**Assignment 2 – Cloud Computing**

# **Question 1**

Getting information ready for steps

1.Convert personal information into Hex.

For my personal information it would be as follows:

Joshua Nathan Cayetano, s3722151, s3722151@student.rmit.edu.au, BP162, Cloud Security (2510)

I first need to convert it into (hex) and use these values for a matrix: <https://www.rapidtables.com/convert/number/ascii-to-hex.html>

= 4A6F73687561204E617468616E204361796574616E6F2C2073333732323135312C2073333732323135314073747564656E742E726D69742E6564752E61752C2042503136322C20436C6F756420536563757269747920283235313029

200 bits = 25 bytes, I need to take first 25 hex pairs (50 hex digits). =

4A6F73687561204E617468616E204361796574616E6F2C2073

NOTE: The above is equal to “Joshua Nathan Cayetano, s”.

* Verify: <https://www.lettercount.com/>

2.Placing information into a 5\*5 State Matrix array.

SHA needs to process information in a 5\*5 matrix. Generally, a entry is split into lanes depending on the SHA method used. For SHA-512 an entry is split into 64-bit lane. In contrast if you use SHA-256 an entry is split across 32-bit lanes. I have been asked to use first 200 bits of my personal information (25 bytes) so I need to pad it later. This is handled in sponge construction padding.

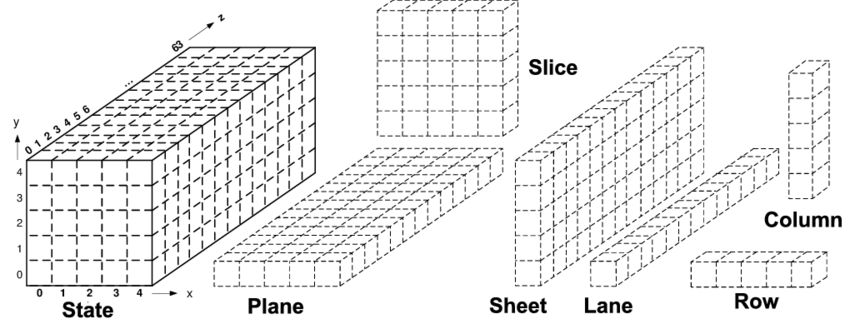


Figure - State Structure

A 5\*5 matrix means a state matrix will have 5 rows(y) and 5 columns(x). How a value is referred/indexed in a state matrix is by A[x][y]. My input has 200 bits so that means I will use 8 bits (1 byte per entry).

How it will look be structured will be in like the following. We start from bottom left (0,0) then fill matrix until top-right (4,4) for SHA-3 Keccak.

*Matrix Structure*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **X=0** | **X=1** | **X=2** | **X=3** | **X=4** |
| **Y=4** | 20 | 21 | 22 | 23 | 24 |
| **Y=3** | 15 | 16 | 17 | 18 | 19 |
| **Y=2** | 10 | 11 | 12 | 13 | 14 |
| **Y=1** | 05 | 06 | 07 | 08 | 09 |
| **Y=0** | 00 | 01 | 02 | 03 | 04 |

*State Matrix*

In my case I will use my hex values of my personal information to produce the following:

INPUT = **~~4A 6F 73 68 75~~ ~~61 20 4E 61 74~~ 68 61 6E 20 43 61 79 65 74 61 6E 6F 2C 20 73**

* **Yellow is y = 0, light blue is y = 1, green is y = 2, pink is y = 3, dark blue is y = 4.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **X=0** | **X=1** | **X=2** | **X=3** | **X=4** |
| **Y=4** | 6E | 6F | 2C | 20 | **73** |
| **Y=3** | 61 | 79 | 65 | 74 | 61 |
| **Y=2** | 68 | 61 | 6E | 20 | 43 |
| **Y=1** | 61 | 20 | 4E | 61 | 74 |
| **Y=0** | **4A** | 6F | 73 | 68 | 75 |

Now we can proceed to doing Theta.

## **1.1) What is the output of mapping in the first round?**

Theta is the first step in Keccak. Generally, the data is arranged in a 3D structure of bits, 5\*5\*W (W representing a lane size). What we do is mix the bits from our state matrix to spread changes evenly.

Concepts

**State Matrix:** This is the 5\*5 grid we used to hold our matrix values. A lane holds a value across different slices. E.g. the hex value 4A. In a space, this can only contain 1 bit. However, a value will be equal or less than 8 bits( 1 byte). So the value is across different another slice.

**Column parity,** C[x]: For each column we XOR all values (checksum).

**Diffusion:** In this step, we XOR each lane with parity bits from neighbouring columns to scramble.

*Steps in Theta*

NOTE: The formulas refer to figure “Theta – University Notes”.

**1. Compute Column Parity (C[x])**

Column Parity refers to summing all the columns in a slice.

For each column x (0 to 4), calculate the XOR of all 5 lanes in that column across all layers:

* C[x]= A[x,0] ⊕A [x,1] ⊕A [x,2] ⊕A[x,3] ⊕A[x,4]
* **What it does**: C[x] captures the parity, sum of a columns value (x) in state matrix.

**2. Compute Diffusion Vector (D[x])**

For each column [x], we compute D[x] using the parity we previously calculated.

* **D[x]** = **C[x−1]** ⊕ **rot(C[x+1],1)**
* **C[x-1]**:
  + This uses the parity next to the left side of X with wrap around (E.g. C[-1] = C[4]).
* **C[x+1]**:
  + This is the parity on the right side of X with wrap around (E.g. C[5] = C[0]).
* Rot (C[x+1], **1**)
  + This rotates the result of C[x+1] left by a position of 1.
  + Rot is used to ensure changes are spread in a non-linear way.

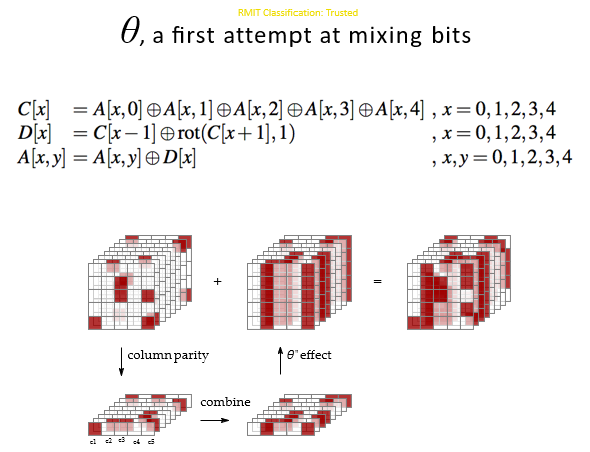
Key idea: D[x] introduces diffusion in non-straightforward approach.

This rotation applies lane wise. It shifts bits within a lane. By making it non-linear, the values would be less predictable.

**3. Apply Theta Effect to the State Matrix (A[x,y] = A[x,y] ⊕ D[x])**

Update every bit in the State Matrix (initial matrix we have) by XORing it with its column’s D[x]:

Result: Each bit is flipped based on its column’s neighbors, creating **non-local diffusion**.



Theta Visualisation with equations (X Yi, 2025)

*Example*

Example State Matrix

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **X=0** | **X=1** | **X=2** | **X=3** | **X=4** |
| **Y=4** | 1 | 2 | 3 | 1 | 2 |
| **Y=3** | 2 | 3 | 1 | 2 | 3 |
| **Y=2** | 3 | 1 | 2 | 3 | 1 |
| **Y=1** | 1 | 2 | 3 | 1 | 2 |
| **Y=0** | 2 | 3 | 1 | 2 | 3 |

Step 1: Calculate Column Parity C[x]

In my case, I reverse the calculations. So I calculated bottom to top for each column.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Column 0** | **Column 1** | **Column 2** | **Column 3** | **Column 4** |
| 1⊕2⊕ 3⊕ 1⊕ 2  =  3 | 2⊕ 3⊕ 1⊕ 2⊕3  =  1 | 3⊕ 1⊕2⊕3⊕1  =  2 | 1⊕2⊕3⊕1⊕2  =  3 | 2⊕ 3⊕1⊕2⊕3  =  1 |

C = 3,1,2,3,1

Step 2: Calculate D[x] Diffusion Vector

The formula is **D[x]** = **C[x−1]** ⊕ **rot(C[x+1],1)**

We ensure we wrap around 5 with x-1 and x+1. 5 comes from mod 5.

* E.g. If X = 0, x-1 = 4, x+1 = 1

a ) Compute **rot(C[x+1],1)**

This means we rotate left by 1 bit. This means we shift all bits by 1 to the left. The most significant bits also wrap around LSB (least significant bits).

For small hex values (4 bits), C[x+1] = 1 (binary 0001 -> rotated 0010 = 2.

|  |  |  |  |
| --- | --- | --- | --- |
| **Columns** | **C[x+1]** | **Binary Form** | **rot(C[x+1],1)** |
| 0 | C[1] = 1 | 0001 –> 0010 | 2 |
| 1 | C[2] = 2 | 0010 -> 0100 | 4 |
| 2 | C[3] = 3 | 0011 -> 0110 | 6 |
| 3 | C[4] = 1 | 0001 -> 0010 | 2 |
| 4 | C[0] = 1 | 0011 -> 0110 | 6 |

B ) **C[x−1]** ⊕ **rot(C[x+1],1)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Columns (x)** | **C[x-1]** | **rot(C[x+1],1)** | **D[x]** = **C[x−1]** ⊕ **rot(C[x+1],1)** |
| 0 | C[4] = 1 | 2 | 1 XOR 2 = **3** |
| 1 | C[0] = 3 | 4 | 3 XOR 4 = **7** |
| 2 | C[1] = 1 | 6 | 1 XOR 6 = **7** |
| 3 | C[2] = 2 | 2 | 2 XOR 2 = **0** |
| 4 | C[3] = 3 | 6 | 3 XOR 6 = **5** |

**Diffusion is [3,7,7,0,5,]**

Step 3: Apply Diffusion to the state

|  |  |  |  |
| --- | --- | --- | --- |
| **X,Y** | **Original value, A[x][y]** | **D[x]** | **New A’ [X][Y] = A[X][Y] XOR D[x]** |
| 0,0 | 2 | 3 | 2 XOR 3 = 1 |
| 0,1 | 1 | 3 | 1 XOR 3 = 2 |
| 0,2 | 3 | 3 | 3 XOR 3 = 0 |
| 0,4 | 2 | 3 | 2 XOR 3 1 |
| 1,0 | 3 | 7 | 3 XOR 7 = 4 |
| 1,1 | 2 | 7 | 2 XOR 7 = 5 |
| 1,2 | 1 | 7 | 1 XOR 7 = 6 |
| 1,3 | 2 | 7 | 2 XOR 7 = 5 |
| 1,4 | 3 | 7 | 3 XOR 7 = 4 |
| 2,0 | 1 | 7 | 1 XOR 7 = 6 |
| 2,1 | 3 | 7 | 3 XOR 7 = 4 |
| 2,2 | 2 | 7 | 2 XOR 7 = 5 |
| 2,3 | 3 | 7 | 3 XOR 7 = 4 |
| 2,4 | 1 | 7 | 1 XOR 7 = 6 |
| 3,0 | 2 | 0 | 2 XOR 0 = 2 |
| 3,1 | 1 | 0 | 1 XOR 0 = 1 |
| 3,2 | 3 | 0 | 3 XOR 0 = 3 |
| 3,3 | 1 | 0 | 1 XOR 0 = 1 |
| 3,4 | 2 | 0 | 2 XOR 0 = 2 |
| 4,0 | 3 | 5 | 3 XOR 5 = 6 |
| 4,1 | 2 | 5 | 2 XOR 5 = 7 |
| 4,2 | 1 | 5 | 1 XOR 5 = 4 |
| 4,3 | 2 | 5 | 2 XOR 5 = 7 |
| 4,4 | 3 | 5 | 3 XOR 5 = 6 |

Step 4 The final state of Theta

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **X=0** | **X=1** | **X=2** | **X=3** | **X=4** |
| **Y=4** | 2 | 5 | 4 | 1 | 7 |
| **Y=3** | 1 | 4 | 6 | 2 | 6 |
| **Y=2** | 0 | 6 | 5 | 3 | 4 |
| **Y=1** | 2 | 5 | 4 | 1 | 7 |
| **Y=0** | 1 | 4 | 6 | 2 | 6 |

Calculations – For Question

First XOR the columns in our state matrix.

