignment Submission Guidelines
9. 📦 Deliverable Folder Structure
10. 💯 Grading Rubric

📢 Important Note!

Please ensure that your implementation works on the course-designated ocelot server: ocelot-bbhatkal.aul.fiu.edu . Claims such as **"it works on my machine"** will not be considered valid, as all testing and grading will be conducted in that environment.

⚠️ **Important:** You are provided with an autograder to support consistent testing and maintain the highest level of grading transparency. The same autograder will be used by the instructor for final evaluation.

If your implementation passes all test cases using the autograder, you are likely to receive full credit for your submission.

⚠️ **There is only one submission - RESUBMISSION IS NOT ALLOWED. You must test your implementation thoroughly with the provided testing framework on the Ocelot server before the final submission.**

---

1. 📧 TA Contact Information

For any questions or concerns related to assignment grading, please reach out to the TA listed below.

- Ensure that all communication is polite and respectful
- You must contact the TA **first** for any grading issues. If the matter remains unresolved, feel free to reach out to me

Name: Sanskar Lohani — sloha002@fiu.edu
Sections: COP 4338-RVG (88613), COP 4338-U01 (84664)

2. 📖 Assignment Overview

This assignment provides hands-on experience with **file encryption using cryptographic techniques** and **systems programming fundamentals**. You will implement **2 core functions** for a **file encryption/decryption system** using **XOR-based cipher operations**, **2D array manipulation**, **character-by-character file I/O**, and **bitwise operations** essential for systems and security programming.

The system implements a simplified XOR based encryption scheme commonly used in educational contexts to teach cryptographic principles. **The instructor-provided framework handles all command parsing, matrix loading, file management, and testing** - your job is to focus purely on implementing the encryption and decryption logic using XOR operations with matrix-based keys.

This assignment introduces the fundamental concepts behind data encryption and security, demonstrating how simple mathematical operations can protect information. While we use XOR cipher for educational purposes, the principles you learn here—including file I/O, bitwise operations, and algorithmic thinking—are the same ones used in real-world encryption systems like AES (Advanced Encryption Standard), ChaCha20, and other modern cryptographic algorithms protecting data in banks, governments, and secure communications worldwide.

✅ **Course Modules Covered:**

- **File I/O** (character-by-character operations)
- **2D Arrays** (matrix operations and indexing)
- **Bitwise Operations** (XOR cipher implementation)
- **Binary Files** (handling encrypted data)
- **Defensive Programming** (input validation and error handling)

3. 🎯 Learning Objectives

By completing this assignment, you will be able to:

1. **Master character-by-character file I/O** using `fgetc()` and `fputc()` for processing text and binary files byte-by-byte.
2. **Apply bitwise XOR operations** for data encryption and understand the mathematical properties that make XOR suitable for cryptography.
3. **Navigate and manipulate 2D arrays** using calculated row/column indices to implement matrix-based key scheduling.
4. **Handle binary file operations** correctly for storing and retrieving encrypted data with proper mode specifications.
5. **Implement comprehensive input validation** and error handling mechanisms for robust file processing.
6. **Understand symmetric encryption concepts** including plaintext, ciphertext, keys, encryption, and decryption operations.

4. 🔐 Understanding Encryption & XOR Cipher

What is Encryption?

Encryption is the process of transforming readable data (called **plaintext**) into an unreadable format (called **ciphertext**) to protect it from unauthorized access. Only someone with the correct **key** can decrypt the ciphertext back to plaintext.

Basic Encryption Flow:

```
Plaintext + Key → Encryption Algorithm → Ciphertext
Ciphertext + Key → Decryption Algorithm → Plaintext
```

Real-World Examples:

- **HTTPS websites** encrypt your passwords and credit card information
- **WhatsApp messages** are encrypted end-to-end
- **Banking transactions** use encryption to protect your financial data
- **Password managers** encrypt your stored passwords

Why XOR for This Assignment?

