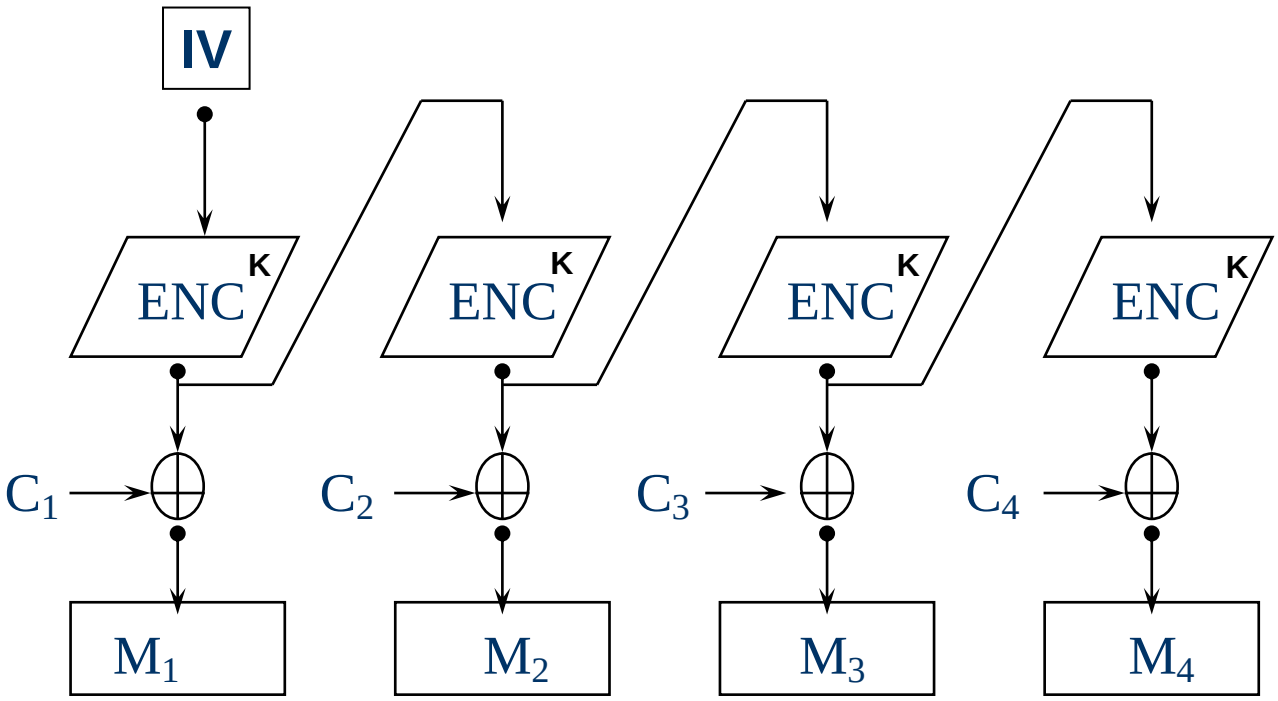
The University of Queensland
School of Information Technology and Electrical Engineering

Semester 2, 2017

COMS3000/7003 – Tutorial 10, Answers

Q1) Draw a block diagram of a block cipher in Output Feedback (OFB) mode (Decryption).

Answer:



b) Due to a transmission error, a Ciphertext block (say C_2) has a single bit error in the first bit of the block. How does this affect the plaintext block after decryption at the receiver?

The corresponding plaintext block (M_2) will have a single bit error at the same position as the ciphertext block. No other plaintext blocks will be affected.

A single bit error in one of the inputs of an XOR operation will always result in a single bit error (at the same position) in the output.

However, a single bit error in input data to a block cipher (encryption or decryption) will result in complete output block being affected. This is due to the so-called *Avalanche effect*, which is a desirable property of block ciphers and cryptographic hash functions. It means that when the input is changed slightly (e.g. flipping a single bit), the output will be completely different.

More formally, a function is said to satisfy the *strict avalanche criterion* if, whenever a single input bit is flipped, each of the output bits should change with a probability of one half.