<https://xor.pw/>

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Column 0** | **Column 1** | **Column 2** | **Column 3** | **Column 4** |
| 6E⊕ 61= f  F ⊕ 68 = 67  67 ⊕61 = 6  6 ⊕ 4A = 4C  Result =  4c | 6F⊕79 = 16  16⊕ 61= 77  77⊕20= 57  57 ⊕6F= 38  Result =  38 | 2C⊕ 65= 49  49 ⊕ 6e = 27  27 ⊕4E = 69  69⊕73= 1a  Result =  1a | 20⊕ 74= 54  54⊕ 20= 74  74⊕ 61= 15  15 ⊕ 68= 7d  Result =  7d | 73⊕ 61= 12  12⊕ 43= 51  51 ⊕ 74= 25  25⊕ 75= 50  Result =  50 |

Note: X refers to the column.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **X** | 0 | 1 | 2 | 3 | 4 |
| **C[X]** | 4C | 38 | 1a | 7d | 50 |

Now we calculate D[x]. We use the following code in java. Run **as Q1Theta.java.**

* It will take way to long to calculate by hand and validate.

public class Q1Theta {

    public static void main(String[] args) {

        // Column parity C[x] (precomputed)

        int[] C = { 0x4C, 0x38, 0x1A, 0x7D, 0x50 };

        // Step 1: Calculate D[x] = C[(x-1)%5] ⊕ rot(C[(x+1)%5], 1)

        int[] D = new int[5];

        for (int x = 0; x < 5; x++) {

            int left = C[(x - 1 + 5) % 5]; // Handle wrap-around

            int right = C[(x + 1) % 5];

            int rotatedRight = rotateLeft(right, 1); // Rotate left by 1 bit

            D[x] = left ^ rotatedRight;

        }

        // State matrix (y=4 at the top, y=0 at the bottom)

        int[][] A = {

                { 0x6E, 0x6F, 0x2C, 0x20, 0x73 }, // y=4

                { 0x61, 0x79, 0x65, 0x74, 0x61 }, // y=3

                { 0x68, 0x61, 0x6E, 0x20, 0x43 }, // y=2

                { 0x61, 0x20, 0x4E, 0x61, 0x74 }, // y=1

                { 0x4A, 0x6F, 0x73, 0x68, 0x75 }  // y=0

        };

        // Step 2: Update state A[x][y] ^= D[x]

        for (int x = 0; x < 5; x++) {

            for (int y = 0; y < 5; y++) {

                A[x][y] ^= D[x];

            }

        }

        // Print results

        System.out.println("D[x] values:");

        for (int x = 0; x < 5; x++) {

            System.out.printf("D[%d] = %02x\n", x, D[x]);

        }

        System.out.println("\nState after Theta step:");

        System.out.println("y\\x   x=0    x=1    x=2    x=3    x=4");

        System.out.println("----------------------------------------");

        for (int y = 4; y >= 0; y--) {

            System.out.printf("y=%d | ", y);

            for (int x = 0; x < 5; x++) {

                System.out.printf("%02x    ", A[x][y]);

            }

            System.out.println();

        }

    }

    private static int rotateLeft(int value, int bits) {

        return ((value << bits) | (value >>> (8 - bits))) & 0xFF;

    }

}

From running the code, we get the following:

**D[x] values:**

**D[0] = 20**

**D[1] = 78**

**D[2] = c2**

**D[3] = ba**

**D[4] = e5**

**State after Theta step:**

**y\x x=0 x=1 x=2 x=3 x=4**

**----------------------------------------**

**y=4 | 53 19 81 ce 90**

**y=3 | 00 0c e2 db 8d**

**y=2 | 0c 1d ac f4 96**

**y=1 | 4f 01 a3 9a 8a**

**y=0 | 4e 19 aa db af**

The updated state matrix is what we use going forward.

Note: While there are 24 rounds, we do all 5 steps of Keccak before proceeding to another round.

## **1.2) What is the output of mapping in the first round?**

Rho is the second step in Keccak. What occurs is the bits in the state matrix are rotate by a fixed offset, predefined table called Rho Offsets. We use this table to spread/diffuse bits for multiple rounds. For this task, we only focus on round 1 of rho.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **X=3** | **X=4** | **X=0** | **X=1** | **X=2** |
| **Y=2** | 25 | 39 | 3 | 10 | 43 |
| **Y=1** | 55 | 20 | 36 | 44 | 6 |
| **Y=0** | 28 | 27 | 0 | 1 | 62 |
| **Y=4** | 56 | 14 | 18 | 2 | 61 |
| **Y=3** | 21 | 8 | 41 | 45 | 15 |

Rho Offsets Table

How the Rho Offsets table works is that we compare the (X,Y) coordinates in the state matrix to the coordinates in the Rho Offsets table. For instance, a value located at X=0, Y=0 has no rotation, while a value located at X=1, Y=0 has a rotation of 1 bit.

Example

Step 1.Fixed Rotation Offset

Rho applies a cyclic bitwise rotation to each lane based on the (x,y) position in matrix. This is calculated using this formula.

A math equation with numbers and symbols

AI-generated content may be incorrect.

Fixed Rotation Offset Formula (Chemejon, 2021).

T must satisfy 2 conditions.

* T must be 0<=**T** < 24
* T must satisfy this conditions.

A number and a symbol

AI-generated content may be incorrect.

This creates our Rho Offsets table.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **X=3** | **X=4** | **X=0** | **X=1** | **X=2** |
| **Y=2** | 25 | 39 | 3 | 10 | 43 |
| **Y=1** | 55 | 20 | 36 | 44 | 6 |
| **Y=0** | 28 | 27 | 0 | 1 | 62 |
| **Y=4** | 56 | 14 | 18 | 2 | 61 |
| **Y=3** | 21 | 8 | 41 | 45 | 15 |

Step 2. Rotation Operation

***Formula for rotation = [Offsets value in table mod [length of lane]***

In both examples I will focus on using values that have a Rho offset of 1, 36 and 78 for simplified purposes. I will use a full 64-bit value in example 1. This will be the same value to make it easy to highlight movements. For example, 2 I will use 8 bits in a 64-bit lane.

*Example 1: 64 bits in a 64-bit lane*

State matrix

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **X = 0** | **X = 1** | **X =2** | **X =3** | **X= 4** |
| **Y = 0** | 8C01E003F007000A |  |  |  |  |
| **Y = 1** |  |  |  |  |  |
| **Y = 2** |  |  |  |  |  |
| **Y = 3** |  |  |  | 8C01E003F007000A | 8C01E003F007000A |
| **Y = 4** |  |  |  |  |  |

For simplicity I will only use the same value

|  |  |  |  |
| --- | --- | --- | --- |
| **(X,y)** | **Hex Value** | **Binary (64-bit, grouped by 4)** | **Decimal Representation** |
| (0,0) | 8C01E003F007000A | 1000 1100 0000 0001 1110 0000 0000 0011 1111 0000 0000 0111 0000 0000 0000 1010 | 10088590947803136010 |
| (4,3) | 8C01E003F007000A | 1000 1100 0000 0001 1110 0000 0000 0011 1111 0000 0000 0111 0000 0000 0000 1010 | 10088590947803136010 |
| (3,3) | 8C01E003F007000A | 1000 1100 0000 0001 1110 0000 0000 0011 1111 0000 0000 0111 0000 0000 0000 1010 | 10088590947803136010 |

Calculations of rotations

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **(x,y)** | **Rho Offset** | **Rotation (Offset mod 64)** | **Binary before rotation** | **Binary After Rotation** | **Hex (Result)** |
| (0,0) | 0 | 0 mod 64 = 0 | (Same) | (Same) | 8C01E003F007000A |
| (4,3) | 8 | 8 mod 64 = 8 | **1000 1100** 0000 0001 1110 0000 0000 0011 1111 0000 0000 0111 0000 0000 0000 1010 | 0000 0001 1110 0000 0000 0011 1111 0000 0000 0111 0000 0000 0000 1010  **1000 1100** | 1E003F007000A8C |
| (3,3) | 21 | 21 mod 64 = 21 | **1000 1100 0000 0001 1110 0**000 0000 0011 1111 0000 0000 0111 0000 0000 0000 1010 | 0000 0000  0111 1110  0000 0000  1110 0000  0000 0001  010**1 0001**  **1000 0000**  **0011 1100** | 7E00E00151803C |

State Matrix after rotations

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **X = 0** | **X = 1** | **X =2** | **X =3** | **X= 4** |
| **Y = 0** | 8C01E003F007000A |  |  |  |  |
| **Y = 1** |  |  |  |  |  |
| **Y = 2** |  |  |  |  |  |
| **Y = 3** |  |  |  | 7E00E00151803C | 1E003F007000A8C |
| **Y = 4** |  |  |  |  |  |

*Example 2: 8 bit in 64 bit lane*

State matrix

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **X = 0** | **X = 1** | **X =2** | **X =3** | **X= 4** |
| **Y = 0** | 8E |  |  |  |  |
| **Y = 1** |  |  |  |  |  |
| **Y = 2** |  |  |  |  |  |
| **Y = 3** |  |  |  | 8E | 8E |
| **Y = 4** |  |  |  |  |  |

For simplicity I will only use the same value

|  |  |  |  |
| --- | --- | --- | --- |
| **(X,y)** | **Hex Value** | **Binary (64-bit, grouped by 4)** | **Decimal Representation** |
| (0,0) | 8E | 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 **1000 1110** | 142 |
| (4,3) | 8E | 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 **1000 1110** | 142 |
| (3,3) | 8E | 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 **1000 1110** | 142 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **(x,y)** | **Rho Offset** | **Rotation (Offset mod 64)** | **Binary before rotation** | **Binary After Rotation** | **Hex (Result)** |
| (0,0) | 0 | 0 mod 64 = 0 | (Same) | (Same) | 8E |
| (4,3) | 8 | 8 mod 64 = 8 | **0000 0000** 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 **1000 1110** | 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 **1000 1110**  **0000 0000** | 8E00 |
| (3,3) | 21 | 21 mod 64 = 21 | **0000 0000** **0000 0000 0000 0**000 0000 0000 0000 0000 0000 0000 0000 0000 **1000 1110** | 0000 0000  0000 0000  0000 0000  0000 0000  000**1 0001**  **1100 0000**  **0000 0000**  **0000 0000** | 11C00000 |

State matrix after rotations

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **X = 0** | **X = 1** | **X =2** | **X =3** | **X= 4** |
| **Y = 0** | 8E |  |  |  |  |
| **Y = 1** |  |  |  |  |  |
| **Y = 2** |  |  |  |  |  |
| **Y = 3** |  |  |  | 11C00000 | 8E00 |
| **Y = 4** |  |  |  |  |  |

Calculations – Question

Here is my matrix calculated from Theta:

**4e 19 aa db af**

**4f 01 a3 9a 8a**

**0c 1d ac f4 96**

**00 0c e2 db 8d**

**53 19 81 ce 90**

**MATRIX READING MUST ALWAYS BE CONSISTENT ACROSS CALCULATIONS.**

Now that we know the rotations, we need to temporarily convert the hex values into binary then move them. This can be tedious to do manually so I will instead use a java script.