For **educational purposes**, we use the **XOR (Exclusive OR) cipher** because:

1. ✅ **Simple to understand** - Basic bitwise operation
2. ✅ **Self-inverse property** - Same operation encrypts and decrypts
3. ✅ **Demonstrates core concepts** - Shows how encryption works fundamentally
4. ✅ **Fast execution** - Single CPU instruction per byte
5. ✅ **Foundation for learning** - Builds understanding for advanced cryptography

XOR Operation Explained

XOR Truth Table:

```
Input A | Input B | A XOR B
--------|---------|--------
   0    |    0    |    0
   0    |    1    |    1
   1    |    0    |    1
   1    |    1    |    0
```

Key Insight - XOR is Symmetric: `(A ⊕ B) ⊕ B = A` (XOR is its own inverse)

```
Encryption: cipher = plain XOR key
Decryption: plain = cipher XOR key

This means:
  (plain XOR key) XOR key = plain

Your decryption code is nearly identical to encryption!
```

Example with Bytes:

```
Character 'H' = 01001000 (72 in decimal)
Key value     = 01010011 (83 in decimal)
                ──────────
Encrypted     = 00011011 (27 in decimal)

To decrypt:
Encrypted     = 00011011 (27 in decimal)
Key value     = 01010011 (83 in decimal)
                ──────────
Original 'H'  = 01001000 (72 in decimal) ✓
```

How Our XOR Cipher Works

We use a **matrix of integers** as our **encryption key**. Each byte of the plaintext is XORed with a value from the matrix in a ***cyclic pattern***:

```
Matrix (2×2 example):
┌─────────┬─────────┐
│   73    │   149   │
├─────────┼─────────┤
│   211   │   57    │
└─────────┴─────────┘

Plaintext: "HELLO"

Byte 0 'H'(72):  row=0, col=0 → 72 ⊕ 73  = 1
Byte 1 'E'(69):  row=0, col=1 → 69 ⊕ 149 = 216
Byte 2 'L'(76):  row=1, col=0 → 76 ⊕ 211 = 167
Byte 3 'L'(76):  row=1, col=1 → 76 ⊕ 57  = 117
Byte 4 'O'(79):  row=0, col=0 → 79 ⊕ 73  = 6  (cycles back!)

This creates a **key stream** where the matrix values repeat cyclically throughout the file.
```

Matrix Position Calculation:

```
For byte at position i with matrix of size n:
    row = (i / n) % n
    col = i % n
```

Real-World Encryption Systems

While XOR cipher is excellent for learning, **real-world applications use more sophisticated algorithms**:

System	Algorithm	Usage
AES	Advanced Encryption Standard	Government, banking, file encryption
ChaCha20	Stream cipher	Google Chrome, TLS connections

System	Algorithm	Usage
RSA	Asymmetric encryption	Digital signatures, key exchange
Blowfish/Twofish	Block ciphers	VPNs, password managers
One-Time Pad	Perfect XOR cipher	Military, diplomatic communications

Important Security Note

🔒 **This assignment's XOR cipher is for EDUCATIONAL PURPOSES ONLY**. Never use simple XOR encryption for protecting real sensitive data. Always use established, peer-reviewed cryptographic libraries like:

- OpenSSL
- libsodium
- Crypto++ (C++)
- Java Cryptography Architecture (JCA)

---

5. ⚙️ Assignment Tasks

What You Must Implement **?**

You will implement **2 functions** that collectively create a **file encryption/decryption system**. The instructor-provided framework handles all matrix loading, command parsing, file verification, and testing.

Core System Capabilities:

- Encrypts **text files** using XOR cipher with matrix-based keys
- Decrypts **encrypted files** to recover original plaintext
- Supports **multiple matrix sizes** (2×2, 3×3, 4×4) for different security levels
- Preserves **exact file size** (no padding needed with XOR)
- Uses **simple, reliable XOR operation** that is symmetric (encryption = decryption)

2 Required Functions:

1. `encryptFile()` - Encrypt plaintext file using XOR with matrix values
2. `decryptFile()` - Decrypt ciphertext file using XOR with matrix values

📋 Detailed Function Specifications

Function 1: `int encryptFile(const char* inputFile, const char* outputFile, int** matrix, int size)`

Purpose: Encrypt a plaintext file using XOR cipher with matrix values as the key stream, processing the file byte-by-byte for secure transformation.