Q2) Which of the following integers are prime numbers: 3, 11, 21, 29, 5677463534656?

3, 11, 29

21 is divisible by 3 and 7
5677463534656 is an even number and therefore divisible by 2, hence not a prime number.

By the way, the currently largest known prime number is $2^{57,885,161} - 1$, a number with 17,425,170 digits.

Even though prime numbers become sparser and sparser the higher the range of integers we consider, it can be proven that there exists an infinite number of prime numbers. This is important, since a lot of public key algorithms (e.g. RSA) need very large prime numbers.

Q3) The Fundamental Theorem of Arithmetic tells us that the primes are the building blocks of the positive integers: every positive integer is a product of prime numbers in one and only one way, except for the order of the factors.

'The Fundamental Theorem of Arithmetic', or 'Unique Factorisation Theorem':
Every positive integer greater than one can be expressed uniquely as a product of primes, apart from the rearrangement of terms.

For example, the prime factors of 18 are as follows: 18 = 2 * 3 *3

Find the prime factors of the following integers: 12, 33, 98, 100

Answer:

12 = 3 * 2 * 2
33 = 3 * 11
98 = 2 * 7 * 7
100 = 2 * 2 * 5 * 5

Q4) Find the results using modular arithmetic, i.e. calculation modulo an integer.

a) 33 mod 13 = ?

b) (44 * 22) mod 13 = ?

c) 3^4 mod 7 = ?

Answers:

a) 33 mod 13 = 7

b) (44 * 22) mod 13 = 968 mod 13 = 6
Or we can use one of the properties of modular arithmetic to simplify this:
(44 * 22) mod 13 = (44 mod 13 * 22 mod 13) mod 13 = (5 * 9) mod 13 = 45 mod 13 = 6

c) 3^4 mod 7 = (3*3*3*3) mod 7 = 81 mod 7 = 4

Q5)

a) Calculate $(5^{64}) \bmod 7 = ?$

You can use the following property of exponentiation: $(a^x)^y = (a^y)^x = a^{xy}$
(This property also holds for modular arithmetic.)

Using this property we can write the following:

$$5^{64} = (5^{32})^2$$
$$5^{32} = (5^{16})^2$$
$$5^{16} = (5^{8})^2$$
$$5^{8} = (5^{4})^2$$
$$5^{4} = (5^{2})^2$$
$$5^{2} = 5 * 5$$

In summary, we can write:
$$5^{64} = (((((5^2)^2)^2)^2)^2)^2$$

To find (5^{64}) we start by calculating $5^2 \bmod 7 = 25 \bmod 7 = 4$
Then we square this result again to get $5^4 \bmod 7 = (5^2)^2 \bmod 7 = 4^2 \bmod 7 = 16 \bmod 7 = 2$
The same for 5^8: $5^8 \bmod 7 = (5^4)^2 \bmod 7 = 2^2 \bmod 7 = 4 \bmod 7 = 4$

$5^{16} \bmod 7 = (5^8)^2 \bmod 7 = 4^2 \bmod 7 = 16 \bmod 7 = 2$
$5^{32} \bmod 7 = (5^{16})^2 \bmod 7 = 2^2 \bmod 7 = 4 \bmod 7 = 4$
$5^{64} \bmod 7 = (5^{32})^2 \bmod 7 = 4^2 \bmod 7 = 16 \bmod 7 = 2$

We have found the answer by doing only 6 multiplications instead of 64. This algorithm is called “square and multiply” and can easily be implemented in software. For any exponent x which is a power of 2, we only need $\log_2 x$ multiplications to find the result.

b) Calculate $5^{13} \bmod 7 = ?$
You can use the following two properties of exponentiation which also hold for modular arithmetic:
$(a^x)^y = (a^y)^x = a^{xy}$
$a^x * a^y = a^{x+y}$

In a) we had a special case since the exponent was a power of 2. The square and multiply algorithm can be applied for any exponent. For this, we first perform a binary expansion of the exponent, i.e. we simply write the exponent in binary format.

