CFB

Cipher feedback (CFB) is a block cipher mode of operation.

A *block cipher* encrypts plaintext in short blocks. For example, AES-128 is a block cipher that encrypts plaintext in 128-bit blocks. A plaintext message is split into plaintext blocks P_i, a block cipher κ encrypts each plaintext block into a ciphertext block C_i, and each ciphertext block is then transmitted.

Plaintext messages often contain well-known strings; an example is an email header. Well-known strings can be a big problem, even if they are encrypted. For instance, if a block cipher is used without something like CFB, a cryptanalyst can collect ciphertexts of email headers, compile a codebook, and then spoof encrypted emails, all without knowing the key.

CFB mode uses the same encryption function in both CFB encode and CFB decode functions. For example, if an CFB encode function calls the AES-128 encryption function, then the inverse CFB decode function also calls the AES-128 encryption function. Decryption functions are not used in CFB mode.

Literature

CFB mode was originally used in conjunction with the Data Encryption Standard (DES) and is specified in the 1980 document *DES Modes of Operation* along with tables of examples. [FIPS PUB 81].

Schneier discusses CFB mode in his 1994 book *Applied Cryptography* in section 8.1.4, and Figure 8.5 offers two block diagrams for 8-bit CFB mode, one for encryption and the other for decryption. [Schneier 1994]

Menezes et al discuss CFB mode in their 1996 *Handbook of Applied Cryptography* in section 7.2.2 (iii). Figure 7.1 (c) offers two block diagrams for r-bit CFB mode, and Algorithm 7.17 provides the details. [HAC 1999]

CFB mode is also specified in the 2001 NIST document *Recommendation for Block Cipher Modes of Operation* in section 6.3 and illustrated in Figure 3. The document also provides example vectors for CFB-128 encipherment and AES-128 encryption in Appendix F.3.13 and CFB-128 decipherment and AES-128 encryption in Appendix F.3.14. [NIST SP 800-38A]

Electronic codebook

To develop an understanding of CFB mode, it's best to start with the simplest block cipher mode. This is *electronic codebook* mode (ECB). The ECB encoder computes each ciphertext block C_i by simply applying the block cipher function κ to each plaintext block P_i.

$$C_i = \kappa(P_i)$$

The ECB decoder recovers each plaintext block P_i by applying the inverse block cipher function $\hat{\kappa}$ to each ciphertext block C_i.

$$P_i = \hat{\kappa}(C_i)$$

Schneier identifies several shortfalls of ECB. "The problem with ECB mode is that if cryptanalysts have the plaintext and ciphertext for several messages, they can start to compile a codebook without knowing the key. In most real-world situations, fragments of messages tend to repeat ... This vulnerability is greatest at the beginning and end of messages, where well-defined headers and footers contain information about the sender, receiver, date, etc."

Another problem is the block replay vulnerability. Mallet opens two bank accounts, one with Bank 1 and the other with Bank 2. He sets up a recording device on the EFT channel between the two banks. He orders funds transfers from his Bank 1 account to his Bank 2 account and identifies the corresponding ECB blocks on the channel. Then he simply replays those blocks on the channel and watches his Bank 2 account grow.

These vulnerabilities do not exist in cipher feedback mode, which uses a simple function to augment the block cipher.

Cipher feedback

The cipher feedback mode uses the exclusive-OR operator and a feedback loop to introduce complexity. The exclusive-OR operator is represented by the symbol $\oplus$. It is also abbreviated *xor* (pronounced "ex or"). It has several useful features: the identity element is 0, the inverse of any element A is itself, and the associative property holds.

$$A \oplus 0 = A \tag{1}$$

$$A \oplus A = 0 \tag{2}$$

$$(A \oplus B) \oplus C = A \oplus (B \oplus C) \tag{3}$$

The CFB encode function computes the current ciphertext block C_i by applying the encryption function κ to the previous ciphertext block C_{i-1} and then xor-ing

that with the current plaintext block P_i.

$$C_i = P_i \oplus \kappa(C_{i-1})$$

The CFB decode function recovers the current plaintext block P_i by applying the encryption function κ to the previous ciphertext block C_{i-1} and then xor-ing that with the current ciphertext block C_i. Note that the encryption function κ is used rather than the decryption function $\hat{\kappa}$.

$$P_i = C_i \oplus \kappa(C_{i-1})$$

The equation for P_i can be derived from C_i. We start with the equation for C_i and "add" $\kappa(C_{i-1})$ to both sides. The associative property lets us shift the square brackets on the right side. The two xored $\kappa(C_{i-1})$ terms cancel because $A \oplus A = 0$. And we can drop the 0 because $A \oplus 0 = A$. Rearranging, we get the equation for P_i.

