CPSC 467: Cryptography and Security

Michael J. Fischer

Lecture 7
September 22, 2020

Stream Ciphers

Block Ciphers

Padding
- Bit padding
- Byte padding

Chaining Modes
- Block chaining modes
- Extending chaining modes to bytes

Stream Ciphers

Letter-by-letter encryption

A *stream cipher* is any cryptosystem that operates in an online fashion:

- The message is encrypted one letter at a time.
- After each message letter is read, one or more ciphertext letters are emitted as output.

Polyalphabetic substitution ciphers such as Caesar, Vigenère, Enigma machines, and even the one-time pad, are all examples of stream ciphers.

Structure of stream cipher

A stream cipher can be built from two components:

1. a cipher that is used to encrypt a given character;
2. a keystream generator that produces a different key to be used for each successive letter.

A commonly-used cipher is the simple XOR cryptosystem, also used in the one-time pad.

Rather than using a long random string for the keystream, we instead use a pseudorandom keystream generated on the fly using a state machine.

Like a one-time pad, a different master key (seed) must be used for each message; otherwise the system is easily broken.

Block Ciphers

Encrypting several letters at a time

A *block cipher* is a cryptosystem that operates on a fixed-length blocks of letters (or bits).

The Hill cipher presented in lecture 5 is an example of a block cipher since it encrypts several letters at a time.

Using a block cipher to encrypt arbitrary messages

Block ciphers are often not very useful by themselves since they can only encrypt messages of the fixed block length.

We can extend block ciphers to handle arbitrarily long messages much as we did for the Basic Caesar cipher.

Two important differences:

1. If the message length is not a multiple of the block size, we need a way to fill out the last block in a way that allows Bob to know where the message ends.
2. Encryption is no longer an online process since a block of characters can't be encrypted and sent out until a block's worth of characters have been read in.

Using a block cipher with arbitrary-length messages

To encrypt an arbitrary message m:

1. Apply a *padding rule* to m to produce a padded message m' whose length is a multiple of the block length b.
2. Split m' into a sequence of blocks. Encrypt them using the block cipher in some *chaining mode* to produce the ciphertext c', another sequence of blocks.

To decrypt, the above must be reversed:

1. Decrypt c' to produce m'.
2. Remove the padding from m' to recover the original message m.

Padding

Padding

Block ciphers are designed to handle sequences of blocks.

To send a message m of arbitrary length, it must first be *encoded* into a new message m' that consists of a sequence of blocks.

m' is then encrypted, transmitted, and decrypted.

After decrypting, m' must be *decoded* to recover the original message m.

A *padding rule* describes the encoding and decoding process.

How to pad

An obvious padding rule is to append 0's to the end of m until its length is a multiple of the block length b.

Unfortunately, this can't be properly decoded, since the receiver does not know how many 0's to discard from m'.

Condition: A padding rule must describe how much padding was added.

Suggestions?

Some easy padding rules

1. Pad with 0's, then *prepend* a block containing the length of m.

   Example: $b = 8, m = 01011, m' = \overbrace{00000101}^{\text{5 in binary}}\ 01011000$.

   Drawback: Must know the length of m before beginning.

2. Pad with 0's, then *append* a block containing the length of m.

   Example: $b = 8, m = 01011, m' = 01011000\ \overbrace{00000101}^{\text{5 in binary}}$.

   Drawback: Wasteful of space

3. Pad with a single 1 bit followed by 0's.

   Example: $b = 8, m = 01011, m' = 01011100$.

   Drawback: Need to count bits.

What happens if the length of m is already a multiple of b?

Compact bit padding

Here's a padding rule that is both space efficient and easy to decode.

- Choose $\ell = \lceil \log_2 b \rceil$. This is the number of bits needed to represent (in binary) any number in the interval $[0 \ldots (b-1)]$.
- Choose p as small as possible so that $|m| + p + \ell$ is a multiple of b.
- Pad m with p 0's followed by a length ℓ binary representation of p. Thus, the padded message is $m \cdot 0^p \cdot \overline{p}$, where $\overline{p}$ is the binary representation of p.

