CS427 Final Project - Stream File Encryption & Key Management

Casey Colley Robert Detjens Lyell Read

CS 427, Winter 2022

Contents

Abstract

To demonstrate proficient Cryptographic knowledge gained from this course, our group designed a NOISE: Nice Ol' Interactive Stream Encryption. This is basically a key manager with built in encryption/decryption of messages with the keys within the manager. This project demonstrated: knowledge of the different definitions of security for different components (encryption schemes, pseudorandom permutations, and compression functions), ability to research further on what was taught in class (to find a suitable PRP, which ended up being AES), ability to write proofs for our cryptographic scheme, and ability to implement in code and test our cryptographic scheme. We wrote this program in Python and utilized a number of different libraries: PyAES, getpass, and docopt.

NOTE: Our Python scripts have a few small bugs that we were unable to iron out prior to submission, and will be broken if you try to run them. Please feel free to read through the code however! We started implementing our modified CTR but ran into complications. We have decided to use regular non-CCA secure CTR and add a MAC to ensure CCA security, but this has not been implemented yet.

Definitions

Stream Encryption and Decryption (`enc`, `dec`)

These define the Encryption and Decryption algorithms used by the program both to encrypt and decrypt the Master Key, and to encrypt and decrypt messages *with* the Master Key.

Primitives

Our design utilizes a secure block cipher/PRP, F. F will be the AES block cipher with a 128-bit key. Our program utilizes a Python library for the AES block cipher implementation called PyAES.

klen = 128

$\underline{F_{AES}(k, d):}$
BLACK BOX

$\underline{F^{-1}_{AES}(k, d):}$
BLACK BOX

Formal Scheme Definition

Our symmetric encryption mode will be a modified CTR mode. For the Decryption algorithm, we don't really need to have r or c_0 as it's simply concatenated with m_i but we have kept it in the definition to better show how our modification resembles true CTR mode. Additionally, r could be used to verify that the decryption was successful, as the resulting block would be $m_i||r$.

blen = 128

$\underline{\text{ENC}_{CTR}(k, m_1||...||m_l):}$
$r \leftarrow \{0,1\}^{blen}$
$c_0 := r$
for $i = 1$ to l:
  $c_i := F(k, m_i||r)$
  $r := r + 1 \% 2^{blen}$
return $c_0||...||c_l$

$\underline{\text{DEC}_{CTR}(k, c_0||...||c_l):}$
$r := c_0$
for $i = 1$ to l:
  $m_i := F^{-1}(k, c_i)$ [blen:]
  $r := r + 1 \% 2^{blen}$
return $m_1||...||m_l$

Security Proof and Reasoning

We will prove that the encryption scheme of our key manager, a modified CTR mode, has security against chosen ciphertext attacks. We assume that F is a secure PRP.

To prove that a scheme has CCA security, we must prove that two random plaintexts (L & R) cannot be distinguished from each other, including any partial information, like so:

$\mathcal{L}^{\Sigma}_{\text{CCA-L}}$

$k \leftarrow \Sigma.\mathsf{KeyGen}$
$\mathcal{S} := \emptyset$

$\underline{\textsc{Eavesdrop}(m_L, m_R) :}$
if $|m_L| \neq |m_R|$:
  return err
$c := \Sigma.\mathsf{Enc}(k, m_L)$
$\mathcal{S} := \mathcal{S} \cup c$
return c

$\underline{\textsc{Decrypt}(c) :}$
if $c \in S$ return err
return $\Sigma.\mathsf{Dec}(k, c)$

$\approx$

$\mathcal{L}^{\Sigma}_{\text{CCA-R}}$

$k \leftarrow \Sigma.\mathsf{KeyGen}$
$\mathcal{S} := \emptyset$

$\underline{\textsc{Eavesdrop}(m_L, m_R) :}$
if $|m_L| \neq |m_R|$:
  return err
$c := \Sigma.\mathsf{Enc}(k, m_R)$
$\mathcal{S} := \mathcal{S} \cup c$
return c

$\underline{\textsc{Decrypt}(c) :}$
if $c \in S$ return err
return $\Sigma.\mathsf{Dec}(k, c)$

From here, we will walk through the proof for the left library.