Parameters:

- `inputFile`: Name of the plaintext input file to encrypt

- `outputFile`: Name of the binary output file for encrypted data
- `matrix`: 2D array containing encryption key values (pre-loaded by framework)
- `size`: Dimension of the matrix (2, 3, or 4)

Return Value:

- `SUCCESS` (1) on successful encryption
- `FAILURE` (0) on any error (file errors, null pointers, invalid parameters)

Requirements:

- **Input Validation:**
    - Check if `inputFile` is NULL → return `FAILURE`
    - Check if `outputFile` is NULL → return `FAILURE`
    - Check if `matrix` is NULL → return `FAILURE`
    - Check if `size` is valid (`MIN_MATRIX_SIZE` to `MAX_MATRIX_SIZE`) → return `FAILURE`
- **File Operations:**
    - Open input file in **text read mode**: `fopen(inputFile, "r")`
    - Open output file in **binary write mode**: `fopen(outputFile, "wb")`
    - Check both file pointers for NULL (handle errors properly)
    - If second `fopen()` fails, close the first file before returning
- **Encryption Algorithm:**

```
Algorithm: Binary File Encryption using XOR with Matrix

Input: Input file, Output file, Encryption matrix of size N×N
Output: Encrypted file

Steps:
1. Open input file in Text read mode (r)
2. Open output file in Binary write mode (rw)
3. Initialize byte counter to 0

4. For each byte read from input file:
   a. Determine position in encryption matrix:
      - Calculate row index using byte counter and matrix size
      - Calculate column index using byte counter and matrix size

   b. Apply encryption:
      - XOR the input byte with corresponding matrix element

   c. Write the encrypted byte to output file

   d. Increment byte counter

5. Close both files

Note: The row and column calculations ensure the matrix pattern repeats
```

```
        cyclically for files larger than the matrix size.
```

- **Error Handling:**
    - If `fputc()` returns `EOF`, cleanup and return `FAILURE`
    - Close both files before returning (success or failure)
    - Return `SUCCESS` only if all operations complete successfully

Example Execution (2×2 matrix [[73, 149], [211, 57]]):

```
Input: "HI" (ASCII: 72, 73)

Byte 0 (H=72):
  - Position: i=0, row=0, col=0
  - Operation: 72 XOR 73 = 1
  - Output: byte value 1

Byte 1 (I=73):
  - Position: i=1, row=0, col=1
  - Operation: 73 XOR 149 = 216
  - Output: byte value 216

Result: Binary file with bytes [1, 216]
```

Integration Role: This function is called by the framework when processing `ENCRYPT` commands from test files. The framework provides the matrix and validates the result.

Function 2: `int decryptFile(const char* inputFile, const char* outputFile, int** matrix, int size)`

Purpose: Decrypt a ciphertext file using XOR cipher, applying the exact same operation as encryption (due to XOR's self-inverse property) to recover the original plaintext.

Parameters:

- `inputFile`: Name of the encrypted binary input file
- `outputFile`: Name of the plaintext output file for decrypted data
- `matrix`: 2D array containing decryption key values (same as encryption matrix)
- `size`: Dimension of the matrix (2, 3, or 4)

Return Value:

- `SUCCESS` (1) on successful decryption
- `FAILURE` (0) on any error

Requirements:

- **Input Validation:**
    - Validate all parameters (same checks as encryption)
    - Return `FAILURE` if any parameter is invalid
- **File Operations:**

- Open input file in **binary read mode**: `fopen(inputFile, "rb")`
- Open output file in **text write mode**: `fopen(outputFile, "w")`
- Check both file pointers for NULL
- Handle file opening errors with proper cleanup

- **Decryption Algorithm:**

```
Algorithm: Binary File Decryption using XOR with Matrix

Input: Encrypted file, Output file, Decryption matrix of size N×N
Output: Decrypted (original) file

Steps:
1. Open encrypted file in Binary read mode (rb)
2. Open output file in Text write mode (w)
3. Initialize byte counter to 0

4. For each byte read from encrypted file:
   a. Determine position in decryption matrix:
      - Calculate row index using byte counter and matrix size
      - Calculate column index using byte counter and matrix size

   b. Apply decryption:
      - XOR the encrypted byte with corresponding matrix element
      (Note: XOR is symmetric - same operation as encryption)

   c. Write the decrypted byte to output file

   d. Increment byte counter

5. Close both files

Key Property: XOR is self-inverse, meaning encryption and decryption
              use the identical operation with the same matrix.
```

