
UNIVERSITI
MALAYA
*Centre for Continuing Education*

# MASTER IN DATA SCIENCE

# ASSIGNMENT 2

**COURSE CODE** : WQD7010

**COURSE TITLE** : NETWORK & SECURITY

**NAME** : NUR HIDAYAH BINTI AHMAD SHAFII

**MATRIC NUMBER** : 22120931

**CLASS** : 1

**LECTURER** : DR. SAAIDAL RAZALLI BIN AZZUHRI

1.

a. To evaluate the hashing algorithm's ability to detect tampering, we can examine XOR and RXOR functions when the input data is tampered with by flipping an odd number of bits. In XOR functions, it returns true (1) if the inputs are different and false (0) if they are the same. It's commutative, associative, and yields 0 when XORed with itself. As for RXOR functions, it computes XOR in reverse bit order and also commutative and associative. When an attacker flips an odd number of bits, it alters an odd number of hash output bits. Both the XOR and RXOR operations used in the algorithm yield non-zero results when comparing original and tampered hashes because XOR will return 1 for each pair of bits that differs between the original and tampered hash and reversal of bit order doesn't affect the outcome when performing RXOR as long as the bit differences remain. The presence of a non-zero result after comparing original and tampered hashes indicates that data integrity has been compromised due to differences revealed in bit patterns between hashes through the nature of XOR and RXOR functions.

b. When an even number of bits are altered, hash function behavior becomes complex. If the tampering involves a complete reversal of all bits, XORing original and tampered hash results in zero, but RXOR may detect the change due to bit order alteration. If the tampering involves flipping bits in pairs, XOR operation may not detect paired bit flips as they cancel each other out. Similarly, RXOR may also fail if pairs maintain their positions after reversal. If the tampering involves altering blocks of bits such that each block has an even number of flipped bits, both XOR and RXOR operations may fail to detect tampering if each block has an even number of flipped bits that cancel out within the block. If an even number of bits are flipped overall with differences between original and tampered hashes, both XOR and RXOR operations may detect the tampering. Therefore, the hash function's effectiveness in identifying even-bit tampering varies depending on the specific pattern of alterations and the characteristics of XOR and RXOR operations.

c. From a security perspective, using a 32-bit hashing algorithm for authentication presents significant challenges in providing strong protection. Due to its limited output space, the algorithm lacks robust resistance to collusion, making it vulnerable to collision attacks, particularly when adversaries can manipulate input data. Even small datasets have the potential to lead to collisions due to the restricted number of possible hash values. Therefore, the algorithm may not provide strong resistance against collusion, particularly when faced with determined attackers possessing significant computational resources. Additionally, preimage resistance is crucial to prevent reversing input from hash value. With only 32 bits of output, preimage space is limited and vulnerable to brute-force attacks. Modern computational power makes brute-force attacks feasible, reducing preimage resistance. Therefore, a 32-bit hashing algorithm may not provide sufficient protection for data integrity and authenticity, especially in security-sensitive applications. Limited hash space increases collision likelihood and makes brute-force attacks more practical, compromising resistance to collusion and preimage. Stronger hashing algorithms with larger output sizes should enhance security by mitigating collision and preimage attacks effectively. Incorporating cryptographic techniques like salting and key stretching can further bolster security by introducing additional complexities for attackers. While a 32-bit hashing algorithm may suffice for simple applications with minimal security requirements, it falls short in scenarios requiring stronger security guarantees.

