Cryptography: Encrypting files with AES

# Abstract

In today’s world, where security, privacy and integrity of information is a prime necessity, modern cryptography plays a vital role in ensuring this and protection from the vast range of cyber-attacks.

In this project we try to protect the files shared on a cloud server from the adversary attacks. We use AES symmetric encryption schemes for security and cryptographic hash functions for authentication (explained below in detail).

# Highlights

* **Security/Privacy**: AES block encryption with OFB mode of operation
* **Authentication:**  SHA-2 cryptographic hash function
* **Secure Server storage:** Amazon S3 server

# Tools/Libraries

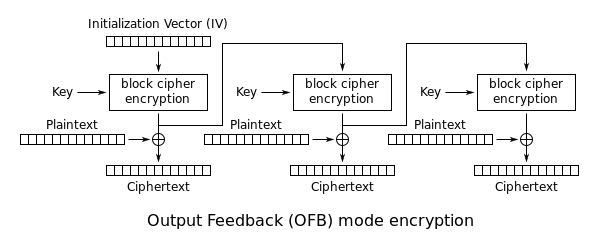
* OpenSSL Library
* Codeblocks development environment

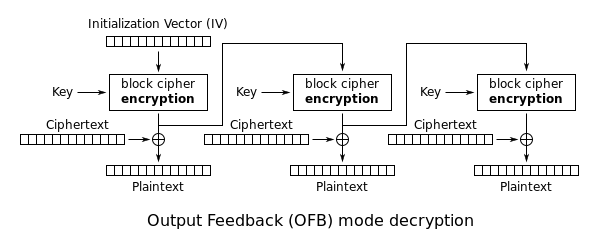
# Implementation approach

# A. Cryptographic primitives used

1. **AES block encryption:** We use symmetric block encryption scheme based on problem requirements and assumption that the key is shared between trusted parties in a secure manner. Since we are working with files, which are large amounts of information, symmetric scheme allows time-efficient encryption/decryption constructions compared to asymmetric schemes.
2. **Block ciphers** are computationally secure and more time-efficient solution for symmetric encryption. But they need to be designed carefully taking into account the confusion-diffusion paradigm and avalanche effect. Otherwise they can be easily broken. We use the standard open-ssl library which takes care of the number of rounds, block length etc internally, ensuring security.
3. **Output Feedback mode of operation (OFB):** A mode of operation describes how to repeatedly apply a cipher's single-block operation to securely transform amounts of data larger than a block. We use the OFB mode because it makes a block cipher into a synchronous stream cipher called the *keystream blocks*. This is a sequential process since encrypting/decrypting each block depends on previous block (or IV in the case of first block). This *keystream* is then XORed with the plaintext blocks to get the ciphertext. Thus the last step of XORing the plaintext (during encryption) or cipthertext (during decryption) can be performed in parallel as and when the streams are available.

Enc/dec process figure courtesy [Wikipedia](http://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Output feedback):



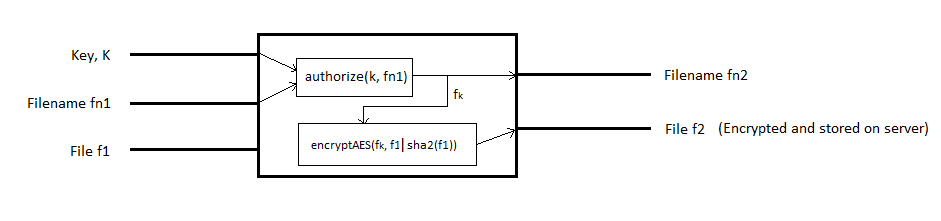


1. **SHA-2 cryptographic hash function:** An n-bit cryptographic hash is a map from arbitrary length messages to n-bit hash values with additional properties: one-wayness and collision-resistance. It is used for authentication purposes and keeps in check the integrity of the message. We use the 256-bit cryptographic hash.

The compression function operates on a 512-bit message block and a 256-bit intermediate hash value. It is essentially a 256-bit block cipher algorithm which encrypts the intermediate hash value using the message block as key. It is meant to provide 128 bits of security against collision attacks which would suffice our authentication purposes.

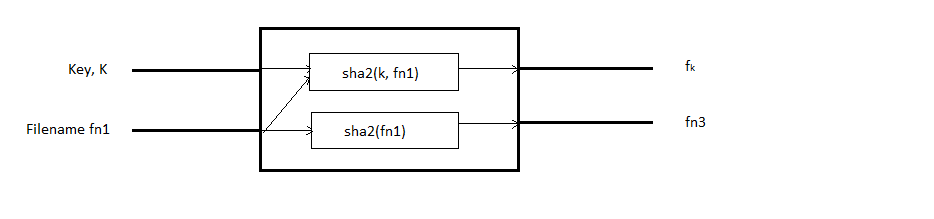
# B. Implementation schematics

Pre-process:



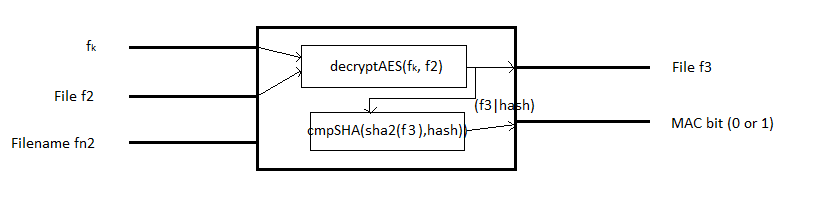
The filename fn2 generated, is the hash value of fn1. The key fk is generated by a combination of k (unique to each user) and fn1 and is thus unique for each file. In *encryptAES*, the file f1 is first appended by its hash value and then encrypted to get f2.

Authorize:



Generates key fk, also used by *encryptAES.* Generates fn3 (equal to fn2), which is used to uniquely locate the file on the server.

Recover:



*decryptAES* produces f3, the decrypted file and its integrity is checked by comparing the hash value of f3 and *hash*. If the values match, the file is authentic.

# Justifying security notions

* **Correctness:**

Clause 1:  f3=f1. During encryption, file f1 is encrypted (Block AES, OFB mode) using a key fk and stored in the cloud server (f2). During decryption this encrypted file (f2) is decrypted using the decryption algorithm from the same scheme and key fk producing file f3. Thus theoretically, f3 = f2 and can be verified using *file-diff.*

Clause 2:  fn3=fn2. During encryption (which produces fn2) and during authorization (which produces fn3) the same module is called with same inputs, ensuring fn2 = fn3.

* **Privacy against server attacks:**

The access rights have to be set in the server. The file owner will have “*rwx*” access permissions whereas the others will have only “*r*” access permission.

* **Authentication and Message integrity**

As explained in previous section, the encrypted file f2 contains the encrypted data of file content + SHA2 hash of the file. After decryption with fk the integrity of the file content can be checked by recalculating the SHA2 hash of the file content (minus the original hash string) and comparing it with the appended hash string. If the hash values do not match the file has been compromised.

# Work distribution

* **Manas**
  + 1. Security scheme designing
    2. Message Authentication schemes and modules
    3. Documentation and presentation
* **Darshan**
  + 1. Security scheme designing
    2. OpenSSl library integration
    3. Code integration

The detailed steps of how to run all programs is provided with the software as *readme.md*