- **Error Handling:**
    - If `fputc()` returns `EOF`, cleanup and return `FAILURE`
    - Close both files properly before returning
    - Return `SUCCESS` only if decryption completes successfully

Example Execution:

```
Encrypted Input: bytes [1, 216]
Matrix: [[73, 149], [211, 57]]

Byte 0 (1):
  - Position: i=0, row=0, col=0
  - Operation: 1 XOR 73 = 72 ('H')
  - Output: 'H'

Byte 1 (216):
  - Position: i=1, row=0, col=1
  - Operation: 216 XOR 149 = 73 ('I')
  - Output: 'I'

Result: Text file containing "HI" ✓
```

Verification: If decryption works correctly, the output file will be identical to the original input file (byte-for-byte match).

Integration Role: This function is called by the framework when processing `DECRYPT` commands. The framework then verifies that the decrypted file matches the original input exactly.

---

6. 🏗 Project Framework

Files Provided by Instructor (DO NOT MODIFY)

File	Purpose	Your Action
`encrypt.h`	Header with function prototypes, constants, and definitions	**DO NOT MODIFY**
`driver.c`	Main program with matrix loading, command execution, and testing	**DO NOT MODIFY**
`Makefile`	Build configuration for compiling the project	**DO NOT MODIFY**
`Testcases/Testing/`	All test files, keys, and expected outputs organized by difficulty	**Use for testing**
`autograder.sh`	Automated grading script testing all 3 levels	**DO NOT MODIFY**
`batchgrader.sh`	Batch testing script for multiple submissions	**DO NOT MODIFY**

Files You Must Implement

File	Your Task
`encryptFile.c`	Implement XOR-based file encryption using matrix values
`decryptFile.c`	Implement XOR-based file decryption using matrix values

Test Structure - Understanding the Test Levels

```
Testcases/Testing/
├── simple/
│   ├── INPUT1.txt           # "HELLO WORLD" (11 bytes, easy test)
│   ├── key_1.txt            # 2x2 matrix for encryption
│   ├── testcases_simple.txt # Test commands
│   └── EXPECTED_OUTPUT.txt  # Reference output for comparison
│
├── moderate/
│   ├── INPUT2.txt           # 97-character sentence (medium complexity)
│   ├── key_2.txt            # 3x3 matrix for encryption
│   ├── testcases_moderate.txt # Test commands
│   └── EXPECTED_OUTPUT.txt      # Reference output for comparison
│
└── rigorous/
    ├── INPUT3.txt               # 504-character paragraph (high complexity)
    ├── key_3.txt                # 4x4 matrix for encryption
    ├── testcases_rigorous.txt   # Test commands
    └── EXPECTED_OUTPUT.txt      # Reference output for comparison
```

Understanding Key Files

Each `key_*.txt` file contains:

```
Line 1: Matrix size (2, 3, or 4)
Lines 2-N: Matrix values (size×size integers)

Example (2×2 matrix):
2
73 149
211 57
```

Understanding Test Commands

Each `testcases_*.txt` file contains commands like:

```
ENCRYPT input.txt key.txt output.bin
DECRYPT output.bin key.txt recovered.txt
VERIFY original.txt recovered.txt
```

The framework reads these commands and calls your functions accordingly.

How the Framework Works

1. **Loads matrix** from key file into 2D array
2. **Executes ENCRYPT** command → calls your `encryptFile()`
3. **Executes DECRYPT** command → calls your `decryptFile()`
4. **Executes VERIFY** command → compares original with decrypted file
5. **Reports results** with detailed pass/fail information

---

7. 🔧 Development & Testing

Required Files for Development

All the following files must be in the same directory:

1. **Framework files**: `driver.c`, `Makefile`, `encrypt.h`
2. **Your implementation files**: `encryptFile.c`, `decryptFile.c`
3. **Testing tools**: `autograder_encryption.sh`,`batchgrader_encryption.h` `/Testing directory`

⚠️ **Critical Testing Information: The instructor will use different and more rigorous test cases for final grading that are not provided to students. However, if you pass all provided test cases with 100% accuracy, you are highly likely to pass the instructor's final grading test cases.**

