COMP 8920: Cryptography — Winter 2025

Lecture 09 — Feb 4, 2025

Prof. Curtis Bright — *Scribe: Mahzabin Chowdhury*

Mode of Operation

CBC (cipher block chaining mode)

In CBC mode, each ciphertext block y_i is x-ored with the next plaintext block, x_{i+1}, before being encrypted with the key K. More formally, we start with an initialization vector, denoted by IV,

$$y_0 = IV$$

Then construct,

$$y_1 = e_k(y_0 \oplus x_1)$$
$$\vdots$$
$$y_n = e_k(y_{n-1} \oplus x_n)$$

This looks secure. Given $y_0, \ldots, y_n$, Eve does not appear to be able to derive $x_1, \ldots, x_n$.

A Padding Oracle Attack

A surprising and devastating attack on CBC with a certain padding scheme was discovered in 2002. It was actually used in practice against web browsers that implemented TLS (Transport Layer Security).

The PKCS #7 padding is as follows:

- 15 bytes of data are padded with one byte `01`.
- 14 bytes are padded with `0202`.
- 13 bytes are padded with `030303`.

For 16 bytes per block, we look at the last byte to see how many padded bytes were added. If the last block is a full 16 bytes, then a full block of zeros is added. To determine how many bytes (1-16) were added, check the very last byte of plaintext.

Given ciphertext blocks $y_0, \ldots, y_n$ (y_0 is the IV), after decryption the block $d_k(y_n)$ gets checked to see if it is padded correctly. If not, an error is raised.

In a "padding oracle attack", an adversary Eve can pass ciphertexts to an oracle and determine whether they were correctly padded, thereby learning the padding but not the resulting plaintext.

Suppose $y_0, \ldots, y_n$ is the ciphertext the adversary wants to decrypt. We'll focus on the first block of actual ciphertext y_1 (y_0 is the IV).

$$y_1 = e_k(y_0 \oplus x_1)$$
$$x_1 = d_k(y_1) \oplus y_0$$

But k is unknown to Eve. However, Eve is free to modify the ciphertext and send the modification to the oracle for padding validation.

Suppose that Eve chooses $y_0' = r_1 r_2 \ldots r_{16}$, where $r_1, \ldots, r_{15}$ are random bytes, and r_{16} will be chosen iteratively from all possible 256 bytes.

For each y_0', Eve will send the two-block ciphertext $y_0' y_1 = (r_1 \ldots r_{16}) y_1$ to the oracle. The oracle computes

$$x_1' = d_k(y_1) \oplus y_0'$$

to check if x_1' is correctly padded. Note that there will be one possibility for r_{16} to make the last byte of x_1' `01`, which is correctly padded.

Since it is unlikely for the last two bytes to be `0202` or the last three to be `030303`, the oracle will likely return "true" only for correctly padded messages. When the oracle returns "true", Eve knows that it is correctly padded and so most likely

$$x_1'[16] = \texttt{01}.$$

The last byte of the original message can be calculated as:

$$x_1[16] = d_k(y_1)[16] \oplus y_0[16]$$
$$x_1'[16] = d_k(y_1)[16] \oplus y_0'[16]$$

XORing these cancels out the term with k:

$$x_1[16] \oplus x_1'[16] = y_0[16] \oplus y_0'[16]$$
$$x_1[16] = y_0[16] \oplus y_0'[16] \oplus \texttt{01}$$

Eve can then move to the second last byte by adjusting r_{15}. Incrementing $\texttt{01} = x_1[16] \oplus y_0[16] \oplus r_{16}$ by 1 (which can be accomplished by updating r_{16} to $\texttt{02} \oplus x_1[16] \oplus y_0[16]$) allows her to find a y_0'' for which the last byte of x_1'' is `02`.

By iterating over all $r_{15} \in \{\texttt{00}, \ldots, \texttt{FF}\}$, Eve can query the oracle until she finds the unique r_{15} that correctly pads the message, resulting in the last two bytes of x_1'' becoming `0202`. She now knows,

$$d_k(y_1)[15] \oplus r_{15} = \texttt{02}$$

XORing with $x_1[15] = d_k(y_1)[15] \oplus y_0[15]$, Eve finds:

$$x_1[15] = y_0[15] \oplus r_{15} \oplus \texttt{02}$$

Repeating this process allows Eve to learn all bytes of x_1 with only $16 \times 256 = 4096$ oracle calls.

This attack works against any byte (say x_2) in the message by sending ciphertexts of the form $y_0 y_1' y_2$ for altered y_1' and using the equations:

$$x_2 = d_k(y_2) \oplus y_1$$
$$x_2' = d_k(y_2) \oplus y_1'$$

Stream Ciphers

Stream ciphers typically encrypt plaintext via XOR with a keystream. We already saw the LFSR method for generating a keystream (but it was insecure). LFSRs are appealing as they are efficient and have long periods. To increase security, three methods are examined:

1. Combination Generator

Multiple (r) LFSRs are combined via a function:

$$f : \mathbb{Z}_2^r \to \mathbb{Z}_2$$

$$z_i = f(z_i', \ldots, z_i^r)$$

2. Filter Generator

One LFSR with a recurrence of degree m is used. Instead of taking the bits produced by the LFSR, a Boolean function is applied:

$$f : \mathbb{Z}_2^m \to \mathbb{Z}_2$$

This derives the keystream from the LFSR state.

3. Shrinking Generator

Two LFSRs are used, with keystream bits taken from the first. If the second LFSR produces `0`, the output of the first LFSR is discarded.

Attack on the Combination Generator

Suppose that we have 3 LFSR with known recurrence, combined by a majority function:

$$f(a, b, c) = (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c)$$

This is called the `MAJ` function as it produces the most common element among (a, b, c). If the triple (a, b, c) is equally likely, the probability that a agrees with $\texttt{MAJ}(a, b, c)$ is $\frac{3}{4}$. This is useful for searching for the key.