Chacha20 application in secure file encryption system on personal computer

Author: Trần Gia Khánh

ID: 23021599

University of Engineering and Technology - Vietnam National University

1. Introduction

In today's digital age, the security of sensitive information is paramount. With the increasing amount of data being stored and transferred electronically, robust encryption methods are essential to protect against unauthorized access and data breaches. This project focuses on implementing a secure file encryption system using the ChaCha20 encryption algorithm. The primary objective is to provide a reliable and efficient solution for encrypting and decrypting files, ensuring the confidentiality and integrity of the data.

2. Background

Introduction to ChaCha20 Encryption algorithm

ChaCha20 is a stream cipher designed by Daniel J. Bernstein, known for its simplicity, speed, and security. It is an improved variant of the Salsa20 cipher and has been widely adopted due to its resistance to cryptographic attacks and its performance advantages over other encryption algorithms like AES. ChaCha20 operates on 512-bit blocks and uses a 256-bit key and a 96-bit nonce, making it suitable for a wide range of applications, including secure communications and data storage.

Comparison with Other Encryption Algorithms

Compared to traditional encryption algorithms such as AES (Advanced Encryption Standard), ChaCha20 offers several advantages:

- Performance: ChaCha20 is designed to be faster and more efficient, especially on systems without hardware acceleration for AES.
- Security: ChaCha20 has been extensively analyzed and is considered secure against known cryptographic attacks.
- Simplicity: The algorithm's design is straightforward, making it easier to implement and less prone to implementation errors.

Use Cases and Application of ChaCha20

ChaCha20 is used in various

ChaCha20 is used in various applications where security and performance are critical. Some notable use cases include:

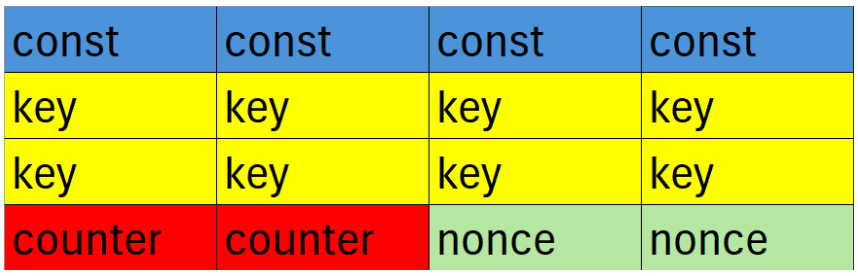
- Secure communications: ChaCha20 is used in protocols like TLS (Transport Layer Security) to encrypt data transmitted over the internet.
- Data storage: It is employed in encrypting files and databases to protect sensitive information.
- Cryptographic libraries: Many modern cryptographic libraries include ChaCha20 as a standard encryption option due to its robustness and efficiency.

This project leverages the strengths of ChaCha20 to create a secure file encryption system, providing users with a reliable tool for protecting their data

3. ChaCha20 algorithm

Like Salsa20, ChaCha's initial state includes a **128**-bit constant, a **256**-bit key, a **32**-bit counter and a **96**-bit nonce. All arranged as a 4×4 matrix of 32-bit words.

This is initial state of ChaCha:



const	const	const	const
key	key	key	key
key	key	key	key
counter	counter	nonce	nonce

- const: Values equal to fixed constants.
- key: 256-bit key.
- counter: Used to count blocks, ensuring no repetition.
- nonce: 96-bit nonce, unique calue for each iteration.

The algorithm processes the input through a series of 20 rounds, each round consisting of four quarter-round operations. Each quarter-round operation involves a series of additions, XORs, and bitwise rotations. The quarter-round function operates on four 32-bit words and is defined as follows:

1. Add the first word to the second word.
2. XOR the second word with the first word and rotate it left by 16 bits.
3. Add the third word to the fourth word.
4. XOR the fourth word with the third word and rotate it left by 12 bits.
5. Add the first word to the second word.
6. XOR the second word with the first word and rotate it left by 8 bits.
7. Add the third word to the fourth word.
8. XOR the fourth word with the third word and rotate it left by 7 bits.

