CPSC 467: Cryptography and Computer Security

Michael J. Fischer

Lecture 7
September 18, 2013

Pseudorandom sequence generator

Pseudorandom sequence generator (PRSG)

Recall that one can build a stream cipher from a one-time pad by using a pseudorandom sequence generator to generate the key instead of choosing a truly random key.

A *pseudoramdom sequence generator (PRSG)* consists of:

1. a *seed* (or *master key*),
2. a *state*,
3. a *next-state generator*,
4. an *output function*.

The initial state is derived from the seed.

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

Security requirements

- The output of the PRSG must "look" random.
- Any regularities in the output of the PRSG give an attacker information about the plaintext.
- A known plaintext-ciphertext pair (m, c) gives the attacker a sample output sequence from the PRSG (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, even knowing a sequence of past outputs, is said to be *cryptographically strong*.

Cryptographically strong PRSGs

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.

Notes:

- 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 Goldwasser & Bellare Chapter 3 for an in-depth discussion of this topic.

Ideas for improving stream ciphers

As with one-time pads, the same keystream 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 keystreams 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.

Using block ciphers on sequences of blocks

Recall: Difference between block and stream ciphers

A block cipher cannot be used directly as a stream cipher.

- A stream cipher must output the current ciphertext byte before reading the next plaintext byte.
- A block cipher waits to output the current ciphertext block until a block's worth of message bytes have been accumulated.

We first return to the problem of using a block cipher to encrypt a sequence of blocks in an on-line fashion and then extend those ideas to become a true stream cipher.

Padding revisited

Earlier we presented a method of *bit padding* to turn an abtirary bit string into one whose length is a multiple of the block length.

Often the underlying message consists of a sequence of bytes, and a block comprises some number b of bytes.

This enables *byte padding* methods to be used, some of which are very simple.

PKCS7 padding

PKCS7 #7 is a message syntax described in internet RFC 2315. It's padding rule is to fill a partially filled last block having *k* "holes" with *k* bytes, each having the value *k* when regarded as a binary number.
At least one byte is always added. Why?

For example, if the last block is 3 bytes short of being full, then the last 3 bytes are set to the values 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.

Chaining mode

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 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.

Output Feedback Mode (OFB)

Similar to a one-time pad, but keystream is generated using E_k.

- To encrypt, Alice repeatedly applies the encryption function to an *initial vector (IV)* k_0 to produce a stream of *block keys* $k_1, k_2, \ldots$, where $k_i = E_k(k_{i-1})$.

  The block keys are XORed with successive plaintext blocks. That is, $c_i = m_i \oplus k_i$.
- 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 keystream 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 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.

Extending chaining modes to bytes

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 numbered $0, \ldots, b-1$.

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.

Some 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.

Active adversary attacks

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.