Run the following as “**Q1KeccakRho.java**”

public class **Q1KeccakRho** {

// Rho offsets (standard table for y=0 at top)

private static final int[][] rhoOffsets = {

{ 0, 36, 3, 41, 18 }, // y=0 (top in standard Keccak)

{ 1, 44, 10, 45, 2 }, // y=1

{ 62, 6, 43, 15, 61 }, // y=2

{ 28, 55, 25, 21, 56 }, // y=3

{ 27, 20, 39, 8, 14 } // y=4 (bottom in standard Keccak)

};

// State matrix after Theta (from your output)

private static final String[][] stateMatrix = {

{ "53", "19", "81", "ce", "90" }, // y=4 (top)

{ "00", "0c", "e2", "db", "8d" }, // y=3

{ "0c", "1d", "ac", "f4", "96" }, // y=2

{ "4f", "01", "a3", "9a", "8a" }, // y=1

{ "4e", "19", "aa", "db", "af" } // y=0 (bottom)

};

// Rotate 8-bit value

public static String rotate(String hexValue, int offset) {

int value = Integer.parseInt(hexValue, 16) & 0xFF;

offset = offset % 8;

int rotated = ((value << offset) | (value >>> (8 - offset))) & 0xFF;

return String.format("%02x", rotated);

}

// Apply Rho with y=4 at top (matches Theta output)

public static String[][] applyRho() {

String[][] transformedMatrix = new String[5][5];

for (int y = 0; y < 5; y++) {

for (int x = 0; x < 5; x++) {

// Your y=4 is standard Keccak's y=0 (top)

// Your y=0 is standard Keccak's y=4 (bottom)

int standardY = 4 - y;

int offset = rhoOffsets[standardY][x];

transformedMatrix[y][x] = rotate(stateMatrix[y][x], offset);

}

}

return transformedMatrix;

}

public static void printMatrix(String[][] matrix) {

System.out.println("y\\x x=0 x=1 x=2 x=3 x=4");

System.out.println("----------------------------------------");

for (int y = 4; y >= 0; y--) {

System.out.printf("y=%d | ", y);

for (int x = 0; x < 5; x++) {

System.out.printf("%s ", matrix[y][x]);

}

System.out.println();

}

}

public static void main(String[] args) {

System.out.println("State Matrix (After Theta):");

printMatrix(stateMatrix);

System.out.println("\nAfter Rho:");

printMatrix(applyRho());

}

}

*After running the code.*

**State Matrix (After Theta):**

y\x x=0 x=1 x=2 x=3 x=4

----------------------------------------

y=4 | 4e 19 aa db af

y=3 | 4f 01 a3 9a 8a

y=2 | 0c 1d ac f4 96

y=1 | 00 0c e2 db 8d

y=0 | 53 19 81 ce 90

**After Rho:**

y\x x=0 x=1 x=2 x=3 x=4

----------------------------------------

y=4 | 4e 91 55 b7 be

y=3 | 9e 10 8e 53 2a

y=2 | 03 47 65 7a d2

y=1 | 00 06 c5 7b 8d

y=0 | 9a 91 c0 ce 24

## **1.3) What is the output of (Pi) mapping in the first round?**

Pie is the 3rd step in Keccak. Here we re-arrange elements of state matrix calculated from Rho. Why we do this is to shuffle elements to allow further diffusion to strengthen security of hash function.

* Each value in the State Matrix holds a hexadecimal value of 64 bits
* We re-arrange elements based on a formula.

It follows this formula:

B [y, 2x+3y (mod 5)] =A [x,y], x,y =0,1,2,3,4

This can be simplified into:

State [x, y] -> State [y, (2x +3y) mod 5 ]

State [x,y] is our input and the right side of the equation is our output, new coordinates. X refers to the column index and y is the row index. We sub the x and y values into formula. The reason why we mod by 5 is so that our answer wraps around a 5\*5 matrix. Elements are dependent on other values which provides better diffusion.

*Example*

Initial State

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **(x,y)** | **X=0** | **X=1** | **X=2** | **X=3** | **X=4** |
| **Y=0** | 1 | 1 | 0 | 0 | 1 |
| **Y=1** | 0 | 0 | 1 | 0 | 0 |
| **Y=2** | 0 | 1 | 1 | 0 | 0 |
| **Y=3** | 1 | 0 | 1 | 0 | 0 |
| **Y=4** | 1 | 0 | 0 | 1 | 1 |

We refer to this table.

We sub values into the formula [x, y] -> State [y, (2x +3y) mod 5 ].

We look at coordinates (1,2). This has a value of 1.

[x, y] -> State [2, (2\*1 +3\*2) mod 5 ].

= (2, 8 mod 5)

= (2, 3)

This means that the value at coordinates moves from (1,2) to (2,3)

Calculation

Here is my code I used to re-arrange elements. Run as **Q1Pie.java**.

public class **Q1Pie** {

    // State matrix after Rho (from your output)

    private static final String[][] stateMatrix = {

            { "4e", "91", "55", "b7", "be" }, // y=4 (top)

            { "9e", "10", "8e", "53", "2a" }, // y=3

            { "03", "47", "65", "7a", "d2" }, // y=2

            { "00", "06", "c5", "7b", "8d" }, // y=1

            { "9a", "91", "c0", "ce", "24" } // y=0 (bottom)

    };

    // Function to apply Pi transformation

    public static String[][] applyPiTransformation() {

        String[][] transformedMatrix = new String[5][5];

        for (int x = 0; x < 5; x++) {

            for (int y = 0; y < 5; y++) {

                // Apply Pi transformation: (x, y) → (y, (2x + 3y) mod 5)

                // Note: Our matrix is stored with y=4 at top (index 0) and y=0 at bottom (index

                // 4)

                // So we need to adjust the y coordinates for the transformation

                // Current y in standard coordinates (where y=0 is bottom)

                int stdY = 4 - y;

                // Calculate new position

                int newX = stdY;

                int newY = (2 \* x + 3 \* stdY) % 5;

                // Convert newY back to our matrix coordinates

                int newMatrixY = 4 - newY;

                transformedMatrix[newMatrixY][newX] = stateMatrix[y][x];

            }

        }

        return transformedMatrix;

    }

    public static void printMatrix(String[][] matrix) {

        System.out.println("y\\x   x=0    x=1    x=2    x=3    x=4");

        System.out.println("----------------------------------------");

        for (int y = 0; y < 5; y++) {

            System.out.printf("y=%d | ", 4 - y); // Show y coordinate in standard form (4 at top)

            for (int x = 0; x < 5; x++) {

                System.out.printf("%4s  ", matrix[y][x]);

            }

            System.out.println();

        }

    }

    public static void main(String[] args) {

        System.out.println("State Matrix (After Rho):");

        printMatrix(stateMatrix);

        System.out.println("\nAfter Pi:");

        printMatrix(applyPiTransformation());

    }

}

**Results from running code.**

State Matrix (After Rho):

y\x x=0 x=1 x=2 x=3 x=4

----------------------------------------

y=4 | 4e 91  **55** b7 be

y=3 | 9e 10 8e 53 2a

y=2 | 03 47 65 7a d2

y=1 | 00 06 c5 7b 8d

y=0 | 9a 91 c0 ce 24

After Pi:

y\x x=0 x=1 x=2 x=3 x=4

----------------------------------------

y=4 | c0 7b d2 9e 91

y=3 | 24 00 47 8e b7

y=2 | 91 c5 7a 2a 4e

y=1 | ce 8d 03 10  **55**

y=0 | 9a 06 65 53 be

## **1.4) What is the output of mapping in the first round?**

This is the 4th step in Keccak SHA-3. This further ensures that the values in the state matrix are unpredictable for an attacker to re-assemble. Chi aims to do this by editing the rows of the 5\*5 matrix. This is by using AND as well as XOR operations to mix bits.

The formula that we use is:

A [x, y] = B [x, y ] XOR ((**B`** [ x + 1, y ], ) ∧ B [x + 2, y] ) , x, y = 0,1,2,3,4

Or

A[x,y] = B[x,y] XOR ( (**NOT B**[x+1,y]) AND B[x+2,y] )