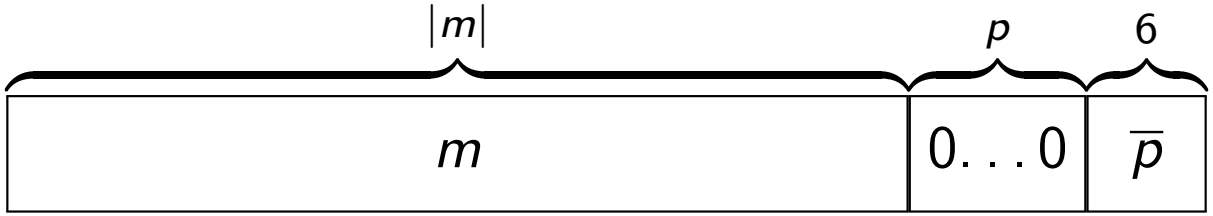
To unpad, interpret the last ℓ bits of the message as a binary number p; then discard a total of $p + \ell$ bits from the right end of the message.

Bit padding examples

1. $b = 8, m = 01011, m' = 01011\overbrace{000}^{0}$.
2. $b = 8, m = 1110, m' = 11100\overbrace{001}^{1}$.
3. $b = 8, m = 111, m' = 11100\overbrace{010}^{2}$.
4. $b = 8, m = 010110, m' = 01011000\ \ 00000\overbrace{111}^{7}$.

Bit padding on 64-bit blocks

- At most 63 0's ever need to be added, so a 6-bit length field is sufficient.
- A message m is then padded to become $m' = m \cdot 0^p \cdot \overline{p}$, where $\overline{p}$ is the 6-bit binary representation of p.
- p is chosen as small as possible so that $|m'| = |m| + p + 6$ is a multiple of 64.



Block codes on byte strings

Often messages and blocks consist of a sequence of 8-bit bytes.

In that case, padding can be done by adding an integral number of bytes to the message.

At least one byte is always added to avoid ambiguity.

PKCS7 padding

PKCS7 #7 is a message syntax described in internet RFC 2315.

- Fill a partially filled last block having k "holes" with k bytes, each having the value k when regarded as a binary number.
- If $k = 0$, an empty block is added before padding.

Example: Block length = 8 bytes.
$m =$ "`hello`".
$m' =$ `68 65 6C 6C 6F` 03 03 03.

On decoding, if the last block of the message does not have this form, then a decoding error is indicated.

Example: The last block cannot validly end in . . . 25 00 03.

What is the last block if $k = 0$?

Possible information leakage from padding

Suppose Alice uses AES (block length 128) in ECB mode to send 129-bit messages.

Eve has a plaintext-ciphertext pair (m', c') and intercepts a new cipher text c for an unknown message m.

Because of padding, both c and c' are two blocks long. Let c_2 and c_2' be the second blocks of each, respectively.

Then the last bit of m is the same as the last bit of m' iff $c_2 = c_2'$, so Eve learns the last bit of m.

Chaining Modes

Encrypting sequences of blocks in ECB mode

Recall from lecture 5 that we used the Basic Caesar Cipher in ECB mode in order to obtain the full Caesar cipher that can encrypt arbitrary length messages.

A *chaining mode* tells how to encrypt a sequence of plaintext blocks $m_1, m_2, \ldots, m_t$ to produce a corresponding sequence of ciphertext blocks $c_1, c_2, \ldots, c_t$, and conversely, how to recover the m_i's given the c_i's.

Electronic Code Book (ECB) mode encrypts/decrypts each block separately using the same key k.

$c_i = E_k(m_i)$, $1 \leq i \leq t$.

Removing cipher block dependence from ECB

ECB has the undesirable property that identical plaintext blocks yield identical ciphertext blocks. For example, $m_3 = m_8$ iff $c_3 = c_8$. This gives Eve possibly useful information about the meesage m.

Various extenstions to ECB fix this particular problem:

- Output Feedback (OFB) mode.
- Cipher Feedback (CFB) mode.
- Cipher Block Chaining (CBC) mode.