Implementation method with C++:

```
inline uint32_t rotate(uint32_t v, uint32_t c) {
    return (v << c) | (v >> (32 - c));
}

inline void quarterRound(uint32_t& a, uint32_t& b, uint32_t& c, uint32_t& d) {
    a += b; d ^= a; d = rotate(d, 16);
    c += d; b ^= c; b = rotate(b, 12);
    a += b; d ^= a; d = rotate(d, 8);
    c += d; b ^= c; b = rotate(b, 7);
}
```

I have a Salsa20 method here:

```
inline uint32_t rotate(uint32_t v, uint32_t c) {
    return (v << c) | (v >> (32 - c));
}

inline void quarterRound(uint32_t& a, uint32_t& b, uint32_t& c, uint32_t& d) {
    b ^= rotate(a + d, 7);
    c ^= rotate(b + a, 9);
    d ^= rotate(c + b, 13);
    a ^= rotate(d + c, 18);
}
```

Notice that ChaCha20's version updates each word twice, while Salsa20's quater round updates each word only once. In addition, ChaCha quater round diffuses changes more quickly. On average, after changing 1 input bit, the Salsa20 quater round will change 8 output bits while ChaCha will change 12.5 output bits.

ChaCha arranges the sixteen 32-bit words in a 4×4 matrix. If we index the matrix elements from 0 to 15:

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

We will quater round each column in order: [0, 4, 8, 12] → [1, 5, 9, 13] → [2, 6, 10, 14] → [3, 7, 11, 15].

Double round in ChaCha is: [0, 5, 10, 15] → [1, 6, 11, 12] → [2, 7, 8, 18] → [3, 4, 9, 14].

```
quarterRound(block[0], block[4], block[8], block[12]);
quarterRound(block[1], block[5], block[9], block[13]);
quarterRound(block[2], block[6], block[10], block[14]);
quarterRound(block[3], block[7], block[11], block[15]);

quarterRound(block[0], block[5], block[10], block[15]);
quarterRound(block[1], block[6], block[11], block[12]);
quarterRound(block[2], block[7], block[8], block[13]);
quarterRound(block[3], block[4], block[9], block[14]);
```

ChaCha20 uses 10 iterations of the double round. Here is an implementation in C++:

```
for (int i = 0; i < 10; ++i) {
    quarterRound(block[0], block[4], block[8], block[12]);
    quarterRound(block[1], block[5], block[9], block[13]);
    quarterRound(block[2], block[6], block[10], block[14]);
    quarterRound(block[3], block[7], block[11], block[15]);

    quarterRound(block[0], block[5], block[10], block[15]);
    quarterRound(block[1], block[6], block[11], block[12]);
    quarterRound(block[2], block[7], block[8], block[13]);
    quarterRound(block[3], block[4], block[9], block[14]);
  }
```

4. System architecture

High-level Architecture

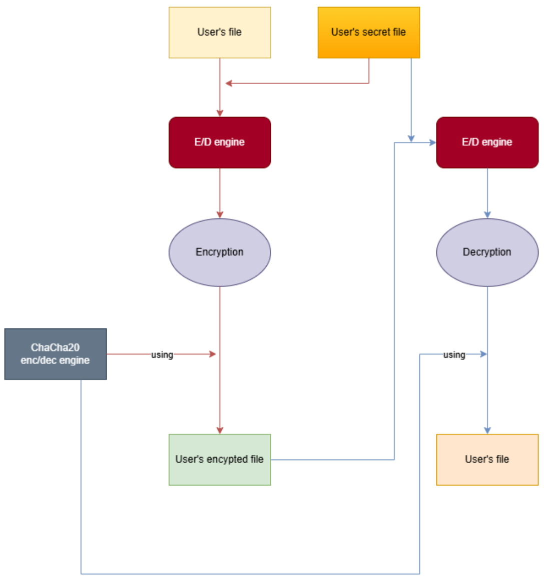
The secure file encryption system using ChaCha20 features a modular architecture that ensures flexibility, scalability, and easy maintenance. Several key components work together seamlessly to handle encryption and decryption. The main components of the architecture include:

- **User Interface (UI)**: A graphical user interface (GUI) that lets users interact with the system, select files for encryption or decryption, and monitor operation status.
- **Encryption/Decryption Engine**: The core component that implements the ChaCha20 algorithm to handle file encryption and decryption.
- **Key Management**: This module will use the user's input secret string to generate the key and nonce used for encryption, decryption.
- **File I/O**: Handles file reading and writing operations, ensuring secure storage and retrieval of encrypted data.
- **Error Handling and Logging**: Manages errors and logs activities for auditing and debugging purposes.

Components and Their Interactions

1. **User Interface (UI)**
   - **Description**: The UI offers an intuitive interface for system interaction. Users can select files, start encryption or decryption operations, and monitor progress.
   - **Interactions**: The UI communicates with the Encryption/Decryption Engine to execute commands and display results, while coordinating with Key Management for key operations.
2. **Encryption/Decryption Engine**
   - **Description**: This core component implements ChaCha20 for file encryption and decryption. It processes plaintext files with encryption keys to generate ciphertext, and vice versa.
   - **Interactions**: The engine works with File I/O for reading and writing files, while coordinating with Key Management to obtain keys and nonces.
3. **Key Management**
   - **Description**: Handles the creation and transformation of the secret string into a 256-bit key and a 96-bit nonce. This is to prevent and increase security, preventing unauthorized access
   - **Interactions**: It supplies keys and nonces to the Encryption/Decryption Engine while providing key management features through the UI.

4. **File I/O**
   - **Description**: This module manages file operations, ensuring secure storage and retrieval of encrypted data while maintaining file integrity.
   - **Interactions**: It coordinates with the Encryption/Decryption Engine for file processing and interfaces with the UI for file selection.
5. **Error Handling and Logging**
   - **Description**: This component tracks system activities and manages errors, ensuring proper handling and recording of any issues during operations.
   - **Interactions**: It monitors all components to log activities and errors, providing user feedback and developer debugging information through the UI.



5. Implementation Details

ChaCha20 encryption/decryption engine

The core of the system is implemented in C++ using the ChaCha20 algorithm. The main function responsible for both encryption and decryption is `chachaProcess` .

```
void chachaProcess(const uint32_t initialState[16],const vector<uint8_t>& input,vector<uint8_t>& output
    output.resize(input.size());
    uint32_t state[16];
    uint32_t block[16];
    vector<uint8_t> keystream(64);
    size_t offset = 0;

    memcpy(state, initialState, sizeof(state));

    while (offset < input.size()) {
        // Generate keystream
        memcpy(block, state, sizeof(state));
        for (int i = 0; i < 10; ++i) {
            quarterRound(block[0], block[4], block[8], block[12]);
            quarterRound(block[1], block[5], block[9], block[13]);
            quarterRound(block[2], block[6], block[10], block[14]);
            quarterRound(block[3], block[7], block[11], block[15]);

            quarterRound(block[0], block[5], block[10], block[15]);
            quarterRound(block[1], block[6], block[11], block[12]);
            quarterRound(block[2], block[7], block[8], block[13]);
            quarterRound(block[3], block[4], block[9], block[14]);
        }

        for (int i = 0; i < 16; ++i) {
            block[i] += state[i];
            keystream[i * 4 + 0] = block[i] & 0xFF;
            keystream[i * 4 + 1] = (block[i] >> 8) & 0xFF;
            keystream[i * 4 + 2] = (block[i] >> 16) & 0xFF;
            keystream[i * 4 + 3] = (block[i] >> 24) & 0xFF;
        }
        size_t bytesToProcess = min<size_t>(64, input.size() - offset);
        for (size_t i = 0; i < bytesToProcess; ++i) {
            output[offset + i] = input[offset + i] ^ keystream[i];
        }

        ++state[12];
        offset += bytesToProcess;
    }
}
```

This function takes an initial state, an input vector and an output vector to store the result.

