CPSC 467b: Cryptography and Computer Security

Lecture 6

Michael J. Fischer

Department of Computer Science
Yale University

January 27, 2010

Chaining mode

Recall:
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 Codebook Mode (ECB)

Each block is encrypted separately.

- To encrypt, Alice computes $c_i = E_k(m_i)$ for each i.
- To decrypt, Bob computes $m_i = D_k(c_i)$ for each i.

This is in effect a monoalphabetic cipher, where the “alphabet” is the set of all possible blocks and the permutation is E_k.

Cipher Block Chaining Mode (CBC)

Prevents identical plaintext blocks from having identical ciphertexts.

- To encrypt, Alice computes 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.

Output Feedback Mode (OFB)

Similar to a one-time pad, but key stream is generated from E_k.

- To encrypt, Alice repeatedly applies the encryption function to an *initial vector (IV)* to produce a stream of block keys, which in turn are XORed with successive plaintext blocks.
  That is, $c_i = m_i \oplus k_i$, where $k_i = E_k(k_{i-1})$ is a *block key*, and k_0 is a fixed initialization vector IV.
- To decrypt, Bob applies exactly the same method to the ciphertext to get the plaintext.
  That is, $m_i = c_i \oplus k_i$, where $k_i = E_k(k_{i-1})$ and $k_0 = IV$.

Cipher-Feedback Mode (CFB)

Similar to OFB, but key stream depends on previous messages as well as on E_k.

- To encrypt, Alice computes the XOR of the current plaintext block with the encryption of the previous ciphertext block. That is, $c_i = m_i \oplus E_k(c_{i-1})$.
  Again, c_0 is a fixed initialization vector.
- To decrypt, Bob computes $m_i = c_i \oplus E_k(c_{i-1})$.

Note that Bob is able to decrypt without using the block decryption function D_k. In fact, it is not even necessary for E_k to be a one-to-one function (but using a non one-to-one function might weaken security).

OFB, CFB, and stream ciphers

Both CFB and OFB are closely related to stream ciphers.
In both cases, c_i is m_i XORed with some function of 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 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.

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 important 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 both 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, a bad block c_i renders m_j unreadable for all $j \geq i$.

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.

Symmetric cryptosystem families

Symmetric (one-key) cryptosystems fall into two broad classes, *block ciphers* and *stream ciphers*.

- A block cipher encrypts large blocks of data at a time.
- A stream cipher process a stream of characters in an on-line fashion, emitting the ciphertext character by character as it goes.

Structure of stream cipher

A stream cipher has two components:

1. a cipher that is used to encrypt a given character;
2. a key stream 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 key stream, we instead generate the key stream on the fly using a state machine.

Key stream generator

A *key stream generator* consists of three parts:

1. an internal state,
2. a next-state generator,
3. an output function.

At each stage, the state is updated and the output function is applied to the state to obtain the next component of the key stream.

The next-state generator and output functions can both depend on the (original) *master* key.

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

Security requirements for key stream generator

The output of the key stream generator must "look" random.

Any regularities in the output give an attacker information about the plaintext.

A known plaintext-ciphertext pair (m, c) gives the attacker a sample output sequence from the key stream generator (namely, $m \oplus c$.)

If the attacker is able to figure out the internal state, then she will be able to predict all future outputs of the generator and decipher the remainder of the ciphertext.

A pseudorandom sequence generator that resists all feasible attempts to predict future outputs given a sequence of past outputs is said to be *cryptographically strong*.

Cryptographically strong pseudorandom sequence generators

Commonly-used linear congruential pseudorandom number generators typically found in software libraries are quite insecure.

After observing a relatively short sequence of outputs, one can solve for the state and correctly predict all future outputs.

(Note that the Linux `random()` is non-linear and hence much better, though still not cryptographically strong.

We will return to pseudorandom number generation later in this course.

See Katz & Lindell Chapter 3 for an in-depth discussion of this topic.)

Ideas for improving stream ciphers

As with one-time pads, the same key stream must not be used more than once.

A possible improvement: Make the next state depend on the current plaintext or ciphertext characters.

Then the generated key streams will diverge on different messages, even if the key is the same.

Serious drawback: One bad ciphertext character will render the rest of the message undecipherable.