$\mathcal{L}^{\Sigma}_{\text{CCA-L}}$

$k \leftarrow \Sigma.\mathsf{KeyGen}$
$\mathcal{S} := \emptyset$

$\underline{\textsc{Eavesdrop}(m_L, m_R) :}$
if $|m_L| \neq |m_R|$:
  return err
$c := \Sigma.\mathsf{Enc}(k, m_{1L}||...||m_{lL})$
$\mathcal{S} := \mathcal{S} \cup c$
return c

$\underline{\textsc{Decrypt}(c) :}$
if $c \in S$:
  return err
return $\Sigma.\mathsf{Dec}(k, c)$

$\diamond$

$\underline{\textsc{Enc}_{CTR}(k, m_{1L}||...||m_{lL}) :}$
$r \leftarrow \{0,1\}^{blen}$
$c_0 := r$
for $i = 1$ to l:
  $c_i := F(k, m_{iL}||r)$
  $r := r + 1 \% 2^{blen}$
return $c_0||...||c_l$

$\approx$

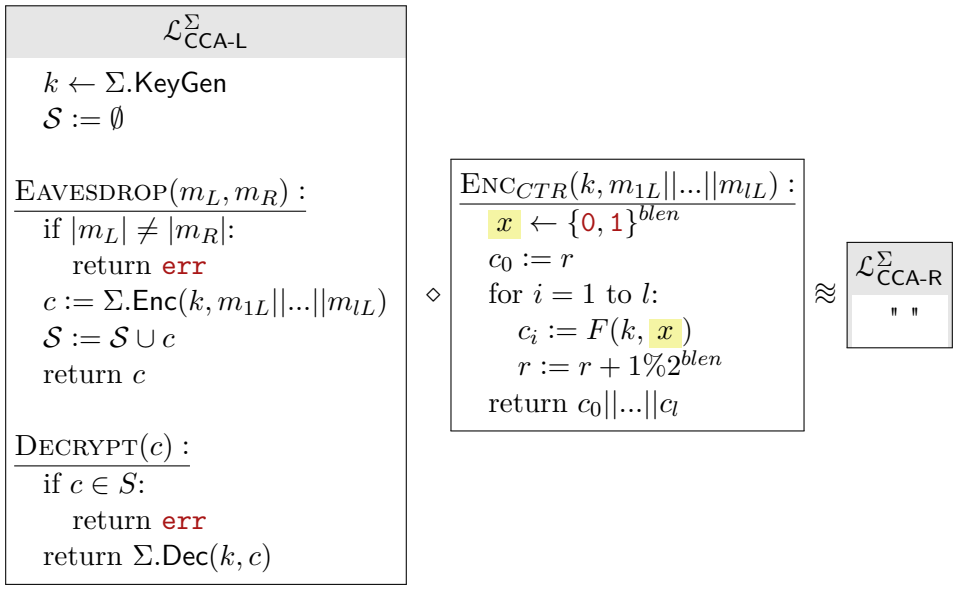
$\mathcal{L}^{\Sigma}_{\text{CCA-R}}$

" "

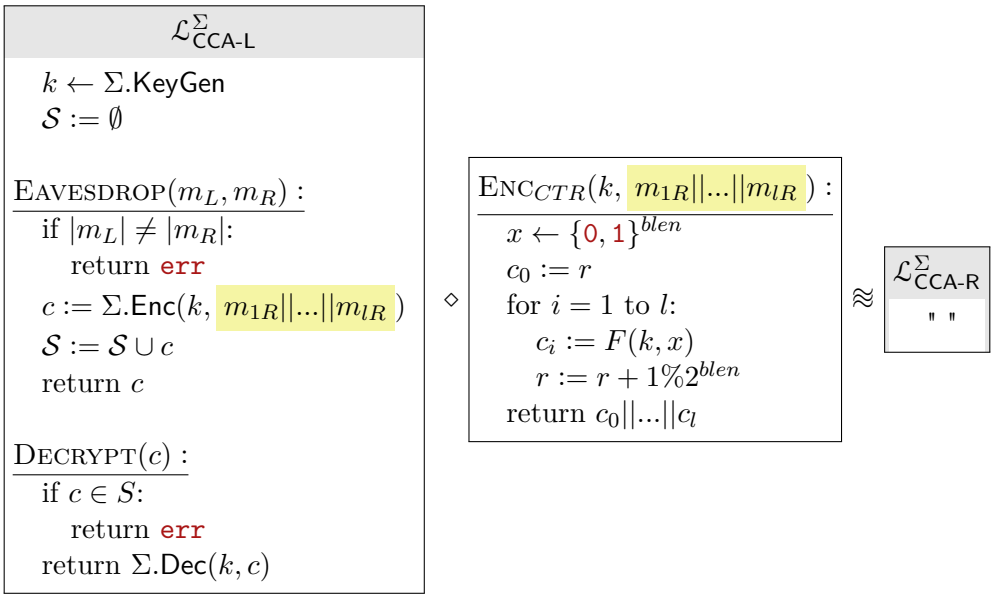
Next, we can turn our attention to the linked encryption scheme. Here we see that for each block, we calculate $F(k, m_i||r)$ for the corresponding ciphertext block. r is sampled randomly, so the chance of collision is $\frac{1}{2^{blen}}$. However, we are doing counter mode, so r for each subsequent block in the message is deterministic, for l blocks in the message. Still, the rate of collision comes to $\frac{l}{2^{blen}}$. The l increases much slower than the 2^{blen}, which means the rate of collisions is still negligible.

