Assignment 1

CYBER SECURITY

Salar Mokhtari Laleh - Ali Abbasi

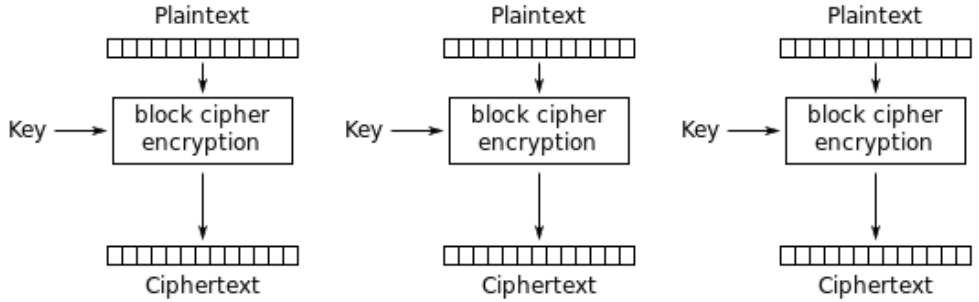
University of Tabriz

Faculty of Electrical & Computer Engineering

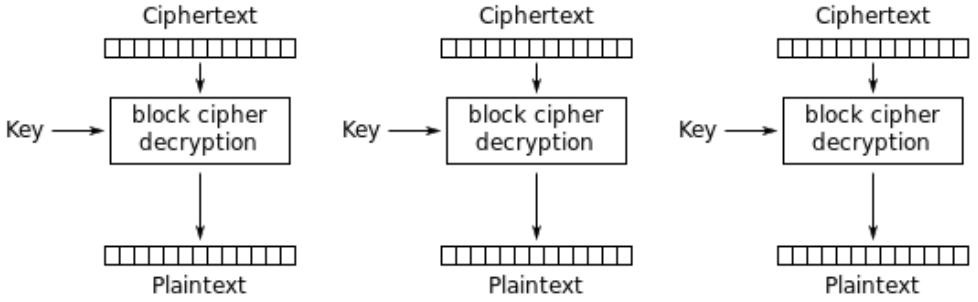
Fall 2022

Q 1

ECB: The answer is Yes. The simplest (and not to be used anymore) of the encryption modes is the Electronic Code Book (ECB). In this mode, the message is divided into blocks, and each block is encrypted separately and then decrypted in the same way. So, in this mode blocks don't depend on each other. Therefore, encryption and decryption can be done simultaneously and parallelly.



Electronic Codebook (ECB) mode encryption



Electronic Codebook (ECB) mode decryption

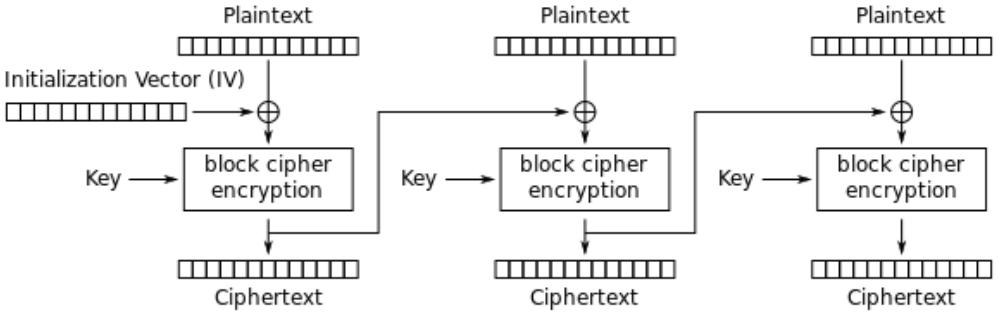
The disadvantage of this method is a lack of diffusion. Because ECB encrypts identical plaintext blocks into identical ciphertext blocks, it does not hide data patterns well. ECB is not recommended for use in cryptographic protocols.

ECB mode can also make protocols without integrity protection even more susceptible to replay attacks, since each block gets decrypted in exactly the same way.