2. a)

| Requirement | Description |
|---|---|
| Pre-image resistance | The given hash function does not provide pre-image resistance. It's easy to find an input that hashes to a given output by simply selecting any combination of integers mi that sum up to the desired output mod $n$ . |
| Second pre-image resistance | It does not satisfied this requirement. Given an input $D = (m_1, m_2, \dots, m_i)$, it's relatively easy to find another input that hashes to the same output. For example, $D' = (m_1 + kn, m_2, \dots, m_i)$ where *k* is any integer, would produce the same s hash value because $\left(\sum_{i=1}^{t} m_1\right) + kn$ mod $n$ would yield the same result. |
| Collision resistance | It does not satisfied this requirement. Due to the infinite nature of combinations of integers $m_i$, it becomes extremely challenging to ensure that no two different inputs will hash to the same output. This is particularly true when considering the periodicity introduced by the modulo operation. |
| Avalanche effect: | This property is partially met. Any slight alteration in the $m_i$ values would yield a distinct sum, and considering we're employing modulo arithmetic, the resulting output would also differ. Nevertheless, the claim of being "significantly different" may not universally apply to all inputs, particularly when $n$ greatly surpasses the combined sum of the $m_i$ values. |
| Efficiency | This requirement is fulfilled through a hash function that consists of a straightforward summation of '$t$ ' integers followed by a modulo operation, which is computationally efficient. |

In conclusion, although the provided hash function is efficient, it does not fulfill the fundamental criteria of a cryptographic hash function, specifically pre-image resistance, second pre-image resistance, and collision resistance. Consequently, it is unsuitable for cryptographic applications.

b)

| Requirement | Description |
|---|---|
| Pre-image resistance | It does not satisfied this requirement. Given a hash output, it's possible to find multiple inputs that hash to the same output due to the non-injective nature of the function. For example, $D = (m_1, m_2, .., m_t)$ and $D' = (-m_1, m_2, .., m_t)$ would both hash to the same value $x$ mod $n$ |
| Second pre-image resistance | It does not satisfied this requirement. it's relatively easy to find another input that hashes to the same output with an input $D = (m_1, m_2, \dots, m_t)$ due to the non-injective nature of the function. |
| Collision resistance | It does not satisfied this requirement. It's easy to identify collisions because of the function's nature, which makes it unsuitable for cryptographic use. |
| Avalanche effect | This requirement is partially met. Minor changes in the input could lead to varied hash outputs because of the non-linear characteristic of squaring. Nevertheless, "Significantly different" may not hold true for all inputs, particularly when $n$ greatly surpasses the combined sum of the $m_1$values. |
| Efficiency | This requirement is fulfilled in some extent since hash function includes a sum of squares followed by a modulo operation, which is computationally efficient. |

In conclusion, although the provided hash function demonstrates efficiency and certain characteristics of an avalanche effect, it does not fulfill the fundamental criteria of a cryptographic hash function. Specifically, it lacks pre-image resistance, second pre-image resistance, and collision resistance due to its non-injective nature. Consequently, it is unsuitable for cryptographic applications.

c) $h = \left(\sum_{i=1}^{t} m_1 \,.\, m_1\right) mod\, n$

$$Sum\ of\ squares = (237^2 + 632^2 + 913^2 + 423^2 + 349^2) = 1592412$$

$$Sum = (237 + 632 + 913 + 423 + 349) = 2554$$

$$h = (2554.\ 1592412)\ mod\ 757 = 73$$

Hence, the hash value for the message D is 73.

3. a) $n = pq = (17)(19) = 323$

$$Let\ A = (p-1)(q-1) = (17-1)(19-1) = 288$$

b)
Public key pairs component:
- Modulus, $n = pq = 323$
$Choose,\ e = 5$

$$public\ key,\ K_p = (n, e) = (323,5)$$

Private key pairs component:
Private exponent, $d$

Using the Extended Euclidean Algorithm,

$$x_{i+1} = x_{i-1} - \left(\left|\frac{x_i}{y_i}\right| \times x_i\right)$$
$$y_{i+1} = y_{i-1} - \left(\left|\frac{x_i}{y_i}\right| \times y_i\right)$$