Because r is sampled randomly and has a neglible rate of collisions, $m_i||r$ also has a collision rate of $\frac{l}{2^{blen}}$ even when the same m_i is inputted. It does not matter what m_i is when we concatenate it

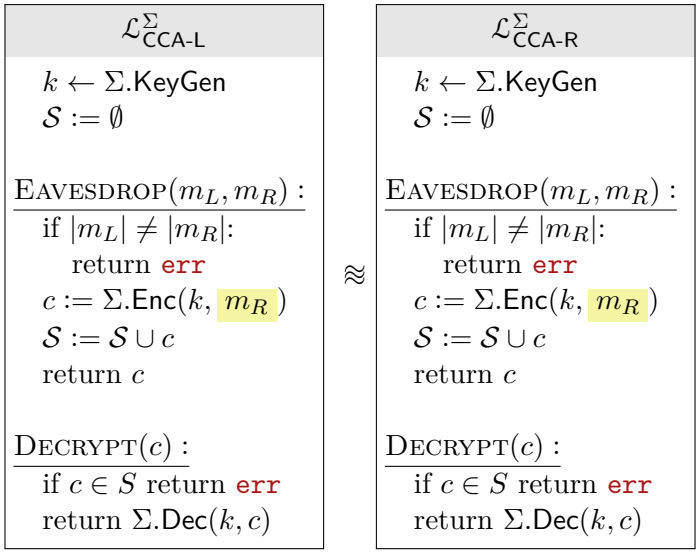
with r and put it through the PRP F. To illustrate this, we can apply the following transformation:



Now, $m_{1L}||...||m_{lL}$ is not being used by the Enc_{CTR} function; we can change it to some other name without disrupting the function of the encryption scheme. We can rename this to $m_{1R}||...||m_{lR}$ and inline it into the library.



Let's inline the whole linked function, and re-consider the right library.



Here we can see in this function, the left and right libraries are indistinguishable. For any calling program A, it will not be able to distinguish between the two libraries - aka, it will not be able to obtain any partial information from the scheme. Therefore, the scheme has CCA security, and by extension, has CPA security.

Key Generation and Storage (`keygen`)

These define the functions that handle generation and storage of keyfiles used by the program. These keyfiles are generated with the function `KeyGen`, which samples a string of length `klen`. This sampling will come from the machine's built-in random device, such as `/dev/urandom`.

These keyfiles are stored encrypted with a password that varies per keyfile. This "password encryption" is implemented by way of Password-Based Key Derivation Function 2 (PBKDF2). This will be expanded upon in the Primitives section.

The keyfiles are encrypted without a MAC as a MAC requires additional secret keys. Our goal is to encrypt with only one password-derived key, so encrypting and MAC'ing is not feasible. To compensate somewhat, a hash is appended to the keyfile before the whole keyfile is encrypted. More discussion on the security properties of this are discussed later.

Primitives

The two biggest primitives we will define here is a hash function, an HMAC function (for use as a PRF), and PBKDF2.

We will also use the F_{AES} block cipher that we defined earlier. When we use F_{AES} in the `Stream Encryption and Decryption` section, the key it takes is the "master key" that is outputted from this section. When we use F_{AES} in here, it will not be used in the same way.

Davies-Meyer compression function

The hash function we will be using is a Davies-Meyer compression function with our same block cipher, F_{AES}. The reason we are using the Davies-Meyer compression function is because we want to implement as much from the ground-up as we can, and this is a very straight-forward function.

A Davies-Meyer function uses a block cipher to produce a hash. The "keys" that are passed into the block cipher are the blocks of the message itself. This is going to be used as an intermediate function to produce an HMAC. The Davies-Meyer algorithm is defined below:

blen = 128

$\underline{\text{HASH}_{DM}(m_1||...||m_l):}$
$h := \{0\}^{blen}$
for $i = 1$ to l:
  $h := F(m_i, h) \oplus h$
return h

A block cipher is not itself a hash function but it can be used as a building block to one. The hashes produced by the Davies-Meyer function are effective, collision-resistant hashes. It is important to note that this is just one component that will be used for turning a password into a key. By itself, the Davies-Meyer function would be insufficient for that task.