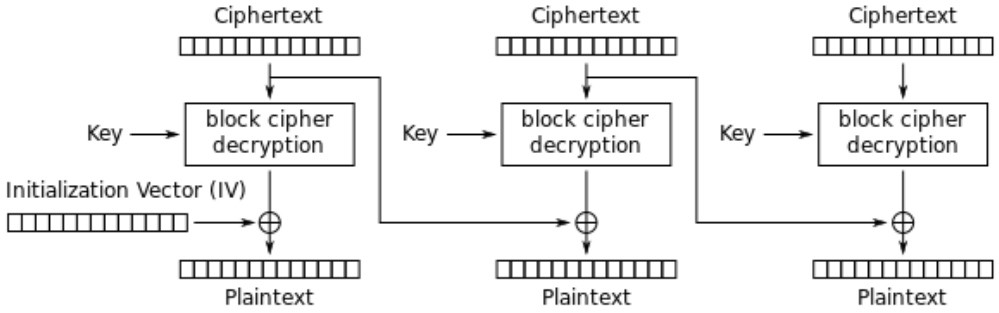
Q 1

CBC: The answer is No for the encryption process. In Cipher Block Channing mode (CBC), each block of plaintext (except the first block) is XORed with the previous ciphertext block before being encrypted. This way, each ciphertext block depends on all plaintext blocks processed up to that point. So, this dependency of each block on previous blocks doesn't let us do the encryption operation in a parallel way.

However, in decryption, we just need the ciphertext block and the key. So, there is no dependency between the blocks and we can decrypt the blocks parallelly.



Cipher Block Chaining (CBC) mode encryption

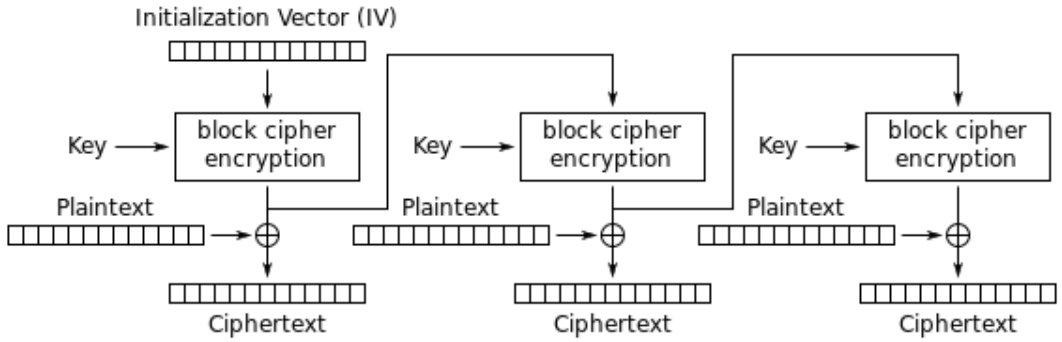


Cipher Block Chaining (CBC) mode decryption

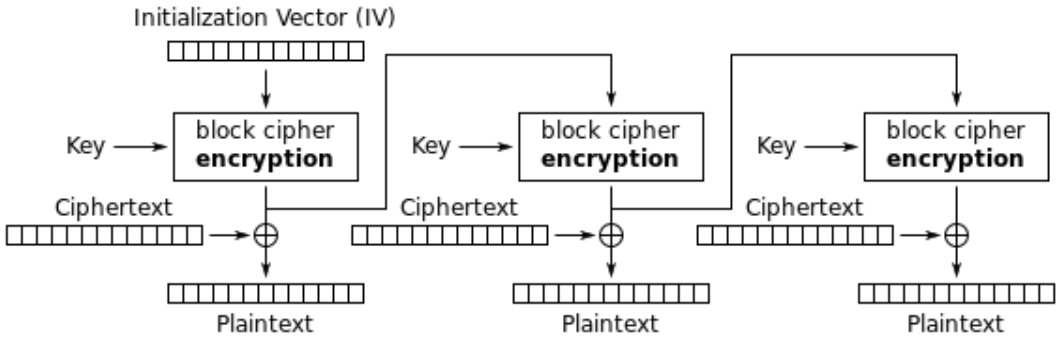
Q 1

OFB: The answer is No. The Output Feedback (OFB) mode makes a block cipher into a synchronous stream cipher. It generates keystream blocks, which are then XORed with the plaintext blocks to get the ciphertext. So, the parallel process in encryption is almost impossible unless we know the value of the Initialization Vector (IV). If we have the IV, we can calculate the output of blocks and save them to use when plaintext blocks come. Therefore, by this way, we can do part of our process simultaneously.