B = Input State

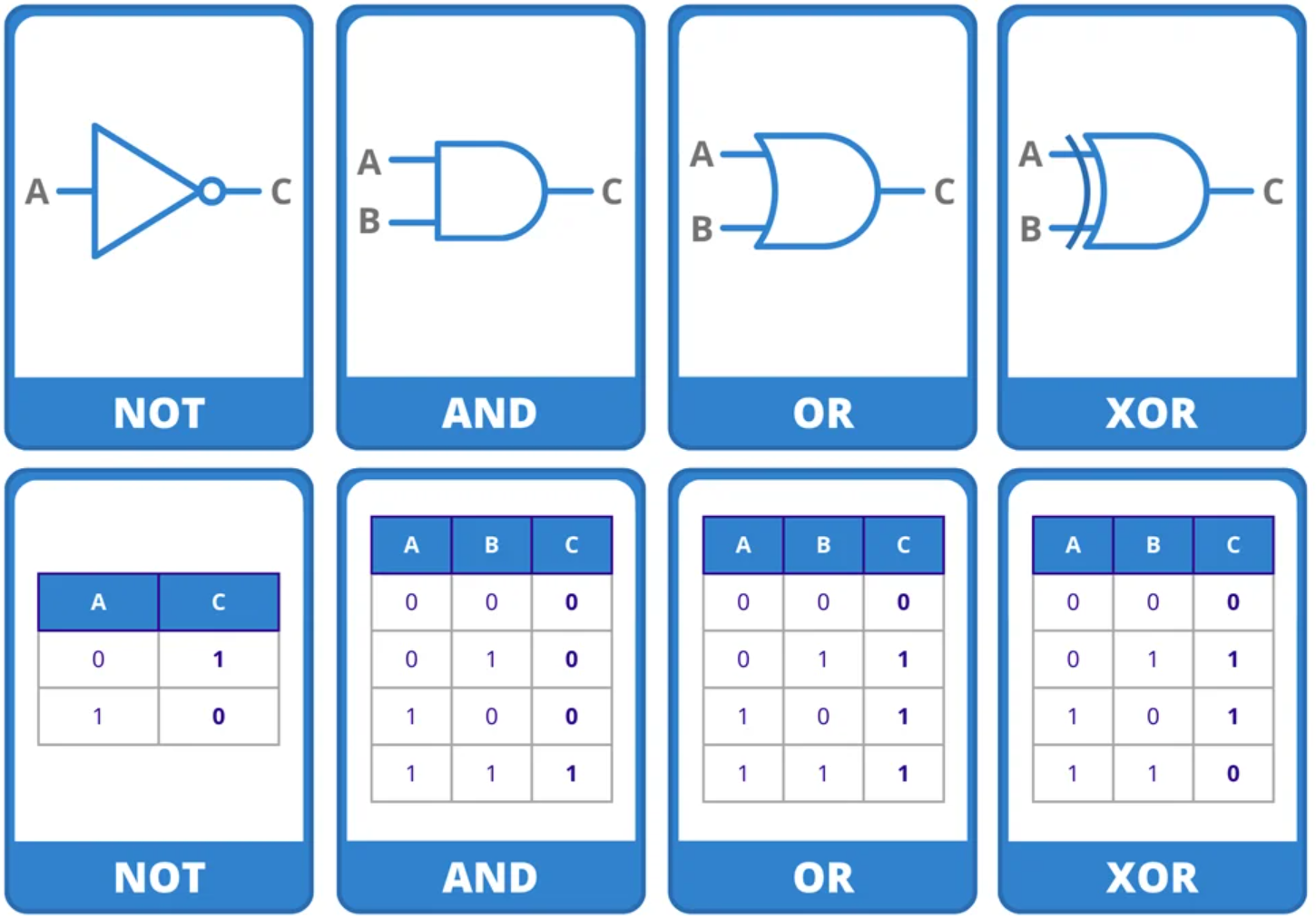
A = Output state after Chi

X,y = refers to values that wrap around modulo 5.

∧ = AND

B` = NOT

*We refer to Truth Tables.*



Truth Table (Truth Table Generator, 2025)

*How Phi works*

1. Look at a value in the 5\*5 State Matrix (x, y)
2. Compute the NOT value of **B`** [ x + 1, y ]. This moves the bit value to the right.
3. Perform AND operation with **B`**[x + 2, y] ). This then moves the bit 2 positions to the right.
4. Perform the XOR operation with the original bit value, B[x, y].
5. Repeat until all bits are moved.

Example

State Matrix

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **X =0** | **X = 1** | **X= 2** | **X=3** | **X=4** |
| **Y= 0** | 1 | 0 | 1 | 0 | 1 |
| **Y= 1** | 0 | 1 | 0 | 1 | 0 |
| **Y = 2** | 1 | 0 | 1 | 0 | 1 |
| **Y = 3** | 0 | 1 | 0 | 1 | 0 |
| **Y = 4** | 1 | 0 | 1 | 0 | 1 |

1.Look at formula

A[x,y] = B[x,y] **XOR** ( (**NOT B**[x+1,y]) **AND** B[x+2,y] )

X+1 and x+2 wrap around modulo 5.

2. Calculate each row

**Y = 0**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **x** | **B[x,0]** | **B[x+1,0]** | **Value of b[x+1,0] from State** | **NOT B [x+1,0]** | **B[x+2,0]** | **Value of B [x+2,0]** | **AND Step** | **XOR Result(A, [x,0]** |
| 0 | 1 | B[1,0] | 0 | 1 | B[2,0] | 1 | 1 AND 1 = 1 | 1 XOR 1  =0 |
| 1 | 0 | B[2,0] | 1 | 0 | B[3,0] | 0 | 0 AND 0 = 0 | 0 X0R 0  =0 |
| 2 | 1 | B[3,0] | 0 | 1 | B[4,0] | 1 | 1 AND 1 = 0 | 1 COR 1 = 0 |
| 3 | 0 | B[4,0] | 1 | 0 | B[0,0] | 1 (Wrap) | 0 AND 1 = 0 | 0 XOR 0 = 0 |
| 4 | 1 | B[0,0] | 1 | 0 | B[1,0] | 0 (Wrap) | 0 AND 0 = 0 | 1 XOR 0 = 1 |

Output = [0,0,0,0,1]

**Y = 1**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **x** | **B[x,1]** | **B[x+1,1]** | **Value of b[x+1,1] from State** | **NOT B [x+1,1]** | **B[x+2,1]** | **Value of B [x+2,1]** | **AND Step** | **XOR Result(A, [x,1]** |
| 0 | 0 | B[1,1] | 1 | 0 | B[2,1] | 0 | 0 AND 0 = 0 | 0 XOR 0  =0 |
| 1 | 1 | B[2,1] | 0 | 1 | B[3,1] | 1 | 1 AND 1 = 0 | 1 X0R 1  =0 |
| 2 | 0 | B[3,1] | 1 | 0 | B[4,1] | 0 | 0 AND 0 = 0 | 0 COR 0 = 0 |
| 3 | 1 | B[4,1] | 0 | 1 | B[0,1] | 0 (Wrap) | 1 AND 0 = 0 | 1 XOR 0 = 1 |
| 4 | 0 | B[0,1] | 0 (wrap) | 1 | B[1,1] | 0 (Wrap) | 1 AND 1 = 0 | 0 XOR 1 = 1 |

Output = [0,0,0,1,1]

**Y = 2**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **x** | **B[x,2]** | **B[x+1,2]** | **Value of b[x+1,2] from State** | **NOT B [x+1,2]** | **B[x+2,2]** | **Value of B [x+2,2]** | **AND Step** | **XOR Result(A, [x,2]** |
| 0 | 1 | B[1,2] | 0 | 1 | B[2,2] | 1 | 1AND 1 = 1 | 1 XOR 1  =0 |
| 1 | 0 | B[2,2] | 1 | 0 | B[3,2] | 0 | 0 AND 0 = 0 | 0 X0R 0  =0 |
| 2 | 1 | B[3,2] | 0 | 1 | B[4,2] | 1 | 1 AND 1 = 1 | 1 XOR 1 = 0 |
| 3 | 0 | B[4,2] | 1 | 0 | B[0,2] | 1 (Wrap) | 0 AND 1 = 0 | 0 XOR 0 = 1 |
| 4 | 1 | B[0,2] | 1 (wrap) | 0 | B[1,2] | 0 (Wrap) | 0 AND 0 = 0 | 1 XOR 0 = 1 |

Output = [0,0,0,0,1]

**Y = 3**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **x** | **B[x,3]** | **B[x+1,3]** | **Value of b[x+1,3] from State** | **NOT B [x+1,3]** | **B[x+2,3]** | **Value of B [x+2,3]** | **AND Step** | **XOR Result(A, [x,3]** |
| 0 | 0 | B[1,3] | 1 | 0 | B[2,3] | 0 | 0 AND 0 = 0 | 0 XOR 1  =0 |
| 1 | 1 | B[2,3] | 0 | 1 | B[3,3] | 1 | 1 AND 1 = 1 | 1 X0R 1  =0 |
| 2 | 0 | B[3,3] | 1 | 0 | B[4,3] | 0 | 0 AND 0 = 0 | 0 XOR 0 = 0 |
| 3 | 1 | B[4,3] | 0 | 1 | B[0,3] | 0 (Wrap) | 1 AND 0 = 0 | 1 XOR 0 = 1 |
| 4 | 0 | B[0,3] | 0 (wrap) | 1 | B[1,3] | 1 (Wrap) | 1 AND 1 = 1 | 0 XOR 1 = 1 |

Output = [0,0,0,1,1]

**Y = 4**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **x** | **B[x,4]** | **B[x+1,4]** | **Value of b[x+1,4] from State** | **NOT B [x+1,4]** | **B[x+2,4]** | **Value of [x+2,4]** | **AND Step** | **XOR Result(A, [x,4]** |
| 0 | 1 | B[1,4] | 0 | 1 | B[2,4] | 1 | 1AND 1 = 1 | 1 XOR 1  =0 |
| 1 | 0 | B[2,4] | 1 | 0 | B[3,4] | 0 | 0 AND 0 = 0 | 0 X0R 0  =0 |
| 2 | 1 | B[3,4] | 0 | 1 | B[4,4] | 1 | 1 AND 1 = 1 | 1 XOR 1 = 0 |
| 3 | 0 | B[4,4] | 1 | 0 | B[0,4] | 1 (Wrap) | 0 AND 1 = 0 | 0 XOR 0 = 0 |
| 4 | 1 | B[0,4] | 1 (wrap) | 0 | B[1,4] | 0 (Wrap) | 0 AND 0 = 0 | 1 XOR 0 = 1 |

Output = [0,0,0,0,1]

3. Have the final output.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **X =0** | **X = 1** | **X= 2** | **X=3** | **X=4** |
| **Y= 0** | 0 | 0 | 0 | 0 | 1 |
| **Y= 1** | 0 | 0 | 0 | 1 | 1 |
| **Y = 2** | 0 | 0 | 0 | 0 | 1 |
| **Y = 3** | 0 | 0 | 0 | 1 | 1 |
| **Y = 4** | 0 | 0 | 0 | 0 | 1 |

*Extra*

The reason for wrapping is because we use a 5\*5 matrix. That means the indexes of x and y range from 0 to 4. If x+1 or x+2 exceeds 4, we wrap around by subtracting 5.

E.g.

x = 3, + 1 = 4. No wrap around/

x=4, +1 = 5. 5 – 5 = 0. X = 0.

X=3 + 2 = 5. 5 – 5 = 0. X = 0.