Building key stream generators from block ciphers

OFB and CFB block chaining modes can be extended to stream ciphers on units smaller than full blocks.

Can't just apply directly because one can't wait for a block's worth of message bytes before outputting the first ciphertext byte.

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.

Some notation

Assume block size $b = 64$ bits and character size $s = 8$ bits.

Let $B = \{0, 1\}$. Define two operations: L_m and $R_m : B^b \to B^m$.

$L_m(x)$ are the leftmost m bits of x, and $R_m(x)$ are the rightmost m bits of x.

Extended CFB and OFB similarities

The extended versions of CFB and OFB are very similar.

Both maintain a b-bit shift register X.

The shift register value X_i at stage i depends only on $c_1, \ldots, c_{i-1}$ and the master key k.

At stage i, Alice

- computes X_i according to CFB or OFB rules;
- computes *byte key* $k_i = L_s(E_k(X_i))$;
- encrypts message byte m_i as $c_i = m_i \oplus k_i$.

Bob decrypts similarly.

Shift register rules

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

Extended CFB mode

$$X_i = R_{b-s}(X_{i-1}) \cdot c_{i-1}$$

Extended OFB mode

$$X_i = R_{b-s}(X_{i-1}) \cdot k_{i-1}$$

('$\cdot$' denotes concatenation.)

Conclusion:

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

Comparison of extended CFB and OFB modes

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

- In CFB mode, loss of ciphertext byte c_i causes m_i and all succeeding message bytes to become undecipherable until c_i is shifted off the end of X. Thus, $\lceil b/s \rceil$ message bytes are lost.
- In OFB mode, X_i depends only on i and the master key k (and the initialization vector IV), so loss of a ciphertext byte causes loss of only the corresponding plaintext byte.

Downside of extended OFB

The downside of OFB is that security is lost of 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 key streams.

Nevertheless, CFB has the undesirable property that the key streams *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 to both problems: Use a different initialization vector for each message. Prefix the ciphertext with the (unencrypted) IV so Bob can still decrypt.

Rotor machines

- Rotor machines are mechanical devices for implementing stream ciphers.
- They played an important role during the Second World War.
- The Germans believed their Enigma machine was unbreakable.
- The Allies, with great effort, succeeded in breaking it and in reading many of the top-secret military communications.
- This is said to have changed the course of the war.

Image from Wikipedia

How a rotor machine works

High-level structure:

- Uses electrical switches to create a permutation of 26 input wires to 26 output wires.
- Each input wire is attached to a key on a keyboard.
- Each output wire is attached to a lamp.
- The keys are associated with letters just like on a computer keyboard.
- Each lamp is also labeled by a letter from the alphabet.
- Pressing a key on the keyboard causes a lamps to light, indicating the corresponding ciphertext character.

The operator types the message one character at a time and writes down the letter corresponding to the illuminated lamp.

The same process works for decryption since $E_{k_i} = D_{k_i}$.

Key stream generation

The encryption permutation.

- Each rotor is individually wired to produce some random-looking fixed permutation π.
- Several rotors stacked together produce the composition of the permutations implemented by the individual rotors.
- In addition, the rotors can rotate relative to each other, implementing in effect a rotation permutation (like the Caeser cipher uses).

Key stream generation (cont.)

Let $\rho_k(x) = x + k \bmod 26$. Then rotor in position k implements permutation $\rho_k \pi \rho_k^{-1}$.

Several rotors stacked together implement the composition of the permutations computed by each.

For example, three rotors implementing permutations π_1, π_2, and π_3, placed in positions r_1, r_2, and r_3, respectively, would produce the permutation

$$\begin{aligned} & \rho_{r_1} \cdot \pi_1 \cdot \rho_{-r_1} \cdot \rho_{r_2} \cdot \pi_2 \cdot \rho_{-r_2} \cdot \rho_{r_3} \cdot \pi_3 \cdot \rho_{-r_3} \\ = \quad & \rho_{r_1} \cdot \pi_1 \cdot \rho_{r_2-r_1} \cdot \pi_2 \cdot \rho_{r_3-r_2} \cdot \pi_3 \cdot \rho_{-r_3} \end{aligned} \tag{1}$$

Changing the permutation

After each letter is typed, some of the rotors change position, much like the mechanical odometer used in older cars.