Encrypting sequences of blocks in OFB mode

Output Feedback (OFB) mode repeatedly applies the block cipher to a fixed initialization vector (IV) to produce a sequence of subkeys. Each block is encrypted/decrypted by XORing with the corresponding subkey.

$k_0 = E_k(IV)$
$k_i = E_k(k_{i-1})$, $c_i = m_i \oplus k_i$, $1 \leq i \leq t$.

It is likely that $k_3 \neq k_8$, so it is also likely that $c_3 \neq c_8$ even when $m_3 = m_8$.

OFB is like the one-time pad where, given a single known plaintext pair (m, c), $m \oplus c$ reveals the sequence of subkeys. Hence, a master key should never be used for more than one message.

Cipher Feedback (CFB)

Cipher Feedback (CFB) mode alleviates this problem by making the sequence of subkeys dependent on the message as well as on the master key k.

Namely, subkey k_i is the encryption of ciphertext block c_{i-1} rather than the encryption of the previous subkey k_{i-1} as is done with OFB.

A curious fact about OFB and CFB

In both OFB and CFB, Bob is able to decrypt without using the block decryption function D_k, so it is not even necessary for E_k tobe one-to-one.

This is because in both modes, E_k is only used for subkey generation, and both encryption and decryption of a message use the same sequence of subkeys.

For example, CFB encryption and decryption are almost identical.

- To encrypt, Alice computes $k_i = E_k(c_{i-1})$ and $c_i = m_i \oplus k_i$.
- To decrypt, Bob computes $k_i = E_k(c_{i-1})$ and $m_i = c_i \oplus k_i$.

c_0 is a fixed initialization vector.

OFB, CFB, and stream ciphers

Both OFB and CFB are closely related to stream ciphers.
In both cases, $c_i = m_i \oplus k_i$, where subkey k_i is computed from the master key and the data that came before stage i.

Like a one-time pad, OFB is insecure if the same key is ever reused, for the sequence of k_i's generated will be the same.
If m and m' are encrypted using the same key k, then $m \oplus m' = c \oplus c'$.

CFB partially avoids this problem, for even if the same key k is used for two different message sequences m_i and m'_i, it is only true that $m_i \oplus m'_i = c_i \oplus c'_i \oplus E_k(c_{i-1}) \oplus E_k(c'_{i-1})$, and the dependency on k does not drop out. However, the problem still exists when m and m' share a prefix.

Cipher Block Chaining Mode (CBC)

Cipher Block Chaining Mode (CBC) prevents identical message blocks from having identical corresponding ciphter blocks by mixing in the *previous* ciphtertext block when encrypting the current block.

- To encrypt, Alice applies E_k to the XOR of the current plaintext block with the previous ciphertext block. That is, $c_i = E_k(m_i \oplus c_{i-1})$.
- To decrypt, Bob computes $m_i = D_k(c_i) \oplus c_{i-1}$.

To get started, we take $c_0 = \mathrm{IV}$, where IV is a fixed *initialization vector* which we assume is publicly known.

Propagating Cipher-Block Chaining Mode (PCBC)

Here is a more complicated chaining rule that nonetheless can be deciphered.

- To encrypt, Alice XORs the current plaintext block, previous plaintext block, and previous ciphertext block.
  That is, $c_i = E_k(m_i \oplus m_{i-1} \oplus c_{i-1})$. Here, both m_0 and c_0 are fixed initialization vectors.
- To decrypt, Bob computes $m_i = D_k(c_i) \oplus m_{i-1} \oplus c_{i-1}$.

Recovery from data corruption

In real applications, a ciphertext block might be damaged or lost. An interesting property is how much plaintext is lost as a result.

- With ECB and OFB, if Bob receives a bad block c_i, then he cannot recover the corresponding m_i, but all good ciphertext blocks can be decrypted.
- With CBC and CFB, Bob needs good c_i and c_{i-1} blocks in order to decrypt m_i. Therefore, a bad block c_i renders both m_i and m_{i+1} unreadable.
- With PCBC, bad block c_i renders m_j unreadable for all $j \geq i$.