$$\begin{aligned}
C_i &= P_i \oplus \kappa(C_{i-1}) \\
C_i \oplus \kappa(C_{i-1}) &= \big[P_i \oplus \kappa(C_{i-1})\big] \ \oplus \ \kappa(C_{i-1}) \\
C_i \oplus \kappa(C_{i-1}) &= P_i \ \oplus \ \big[\kappa(C_{i-1}) \oplus \kappa(C_{i-1})\big] \\
C_i \oplus \kappa(C_{i-1}) &= P_i \oplus 0 \\
C_i \oplus \kappa(C_{i-1}) &= P_i \\
P_i &= C_i \oplus \kappa(C_{i-1}) \qquad \square
\end{aligned}$$

The CFB encode function computes successive ciphertext blocks C_i. It takes a plaintext block P_i and xors it with an encryption of the previous ciphertext block, $\kappa(C_{i-1})$. This works great except for the initial case where we compute C_0. That is, what do we use for C_{-1}? It doesn't exist!

$$C_0 = P_0 \oplus \kappa(C_{-1})$$

The CFB decode function computes successive plaintext blocks P_i. It takes a ciphertext block C_i and xors it with an encryption of the previous ciphertext block, $\kappa(C_{i-1})$. We have the same problem with C_{-1}.

$$P_0 = C_0 \oplus \kappa(C_{-1})$$

And we have another problem with CFB decode. It's possible to decode thousands of blocks of ciphertext with the wrong key. We need some way to verify, at the beginning of CFB decode, that we have the correct key.

The C_{-1} problem is solved by replacing $\kappa(C_{-1})$ with $\kappa(0)$. The key verification problem is solved by using a random block R. With these two solutions, we can write complete algorithms for the basic CFB encode and decode functions.

Basic CFB

We need a random block R. For a block size of n bits, R is a randomly-selected value anywhere between 0 and $2^n - 1$ inclusive.

$$R \in \{0, 1, \ldots, (2^n - 1)\}$$

Basic CFB encoder:
Select a random R. Compute and output both preliminary blocks C_x and C_y. Then input plaintext block P_i and output ciphertext block C_i.

$$\begin{aligned} R &= \text{rand}(2^n) \\ C_x &= R \oplus \kappa(0) \\ C_y &= R \oplus \kappa(C_x) \\ C_0 &= P_0 \oplus \kappa(C_y) \\ C_1 &= P_1 \oplus \kappa(C_0) \\ C_2 &= P_2 \oplus \kappa(C_1) \\ &\vdots \end{aligned}$$

Basic CFB decoder:
Input ciphertext blocks C_x and C_y. Compute both R and R' and make sure they're equal. Then input ciphertext block C_i and output plaintext block P_i.

$$\begin{aligned} R &= C_x \oplus \kappa(0) \\ R' &= C_y \oplus \kappa(C_x) \\ R &\stackrel{?}{=} R' \\ P_0 &= C_0 \oplus \kappa(C_y) \\ P_1 &= C_1 \oplus \kappa(C_0) \\ P_2 &= C_2 \oplus \kappa(C_1) \\ &\vdots \end{aligned}$$

In OpenPGP, these two algorithms are more complicated. They require three preliminary blocks instead of two. C_x is the same, but C_y is truncated, and an additional *resynchronization* block C_z is required.

OpenPGP CFB

The definitive guide for OpenPGP CFB mode is RFC-4880. Section 13.9 provides the following twelve step algorithm for the CFB encode function. An

algorithm for the CFB decode function is not specified in RFC-4880. Unfortunately, the authors describe the algorithm without mathematical notation. Instead they employ a sort of register transfer language. FR is the feedback register. FRE is the encrypted feedback register. IV is the initialization vector. They also use the term "octet" instead of "byte" which is more common. BS is the block size, either 8 octets (64 bits) or 16 octets (128 bits).

1. The feedback register (FR) is set to the IV, which is all zeros.
2. FR is encrypted to produce FRE (FR Encrypted). This is the encryption of an all-zero value.
3. FRE is xored with the first BS octets of random data prefixed to the plaintext to produce C[1] through C[BS], the first BS octets of ciphertext.
4. FR is loaded with C[1] through C[BS].
5. FR is encrypted to produce FRE, the encryption of the first BS octets of ciphertext.
6. The left two octets of FRE get xored with the next two octets of data that were prefixed to the plaintext. This produces C[BS+1] and C[BS+2], the next two octets of ciphertext.
7. (The resynchronization step) FR is loaded with C[3] through C[BS+2].
8. FR is encrypted to produce FRE.
9. FRE is xored with the first BS octets of the given plaintext, now that we have finished encrypting the BS+2 octets of prefixed data. This produces C[BS+3] through C[BS+(BS+2)], the next BS octets of ciphertext.
10. FR is loaded with C[BS+3] to C[BS + (BS+2)] (which is C11-C18 for an 8-octet block).
11. FR is encrypted to produce FRE.
12. FRE is xored with the next BS octets of plaintext, to produce the next BS octets of ciphertext. These are loaded into FR, and the process is repeated until the plaintext is used up.