$13 = 1101_{\text{(binary)}}$

This basically means: $13 = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 8 + 4 + 1$

Using the second property mentioned above, we can write:

$5^{13} \bmod 7 = 5^{(8+4+1)} \bmod 7 = (5^8 * 5^4 * 5^1) \bmod 7 = (5^8 \bmod 7 * 5^4 \bmod 7 * 5^1 \bmod 7) \bmod 7$
We can now calculate each of these terms individually using the square-and-multiply algorithm and then multiply them to find the final result.

$5^2 \bmod 7 = 25 \bmod 7 = 4$
$5^4 \bmod 7 = (5^2)^2 \bmod 7 = 4 * 4 \bmod 7 = 16 \bmod 7 = 2$
$5^8 \bmod 7 = (5^4)^2 \bmod 7 = 2 * 2 \bmod 7 = 4 \bmod 7 = 4$

$5^{13} \bmod 7 = (5^8 * 5^4 * 5^1) \bmod 7 = (4 * 2 * 5) \bmod 7 = 40 \bmod 7 = 5$

In this way, we can use the square-and-multiply algorithm to do modular exponentiation. The cost of the algorithm for an exponent x is $O(\log_2 x)$, which is very efficient, even for very large exponents.

No such efficient algorithm exists to calculate, the inverse, i.e. the Discrete Logarithm. Therefore, modular exponentiation is a one-way operation.

Q6) We are doing calculations modulo n=5.

The discrete logarithm $\log_a c \bmod 5$ exists for all *c* if *a* is a '*generator'*.
(*a* is a generator if by computing a^b for b = {1,2…n-1}, all elements 1,2 …n-1 are "generated")

a) Verify that *a* = 3 is a generator.

$a^1 = 3 \bmod 5 = 3$
$a^2 = 9 \bmod 5 = 4$
$a^3 = 27 \bmod 5 = 2$
$a^4 = 81 \bmod 5 = 1$

a is a generator since it "generates" all the elements, i.e. 1,2,3,4.

b) Calculate the following discrete logarithms:

$\log_3 4 \bmod 5 = ?$
$\log_3 2 \bmod 5 = ?$

$3^1 \bmod 5 = 3$
$3^2 \bmod 5 = 9 \bmod 5 = 4$
$3^3 \bmod 5 = (3^2 * 3) \bmod 5 = (3^2 \bmod 5 * 3 \bmod 5) \bmod 5 = 4 * 3 \bmod 5 = 2$
$3^4 \bmod 5 = (3^2 * 3^2) \bmod 5 = (3^2 \bmod 5 * 3^2 \bmod 5) \bmod 5 = 4 * 4 \bmod 5 = 16 \bmod 5 = 1$

$\log_3 4 \bmod 5 = 2$
$\log_3 2 \bmod 5 = 3$

For small numbers, we can find the answer via trial and error. For large numbers, there exists no efficient algorithm to compute discrete logarithms.
(No polynomial time algorithm is known. If such an algorithm exists is one of the open questions in Computer Science. The same applies to the problem of factoring large integers.)

Q7) The Diffie-Hellman Key exchange protocol relies on the fact that it is easy to do modular exponentiation, but it is hard to do the inverse, i.e. to compute the "Discrete Logarithm."

Let's assume Alice and Bob want to establish a shared secret key using the Diffie-Hellman protocol.

They agree on a value n = p = 29 (prime number) and a generator a = g = 3. (The fact that gcd(3,p-1=28)=1 guarantees that 3 is a generator.)
These values do not have to be secret and can remain constant over a long period of time.

According to the protocol, Alice chooses a random number x (modulo 29), let's say x = 5.
Bob also chooses a random number y = 16.

Execute step by step the Diffie-Hellman protocol and calculate the shared secret key.
What can you say about the security of this protocol?

Answer:

Alice computes $X = g^x \bmod p = 3^5 \bmod 29 = 243 \bmod 29 = 11$
She sends this value to Bob over an insecure channel on which Eve is eavesdropping.