Comprehensive Development Workflow

> ⚠️ **Important: Please refer to the document BONUS_Testing_Guide.pdf for the detailed testing instructions.**

Step 1: Understand Expected Output Format

```
# Generate expected output from instructor sample
./BONUS_sample Testcases/simple/testcases_simple.txt> EXPECTED_OUTPUT.txt

# View the expected output format to understand correct behavior
cat EXPECTED_OUTPUT.txt
```

Step 2: Recommended Implementation Strategy

Follow this sequence to build your implementation incrementally:

- **Phase 1:** Understand the Algorithm
- **Phase 2:** Implement Encryption
- **Phase 3:** Implement Decryption
- **Phase 4:** Testing (Incremental Validation)

Step 3: Use Autograders for Final Validation

- ⚠️ **It is very importat to test your implelentation with both the autograders before the final submission**
- ⚠️ **Important: Please note that instructor will use the same autograders for the final grading with different test cases.**

🤖 Autograder System

The autograder is your **best friend** for this assignment. It:

1. **Compiles your code** using the Makefile
2. **Runs all 3 test levels** automatically (simple, moderate, rigorous)
3. **Compares your output** with expected results line-by-line
4. **Verifies actual files** to prevent cheating:
   - Checks that `ENCRYPTED.bin` was created
   - Verifies `ENCRYPTED.bin` is different from input (proves encryption happened)
   - Confirms `OUTPUT.txt` matches original input exactly (proves correct decryption)
5. **Provides detailed feedback** showing exactly what went wrong if tests fail

What the Autograder Tests

For Each Test Level:

1. ✅ Program output messages match expected format
2. ✅ `ENCRYPTED.bin` file is created
3. ✅ `ENCRYPTED.bin` differs from input (actual encryption occurred)
4. ✅ `ENCRYPTED.bin` is not empty
5. ✅ `OUTPUT.txt` file is created
6. ✅ `OUTPUT.txt` matches original input byte-for-byte

Anti-Cheat Measures:

- Cannot hardcode output messages (files are verified)
- Cannot copy input to output (encrypted file must differ)
- Cannot produce empty files (size is checked)
- Cannot fake decryption (byte-by-byte comparison with `diff`)

⚠️ *IMPORTANT: Final Grading Notice*

🚨 **The instructor will use the SAME autograder script for final grading, but may include additional edge case testing.**

Your implementation must:

- ✅ **Pass the provided autograder with 100/100 points**
- ✅ **Handle all edge cases** correctly (empty arrays, null pointers, boundary conditions)
- ✅ **Produce deterministic output** for identical inputs
- ✅ **Compile cleanly** with the provided Makefile

File I/O Best Practices

Critical File Operation Guidelines:

- Always check if `fopen()` returns NULL before using the file pointer
- **Use correct file modes - THIS IS CRITICAL:**
  - Reading text: `"r"`
  - Writing text: `"w"`
  - Reading binary: `"rb"`
  - Writing binary: `"wb"`

⚠️ **Why Binary Mode Matters for Encrypted Files:**

Encrypted data contains **all possible byte values (0-255)**, including special characters like null bytes (`\0`), newlines (`\n`), and carriage returns (`\r`).

- **Text mode (`"r"`)**: On some systems (Windows), may perform line-ending conversions (`\r\n` ↔ `\n`) or stop at null bytes, **corrupting your encrypted data**.
- **Binary mode (`"rb"`)**: Reads/writes bytes **exactly as they are** with no transformations, ensuring data integrity.

For this assignment:

- ✅ Encryption **output** must use `"wb"` (encrypted data is binary)
- ✅ Decryption **input** must use `"rb"` (reading encrypted binary data)
- ✅ Always use binary mode for encrypted files to ensure **cross-platform compatibility** and **data integrity**

Even though it may work with `"r"` on Linux/Ocelot, always use `"rb"` for binary data as a professional best practice.