Distillation

The twelve-step procedure may be distilled. Consider that the crucial steps are 3, 6, 9, and 12. These are the steps where output is produced. Step 3 produces "C[1] through C[BS]." Step 6 produces "C[BS+1] and C[BS+2]." Step 9 produces "C[BS+3] through C[BS+(BS+2)]." Step 12 produces "the next BS octets of ciphertext."

We specify a distilled procedure with BS = 16 and AES-128 encryption. The distilled procedure uses only four steps (A, B, C, D). Step D is repeated until the plaintext is exhausted. The Plaintext register (P) contains all of the plaintext octets (bytes). The AES session key register (K) contains a 16-byte (128-bit) value. The Ciphertext register (C) is initially empty. The Feedback Register (FR) and the Feedback Register Encrypted (FRE) are 16-byte values. The Random value register (R) contains a 16-byte random value. The most significant byte of a register has index 1. For example, the most significant byte of FRE is FRE[1].

Step A: C[1] C[2] ... C[16]
The FR is reset to all zeros. Then the contents of FR is AES-128 encrypted using the session key K and the 16-byte result is written into FRE. The 16 bytes in FRE is xored with the 16 bytes in R and the 16-byte result is written to C[1] through C[16].

Step B: C[17] C[18]
The FR is set to the 16-byte value in C[1] through C[16]. Then the contents of FR is AES-128 encrypted using the session key K and the 16-byte result is written into FRE. The two bytes in FRE[1] and FRE[2] are xored with the two bytes in R[15] and R[16] and the two-byte result is written to C[17] and C[18].

Step C: C[19] C[20] ... C[34]
The FR is set to the 16-byte value in C[3] through C[18]. Then the contents of FR is AES-128 encrypted using the session key K and the 16-byte result is written into FRE. The 16 bytes of FRE is xored with the 16 bytes of P[1] through P[16] and the 16-byte result is written to C[19] through C[34].

Step D: C[35] C[36] ... C[50]
The FR is set to the value in C[19] through C[34]. Then the contents of FR is AES-128 encrypted using the session key K and the 16-byte result is written into FRE. The 16 bytes of FRE is xored with the 16 bytes of P[17] through P[32] and the 16-byte result is written to C[35] through C[50].

The algorithm in RFC-4880 does not specify what to do when the number of plaintext bytes is not a multiple of the block size BS. But it turns out that Steps C and D must be slightly modified to account for these cases. If the number of bytes remaining in the plaintext queue is $0 < n < 16$, then the 16 bytes of FRE are xored with a special 16-byte padded block, but only n ciphertext octets are actually emitted. The 16-byte padded block consists of the remaining n bytes and $16 - n$ padding bytes. The positions of these bytes does not matter as long as the decoder follows that same pattern.

Encode algorithm

Now we specify the algorithm for the OpenPGP CFB encode function. We use the same four steps as above. But instead of register transfer language, we use mathematical notation. This avoids the feedback register and the encrypted feedback register. We use uppercase variables for 128-bit values (e.g. C_0) and lowercase variables for 8-bit values (e.g. c_{19}).

Step A:
Select a random value for R between 0 and 2^{128}. Compute C_x as the xor of R and $\kappa(0)$. Split C_x into 16 ciphertext bytes and output them as $c_1\ c_2\ \ldots\ c_{16}$.

$$\begin{aligned} R &= \text{rand}(2^{128}) \\ C_x &= R \oplus \kappa(0) \\ c_j &= \text{byte}_{16-j}(C_x) \qquad j \in [1, 16] \end{aligned}$$

Step B:
Compute C_y as the xor of the most-significant word (msw) of the encryption of C_x with the least-significant word (lsw) in R. Split C_y into 2 ciphertext bytes and output them as $c_{17}\ c_{18}$.

$$\begin{aligned} C_y &= \text{msw}(\kappa(C_x)) \oplus \text{lsw}(R) \\ c_{17} &= \text{byte}_1(C_y) \\ c_{18} &= \text{byte}_0(C_y) \end{aligned}$$