In decryption, as the structure of the blocks is the same so we can do the decryption operation parallelly. Otherwise, there is no way to do the decryption process simultaneously.



Output Feedback (OFB) mode encryption



Output Feedback (OFB) mode decryption

Q 2

In operation, the secret 56-bit key is broken up into 16 subkeys according to the DES key schedule; one subkey is used in each of the sixteen DES rounds. DES weak keys produce sixteen identical subkeys. This occurs when the key (expressed in hexadecimal) is:

- Alternating ones + zeros (0x0101010101010101)
- Alternating 'F' + 'E' (0xFEFEFEFEFEFEFEFE)
- '0xE0E0E0E0F1F1F1F1'
- '0x1F1F1F1F0E0E0E0E'

If an implementation does not consider the parity bits, the corresponding keys with the inverted parity bits may also work as weak keys:

- all zeros (0x0000000000000000)
- all ones (0xFFFFFFFFFFFFFFFF)
- '0xE1E1E1E1F0F0F0F0'
- '0x1E1E1E1E0F0F0F0F'

Using weak keys, the outcome of the Permuted Choice 1 (PC-1) in the DES key schedule leads to round keys being either all zeros, all ones or alternating zero-one patterns.

Since all the subkeys are identical, and DES is a Feistel network, the encryption function is self-inverting; that is, despite encrypting once giving a secure-looking cipher text, encrypting twice produces the original plaintext.

DES also has semi-weak keys, which only produce two different subkeys, each used eight times in the algorithm: This means they come in pairs K1 and K2, and they have the property that:

$$E_{k_1}\left(E_{k_2}(M)\right) = M$$

Q 2

where $E_k(M)$ is the encryption algorithm encrypting message M with key K. There are six semi-weak key pairs:

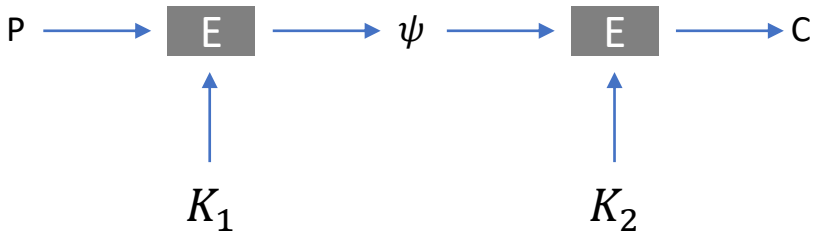
- 0x011F011F010E010E and 0x1F011F010E010E01
- 0x01E001E001F101F1 and 0xE001E001F101F101
- 0x01FE01FE01FE01FE and 0xFE01FE01FE01FE01
- 0x1FE01FE00EF10EF1 and 0xE01FE01FF10EF10E
- 0x1FFE1FFE0EFE0EFE and 0xFE1FFE1FFE0EFE0E
- 0xE0FEE0FEF1FEF1FE and 0xFEE0FEE0FEF1FEF1

There are also 48 possibly weak keys that produce only four distinct subkeys (instead of 16). They can be found in a NIST publication

These weak and semi-weak keys are not considered "fatal flaws" of DES. There are 2^{56} $(7.21 \times 10^{16}, about\ 72\ quadrillion)$ possible keys for DES, of which four are weak and twelve are semi-weak. This is such a tiny fraction of the possible keyspace that users do not need to worry. If they so desire, they can check for weak or semi-weak keys when the keys are generated. They are very few, and easy to recognize. Note, however, that currently DES is no longer recommended for general use since all DES keys can be brute-forced it's been decades since the Deep Crack machine was cracking them on the order of days, and as computers tend to do, more recent solutions are vastly cheaper on that time scale. Examples of progress are in Deep Crack's article

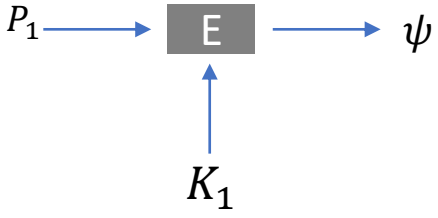
Q 3

We can use Meet in the Middle attack. In this attack, instead of focusing on the input and output of the entire encryption chain, we focus on the middle value. Suppose this is the value for ψ



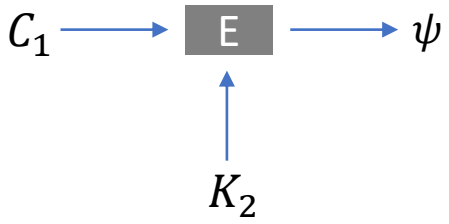
By having several pairs of C and P, you can easily get $K_1 \& K_2$

Suppose the first pair is C and p, because the length of each key is 56 bits, so all states of a key are 2^{56}. Now, having P_1 in hand and checking all 2^{56} states for k, we get all possible ψ's and store all pairs of k_1 and ψ.