Calculation for Chi - Code

Run the following code as **Q1Chi.java.**

public class **Q1Chi** {

public static void main(String[] args) {

// Input matrix (5x5) in hexadecimal (from Pi step output)

// Note: y=0 is bottom, y=4 is top (same as your Pi output)

int[][] inputMatrix = {

{ 0xc0, 0x7b, 0xd2, 0x9e, 0x91 }, // y=4 (top)

{ 0x24, 0x00, 0x47, 0x8e, 0xb7 }, // y=3

{ 0x91, 0xc5, 0x7a, 0x2a, 0x4e }, // y=2

{ 0xce, 0x8d, 0x03, 0x10, 0x55 }, // y=1

{ 0x9a, 0x06, 0x65, 0x53, 0xbe } // y=0 (bottom)

};

// Apply χ (Chi) step

int[][] outputMatrix = computeChi(inputMatrix);

// Print the result in the same format as your Pi output

System.out.println("Output after χ (Chi) step:");

printMatrix(outputMatrix);

}

// Computes χ (Chi) for a 5x5 state matrix

public static int[][] computeChi(int[][] input) {

int[][] output = new int[5][5];

// Loop through each row (y)

for (int y = 0; y < 5; y++) {

// Loop through each column (x)

for (int x = 0; x < 5; x++) {

// Get B[x,y], B[x+1,y], B[x+2,y] (with wrap-around)

int a = input[y][x];

int b = input[y][(x + 1) % 5];

int c = input[y][(x + 2) % 5];

// Compute χ: A[x,y] = B[x,y] XOR (NOT B[x+1,y] AND B[x+2,y])

// Mask with 0xFF to keep only 8 bits (byte)

output[y][x] = a ^ ((~b) & c) & 0xFF;

}

}

return output;

}

// Prints a 5x5 matrix in hex format (same format as your Pi output)

public static void printMatrix(int[][] matrix) {

System.out.println("y\\x x=0 x=1 x=2 x=3 x=4");

System.out.println("----------------------------------------");

for (int y = 0; y < 5; y++) {

System.out.printf("y=%d | ", 4 - y); // Show y coordinate in standard form (4 at top)

for (int x = 0; x < 5; x++) {

System.out.printf("%02x ", matrix[y][x]);

}

System.out.println();

}

}

}

**Output after running code:**

Output after (Chi) step:

y\x x=0 x=1 x=2 x=3 x=4

----------------------------------------

y=4 | 40 77 d3 de aa

y=3 | 63 88 76 8e b7

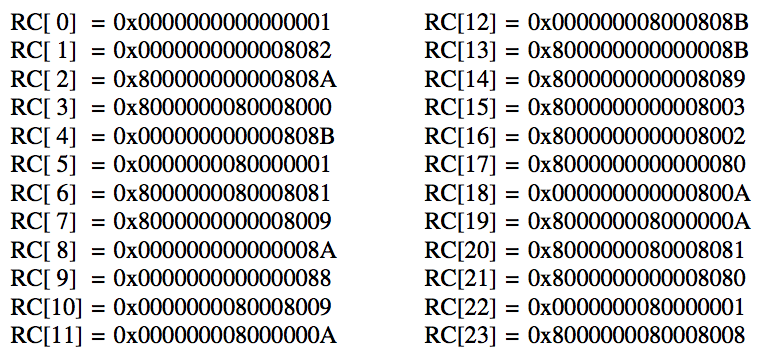
y=2 | ab c5 3e bb 0a

y=1 | cc 9d 46 9a 54

y=0 | fb 14 c9 53 ba

## **1.5) What is the output of (Iota) mapping in the first round?**

Iota is the final step in Keccak. This step’s objective is to break symmetry in a state matrix by XORING the values with a round-dependent constant (RC). These constants are referred in the **Iota Round Constants** table. Generally, how it works is Iota will only change the first lane of the 5\*5 matrix (0,0). So only one value will change, the rest of the state matrix will stay the same.



Iota Round Constants (2025)

Output after (Chi) step:

y\x x=0 x=1 x=2 x=3 x=4

----------------------------------------

y=4 | 40 77 d3 de aa

y=3 | 63 88 76 8e b7

y=2 | ab c5 3e bb 0a

y=1 | cc 9d 46 9a 54

y=0 | fb 14 c9 53 ba

The value we multiply in the matrix is fb. We use <https://xor.pw/> and XOR it with RC[0] = 0x0000000000000001

= fa

Output after (Chi) step:

y\x x=0 x=1 x=2 x=3 x=4

----------------------------------------

y=4 | 40 77 d3 de aa

y=3 | 63 88 76 8e b7

y=2 | ab c5 3e bb 0a

y=1 | cc 9d 46 9a 54

y=0 | fa 14 c9 53 ba

# **Question 2 HMAC**

Suppose that a string to sign is as follows:

AWS4-HMAC-SHA256

20250415M123600Z

20250415/us-east-1/iam/aws4\_request

f536975d06c0309214f805bb90ccff089219ecd68b2577efef23edd43b7e1a59

Assume that kSecret = **your student ID**/K7MDENG+bPxRfiCYEXAMPLEKEY.

* My Student ID: 3722151

First, we identify our keys, messages and the string we sign at the end:

* **kSecret =**3722151/K7MDENG+bPxRfiCYEXAMPLEKEY  
  **Date =**20250415
* **Region =**us-east-1
* **Service =**iam
* **String to sign =**

AWS4-HMAC-SHA256

20250415M123600Z

20250415/us-east-1/iam/aws4\_request

f536975d06c0309214f805bb90ccff089219ecd68b2577efef23edd43b7e1a59

## 2.1) Compute kDate = HMAC("AWS4" + kSecret, Date), where Date = 20250415;

1)First we calculate kDate.

**Steps:**

1. Take kSecret and append it to "AWS4" to it:  
   Key

= "AWS4" + kSecret

= AWS43722151/K7MDENG+bPxRfiCYEXAMPLEKEY

1. The message is the Date (20250415).
2. Compute HMAC-SHA256 using the **Key** and **Message** (20250415).

1)Get Key and Message ready

I am calculating this manually. I first convert kSecret to hex. I then convert my message Date into Hex.

Key = “AWS4" + kSecret

* kSecret = 3722151/K7MDENG+bPxRfiCYEXAMPLEKEY

AWS43722151/K7MDENG+bPxRfiCYEXAMPLEKEY

* Use <https://www.rapidtables.com/convert/number/ascii-to-hex.html> to convert into Hex.
* **Key(Hex)**:

41575334333732323135312F4B374D44454E472B62507852666943594558414D504C454B4559

HMAC-SHA256 requires the key to be **64 bytes long**. If the key is longer, we first hash it with SHA256 (resulting in 32 bytes), then pad to 64 bytes with zeros. Length of my key is 40 bytes (which is **less than 64**), so we **pad with zeros** to 64 bytes.

* ***This is because it is 40 bytes in ASCII. Also, this is required for the first HMAC step only.***

I now need to append 0’s to my key to ensure it is 512 bits.

Step 1: Calculate Current Length

* **Hex length**: Count the characters → **80 hex digits** (not bits!).
* **Bits**: Each hex digit = 4 bits → 80 × 4 = 320 bits.

Step 2: Target Length for HMAC-SHA256

* HMAC-SHA256 requires a **512-bit (64-byte) key**.
* **Hex equivalent**: 512 bits ÷ 4 = 128 hex digits.

Step 3: Padding Needed

* **Current hex digits**: 80
* **Target hex digits**: 128
* **Zeros to append**: 128 - 80 = 48 zeros (00 × 24, since each 00 = 1 byte).

**K**(padded) =

41575334333732323135312F4B374D44454E472B62507852666943594558414D504C454B455900000000000000000000000000000000000000000000000000

* This allows 128 hex characters.

Message: "20250415"

* + Convert to hex using RapidTables.
  + Hex: 3230323530343135

2)Calculate **output1 = (kipad) ||m ):**

We use [https://xor.pw/#](https://xor.pw/)

**Input 1** =

41575334333732323135312F4B374D44454E472B62507852666943594558414D504C454B455900000000000000000000000000000000000000000000000000

**Input 2**(ipad) = repeat 36, 64 times

=

36363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636

kipad=

367761650205010404070307197d017b727378711d54664e64505f756f736e777b667a737d736f36363636363636363636363636363636363636363636363636

Next, we need to append the message to the end of this. (kipad).

(kipad)||m =

367761650205010404070307197d017b727378711d54664e64505f756f736e777b667a737d736f363636363636363636363636363636363636363636363636363230323530343135

3)Calculate **output2 = SHA256(output1)**

Now we need to use <https://emn178.github.io/online-tools/sha256.html>

* **Input hex, output hex.**

SHA256[(kipad)||m] =

763fd92c1fd3894fc227e22ac236ab1c6bf65dbf58d46c7c09c8805e68b847cf

4)Calculate **output3 = kopad**

kopad compute using <https://xor.pw>

K(padding)=

41575334333732323135312F4B374D44454E472B62507852666943594558414D504C454B455900000000000000000000000000000000000000000000000000

**opad** = repeat 5c 64 times

=

5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C

(512-bit)

Result=

5c1d0b0f686f6b6e6e6d696d73176b111819121b773e0c240e3a351f0519041d110c10191719055c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c

5)Calculate **output4 = output3output2 ):**

kopad|| SHA256[(kipad)||m]

What we need to do here is concatenate output 3 with output 2 at the end.

5c1d0b0f686f6b6e6e6d696d73176b111819121b773e0c240e3a351f0519041d110c10191719055c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c763fd92c1fd3894fc227e22ac236ab1c6bf65dbf58d46c7c09c8805e68b847cf

6)Result = SHA256(output4) ):

HMAC(k,m)

SHA256[kopad|| SHA256[(kipad)||m]]

Use this tool: <https://emn178.github.io/online-tools/sha256.html>

* **Input Hex**, Output hex

=

44860973422fd434c2a89ec8b67af8945bbe05f4433ac17f69bc3dd5ac46e78f

7)Validation

We can also use [**https://emn178.github.io/online-tools/sha256.html**](https://emn178.github.io/online-tools/sha256.html) to check if our results are correct.

* Input Hex, output hex
* Enable HMAC
* Encoding Hex
* Key (before padding):

41575334333732323135312F4B374D44454E472B62507852666943594558414D504C454B4559

Enter original input/message(HEX) – This is the date “20250415”:

3230323530343135

Output: dd1b50bf12bbeefe8ecd6953cea134f07973a57c6d35ca70a60490ccbd9ed2b2

* Not the same as Step 6 Calculation. Therefore, calculation is incorrect.

**kDate**: dd1b50bf12bbeefe8ecd6953cea134f07973a57c6d35ca70a60490ccbd9ed2b2

## 2.2) Compute kRegion = HMAC(kDate, Region), where Region = us-east-1;

**Compute**kRegion

**Formula:**  
kRegion = HMAC-SHA256(kDate, Region)

**Steps:**

1. Use kDate from Step 1 as the **Key**.
2. The message is the Region (us-east-1).
3. Compute HMAC-SHA256.  
   kRegion = HMAC-SHA256(<kDate\_from\_step1>, "us-east-1")

**kDate**: dd1b50bf12bbeefe8ecd6953cea134f07973a57c6d35ca70a60490ccbd9ed2b2

1)Get Key and Message ready

Key = **kDate**

dd1b50bf12bbeefe8ecd6953cea134f07973a57c6d35ca70a60490ccbd9ed2b2

*Determine if we hash key.*

1. **For Keys *Longer* Than 64 Bytes(Hex):**
   * **Step 1:** Hash the key with SHA-256 (reduces it to 32 bytes).
   * **Step 2:** Pad the hashed key with zeros to 64 bytes.
2. **For Keys *Shorter* Than 64 Bytes(Hex):**
   * **Directly use the key as-is** (HMAC automatically pads it internally with zeros to 64 bytes during processing).
3. **For Keys *Exactly* 64 Bytes(Byte):**

* Use the key directly (no hashing or padding needed).

My key has 64 hex characters = 32 bytes(hex). This is shorter than 64 bytes(hex). We use it directly.