Error-correcting codes applied to the ciphertext are often used in practice since they minimize lost data and give better indications of when irrecoverable data loss has occurred.

Other modes

Other modes can easily be invented.

In all cases, c_i is computed by some expression (which may depend on i) built from $E_k()$ and $\oplus$ applied to available information:

- ciphertext blocks $c_1, \ldots, c_{i-1}$,
- message blocks $m_1, \ldots, m_i$,
- any initialization vectors.

Any such equation that can be "solved" for m_i (by possibly using $D_k()$ to invert $E_k()$) is a suitable chaining mode in the sense that Alice can produce the ciphertext and Bob can decrypt it.

Of course, the resulting security properties depend heavily on the particular expression chosen.

Stream ciphers from OFB and CFB block ciphers

OFB and CFB block modes can be turned into stream ciphers.

Both compute $c_i = m_i \oplus k_i$, where

- $k_i = E_k(k_{i-1})$ (for OFB);
- $k_i = E_k(c_{i-1})$ (for CFB).

Assume a block size of b bytes. Number the bytes in block m_i as $m_{i,0}, \ldots, m_{i,b-1}$ and similarly for c_i and k_i.

Then $c_{i,j} = m_{i,j} \oplus k_{i,j}$, so each output byte $c_{i,j}$ can be computed before knowing $m_{i,j'}$ for $j' > j$; no need to wait for all of m_i.

One must keep track of j. When $j = b$, the current block is finished, i must be incremented, j must be reset to 0, and k_{i+1} must be computed.

Extended OFB and CFB modes

Simpler (for hardware implementation) and more uniform stream ciphers result by also computing k_i a byte at a time.

The idea: Use a shift register X to accumulate the feedback bits from previous stages of encryption so that the full-sized blocks needed by the block chaining method are available.

X is initialized to some public initialization vector.

Extended OFB and CFB notation

Assume block size $b = 16$ bytes.

Define two operations: L and R on blocks:

- $L(x)$ is the leftmost byte of x;
- $R(x)$ is the rightmost $b - 1$ bytes of x.

Extended OFB and CFB similarities

The extended versions of OFB and CFB are very similar.

Both maintain a one-block shift register X.

The shift register value X_s at stage s depends only on $c_1, \ldots, c_{s-1}$ (which are now single bytes) and the master key k.

At stage i, Alice

- computes X_s according to Extended OFB or Extended CFB rules;
- computes *byte key* $k_s = L(E_k(X_s))$;
- encrypts message byte m_s as $c_s = m_s \oplus k_s$.

Bob decrypts similarly.

Shift register rules

The two modes differ in how they update the shift register.

Extended OFB mode

$$X_s = R(X_{s-1}) \cdot k_{s-1}$$

Extended CFB mode

$$X_s = R(X_{s-1}) \cdot c_{s-1}$$

('$\cdot$' denotes concatenation.)

Summary:

- Extended OFB keeps the most recent b key bytes in X.
- Extended CFB keeps the most recent b ciphertext bytes in X,

Comparison of extended OFB and CFB modes

The differences seem minor, but they have profound implications on the resulting cryptosystem.

- In eOFB mode, X_s depends only on s and the master key k (and the initialization vector IV), so loss of a ciphertext byte causes loss of only the corresponding plaintext byte.
- In eCFB mode, loss of ciphertext byte c_s causes m_s and all succeeding message bytes to become undecipherable until c_s is shifted off the end of X. Thus, b message bytes are lost.

Downside of extended OFB

The downside of eOFB is that security is lost if the same master key is used twice for different messages. CFB does not suffer from this problem since different messages lead to different ciphertexts and hence different keystreams.

Nevertheless, eCFB has the undesirable property that the keystreams *are the same* up to and including the first byte in which the two message streams differ.

This enables Eve to determine the length of the common prefix of the two message streams and also to determine the XOR of the first bytes at which they differ.

Possible solution

Possible solution to both problems: Use a different initialization vector for each message. Prefix the ciphertext with the (unencrypted) IV so Bob can still decrypt.