CS427 Final Project - Stream File Encryption & Key Management

Casey Colley Robert Detjens Lyell Read

CS 427, Winter 2022

Contents

Abstract **2**

Stream Encryption and Decryption (`enc`, `dec`) **3**

Primitives 3

Formal Scheme Definition 3

Security Proof and Reasoning 3

Key Generation and Storage (`keygen`) **7**

Primitives 7

Formal Scheme Definition 7

Security Proof and Reasoning 7

Conclusion and Discussion **7**

Abstract

placeholder

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

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

$\underline{\text{Dec}_{CTR}(k, c_0||...||c_l)\text{:}}$
  $r := c_0$
  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:

```
$\mathcal{L}^{\Sigma}_{\textsf{CCA-L}}$
  $k \leftarrow \Sigma.\textsf{KeyGen}$
  $\mathcal{S} := \emptyset$

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

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

$\approx$

```
$\mathcal{L}^{\Sigma}_{\textsf{CCA-R}}$
  $k \leftarrow \Sigma.\textsf{KeyGen}$
  $\mathcal{S} := \emptyset$

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

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

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

```
$\mathcal{L}^{\Sigma}_{\textsf{CCA-L}}$
  $k \leftarrow \Sigma.\textsf{KeyGen}$
  $\mathcal{S} := \emptyset$

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

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

$\diamond$

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

$\approx$

```
$\mathcal{L}^{\Sigma}_{\textsf{CCA-R}}$
  " "
```

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:

```
$\mathcal{L}^{\Sigma}_{\text{CCA-L}}$
  $k \leftarrow \Sigma.\mathsf{KeyGen}$
  $\mathcal{S} := \emptyset$

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

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

$\diamond$

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

$\approx$

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

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.

```
$\mathcal{L}^{\Sigma}_{\text{CCA-L}}$
  $k \leftarrow \Sigma.\mathsf{KeyGen}$
  $\mathcal{S} := \emptyset$

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

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

$\diamond$

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

$\approx$

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

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

```
$\mathcal{L}^{\Sigma}_{\textsf{CCA-L}}$
    $k \leftarrow \Sigma.\textsf{KeyGen}$
    $\mathcal{S} := \emptyset$

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

DECRYPT$(c)$ :
    if $c \in S$ return err
    return $\Sigma.\textsf{Dec}(k, c)$
```

$\approx$

```
$\mathcal{L}^{\Sigma}_{\textsf{CCA-R}}$
    $k \leftarrow \Sigma.\textsf{KeyGen}$
    $\mathcal{S} := \emptyset$

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

DECRYPT$(c)$ :
    if $c \in S$ return err
    return $\Sigma.\textsf{Dec}(k, c)$
```

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 the Master Key and the keys it protects. The Master Key is generated with 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`.

This Master Key will be stored on the machine, with a hash and encrypted. The encryption and decryption of the Master Key is done in through the modded CTR mode. The hash of the key will be appended before being encrypted, which ensures that it has not been tampered with and that the password was correct.

Primitives

The primitives we need are the F_{AES} block cipher that we identified earlier. The key to this block cipher will be derived by hashing the text password entered by the user (hence, it must have 128-bit output). The hash we will be using is a Davies-Meyer compression function with our same AES block cipher, F. A Davies-Meyer compression function functionally turns a block cipher into a hashing function. No key is needed by the scheme; the "keys" are the blocks of the message itself. The algorithm is defined below:

```
blen = 128

$\underline{\textsc{Hash}_{D-M}(m_1||...||m_l)}$:
  $h := \{0\}^{blen}$
  for $i = 1$ to $l$:
    $h := F(m_i, h) \oplus h$
  return $h$
```

Formal Scheme Definition

```
KeyFile := KeyGen()
```

```
$\underline{\text{KeyGen}()}$:
  $p :=$ getpass()
  $ph := \textsc{Hash}_{D-M}(p)$
  $k \leftarrow \{0,1\}^{\lambda}$
  $k+ = \textsc{Hash}_{D-M}(k)$
  $E := \textsc{Enc}_{CTR}(ph, k)$
  return $E$
```

```
$\underline{\text{DecryptKey}()}$:
  $p :=$ getpass()
  $ph := \textsc{Hash}_{D-M}(p)$
  $k, kh := \textsc{Dec}_{CTR}(h, KeyFile)$
  $keyH := \textsc{Hash}_{D-M}(k)$
  if $kh \neq keyH$:
    return err
  return $k$
```

Security Proof and Reasoning

Conclusion and Discussion

placeholder