HMAC (as a PRF)

Now we will build an HMAC function with our Davies-Meyer compression function. This HMAC is an intermediate function for the PBKDF2 which will be transforming the chosen password into a key which can be used to encrypt and decrypt keyfiles.

hlen = 128
opad = $0x5c^{\lambda}$
ipad = $0x36^{\lambda}$
$\text{HMAC}_{DM}(k, m)$:
if |k| > hlen:
$k := \text{HASH}_{DM}(k)$
x := $\text{HASH}_{DM}(k \oplus \text{ipad})$
y := $k \oplus$ opad
return $\text{HASH}_{DM}(y||x||m)$

PBKDF2 requires a pseudorandom function as part of its functioning. In RFC2898, an example PRF given is an HMAC. Therefore, we have defined an HMAC here utilizing our HASH_{DM} function.

Password-Based Key Derivation Function 2 (PBKDF2)

PBKDF2 is an established Key Derivation Function that will be doing the heavy lifting in turning a keyfile's password into a usable "master key" to decrypt it. This function repeatedly calls the previously-defined HMAC on the password (with a salt, etc) to generate each block of the key. After this key is generated, we will use it to decrypt the keyfile.

A few parameters are seen below. s is a salt that can be an arbitrary length (as it will be hashed down). *klen* is the desired length of the key. We can change this, however we are constricted to the key-lengths that our encryption algorithm can take, which is 128. *hlen* is the fixed length of our hash output. In this scheme, our HMAC depends on our HASH_{DM} function that spits out 128 bit output. By using the same F_{AES} for both our hashing output and our encryption, we do constrict ourselves to specific input and output lengths throughout our component functions (namely, 128 bits). c is the number of iterations that the HMAC should be applied. This should be a very large number.

```
   klen := 128
   hlen := 128
   c :=
PBKDF2(p, s):
   for i = 1 to (klen/hlen):
      U_1 := HMAC(p, s || i)
      T_i := U_1
      for j = 2 to c:
         U_j := HMAC(p, U_{i-1})
         T_i := T_i ⊕ U_j
      T_i return T
```

Formal Scheme Definition

Our program relies on three secret keys: a key for encryption and decryption of a file, and two keys to generate a MAC for the encrypted file's contents.

The encrypted keys and their hash will be kept in a file, and the decrypted keys will be extracted and used internally within the program only. This is reflected below:

KeyFile := KeyGen()

KeyGen():
- $p :=$ getpass()
- $s \leftarrow \{0,1\}^\lambda$
- $K :=$ PBKDF2(p, s)
- $key \leftarrow \{0,1\}^\lambda$
- $mac1 \leftarrow \{0,1\}^\lambda$
- $mac2 \leftarrow \{0,1\}^\lambda$
- $kh =$ Hash$_{DM}(k||mac||mac2)$
- $E :=$ Enc$_{CTR}(K, key||mac1||mac2||kh)$
- return $E||salt$

DecryptKey():
- $p :=$ getpass()
- $s := Keyfile[-\lambda :]$
- $K :=$ PBKDF2(p, s)
- $k, mac1, mac2, H :=$ Dec$_{CTR}(K,$
- $keyH :=$ Hash$_{DM}(k||mac1||mac$
- if $H \neq keyH$:
  - return err
- return $k, mac1, mac2$

Security Proof and Reasoning

There are a number of components to these functions. We use the PBKDF2 algorithm to transform a keyfile's password into a 128-bit key used to decrypt the Keyfile. PBKDF2 uses an HMAC as its PRF calls. This HMAC uses a Davies-Meyer function as its internal hash component. These layers of security depend on the layer below it.

We start with the Davies-Meyer hash. We are interested in the collision-resistance of the hash function. We know that F is a secure block cipher.

F takes as its key the blocks of message m.

There are two possibilities here: either each block of m is distinct, in which case there are as many F(*)s as there are blocks; or some of the blocks are the same. This second possibility would be more

interesting to an attacker. If they passed in several message blocks of all zeros (for instance), would they be able to reverse-engineer the hash in order to reveal the original message of a different hash:

The actual keyfile itself contains three keys, as described above. These three keys are concatenated together and then hashed through our Davies-Meyer function. This hash is used as a verification that a) the password is correct, and b) the keyfile is not corrupted. If either of these conditions does not hold, then the program will return an error instead of the correctly-decrypted keys. While a MAC would be ideal here, a MAC requires the use of an additional key. We are unable to do that while continuing use a password to encrypt the keyfile.

The usage (or not) of a MAC for the key storage functions is not important. It is important for the encryption and decryption of files, as those are intended to leave the computer, where eavesdroppers could corrupt and edit the files before they get to their destination. However, for the storage of keyfiles on a single computer, the risk for corruption is a lot lower, and it's not imperative that the keyfiles have MACs computed for them.

Conclusion and Discussion

In this report, we have methodically gone through each component of our key manager, including the encryption scheme, the Master Key generation and storage, and how we apply our encryption and decryption schemes to the KeyFile in a way that ensures that an attacker cannot gain partial knowledge of either the Master Key, the keys in the key manager, or the messages sent be NOISE.