Lecture Notes 4

1 Using Symmetric Cryptosystems

Symmetric (one-key) cryptosystems fall into two broad classes, *block ciphers* and *stream ciphers*. A block cipher takes as inputs a key and a fixed-length plaintext block and produces a fixed-length ciphertext block as output. Most of the ciphers we have been discussing so far are of this type. A *stream cipher* on the other hand process a stream of characters in an on-line fashion, emitting the ciphertext characters as it goes. The rotor machines are examples of stream ciphers.

1.1 Stream ciphers

A stream cipher has two components, the cipher that is used to encrypt a given character, and a keystream generator that produces a different key to be used for each successive letter.

A simple stream cipher can be built from the XOR cryptosystem used in the one-time pad. However, rather than using a random key as long as the message, we instead generate the keystream on the fly using a state machine. A *keystream generator* consists of three parts: an internal state, a next-state generator, and an output function. The next-state generator and output functions can both depend on (original) key. At each stage, the state is updated and the output function applied to obtain the next component of the keystream. Like a one-time pad, one must use different key for each message; otherwise the system is easy to break.

To be secure, the keystream generator must be a good pseudorandom sequence generator. Any regularities in the output of the keystream generator will give an attacker information about the plaintext. In particular, if the attacker is ever able to figure out the internal state of the keystream generator, then she will be able to predict all future outputs of the generator and decipher the remainder of the ciphertext. It turns out that the linear congruential pseudorandom number generators typically found in software libraries are quite insecure. After observing a relatively short sequence of outputs from the generator, one can solve for the state and correctly predict all future outputs. For the simple XOR cipher to be secure, a *cryptographically strong* pseudorandom number generator must be used. Even so, the fact that a different key must be used for each message sent makes it problematic in practice. It's at least an improvement to make the next-state generator depend on the current plaintext or ciphertext characters so that the generated keystreams will diverge on different messages, even if the key is the same.

1.2 Block ciphers

A block cipher operates on an entire *block* of plaintext to produce a *block* of ciphertext. So does a stream cipher, but with a stream cipher the block size is very small (typically one bit or one byte), whereas a block cipher operates on fairly long blocks, e.g., 64-bits for DES, 128-bits for Rijndal (AES). With a stream cipher, security rests with the keystream generator, for the ciphers used to encrypt the individual characters are all subject to known-plaintext attacks. On the other hand, block ciphers can be designed to resist known-plaintext attacks and can therefore be pretty secure, even if the same key is used to encrypt a succession of blocks, as is often the case.

Of course, the length messages one wants to send are rarely exactly the block length. To use a block cipher to encrypt long messages, one first divides the message into blocks of the right length, padding the last partial block according to a suitable padding rule. Then the block cipher is used in some *chaining mode* to encrypt the sequence of resulting blocks. 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.

Some standard chaining modes are:

- *Electronic Codebook Mode (ECB)* – apply cipher to each plaintext block. That is, $c_i = E_k(m_i)$ for each i. This becomes in effect a monoalphabetic cipher, where the "alphabet" is the set of all possible blocks and the permutation is defined by E_k. To decrypt, Bob computes $m_i = D_k(c_i)$.

- *Cipher Block Chaining Mode (CBC)* – encrypt the XOR of the current plaintext block with the previous ciphertext block to produce the current ciphertext block. That is, $c_i = E_k(m_i \oplus c_{i-1})$. To get started, we take $c_0 = \text{IV}$, where IV is a fixed *initialization vector* which we assume is publicly known. To decrypt, Bob computes $m_i = D_k(c_i) \oplus c_{i-1}$.

- *Cipher-Feedback Mode (CFB)* – XOR the current plaintext block with the encryption of the previous ciphertext block. That is, $c_i = m_i \oplus E_k(c_{i-1})$, where 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).

- *Output Feedback Mode (OFB)* – the encryption function is iterated on an *initial vector (IV)* to produce a stream of block keys, which in turn are XORed with the successive plaintext blocks to produce the successive ciphertext blocks. (This is similar to a simple keystream generator.) That is, $c_i = m_i \oplus k_i$, where $k_i = E_k(k_{i-1})$ is a *block key*. k_0 is a fixed initialization vector IV. To decrypt, Bob can apply exactly the same method to the ciphertext to get the plaintext, that is, $m_i = c_i \oplus k_i$, where $k_i = E_k(k_{i-1})$.

- *Propagating Cipher-Block Chaining Mode (PCBC)* – encrypt the XOR of 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 $m_i = D_k(c_i) \oplus m_{i-1} \oplus c_{i-1}$.

Remarks

1. Both CFB and OFB are closely related to stream ciphers since in both cases, c_i is m_i XORed with some function of stuff that came before stage i. Like a one-time pad and other simple XOR stream ciphers, OFB becomes insecure if the same key is ever reused, for the sequence of k_i's generated will be the same. CFB, however, avoids this problem, for even if the same key k is used for two different message sequences m_i and m'_i, it will not generally be the case that $m_i \oplus m'_i = c_i \oplus c'_i$; rather, $m_i \oplus m'_i = c_i \oplus c'_i \oplus E_k(c_{i-1}) \oplus E_k(c'_{i-1})$, and the dependency on k does not drop out.

2. The different modes differ in their sensitivity to data corruption. 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, he 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 $j \geq i$.

3. Other modes can be easily invented. We see that in all cases, c_i is computed by some expression (which may depend on i) built from $E_k()$ and $\oplus$ applied to blocks $c_1, \ldots, c_{i-1}$, $m_1, \ldots, m_i$, and the 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 is able to produce the ciphertext and Bob is able to decrypt it. Of course, the resulting security properties depend heavily on the particular expression chosen.