Key and Nonce Generation

In this implementation, the key and nonce are derived from a secret string entered by the user. From that string, we can understand that the key and nonce are created by own. This approach ensures that 256-bit key and 96-bit nonce are unique and can be easily reproduced by the user. The process involves the following steps:

1. **User Input**: The user is prompted to enter a secret string through the GUI.
2. **Derivation Function**: The secret string is passed to a derivation function that generates the key and nonce.
3. **Hashing**: The derivation function uses a cryptographic hash function (e.g., SHA-256) to hash the secret string.
4. **Key Extraction**: The first 32 bytes of the hash output are used as the 256-bit key.
5. **Nonce Extraction**: The next 12 bytes of the hash output are used as the 96-bit nonce.

In this part, using Python beside C++:

```
def derive_key_and_nonce(input_string):
    if not input_string:
        raise ValueError("Input string must not be empty.")
    hash_object = hashlib.sha256(input_string.encode())
    hash_bytes = hash_object.digest()
    key = hash_bytes[:32]
    nonce_hash_object = hashlib.sha256((input_string + "nonce").encode())
    nonce_bytes = nonce_hash_object.digest()
    nonce = nonce_bytes[:12]
    return key, nonce
```

Integration with the C++ Executable

The Python GUI application integrates with the C++ executable to perform the actual encryption and decryption. This is achieved through the following steps:

1. **Command Execution**: The Python application uses the `subprocess` module to execute the C++ executable with the appropriate command-line arguments (e.g., file paths, key and nonce).
2. **Input and Output Handling**: The Python application passes the input file path and other parameters to the C++ executable. It also retrieves the output file path and displays the results to the user.
3. **Error Handling**: The Python application captures any errors or exceptions from the C++ executable and displays appropriate messages to the user.

This integration ensures that the user can easily perform encryption and decryption operations through a graphical interface, while the C++ implementation handles the core cryptographic processing.

6. Security Considerations

Security Features of ChaCha20

This algorithm is designed with several security features that make it a robust choice for secure file encryption:

- **High Security Margin**: ChaCha20 has a high security margin, making it resistant to various cryptographic attacks, including differential and linear cryptanalysis.
- **Simplicity**: The algorithm's design is straightforward, reducing the risk of implementation errors that could lead to vulnerabilities.
- **Speed and Efficiency**: ChaCha20 is designed to be fast and efficient, even on systems without hardware acceleration, making it suitable for a wide range of applications.

Potential Vulnerabilities and Mitigations

While ChaCha20 is a secure algorithm but security of the file encryption system depends on proper implementation and usage.

- Nonce and key reuse: Reusing a nonce and a key, generate from user’s secret string can lead to serious issues, including the possibility of plaintext recovery.
- Weak secret key: A user's secret string that is too weak can lead to being too predictable, which can be predicted by a bad actor using calculation or string generation methods to get the secret string, thereby getting the file.

Best Practices for Secure Usage

To maximize the security of the file encryption system, users can follow these best practices:

- Use strong secret key: make a complicated string, including special characters like ‘!’, ‘@’, ‘#’,...
- Update the system core: In this system, we can replace ChaCha20 by another encryption algorithm like AES-GCM (Galois Counter Mode),... Beside, we can combined it as one.

7. Conclusion

This project primarily focuses on the ChaCha20 algorithm, how it works, and its application in a secure file encryption system. Additionally, it involves the combination of Python for the GUI and C++ for implementing the algorithm.

The ChaCha20 algorithm was chosen for its simplicity, speed, and security. It has been extensively analyzed and is considered secure against known cryptographic attacks. By deriving the encryption key and nonce from a user-provided secret string, we ensured that the system is both secure and user-friendly.

In summary, the secure file encryption system using ChaCha20 provides a high level of protection for sensitive data. It is a reliable, efficient, and user-friendly solution that meets modern data security needs. I believe this application will help people safeguard their important files on their personal computers.