Q 3

Then, having C_1 and checking all 2^{56} states of K_2, we obtain all possible ψ's and store all possible pairs of K_2 andψ.



Then we compare pairs (ψ, K_1) and (ψ, K_2) and select pairs that have equal ψ. Now we try the obtained K_1 and K_2 on other pairs of C and P. The K_1 and K_2 that answer correctly on the other C and P are actually real keys

Finally we have to check 2×2^{56} (2^{57}) states which is way faster than Brute-force attack with 2^{112} states.

Q 5

PKCS (Public-Key Cryptography Standards) were designed and published, in the 1990s, by RSA Security Inc, and have now been standardised in the form of RFCs. PKCS #5 (RFC 2859) is a standard used for password-based encryption, and PKCS #7 (RFC 2815) is used to sign and/or encrypt messages for PKI.

PKCS #7 (Cryptographic Message Syntax) is a standard padding method that determines the number of padding bytes and then ads that as a value. For example, for a 128-bit block size, and if we have “testing”, then there are seven bytes (for ASCII coding) that represent the data, and we then have 9 (0x09) padding values. The padding block becomes:

74657374696e670909090909090909

Padding is used in certain block cipher modes (like ECB and CBC) when the plain-text needs to be a multiple of the block size. If we are using a 16 byte block cipher, but our plain-text is only 9 bytes, then we need to pad out our data with 7 additional bytes.

To do this we append 7 bytes all with the value of 0x07. The general case is if we need to add N bytes to make a full block, we append N bytes each with a value of N. This works only if the block size is less than 256 bytes, since a byte can only be a value between 0 and 255.

Q 7

The basis of the FMS attack lies in the use of weak initialization vectors (IVs) used with RC4. RC4 encrypts one byte at a time with a keystream output from $prga()$; RC4 uses the key to initialize a state machine via ksa(), and then continuously modifies the state and generates a new byte of the keystream from the new state. Theoretically, the key stream functions as a random one-time pad, as a pseudo-random number generator controls the output at each step.

With certain IVs, an attacker knowing the first byte of the keystream and the first m bytes of the key can derive the $(m + 1)$th byte of the key due to a weakness in the KSA. Because the first byte of the plaintext comes from the WEP SNAP header, an attacker can assume he can derive the first byte of the keystream from $B \oplus 0xAA$ (the SNAP header is almost always 0xAA). From there, he only needs an IV in the form $(a + 3, n - 1, x)$ for key index a equal to 0, element value space n equal to 256 (since 8 bits make a byte), and any x. To start, the attacker needs IVs of (3, 255, x). WEP uses 24-bit IVs, making each value one byte long.

To start, the attacker utilizes the IV as the first 3 elements in K[ ]. He fills the S-box $S[\,]$ with sequential values from 0 to n as RC4 does when initializing the S-box from a known $K[\,]$. He then performs the first 3 iterations of $ksa()$ to begin initializing the S-box.

After the third step, the attacker can possibly, but not definitely, derive the fourth byte of the key using the keystream output O by computing $(O - j - S[i]) \bmod n = K[i]$, with the value $i = 3$ at this step.

At this point, the attacker does not yet have the fourth byte of the key. This algorithm does not regenerate the next byte of the key; it generates a possible value of the key. By collecting multiple messages—for example WEP packets—and repeating these steps, the attacker will generate a number of different possible values. The correct value appears significantly more frequently than any other; the attacker can determine the value of the key by recognizing this value and selecting it as the next byte. At this point, he can start the attack over again on the fifth byte of the key.

Although the attacker cannot attack words of the key out of order, he can store messages for later sequential attack on later words once he knows earlier words. Again, he only needs messages with weak IVs, and can discard others. Through this process, he can gather a large number of messages for attack on the entire key; in fact, he can store only a short portion of the beginning of those messages, just enough to carry the attack out as far as the word of the key the IV will allow him to attack.