Binary File Mode Usage Example :

```c
FILE* inFile = fopen("input.bin", "rb");
if (inFile == NULL) {
    return FAILURE;
}
FILE* outFile = fopen("output.bin", "wb");
if (outFile == NULL) {
    fclose(inFile);  // Important: close first file!
    return FAILURE;
}
// Process files...
fclose(inFile);
fclose(outFile);
return SUCCESS;
```

8. 📤 Assignment Submission Guidelines

- All assignments **must be submitted via Canvas** by the specified deadline as a **single compressed (.ZIP) file**. Submissions by any other means **WILL NOT** be accepted and graded!
- The filename must follow the format: `BONUS_FirstName_LastName.zip` (e.g., **BONUS_Harry_Potter.zip**)
- ⚠️ **Important:** Include the Firstname and Lastname of the **team's submitting member**, or your own name if **submitting solo**.
- ⚠️ **Important:** For **group submissions**, each group must designate one **corresponding member**. This member will be responsible for submitting the assignment and will serve as the primary point of contact for all communications related to that assignment.
- ⚠️ **Important Policy on Teamwork and Grading**

  All members of a team will receive the **same grade** for collaborative assignments. It is the responsibility of the team to ensure that work is **distributed equitably** among members. Any disagreements or conflicts should be addressed and resolved within the team in a **professional manner**. **If a resolution cannot be reached internally, the instructor may be requested to intervene**.
- ⚠️ **Important:** Your implementation **must be compiled and tested on the course designated Ocelot server** (`ocelot-bbhatkal.aul.fiu.edu`) **to ensure correctness and compatibility**. **Submissions that fail to compile or run as expected on Ocelot will receive a grade of ZERO.**
- Whether submitting as a **group or individually**, every submission must include a **README** file containing the following information:
    1. Full name(s) of all group member(s)

2. **Prefix the name of the corresponding member with an asterisk** `*`
3. Panther ID (PID) of each member
4. FIU email address of each member
5. Include any relevant notes about compilation, execution, or the programming environment needed to test your assignment. **Please keep it simple—avoid complex setup or testing requirements.**

- ⚠️ **Important - Late Submission Policy:** For each day the submission is delayed beyond the **Due date** and until the **Until date** (as listed on Canvas), a **10% penalty** will be applied to the total score. **Submissions after the Until date will not be accepted or considered for grading.**
    - **Due Date**: ANNOUNCED ON CANVAS
    - **Until Date**: ANNOUNCED ON CANVAS
- Please refer to the course syllabus for any other details.

9. 📦 Deliverable Folder (`ZIP file`) Structure - ⚠️ exact structure required - no additional files/subfolders

> ⚠️ **DO NOT submit any other files other than these required files. The batch autograder may treat any additional items in the ZIP file as invalid, which will result in a grade of zero: encryptFile.c, decryptFile.c, README.txt**
>
> 📁 All other required framework files (encrypt.h, driver.c, Makefile, test data) will be supplied by the instructor to the autograder during final grading.

- ⚠️ You are required to submit your work as **a single ZIP archive file**.
- ⚠️ **Filename Format**: your `BONUS_Firstname_Lastname.zip` (**exact format required**).
- **You will be held responsible for receiving a ZERO grade if the submission guidelines are not followed**.

```
BONUS_FirstName_LastName.zip
├── encryptFile.c       # Function 1: XOR-based encryption
├── decryptFile.c       # Function 2: XOR-based decryption
└── README.txt          # Team details (see submission guidelines)
```

10. 💯 Grading Rubric

Criteria	Marks
⚠️ **Program fails to run or crashes during testing**	**0**
⚠️ **Partial Implementation** *(Evaluated based on completeness and at the instructor's discretion)*	**0 - 90**

Detailed Breakdown of 100 Points

Autograding Component	Points
Simple Test (2×2 matrix, 11 bytes, basic encryption/decryption cycle)	20
Moderate Test (3×3 matrix, 97 bytes, sentence-level complexity)	30
Rigorous Test (4×4 matrix, 504 bytes, paragraph-level complexity)	40
AUTOGRADER TOTAL	**90**
Manual Grading: Clear code structure, Meaningful comments, well-written README, Overall readability, Following the submission guidelines	10
FINAL TOTAL	**100**

⚠️ **Note**: The autograder awards 90 points automatically based on correctness. The remaining 10 points are manually graded for code quality, comments, structure, and README completeness.

Good luck! 💪