$$x_1 = 1,\ y_1 = 0$$
$$x_2 = 0,\ y_2 = 1$$
$$x_3 = 1 - \left(\left|\frac{5}{288}\right| \times 0\right) = 1,\ y_3 = 0 - \left(\left|\frac{5}{288}\right| \times 1\right) = -57$$
$$x_4 = 0 - \left(\left|\frac{288}{5}\right| \times 1\right) = -57,\ y_4 = 1 - \left(\left|\frac{288}{5}\right| \times (-57)\right) = 4$$
$$x_5 = 1 - \left(\left|\frac{5}{4}\right| \times (-57)\right) = 229,\ y_5 = -57 - \left(\left|\frac{5}{4}\right| \times 4\right) = -13$$
$$x_6 = -57 - \left(\left|\frac{4}{(-13)}\right| \times 229\right) = 173,\ y_6 = 4 - \left(\left|\frac{4}{(-13)}\right| \times (-13)\right) = 3$$

Hence, d=173 since it is the modular multiplication inverse of e=5 and A=288.

Private key = (d,n) = (173, 323)

c) R=83, S=83, A=65

The numerical representation of the plaintext message "RSA" is the sequence of these ASCII values: M=[82 83 65]

d) M1 = 82, M2 = 83, M3 = 65
Using $C = M^e \ mod \ N$
For M1 = 82,

$$C_1 = 82^5 \ mod \ 323 \ = 233 \ mod \ 323 \ = 233$$

For M2 = 83,

$$C_2 = 83^5 \ mod \ 323 \ = 87 \ mod \ 323 = 87$$

For M3 = 65,

$$C_3 = 65^5 \ mod \ 323 \ = 12 \ mod \ 323 \ = \ 12$$

Hence, the ciphertext values are [127 184 135]

e) Given ciphertext value = 2190236
Using $M = C^d \ mod \ N$

$$M = 2190236^{173} \ mod \ 323 \ = 159 \ mod \ 323 = 159$$

From ASCII table, 159 equivalent to character “f”. Hence, the decrypted message is f.

4.

a) $public\ key,\ YA = \alpha^{XA} mod\ q \quad where\ XA = \ private\ key$

$$40 = 3^{XA} mod\ 353$$

Using brute-force search to find the value of secret key XA. The brute-force attack is to calculate powers of 3 modulo 353, stopping when result equals 40. The desired answer is reached with the exponent value of 97, which provides $40 = 3^{97} mod\ 353$ .

B) $248 = 3^{XB} mod\ 353$

Using brute-force search to find the value of secret key XB. The brute-force attack is to calculate powers of 3 modulo 353, stopping when result equals 248. The desired answer is reached with the exponent value of 233, which provides $248 = 3^{233} mod\ 353$

The shared secret key:

A computes $K = (YB)^{XA}\ mod\ q = 248^{97} mod\ 353 = 160$

B computes $K = (YA)^{XB}\ mod\ q = 40^{233} mod\ 353 = 160$

c) XE=201, User B private key XB =156

$$YE = 3^{201}\ mod\ 353 = 264\ mod\ 353 = 264$$

Hence, Eve's public key is 264.

$$K_{BE} = 264^{156}\ mod\ 353 = 285\ mod\ 353 = 385$$

The shared key between user B and Eve is approximately 385.

d) YB=248, XE=201, q=353, XA=97,YE=264

$$K_{EB} = 248^{201}\ mod\ 353 = 55\ mod\ 353 = 55$$

Hence, the shared key between Even and User B is approximately 55.

$$K_{AE} = 264^{97}\ mod\ 353 = 301\ mod\ 353 = 301$$

Hence, the shared key between User A and Eve is approximately 301.

e) YA = 40, XE = 201

$$K = 40^{201}\ mod\ 353 = 301\ mod\ 353 = 301$$

Hence, the shared secret key between user A and Eve is approximately 301.

YB = 248, XE = 201

$$K = 248^{201}\ mod\ 353 = 55\ mod\ 353 = 55$$

Hence, the shared secret key between user B and Eve is approximately 55.