Bob computes $Y=g^y \bmod p = 3^{16} \bmod 29 = (((3^2)^2)^2)^2 \bmod 29$
(using square and multiply)

$3^2 \bmod 29 = 9$
$3^4 \bmod 29 = 9*9 \bmod 29 = 81 \bmod 29 = 23$
$3^8 \bmod 29 = 23*23 \bmod 29 = 529 \bmod 29 = 7$
$3^{16} \bmod 29 = 7*7 \bmod 29 = 49 \bmod 29 = 20$

Y=20
Bob sends this number to Alice over the same insecure channel.

Alice now has x and $Y=g^y$
She computes $K = Y^x = (g^y)^x = g^{xy}$
$K = Y^x = 20^5 \bmod 29 = (20^2 * 20^2 * 20) \bmod 29 = (400 \bmod 29 * 400 \bmod 29 * 20) \bmod 29$
$= (23 * 23 * 20) \bmod 29 = ((23*23) \bmod 29 * 20 \bmod 29) \bmod 29 = (7 * 20) \bmod 29 = 24$

Bob knows y and $X = g^x$
He computes $K = X^y = (g^x)^y = g^{xy}$
$K = 11^{16} \bmod 29 = ?$

$11^2 \bmod 29 = 121 \bmod 29 = 5$
$11^4 \bmod 29 = 5*5 \bmod 29 = 25$
$11^8 \bmod 29 = 25*25 \bmod 29 = 16$
$11^{16} \bmod 29 = 16*16 \bmod 29 = 24$
K = 24

Alice and Bob now know the shared secret key K=24

Eve knows $X = g^x$ and $Y = g^y$, but cannot compute g^{xy} without computing either x or y. To find x or y, Eve needs to compute the Discrete logarithm of X or Y respectively. In this special case, Eve could simply compute the discrete logarithm by trying all the possible 28 numbers. However, for large p (>1000 bits) this is computationally infeasible.

However, the Diffie Hellman protocol is susceptible to the Man-in-the-middle attack.

Q8) Describe how Trudy could use a Man-in-the-middle-attack to attack the Diffie-Hellman session establishment protocol between Alice and Bob.

In this attack, Trudy intercepts Alice's public value X and sends her own public value F to Bob. When Bob transmits his public value Y, Trudy substitutes it with her own version H and sends it to Alice. Trudy and Alice thus agree on one shared key Ka and Trudy and Bob agree on another shared key Kb. After this exchange, Trudy simply decrypts any messages sent out by Alice or Bob, and then reads and possibly modifies them before re-encrypting with the appropriate key and transmitting them to the other party. This vulnerability is present because Diffie-Hellman key exchange does not authenticate the participants.

Q9) Consider an RSA system with p=7 and q=11. Find the parameters *n* and *z* and also find a valid parameter *e*.

Answer:
n = p*q = 77
z = (p-1)(q-1) = 6 * 10 = 60
e needs to be relatively prime to z, i.e. it cannot share a common factor with z except for 1.
The prime factors of 60 are: 60 = 2*2*3*5

Possible choices for e are: 7, 11, 13, 17 …

Q10) Consider an RSA system with the following parameters:
p=3, q=11
n = p * q = 33
z = (p-1)(q-1) = 2 * 10 = 20
e = 7

a) Find the secret key *d*.

Answer:
The extended Euclid algorithm provides an efficient method for finding d, given the prime factors p and q of n are known.

In our simple example, we can just find the answer by trying different values of d. We know that the following condition must hold: e*d mod z = 1

Trial and error:

d=1: e*d mod z = 7 * 1 mod 20 = 7
d=2: e*d mod z = 7 * 2 mod 20 = 14
d=3: e*d mod z = 7 * 3 mod 20 = 1 We have found d = 3

b) Use this system to encrypt the following binary data: “01100111”

Since we are using a modulus of n=33, we can encode, and therefore encrypt, the numbers {0,1, 2, …, 32}. We therefore need to segment our plaintext message into blocks of no greater than 32 (i.e. five or six bits – 0-31 requires 5 bits, to do 32 would require 6 bits). Since we cannot encrypt an 8-bit byte with such a small modulus, the easiest method would be to encrypt 4-bit nibbles (i.e. 4-bit blocks).