Step C:
Initialize i to 0. Compute C_z as the lower 7 words of C_x multiplied by 65536 and then added to C_y. Let n be the number of remaining plaintext bytes. Input up to 16 plaintext bytes $p_1\ p_2\ \ldots$ and merge them into P_0. If $n < 16$ then pad P_0 in the least-significant bytes. Compute C_0 as the xor of P_0 and $\kappa(C_z)$. Split C_0 into 16 ciphertext bytes and output n of them as $c_{19}\ c_{20}\ \ldots$

$$\begin{aligned} i &= 0 \\ C_z &= 65536 \cdot \text{L}_7(C_x) + C_y \\ P_0 &= \sum_{j=1}^{16} 256^{16-j} \cdot p_j \\ C_0 &= P_0 \oplus \kappa(C_z) \\ c_{18+j} &= \text{byte}_{16-j}(C_0) \qquad j \in [1, 16] \end{aligned}$$

Step D:
Increment i. Let n be the number of remaining plaintext bytes. If $n = 0$ then terminate. Input up to 16 plaintext bytes $p_{16i+1}\ p_{16i+2}\ \ldots$ and merge them into P_i. If $n < 16$ then pad P_i in the least-significant bytes. Compute C_i as the xor of P_i and $\kappa(C_{i-1})$. Split C_i into 16 ciphertext bytes and output n of them as $c_{16i+19}\ c_{16i+20}\ \ldots$ Repeat this step.

$$\begin{aligned} i &= i+1 \\ P_i &= \sum_{j=1}^{16} 256^{16-j} \cdot p_{16i+j} \\ C_i &= P_i \oplus \kappa(C_{i-1}) \\ c_{16i+18+j} &= \text{byte}_{16-j}(C_i) \qquad j \in [1,16] \end{aligned}$$

Decode algorithm

As mentioned above, RFC-4880 does not specify an algorithm for the OpenPGP CFB decode function. However, it is easy enough to derive the decoder equations from the encoder equations. For R, P_0, and P_i we use the same trick that we used in the proof, where we add $\kappa(X)$ to both sides. For C_y and C_z we use the same equation as the encoder function.

Step A:
Input 16 ciphertext bytes $c_1\ c_2\ \ldots\ c_{16}$ and merge them into C_x. Then compute the random block R as the xor of C_x and $\kappa(0)$.

$$\begin{aligned} C_x &= \sum_{j=1}^{16} 256^{16-j} \cdot c_j \\ R &= C_x \oplus \kappa(0) \end{aligned}$$

Step B:
Compute C_y as the xor of the most-significant word (msw) of the encryption of C_x with the least-significant word (lsw) in R. Input the 2 ciphertext bytes $c_{17}\ c_{18}$ and merge them into C'_y. Verify that C_y and C'_y are equal.

$$\begin{aligned} C_y &= \text{msw}(\kappa(C_x)) \oplus \text{lsw}(R) \\ C'_y &= 256 \cdot c_{17} + c_{18} \\ C_y &\stackrel{?}{=} C'_y \end{aligned}$$

Step C:
Initialize i to 0. Compute C_z as the lower 7 words of C_x multiplied by 65536 and then added to C_y. Let n be the number of remaining ciphertext bytes. Input up to 16 ciphertext bytes $c_{19}\ p_{20}\ \ldots$ and merge them into C_0. If $n < 16$ then pad C_0 in the least-significant bytes. Compute P_0 as the xor of C_0 and $\kappa(C_z)$. Split P_0 into 16 plaintext bytes and output n of them as $p_1\ p_2\ \ldots$

$$\begin{aligned} i &= 0 \\ C_z &= 65536 \cdot \mathrm{L}_7(C_x) + C_y \\ C_0 &= \sum_{j=1}^{16} 256^{16-j} \cdot c_{18+j} \\ P_0 &= C_0 \oplus \kappa(C_z) \\ p_j &= \mathrm{byte}_{16-j}(P_0) \qquad j \in [1, 16] \end{aligned}$$

Step D:
Increment i. Let n be the number of remaining ciphertext bytes. If $n = 0$ then terminate. Input up to 16 ciphertext bytes $c_{16i+19}\ c_{16i+20}\ \ldots$ and merge them into C_i. If $n < 16$ then pad C_i in the least-significant bytes. Compute P_i as the xor of C_i and $\kappa(C_{i-1})$. Split P_i into 16 plaintext bytes and output n of them as $p_{16i+1}\ p_{16i+2}\ \ldots$ Repeat this step.

$$\begin{aligned} i &= i + 1 \\ C_i &= \sum_{j=1}^{16} 256^{16-j} \cdot c_{16i+18+j} \\ P_i &= C_i \oplus \kappa(C_{i-1}) \\ p_{16i+j} &= \mathrm{byte}_{16-j}(P_i) \qquad j \in [1, 16] \end{aligned}$$