Project 1: Encrypted File System Report

Charles Averill

charles@utdallas.edu

Field	Length (bits)	Start	End
Username	1024	0	1023
Salt	1024	1024	2047
AES-CTR IV	128	2048	2175
HMAC	256	2176	2431

Table 1. Public metadata fields

Field	Length (bits)	Start	End
File length	32	0	31
Password hash	256	32	287

Table 2. Secret metadata fields

1. Design

1.1. Metadata

Metadata is stored in a file named `meta` in the file's directory. Public fields are stored as described in Table 1. Secret fields are assembled as described in Table 2, hashed using AES-ECB, and stored directly following the public fields.

1.2. User Authentication

The encrypted file's password salt is stored publicly as described in Table 1. Its password hash is encrypted via AES-ECB and stored as described in Table 2.

When performing actions requiring authentication, the salt is retrieved from public metadata. The current user's plaintext password is concatenated with the salt and hashed with SHA256. This hash is then used as a decryption key with AES-ECB to decrypt the file's secret metadata. The stored hash is then compared against the computed hash, and authentication succeeds if the two are identical.

1.3. Encryption

When storing files, the full file plaintext is split into blocks of 1KB each. Each of these blocks is encrypted via AES-CTR as described in Algorithm 1. Because AES-ECB is only being called on the initialization vectors, index, and password hash, AES-CTR functions as both the encryption and decryption procedure, as each of these fields is consistent in both scenarios.

This implementation is secure, even if an adversary has access to all of the encrypted blocks, because each of the subsequences of length 16 in the plaintext is encrypted with the index of the subsequence. This ensures that blocks of similar plaintext have unique encrypted equivalents, preventing an attacker from extracting secret information about the encrypted file by aggregating knowledge of each block.

Algorithm 1 Implementation of the AES-CTR algorithm using AES-ECB as a backend

▷ *Performed on each 16-byte subsequence of AES-CTR's input array*
procedure AES-CTR-BLOCK($arr, hash, IV, idx$)
  $IV \leftarrow$ ENCRYPTAES$_{ECB}(IV + idx, hash)$
  return $IV \oplus arr$
▷ *Used for both encryption and decryption*
procedure AES-CTR($arr, hash, IV$)
  $out \leftarrow []$
  for $B =$ sub-array of length 16 $\in arr$ **do**
    $out \leftarrow out \|$AES-CTR-BLOCK$(B, hash, IV, B.idx)$
  return out

1.4. File Length Concealment

With my encryption scheme, the length of a file can only be known within a range of 1KB. This is because plaintext is padded before being encrypted via AES-CTR, meaning each block of the file has the same length when encrypted.

1.5. Message Authentication

Message authentication is accomplished with a standard hash-based message authentication code (HMAC) implementation. This implementation is described in Algorithm 2.

After modifications to file contents via WRITE and CUT, the value of the HMAC stored in secret metadata is recomputed using the new contents of the file.

Algorithm 2 Message Authentication Scheme

Require: $hash.length = 32$ ▷ *SHA256 key length in bytes*
procedure HMAC($message, hash$)
  $okey \leftarrow key \oplus$ `0x5C`
  $ikey \leftarrow key \oplus$ `0x36`
  return SHA256($okey \|$SHA256$(ikey, message)$)

1.6. Efficiency

CREATE and LENGTH are $O(1)$. All loops within encryption functions are performed on constant-length structures and known values.

READ is $O(len)$. Password verification and length checks are constant time as in CREATE and LENGTH. Its loop is dependent on the number of blocks being read, which is computed from len.

WRITE is $O(len +$ LENGTH$(filename, password))$. Its main loop is dependent on the number of blocks being read or written, which is computed from len. The length update in WRITE is constant-time, and the integrity value update is $O($LENGTH$(filename, password)$, as it reads the entire file's contents via READ and calls SHA256 on them.

CHECKINTEGRITY is $O($LENGTH$(filename, password))$. It reads the entire file's contents via READ, and then performs a constant-time check against the stored integrity value.