Thus our binary data consists of two messages “0110” and “0111”, that is 6 and 7:
$m_1 = 6$
$m_2 = 7$

Now we encrypt each of these messages individually.

$c = m^e \bmod n$

$c_1 = m_1^e \bmod 33 = 6^7 \bmod 33$

We can use square-and-multiply for this:
$7 = 4 + 2 + 1 = 2^2 + 2^1 + 2^0$
$6^7 \bmod 33 = (6^4 \bmod 33 * 6^2 \bmod 33 * 6 \bmod 33) \bmod 33$

$6^2 \bmod 33 = 36 \bmod 33 = 3$
$6^4 \bmod 33 = ((6^2 \bmod 33) * (6^2 \bmod 33)) \bmod 33 = 3*3 \bmod 33 = 9 \bmod 33 = 9$

$\mathbf{c_1} = 6^7 \bmod 33 = (9*3 \bmod 33 * 6 \bmod 33) \bmod 33 = 27 * 6 \bmod 33 = 162 \bmod 33 = \mathbf{30}$

$c_2 = m_2^e \bmod 33 = 19^7 \bmod 33 =$
$7^7 \bmod 33 = (7^4 \bmod 33 * 7^2 \bmod 33 * 7 \bmod 33) \bmod 33$
$7^2 \bmod 33 = 49 \bmod 33 = 16$
$7^4 \bmod 33 = ((7^2 \bmod 33) * (7^2 \bmod 33)) \bmod 33 = 16*16 \bmod 33 = 256 \bmod 33 = 25$

$\mathbf{c_2} = 7^7 \bmod 33 = (25*16 \bmod 33 * 7) \bmod 33 = (400 \bmod 33 * 7) \bmod 33 = 4*7 \bmod 33 = 28 \bmod 33 = \mathbf{28}$

Our ciphertext is therefore as follows: 30,28 or “011110”,”011100” i.e. “011110011100”
(note this ciphertext is longer than our plaintext, because we only encrypt every 4 bits with a 6-bit modulus).

Such a small modulus is not secure – nor is the ECB method we used here!

Q11) Explain why in a real RSA system an attacker cannot find the secret key *d*.

Answer:
For large integers, there exists an efficient algorithm to find the secret key d that corresponds to the public key e, only if the prime factors p and q of the modulus n are known. The factors p and q are kept secret and computing these values from the modulus n, i.e. factoring n, is considered a hard problem for large n.

No efficient algorithm for factoring large integers has been found so far. However, nobody has proven that no such algorithm exists. The security of RSA relies on the fact that factoring remains a hard problem.

Without the (secret) factors p and q, it is computationally impossible to find the secret key d from the public key e.

Q12) Encrypting or signing large files with RSA is computationally very expensive due to the large number of modular exponentiations with very large integers that are involved. Explain how cryptographic one-way hash functions can be used to increase the efficiency of digital signatures using RSA.
Explain why this mechanism also provides integrity.

Answer:
We assume we have a cryptographic one-way hash function h() and a signature algorithm sigK(). The signature algorithm could be encryption with the secret key d in RSA. We assume we want to sign a large file f.

We first compute the hash of the file h(f). Then, we apply the signature algorithm to that and we compute s = sigK(h(f)).

The signature s is stored with the file.

Verification of the signature is as follows:

The verifier verifies the signature by decrypting it with the public RSA key. If the result is the same as the result from computing h(f), then the signature is valid.

Let's assume Trudy tampers with the file and flips the first bit, resulting in a different file f2. She can compute the hash of the new file h(f2) but she cannot provide a valid signature since she does not have the secret RSA key required for that.

A verifier would compute the hash of the file h(f2) and compare it with the value h(f) obtained by decrypting the signature with the public RSA key. These two values do not match and therefore the signature is not valid.