Eve intercepts the encrypted messages exchanged between User A and User B. Leveraging the established shared secret keys (K_AE for User A and K_EB for User B), Eve decrypts the intercepted messages. Possessing knowledge of these shared secret keys enables Eve to decipher messages encrypted with the corresponding public keys, thus granting her access to the plaintext content of the communication between User A and User B without their awareness. This unauthorized access allows Eve to eavesdrop on their conversation, potentially obtaining sensitive information.

f) The steps that Eve can take to modify the message encrypted with the shared secret key K_AE and then send it to User B, making User B believe it came securely from User A are Eve need to intercept the encrypted message that User A sends to User B. Next, shw will need to proceed with decrypting it by using her private key XE and the shared secret key K_AE. After obtaining the original plaintext message, she modifies it according to her intentions and encrypts it using the shared secret key K_BE before sending it to User B. From User B's perspective, the message appears to be securely sent from User A due to encryption with the correct shared secret key. However, in reality, Eve has manipulated its content before transmission. Upon receiving this modified message, User B decrypts it using their shared secret key K_BE and may mistakenly believe that it originated from User A. Thus, Eve gets to see every message sent, and User A and B never suspect a thing.

The absence of authentication in the Diffie-Hellman key exchange protocol makes it vulnerable to man-in-the-middle attacks. This means that an adversary could intercept and manipulate the key exchange process, allowing them to establish separate shared secret keys with each party involved. This can lead to decryption and potential alteration of communication without detection. To address this vulnerability, additional security measures such as digital signatures or message authentication codes (MACs) can be introduced. Digital signatures validate the authenticity of messages by requiring them to be signed with the sender's private key, which can then be verified using their public key. Similarly, MACs ensure message integrity by generating a cryptographic tag based on the message content and a shared secret key, which the receiver can verify upon message receipt. Incorporating these measures strengthens the Diffie-Hellman key exchange protocol against man-in-the-middle attacks, enhancing both confidentiality and integrity of communication.

5.
Variant MAC function: VMAC(K,M) = CBC(K,M) XOR K1
Messages 0 = 0^n
Messages 1 = 1^n
message (1||0) which is the message 1 concatenated with message 0

MACs obtained by the adversary:
T0 = CBC(K,0) XOR K1
T1 = CBC(K,0) XOR K1
T2 = CBC(K,[CBC(K,1)]) XOR K1

Since T0 = T1, new message will be 0 || (T0 XOR T1)

$$T0 \bigoplus T1 \;=\; (CBC(K,0)\; XOR\; K1) \bigoplus CBC(K,0)\; XOR\; K1$$

$$T0 \bigoplus T1 \;=\; \Big(CBC(K,0) \bigoplus CBC(K,0)\Big) XOR\; K1 \bigoplus K1$$

$$T0 \bigoplus T1 \;= 0^n\; XOR\; 0^n$$

$$T0 \bigoplus T1 \;= 0^n$$

Hence, the new message is $0^n$

The MAC for the new message:

$$VMAC(K, 0^n) = CBC(K, 0^n)\; XOR\; K1$$

Since message 0 is all zeroes,

$$VMAC(K, 0^n) = CBC(K, 0)\; XOR\; K1$$

$$VMAC(K, 0^n) = 0$$

The MAC for the new message 0||(T0 XOR T1) is the same as the MAC for the message 0. Therefore, the adversary can compute the correct MAC for the new message without making any additional queries to the MAC oracle.

When modifying the Cipher-based Message Authentication Code (CMAC) structure to accommodate message lengths that do not evenly divide the block size, an important adjustment is incorporating padding. In the original CMAC configuration, when the message length aligns precisely with the block size, each block undergoes direct processing through the ciphering function to produce the final MAC. However, in cases where the message length is not a perfect multiple of the block size, padding becomes necessary to enable proper processing. A commonly used approach involves bit padding, wherein a '1' bit is appended at the end of the message followed by '0' bits until the total message length reaches a congruent state modulo with respect to the block size. If the message length already satisfies this congruency, a single block of '0's followed by a '1' bit is added. This padding ensures that correct processing of messages occurs within CMAC algorithm and helps maintain its integrity and security.