Message: **us-east-1**

Convert to hex <https://www.rapidtables.com/convert/number/ascii-to-hex.html>

* + Hex: 75732D656173742D31

2)Calculate **output1 = (kipad) ||m ):**

We use [https://xor.pw/#](https://xor.pw/)

**Input 1 = K = kdate**

dd1b50bf12bbeefe8ecd6953cea134f07973a57c6d35ca70a60490ccbd9ed2b2

**Input 2(ipad)**  = repeat 36, 64 times

=

36363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636

kipad=

3636363636363636363636363636363636363636363636363636363636363636eb2d6689248dd8c8b8fb5f65f89702c64f45934a5b03fc469032a6fa8ba8e484

Next, we need to append the message to the end of this. (kipad).

(kipad)||m =

3636363636363636363636363636363636363636363636363636363636363636eb2d6689248dd8c8b8fb5f65f89702c64f45934a5b03fc469032a6fa8ba8e48475732D656173742D31

3)Calculate **output2 = SHA256(output1)**

Now we need to use [https://emn178.github.io/online-tools/sha256.html to SHA output 1](https://emn178.github.io/online-tools/sha256.html%20to%20SHA%20output%201)

* **Input hex, output hex.**

SHA256[(kipad)||m] =

ea4e8fc89cf349074bf82e55866bb6c1705f8d55de65d17664c76323e49dbf3f

4)Calculate **output3 = kopad**

kopad compute using <https://xor.pw>

K(padding)=

dd1b50bf12bbeefe8ecd6953cea134f07973a57c6d35ca70a60490ccbd9ed2b2

**opad** = repeat 5c 64 times

=

5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C

(512-bit)

Result=

5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c81470ce34ee7b2a2d291350f92fd68ac252ff9203169962cfa58cc90e1c28eee

5)Calculate **output4 = output3output2 ):**

kopad|| SHA256[(kipad)||m]

What we need to do here is concatenate output 3 with output 2 at the end.

5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c81470ce34ee7b2a2d291350f92fd68ac252ff9203169962cfa58cc90e1c28eeeea4e8fc89cf349074bf82e55866bb6c1705f8d55de65d17664c76323e49dbf3f

6)Result = SHA256(output4) ):

HMAC(k,m)

SHA256[kopad|| SHA256[(kipad)||m]]

Use this tool: <https://emn178.github.io/online-tools/sha256.html>

* Input Hex, Output hex

=

babc4295c993666650df5d7a0a133274b5f3061b5ba13515b612be75102c621e

7)Validation

We can also use [**https://emn178.github.io/online-tools/sha256.html**](https://emn178.github.io/online-tools/sha256.html) to check if our results are correct.

* Input Hex, output hex
* Enable HMAC
* Encoding Hex
* Key (before padding):

dd1b50bf12bbeefe8ecd6953cea134f07973a57c6d35ca70a60490ccbd9ed2b2

Enter original input/message(HEX) – This is region “us-east-1”:

75732D656173742D31

Output:

a8233e8b6ff752fc106d13900b8f4534678fb32fa63ba770a9b5384ec5151db1

* Not the same as Step 6 Calculation. Therefore, calculation is incorrect.

**kRegion**:

a8233e8b6ff752fc106d13900b8f4534678fb32fa63ba770a9b5384ec5151db1

## 2.3) Compute kService = HMAC(kRegion, Service), where Service = iam;

**Step 3: Compute**kService

**Formula:**  
kService = HMAC-SHA256(kRegion, Service)

**Steps:**

1. Use kRegion from Step 2 as the **Key**.
2. The message is the Service (iam).
3. Compute HMAC-SHA256.

**Example:**  
kService = HMAC-SHA256(<kRegion\_from\_step2>, "iam")

1)Get Key and Message ready

**Key(kRegion)** = a8233e8b6ff752fc106d13900b8f4534678fb32fa63ba770a9b5384ec5151db1

*Determine if we hash key.*

1. **For Keys *Longer* Than 64 Bytes(Hex):**
   * **Step 1:** Hash the key with SHA-256 (reduces it to 32 bytes).
   * **Step 2:** Pad the hashed key with zeros to 64 bytes.
2. **For Keys *Shorter* Than 64 Bytes(Hex):**
   * **Directly use the key as-is** (HMAC automatically pads it internally with zeros to 64 bytes during processing).
3. **For Keys *Exactly* 64 Bytes(Byte):**

Use the key directly (no hashing or padding needed).

My key has 64 hex characters = 32 bytes(hex). This is shorter than 64 bytes(hex). We use it directly.

Message = **iam**

Convert to hex using <https://www.rapidtables.com/convert/number/ascii-to-hex.html>

* + Hex: 69616D

2)Calculate **output1 = (kipad) ||m ):**

We use [https://xor.pw/#](https://xor.pw/)

**Input 1 = k = kRegion =**

a8233e8b6ff752fc106d13900b8f4534678fb32fa63ba770a9b5384ec5151db1

**Input 2(ipad)**  = repeat 36, 64 times =

36363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636

kipad=

36363636363636363636363636363636363636363636363636363636363636369e1508bd59c164ca265b25a63db9730251b98519900d91469f830e78f3232b87

Next, we need to append the message to the end of this. (kipad).

(kipad)||m =

36363636363636363636363636363636363636363636363636363636363636369e1508bd59c164ca265b25a63db9730251b98519900d91469f830e78f3232b8769616D

3)Calculate **output2 = SHA256(output1)**

Now we need to use [https://emn178.github.io/online-tools/sha256.html to SHA output 1](https://emn178.github.io/online-tools/sha256.html%20to%20SHA%20output%201)

* Input hex, output hex.

SHA256[(kipad)||m] =

01023b7cad6a3c0d455d34d5d52e916efae6b2c2cd6523ecbbaa39ad81be4e88

4)Calculate **output3 = kopad**

kopad compute using <https://xor.pw>

K(padding)=

a8233e8b6ff752fc106d13900b8f4534678fb32fa63ba770a9b5384ec5151db1

**opad** = repeat 5c 64 times

=

5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C

(512-bit)

Result=

5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5cf47f62d733ab0ea04c314fcc57d319683bd3ef73fa67fb2cf5e96412994941ed

5)Calculate **output4 = output3output2 ):**

kopad|| SHA256[(kipad)||m]

What we need to do here is concatenate output 3 with output 2 at the end.

5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5cf47f62d733ab0ea04c314fcc57d319683bd3ef73fa67fb2cf5e96412994941ed01023b7cad6a3c0d455d34d5d52e916efae6b2c2cd6523ecbbaa39ad81be4e88

6)Result = SHA256(output4) ):

HMAC(k,m)

SHA256[kopad|| SHA256[(kipad)||m]]

Use this tool: <https://emn178.github.io/online-tools/sha256.html>

* Input Hex, Output hex

=

4a0f89f772cc53be9138a76c1a2073e548e463447fe48fd1c32c7c7963fc2724

7)Validation

We can also use [**https://emn178.github.io/online-tools/sha256.html**](https://emn178.github.io/online-tools/sha256.html) to check if our results are correct.

* Input Hex, output hex
* Enable HMAC
* Encoding Hex
* Key (before padding):

a8233e8b6ff752fc106d13900b8f4534678fb32fa63ba770a9b5384ec5151db1

Enter original input/message(HEX) – This is service **“iam**”:

69616D

Output:

895ccd437d977822228695e45c628f5142362fdd10fbe0dca1160c1d2f4c51c1

* Not the same as Step 6 Calculation. Therefore, calculation is incorrect.

**kService** =

895ccd437d977822228695e45c628f5142362fdd10fbe0dca1160c1d2f4c51c1

## 2.4) Compute kSigning = HMAC(kService, "aws4\_request");

**Step 4: Compute**kSigning

**Formula:**  
kSigning = HMAC-SHA256(kService, "aws4\_request")

**Steps:**

1. Use kService from Step 3 as the **Key**.
2. The message is the fixed string "aws4\_request".
3. Compute HMAC-SHA256.

**Example:**  
kSigning = HMAC-SHA256(kService, "aws4\_request")

1)Get Key and Message ready

Get kService as key. Then convert my message Date into Hex.

**Key (kService)** =

895ccd437d977822228695e45c628f5142362fdd10fbe0dca1160c1d2f4c51c1

*Determine if we hash key.*

1. **For Keys *Longer* Than 64 Bytes(Hex):**
   * **Step 1:** Hash the key with SHA-256 (reduces it to 32 bytes).
   * **Step 2:** Pad the hashed key with zeros to 64 bytes.
2. **For Keys *Shorter* Than 64 Bytes(Hex):**
   * **Directly use the key as-is** (HMAC automatically pads it internally with zeros to 64 bytes during processing).
3. **For Keys *Exactly* 64 Bytes(Byte):**

* Use the key directly (no hashing or padding needed).

My key has 64 hex characters = 32 bytes(hex). This is shorter than 64 bytes(hex). We use it directly.

Message: **aws4\_request**

Convert to hex using <https://www.rapidtables.com/convert/number/ascii-to-hex.html>

* + Hex: 617773345F72657175657374

2)Calculate **output1 = (kipad) ||m ):**

We use [https://xor.pw/#](https://xor.pw/)

**Input 1 = k = kService =**

895ccd437d977822228695e45c628f5142362fdd10fbe0dca1160c1d2f4c51c1

**Input 2(ipad)**  = repeat 36, 64 times =

36363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636

kipad=

3636363636363636363636363636363636363636363636363636363636363636bf6afb754ba14e1414b0a3d26a54b967740019eb26cdd6ea97203a2b197a67f7

Next, we need to append the message to the end of this. (kipad).

(kipad)||m =

3636363636363636363636363636363636363636363636363636363636363636bf6afb754ba14e1414b0a3d26a54b967740019eb26cdd6ea97203a2b197a67f7617773345F72657175657374

3)Calculate **output2 = SHA256(output1)**

Now we need to use <https://emn178.github.io/online-tools/sha256.html> to SHA-256.

* **Input hex, output hex.**

SHA256[(kipad)||m] =

ee8e84511d08865801ad6957d8e0d0555b42b08841ea3c91dbf8ed68e6e55c16

4)Calculate **output3 = kopad**

kopad compute using <https://xor.pw>

**K(padding**)=

895ccd437d977822228695e45c628f5142362fdd10fbe0dca1160c1d2f4c51c1

**opad** = repeat 5c 64 times

=

5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C

(512-bit)

Result=

5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5cd500911f21cb247e7edac9b8003ed30d1e6a73814ca7bc80fd4a504173100d9d

5)Calculate **output4 = output3output2 ):**

kopad|| SHA256[(kipad)||m]

What we need to do here is concatenate output 3 with output 2 at the end.

5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5cd500911f21cb247e7edac9b8003ed30d1e6a73814ca7bc80fd4a504173100d9dee8e84511d08865801ad6957d8e0d0555b42b08841ea3c91dbf8ed68e6e55c16

6)Result = SHA256(output4) ):

HMAC(k,m)

SHA256[kopad|| SHA256[(kipad)||m]]

Use this tool: <https://emn178.github.io/online-tools/sha256.html>

* **Input Hex**, Output hex

=

b0155b92470419e1d22278b8da63b5c1df693ca141488448863a63b242393b44

7)Validation

We can also use [**https://emn178.github.io/online-tools/sha256.html**](https://emn178.github.io/online-tools/sha256.html) to check if our results are correct.