CUT is $O($LENGTH$(filename, password))$. Its core behavior is actuall $O(1)$ as it only ever modifies one block of the file. However, its update to the stored integrity value in the file's metadata relies on reading the contents of the entire file.

The implementation I have chosen is maximally storage-efficient, as it only stores as much data as is required to decrypt and retrieve the plaintext.

An implementation that reaches maximum speed efficiency must necessarily sacrifice security. Backtracking to use AES-ECB for the plaintext encryption and sacrificing message authentication allows for an $O(1)$ CUT and an $O(len)$ WRITE.

One update that could improve my speed efficiency would be to compute HMAC on the ciphertext of the encrypted file, rather than decrypting and re-hashing the plaintext. I chose to perform HMAC the less-efficient way because it reduces the complexity of my code, reducing the risk that the code is incorrect.

2. Pseudocode

Algorithm 3 File Creation

```
procedure CREATE(filename, username, password)
    ▷ Create folder for blocks and metadata file
    CREATEFOLDER(filename)
    meta ← CREATEFILE(filename/"meta")
    ▷ Generate metadata
    salt ← random()
    iv ← random()
    hash ← SHA256(salt||password)
    ▷ Write public metadata
    WRITE(username, meta)
    WRITE(salt, meta)
    WRITE(iv, meta)
    WRITE(HMAC([], hash), meta)
    ▷ Write secret metadata (file is currently length 0)
    WRITE(ENCRYPTAES_ECB(0||hash), meta)
```

Algorithm 4 Length Extraction

```
procedure LENGTH(filename, password)
    salt ← read salt from public metadata
    hash ← SHA256(salt||password)
    return DECRYPTAES_ECB(secret metadata, hash).length
```

Algorithm 5 Perform a check on the integrity of the encrypted file contents

```
procedure CHECKINTEGRITY(filename, password)
    salt ← read salt from public metadata
    hash ← SHA256(salt||password)
    filehash ← DECRYPTAES_ECB(secret metadata, hash).hash
    ▷ Verify password
    if hash ≠ filehash then
        return PASSWORDINCORRECTEXCEPTION
    ▷ Compare stored HMAC and computed HMAC values
    HMAC_file ← read HMAC from public metadata
    len ← LENGTH(filename, password)
    contents ← READ(filename, 0, len, password)
    HMAC_current ← HMAC(contents, hash)
    return HMAC_file = HMAC_current
```

Algorithm 6 Truncate a file

```
procedure CUT(filename, length, password)
    salt ← read salt from public metadata
    hash ← SHA256(salt||password)
    filehash ← DECRYPTAES_ECB(secret metadata, hash).hash
    IV ← read IV from public metadata
    ▷ Verify password
    if hash ≠ filehash then
        return PASSWORDINCORRECTEXCEPTION
    ▷ Read, decrypt, truncate, encrypt, and write new end block
    block_end ← length / size_block
    len_end ← length mod size_block
    block_plain ← DECRYPTAES_CTR(block_end, hash, IV)
    block_truncated ← block_plain[0 upto len_end]
    WRITE(ENCRYPTAES_CTR(block_truncated, hash, IV), block_end)
    ▷ Update length and integrity metadata
    WRITE(ENCRYPTAES_ECB(length||hash), meta)
    contents ← READ(filename, 0, length, password)
    WRITE(HMAC(contents, hash), meta)
```