The period before the rotor positions repeat is quite long, allowing long messages to be sent without repeating the same permutation.

Thus, a rotor machine is much like a polyalphabetic substitution cipher but with a very long period.

Unlike a pure polyalphabetic cipher, the successive permutations until the cycle repeats are not independent of each other but are related by equation (1).

This gives the first toehold into methods for breaking the cipher (which are far beyond the scope of this course).

History

Several different kinds of rotor machines were built and used, both by the Germans and by others, some of which work somewhat differently from what I described above.

However, the basic principles are the same.

The interested reader can find much detailed material on the web by searching for "enigma cipher machine" and "rotor cipher machine". Nice descriptions may be found at `http://en.wikipedia.org/wiki/Enigma_machine` and `http://www.quadibloc.com/crypto/intro.htm`.

Steganography

Steganography, hiding one message inside another, is an old technique that is still in use.

For example, a message can be hidden inside a graphics image file by using the low-order bit of each pixel to encode the message. The visual effect of these tiny changes is probably too small to be noticed by the user.

The message can be hidden further by compressing it or by encrypting it with a conventional cryptosystem.

Unlike conventional cryptosystems, steganography relies on the secrecy of the method of hiding for its security.

If Eve does not even recognize the message as ciphertext, then she is not likely to attempt to decrypt it.

Active adversary

Recall from lecture 3 the active adversary "Mallory" who has the power to modify messages and generate his own messages as well as eavesdrop.

Alice sends $c = E_k(m)$, but Bob may receive a corrupted or forged $c' \neq c$.

How does Bob know that the message he receives really was sent by Alice?

The naive answer is that Bob computes $m' = D_k(c')$, and if m' "looks like" a valid message, then Bob accepts it as having come from Alice. The reasoning here is that Mallory, not knowing k, could not possibly have produced a valid-looking message. For any particular cipher such as DES, that assumption may or may not be valid.

Some active attacks

Three successively weaker (and therefore easier) active attacks in which Mallory might produce fraudulent messages:

1. Produce valid $c' = E_k(m')$ for a message m' of his choosing.
2. Produce valid $c' = E_k(m')$ for a message m' that he cannot choose and perhaps does not even know.
3. Alter a valid $c = E_k(m)$ to produce a new valid c' that corresponds to an altered message m' of the true message m.

Attack (1) requires computing $c = E_k(m)$ without knowing k.

This is similar to Eve's ciphertext-only passive attack where she tries to compute $m = D_k(c)$ without knowing k.

It's conceivable that one attack is possible but not the other.

Replay attacks

One form of attack (2) clearly *is* possible.

In a *replay* attack, Mallory substitutes a legitimate old encrypted message c' for the current message c.

It can be thwarted by adding timestamps and/or sequence numbers to the messages so that Bob can recognize when old messages are being received.

Of course, this only works if Alice and Bob anticipate the attack and incorporate appropriate countermeasures into their protocol.

Fake encrypted messages

Even if replay attacks are ruled out, a cryptosystem that is secure against attack (1) might still permit attack (2).

There are all sorts of ways that Mallory can generate values c'.

What gives us confidence that Bob won't accept one of them as being valid?

Message-altering attacks

Attack (3) might be possible even when (1) and (2) are not.

For example, if c_1 and c_2 are encryptions of valid messages, perhaps so is $c_1 \oplus c_2$.

This depends entirely on particular properties of E_k unrelated to the difficulty of decrypting a given ciphertext.

We will see some cryptosystems later that do have the property of being vulnerable to attack (3). In some contexts, this ability to do meaning computations on ciphertexts can actually be useful, as we shall see.

Encrypting random-looking strings

Cryptosystems are not always used to send natural language or other highly-redundant messages.

For example, suppose Alice wants to send Bob her password to a web site. Knowing full well the dangers of sending passwords in the clear over the internet, she chooses to encrypt it instead. Since passwords are supposed to look like random strings of characters, Bob will likely accept anything he gets from Alice.

He could be quite embarrassed (or worse) claiming he knew Alice's password when in fact the password he thought was from Alice was actually a fraudulent one derived from a random ciphertext c' produced by Mallory.