* Input Hex, output hex
* Enable HMAC
* Encoding Hex
* Key (before padding):

895ccd437d977822228695e45c628f5142362fdd10fbe0dca1160c1d2f4c51c1

* Enter original input/message (HEX) – This is **aws\_request**:

617773345F72657175657374

Output:

f7cf6f9dad80bb345418e4d5827234b0d9d5f4df82946ad39c63777d0e0a4df8

Not the same as Step 6 Calculation. Therefore, calculation is incorrect.

**K(kSigning)** =

f7cf6f9dad80bb345418e4d5827234b0d9d5f4df82946ad39c63777d0e0a4df8

## 2.5) Compute the signature = HexEncode(HMAC(kSigning, string to sign))

**Formula:**  
signature = HexEncode(HMAC-SHA256(kSigning, string\_to\_sign))

**Steps:**

1. Use kSigning from Step 4 as the **Key**.
2. The message is the entire **string to sign** (given at the start).
3. Compute HMAC-SHA256.
4. Convert the result to hexadecimal (HexEncode).

**Final Output:**  
The hex-encoded string is the AWS SigV4 signature.

1)Get Key and Message ready

The key = kSigning from last step. I then convert my message Date into Hex.

**K(kSigning) =**

f7cf6f9dad80bb345418e4d5827234b0d9d5f4df82946ad39c63777d0e0a4df8

*Determine if we hash key.*

1. **For Keys *Longer* Than 64 Bytes(Hex):**
   * **Step 1:** Hash the key with SHA-256 (reduces it to 32 bytes).
   * **Step 2:** Pad the hashed key with zeros to 64 bytes.
2. **For Keys *Shorter* Than 64 Bytes(Hex):**
   * **Directly use the key as-is** (HMAC automatically pads it internally with zeros to 64 bytes during processing).
3. **For Keys *Exactly* 64 Bytes(Byte):**
   * Use the key directly (no hashing or padding needed).

My key has 64 hex characters = 32 bytes(hex). This is shorter than 64 bytes(hex). We use it directly.

Message (text):

AWS4-HMAC-SHA256

20250415M123600Z

20250415/us-east-1/iam/aws4\_request

f536975d06c0309214f805bb90ccff089219ecd68b2577efef23edd43b7e1a59

Above has line breaks so we have to compensate. I convert each line into hex using <https://www.rapidtables.com/convert/number/ascii-to-hex.html> and then i add “**0d0a**” to represent each new line

Line 1=

415753342D484D41432D534841323536

Line 2=

32303235303431354D3132333630305A

Line 3=

32303235303431352F75732D656173742D312F69616D2F617773345F72657175657374

Line 4= 66353336393735643036633033303932313466383035626239306363666630383932313965636436386232353737656665663233656464343362376531613539

=

Message (Hex):

415753342D484D41432D534841323536**0d0a**32303235303431354D3132333630305A**0d0a**32303235303431352F75732D656173742D312F69616D2F617773345F72657175657374**0d0a**66353336393735643036633033303932313466383035626239306363666630383932313965636436386232353737656665663233656464343362376531613539

2)Calculate **output1 = (kipad) ||m ):**

We use [https://xor.pw/#](https://xor.pw/)

**Input 1 = K(kSigning) =**

f7cf6f9dad80bb345418e4d5827234b0d9d5f4df82946ad39c63777d0e0a4df8

**Input 2(ipad)**  = repeat 36, 64 times =

36363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636363636

kipad=

3636363636363636363636363636363636363636363636363636363636363636c1f959ab9bb68d02622ed2e3b4440286efe3c2e9b4a25ce5aa55414b383c7bce

Next, we need to append the message to the end of this. (kipad).

(kipad)||m =

3636363636363636363636363636363636363636363636363636363636363636c1f959ab9bb68d02622ed2e3b4440286efe3c2e9b4a25ce5aa55414b383c7bce415753342D484D41432D534841323536**0d0a**32303235303431354D3132333630305A**0d0a**32303235303431352F75732D656173742D312F69616D2F617773345F72657175657374**0d0a**66353336393735643036633033303932313466383035626239306363666630383932313965636436386232353737656665663233656464343362376531613539

3)Calculate **output2 = SHA256(output1)**

Now we need to use <https://emn178.github.io/online-tools/sha256.html> to SHA-256.

- **Input hex, output hex.**

SHA256[(kipad)||m] =

a1d468f85106a5bb554bb25f06e75db0c1f236d596787459c324f3cd07b9d47d

4)Calculate **output3 = kopad**

kopad compute using <https://xor.pw>

K(padding)=

f7cf6f9dad80bb345418e4d5827234b0d9d5f4df82946ad39c63777d0e0a4df8

**opad** = repeat 5c 64 times

=

5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C

(512-bit)

Result=

5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5cab9333c1f1dce7680844b889de2e68ec8589a883dec8368fc03f2b21525611a4

5)Calculate **output4 = output3output2 ):**

kopad|| SHA256[(kipad)||m]

What we need to do here is concatenate output 3 with output 2 at the end.

5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5cab9333c1f1dce7680844b889de2e68ec8589a883dec8368fc03f2b21525611a4a1d468f85106a5bb554bb25f06e75db0c1f236d596787459c324f3cd07b9d47d

6)Result = SHA256(output4) ):

HMAC(k,m)

SHA256[kopad|| SHA256[(kipad)||m]]

Use this tool: <https://emn178.github.io/online-tools/sha256.html>

* **Input Hex,** Output hex

=

60b71bb1ebef61aa98533b9120a75ae49125bda73dcbdaccba65478b60dc8d32

7)Validation

We can also use [**https://emn178.github.io/online-tools/sha256.html**](https://emn178.github.io/online-tools/sha256.html) to check if our results are correct.

* ***Input Hex, output hex***
* Enable HMAC
* Encoding Hex
* Key (before padding):

f7cf6f9dad80bb345418e4d5827234b0d9d5f4df82946ad39c63777d0e0a4df8

Enter original input/message(HEX) – This is the string that we sign:

415753342D484D41432D534841323536**0d0a**32303235303431354D3132333630305A**0d0a**32303235303431352F75732D656173742D312F69616D2F617773345F72657175657374**0d0a**66353336393735643036633033303932313466383035626239306363666630383932313965636436386232353737656665663233656464343362376531613539

Output:

707a3b1acda2b44a57c02a5e9717d79b383c3cfae9d67f06f1cf1802a5f62cbd

* Not the same as Step 6 Calculation. Therefore calculation is incorrect.

**Signature =**

707a3b1acda2b44a57c02a5e9717d79b383c3cfae9d67f06f1cf1802a5f62cbd

# **Question 3: Simplified Kerberos**

Preparation

Student ID =3722151

C=your first name = Joshua

S=your surname = Cayetano

KC=MD5(C, your student ID),

KS=MD5(S, your student ID),

Lt=8 hours,

nC=MD5(C),

<https://emn178.github.io/online-tools/md5.html>

* Input: UTF-8
* Output: Hex

KC=MD5(C, your student ID),

=Joshua3722151

= 26e5d99e6cf00eee433a5526ad36daad

KS=MD5(S, your student ID)

=Cayetano3722151

= 0ceec716b4723e1ffab33519a5e48a3b

Nc= MD5(C)

=Joshua

= 85b103482a20682da703aa388933a6d8

## **3.1) Implement AES-128 encryption and decryption algorithms in CBC mode.**

**Note: run as** index.html

<!DOCTYPE html>

<html>

<head>

    <title>AES-128-CBC Tool (Task 3.1)</title>

    <!-- Load CryptoJS library for AES and MD5 operations -->

    <script src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/4.1.1/crypto-js.min.js"></script>

</head>

<body>

    <h1>AES-128-CBC Encryption/Decryption</h1>

    <!--

    Key Input Section

    - KC (Client Key) is pre-filled with my MD5 hash of "Joshua3722151"

    - IV (Initialization Vector) is set to a default zero value for simplicity

  -->

    <label>Client Key (KC):</label>

    <input type="text" id="key" value="26e5d99e6cf00eee433a5526ad36daad" readonly><br>

    <label>IV (16-byte Hex):</label>

    <input type="text" id="iv" value="00000000000000000000000000000000"><br>

    <!-- Plaintext input for encryption -->

    <label>Plaintext:</label>

    <input type="text" id="plaintext" value="Hello, Kerberos!"><br>

    <!-- Encryption button and output display -->

    <button onclick="encrypt()">Encrypt</button>

    <p id="ciphertext"></p>

    <!-- Ciphertext input for decryption -->

    <label>Ciphertext (Hex):</label>

    <input type="text" id="ciphertextInput"><br>

    <!-- Decryption button and output display -->

    <button onclick="decrypt()">Decrypt</button>

    <p id="decrypted"></p>

    <script>

        /\*\*

         \* Encrypts plaintext using AES-128-CBC

         \* - Uses the key and IV from input fields

         \* - Outputs ciphertext in Hex format

         \*/

        function encrypt() {

            // Get values from input fields

            const key = document.getElementById("key").value;

            const iv = document.getElementById("iv").value;

            const plaintext = document.getElementById("plaintext").value;

            // Perform AES-128-CBC encryption

            const ciphertext = CryptoJS.AES.encrypt(

                plaintext,                          // Plaintext to encrypt

                CryptoJS.enc.Hex.parse(key),        // Key (converted from Hex string)

                {

                    iv: CryptoJS.enc.Hex.parse(iv),   // IV (converted from Hex string)

                    mode: CryptoJS.mode.CBC           // CBC mode

                }

            ).ciphertext.toString(CryptoJS.enc.Hex); // Convert result to Hex string

            // Display encrypted result

            document.getElementById("ciphertext").innerText = `Encrypted: ${ciphertext}`;

        }

        /\*\*

         \* Decrypts ciphertext using AES-128-CBC

         \* - Uses the key and IV from input fields

         \* - Expects ciphertext in Hex format

         \* - Outputs decrypted plaintext

         \*/

        function decrypt() {

            // Get values from input fields

            const key = document.getElementById("key").value;

            const iv = document.getElementById("iv").value;

            const ciphertext = document.getElementById("ciphertextInput").value;

            // Perform AES-128-CBC decryption

            const decrypted = CryptoJS.AES.decrypt(

                {

                    ciphertext: CryptoJS.enc.Hex.parse(ciphertext) // Parse Hex ciphertext

                },

                CryptoJS.enc.Hex.parse(key),        // Key (converted from Hex string)

                {

                    iv: CryptoJS.enc.Hex.parse(iv),   // IV (converted from Hex string)

                    mode: CryptoJS.mode.CBC           // CBC mode

                }

            ).toString(CryptoJS.enc.Utf8);        // Convert result to UTF-8 string

            // Display decrypted result

            document.getElementById("decrypted").innerText = `Decrypted: ${decrypted}`;

        }

    </script>

</body>

</html>

A screenshot of a computer

AI-generated content may be incorrect.