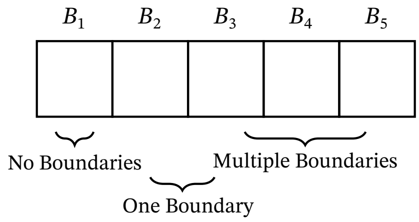

Figure 1. All possible read/write accesses

Algorithm 7 Reading Encrypted Data

```
procedure READ(filename, start, len, password)
    salt ← read salt from public metadata
    hash ← SHA256(salt||password)
    filehash ← DECRYPTAES_ECB(secret metadata, hash).hash
    IV ← read IV from public metadata
    ▷ Verify password, read bounds
    if hash ≠ filehash then
        return PASSWORDINCORRECTEXCEPTION
    else if start + len > LENGTH(filename, password) then
        return OUTOFBOUNDSEXCEPTION
    block_start, block_end ← start / size_block, (start+len) / size_block
    ▷ Decrypt and read one block at a time
    out ← []
    for i = block_start to block_end do
        block_plain ← DECRYPTAES_CTR(block_i, hash, IV)
        Compute range of read based on cases in Fig. 1
        out ← out||block_plain[range]
    return out
```

Algorithm 8 Writing Encrypted Data

```
procedure WRITE(filename, start, plain, password)
    salt ← read salt from public metadata
    hash ← SHA256(salt||password)
    filehash ← DECRYPTAES_ECB(secret metadata, hash).hash
    IV ← read IV from public metadata
    ▷ Verify password, write bounds
    if hash ≠ filehash then
        return PASSWORDINCORRECTEXCEPTION
    else if start + plain.length > LENGTH(filename, password) then
        return OUTOFBOUNDSEXCEPTION
    if LENGTH(filename, password) = 0 then ▷ File is empty
        for B = sub-array of length size_block ∈ plain do
            f ← CREATEFILE(filename/B.idx)
            WRITE(ENCRYPTAES_CTR(B, hash, IV), f)
    else ▷ File already contains data
        block_start, block_end ← start / size_block, (start+len) / size_block
        ▷ Read, decrypt, update, encrypt, write one block at a time
        for i = block_start to block_end do
            block_plain ← DECRYPTAES_CTR(block_i, hash, IV)
            Compute range of write based on cases in Fig. 1
            block_updated ← OVERWRITE(block_plain, plain, range)
            WRITE(ENCRYPTAES_CTR(block_updated, hash, IV), block_i)
    ▷ Update length and integrity metadata
    len ← max(LENGTH(filename, password), start+plain.len)
    WRITE(ENCRYPTAES_ECB(len||hash), meta)
    contents ← READ(filename, 0, len, password)
    WRITE(HMAC(contents, hash), meta)
```

3. Variations

1. **Suppose that the only write operation that could occur is to append at the end of the file. How would you change your design to achieve the best efficiency (storage and speed) without affecting security?**
   If the only write operation is an append, this update remains $O(len + \textsc{Length}(filename, password))$, however it is a tighter upper bound. This is because instead of having to decrypt part of the file, update it, re-encrypt it, and write it back to the file, we need only encrypt the appended text and append it to the existing ciphertext, without sacrificing security.
2. **Suppose that we are concerned only about adversaries who steal the disks. That is, the adversary can read only one version of the same file. How would you change your design to achieve the best efficiency?**
   If an adversary is only concerned with stealing disks, we can eliminate message authentication entirely, as there is no chance of files being manipulated (short of cosmic ray bit flips and failing memory, which are mitigated by ECC memory and maintenance). This results in an $O(len)$ WRITE and an $O(1)$ CUT.
3. **Can you use the CBC mode in the EFS? If yes, describe how your design would change and analyze the efficiency of the resulting design. If no, describe why.**
   Yes, CBC mode could be utilized in the EFS, however with significant drawbacks:
   - Overwriting, which was $O(len)$ is now $O(\textsc{Length}(filename, password))$, as each block before the target block must be decrypted to decrypt the target block
   - Corruption in block n results in all blocks greater than n also becoming corrupted
   - CUT remains $O(\textsc{Length}(filename, password))$, but is actually $O(2\textsc{Length}(filename, password))$ because the final block must be re-encrypted
4. **Can you use the ECB mode in the EFS? If yes, describe how your design would change and analyze the efficiency of the resulting design. If no, describe why.**
   No, ECB mode cannot be utilized in the EFS without sacrificing security requirements. Utilizing ECB mode would reveal massive amounts of information about the structure of the plaintext, as identical blocks would have identical ciphertexts.