## **3.2) Implement each step of Phase 1 & 2 in Simplified Kerberos using AES-128**

**C=your first name, S=your surname,**

**KC=MD5(C, your student ID),**

**KS=MD5(S, your student ID),**

**Lt=8 hours, nC=MD5(C),**

General overview of Simplified Kerberos

Phase 1: Client to Authentcation Server (AS)

* Client sends (C = “Joshua”, S=”Cayetano”, nc) to AS

AS responds with the encrypted ticket {sk, ticket}

* Ticket = {S, sk, ts Lt} encrypted with AS
* Response is encrypted by KC

Phase 2: Client to Server

Client sends the ticket and authenticator

* Authenticator = {C, ts} encrypted with sk
* Server decrypt ticket with KS to get sk then verifies authenticator

Code

Phase 1 (Automatic)

* Click buttons to generate each step
* Outputs clean JSON/hex values like 3.1
* Shows the encrypted ticket separately

Phase 2 (Interactive)

* Paste the encrypted AS response from Phase 1
* Client decrypts is manually
* Generate authenticator using session key
* Server verifies ticket independently

Extra

IV or Initialisation Vector is zeroed for simplicity.

**Run as “Q3two.html”**

<!DOCTYPE html>

<html>

<head>

    <title>Simplified Kerberos (Task 3.2)</title>

    <script src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/4.1.1/crypto-js.min.js"></script>

    <style>

        .phase {

            margin-bottom: 20px;

            padding: 15px;

            border: 1px solid #ddd;

        }

        .step {

            margin-left: 20px;

        }

    </style>

</head>

<body>

    <h1>Simplified Kerberos Protocol</h1>

    <!-- Predefined Values -->

    <div>

        <h3>Precomputed Values (MD5):</h3>

        <p>KC (Client Key): <code>26e5d99e6cf00eee433a5526ad36daad</code></p>

        <p>KS (Server Key): <code>0ceec716b4723e1ffab33519a5e48a3b</code></p>

        <p>nC (Client Nonce): <code>85b103482a20682da703aa388933a6d8</code></p>

    </div>

    <!-- Phase 1: Client → AS -->

    <div class="phase" id="phase1">

        <h2>Phase 1: Client Requests Ticket from AS</h2>

        <div class="step">

            <h3>Step 1: Client Sends (C, S, nC)</h3>

            <button onclick="sendClientRequest()">Send Request to AS</button>

            <p id="clientRequest"></p>

        </div>

        <div class="step">

            <h3>Step 2: AS Responds with Encrypted {sk, ticket}</h3>

            <p id="asResponse"></p>

            <p id="ticketHex"></p>

        </div>

    </div>

    <!-- Phase 2: Client → Server -->

    <div class="phase" id="phase2">

        <h2>Phase 2: Client Authenticates to Server</h2>

        <div class="step">

            <h3>Step 3: Client Sends (ticket, authenticator)</h3>

            <button onclick="sendToServer()">Send to Server</button>

            <p id="clientToServer"></p>

        </div>

        <div class="step">

            <h3>Step 4: Server Verifies and Responds</h3>

            <p id="serverResponse"></p>

        </div>

    </div>

    <script>

        // Predefined values

        const C = "Joshua";

        const S = "Cayetano";

        const studentID = "3722151";

        const KC = "26e5d99e6cf00eee433a5526ad36daad";

        const KS = "0ceec716b4723e1ffab33519a5e48a3b";

        const nC = "85b103482a20682da703aa388933a6d8";

        const Lt = "8h"; // Ticket lifetime

        // Generate random 128-bit session key (for demo)

        const sk = CryptoJS.lib.WordArray.random(16).toString(CryptoJS.enc.Hex);

        // Phase 1: Client → AS

        function sendClientRequest() {

            // Step 1: Client sends (C, S, nC) to AS

            document.getElementById("clientRequest").innerHTML =

                `<strong>Sent:</strong> ${JSON.stringify({ C, S, nC })}`;

            // Step 2: AS generates ticket and responds

            const ts = new Date().toISOString(); // Timestamp

            const ticket = { S, sk, ts, Lt };

            // Encrypt ticket with KS (AES-128-CBC)

            const encryptedTicket = CryptoJS.AES.encrypt(

                JSON.stringify(ticket),

                CryptoJS.enc.Hex.parse(KS),

                { iv: CryptoJS.enc.Hex.parse("0".repeat(32)) }

            ).toString();

            // Encrypt {sk, ticket} with KC

            const responseToClient = { sk, ticket: encryptedTicket };

            const encryptedResponse = CryptoJS.AES.encrypt(

                JSON.stringify(responseToClient),

                CryptoJS.enc.Hex.parse(KC),

                { iv: CryptoJS.enc.Hex.parse("0".repeat(32)) }

            ).toString();

            document.getElementById("asResponse").innerHTML =

                `<strong>AS → Client (Encrypted):</strong> ${encryptedResponse}`;

            document.getElementById("ticketHex").innerHTML =

                `<strong>Ticket (Hex):</strong> ${encryptedTicket}`;

        }

        // Phase 2: Client → Server

        function sendToServer() {

            // Client decrypts AS response to get sk and ticket

            const ts = new Date().toISOString();

            const authenticator = { C, ts };

            // Encrypt authenticator with sk

            const encryptedAuth = CryptoJS.AES.encrypt(

                JSON.stringify(authenticator),

                CryptoJS.enc.Hex.parse(sk),

                { iv: CryptoJS.enc.Hex.parse("0".repeat(32)) }

            ).toString();

            document.getElementById("clientToServer").innerHTML = `

        <strong>Sent to Server:</strong><br>

        Ticket: ${document.getElementById("ticketHex").innerText}<br>

        Authenticator (Hex): ${encryptedAuth}`;

            // Server verification would happen here

            document.getElementById("serverResponse").innerHTML =

                "<strong>Server:</strong> Ticket and authenticator verified! Access granted.";

        }

    </script>

</body>

</html>

A close up of a computer screen

AI-generated content may be incorrect.

## **A close up of a computer screen AI-generated content may be incorrect.3.3) What is the ticket (Hex numbers) in your implementation? What is the authenticator (Hex numbers) in your implementation?**

**Ticket: Ticket (Hex):** vC7NIaiL6ROERurlUcgKCPIF4+QE3U0577V8bJhRIGfB1qb9GaXIb86maKUr7gSXJMaO5vR12C/kIcRqDvlchQtANeQRJLD9kFAulrrcs/DRWXCy+lujjOYR1zIYfLZYjpyohLvLz+pOtkWUeUEGMA==

**Authenticator (Hex):**

7KZycARHeKv6G47JOvOPZholryxvNhcHqcl+otltPAK/2s+b+4quQdac6gf5p0E/

How does this work?

Phase 1

AS sends back to the client after ticket request {sk, ticket}. This is encrypted by KC.

Client decrypts this value with KS in order to get sk.

Phase 2

Client then sends to Server (Ticket, authenticator).

The ticket is encrypted with Ks, which was done by AS in Phase 1.

Authenticator is encrypted with Sk by Client.

Server then decrypts the ticket with their Ks, gets the Sk, then verifies the authenticator value.

*Extra*

The ticket is encrypted with the Server Key (Ks), not to be confused with session key Sk.

* In Phase 1, Step 2 the AS:
  + Created a random 128 bit Sk.
  + Sk was embedded into the ticket. This is encrypted by Ks.
  + This value is sent to the client separately and encrypted with Kc.
* Sk is a short-lived key for session,

## **3.4) Please explain how the user can be authenticated by the server and how the server can be authenticated by the user in the protocol.**

To tackle this problem, we will first discuss the Server.

Server Authentication

The Server (Cayetano) verifies the Client (Joshua) by looking at the Ticket and Authenticator that is sent to it by the client in phase 2. First the Server decrypts the Ticket by using its secret key, Ks. The reason it can do this is the Authentication Server (AS) encrypted the ticket with Ks which it also knows. The ticket contains {S, sk, ts, Lt}.

S refers to Server ID, sk is session key, ts s timestamp and Lt is lifetime. S is obtained by Client receiving this from the AS from Phase 1. This is also included in ticket to ensure the ticket is only useable by the right server.

The Server needs to check 2 things. The first is if the Server ID is matches its own ID, the second is that if the ticket is valid. A ticket is valid only if ts +Lt >= current time when decrypted. The focus of decrypting the ticket is obtaining the session key, sk from the ticket. Sk will be used to decrypt the Authenticator. We only do this once we confirm if server ID is correct and if the ticket is still valid.

The Authenticator contains {C, ts}.

This refers to the Client ID, and timestamp. The server by using the session key (sk) obtained from the ticket, can decrypt the Authenticator. Here the Server checks if the C matches Client and if and if the ts is recent. The reason why the time stamp needs to be recent is to avoid replay attacks. If a session key was obtained from an attacker, they could only decrypt a previous message but not future messages as they are isolated by sessions. If both checks are valid the Client is authenticated by the Server.

Client Authentication

The Client (Joshua) verifies the Server’s (Cayetano) identity by obtaining the session key (sk) from the Authentication Server (AS) in Phase 1. The sk is decrypted with Kc. Both the Client and AS share the same Kc. The sk is then used by the client to encrypt the Authenticator and send it to server. The server should be able to decrypt the Authenticator by obtaining the sk from ticket which they decrypted with their Ks. This means the Server is who they say they are as only real Server should have Ks.

What is important to distinguish is that the Server does not know the Client beforehand. The Server trusts that it is Client as only real client could have obtained sk from AS. The real client could only encrypt authenticator with sk.

# **Question 4: Diffie-Hellman**

## **4.1)Implemenent 160-bit random number geneator and secure hash algorithm SHA1;**

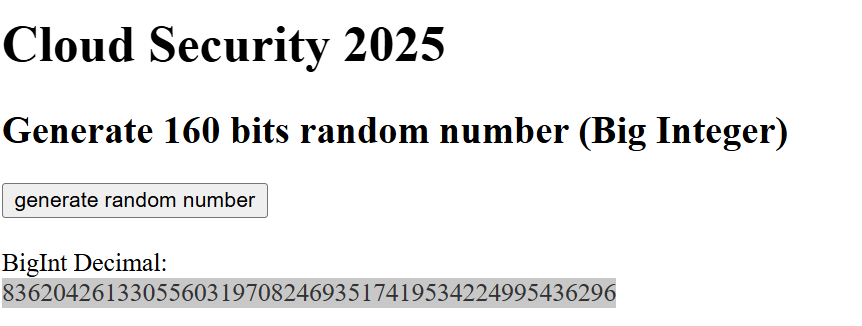
I used the following code <https://titan.csit.rmit.edu.au/~e31435/2025/lab8/random.html> in order to generate 2 random 160 bit numbers (decimal). This will represent the randomised values, “**a**” for Client and “**b**” for Server.

* Copy and run as “**Q4random.html**”

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  | <!DOCTYPE html> | |  | <html> | |  | <body> | |  |  | |  | <h1>Cloud Security 2025</h1> | |  |  | |  | <h2>Generate 160 bits random number (Big Integer)</h2> | |  |  | |  | <button type="button" onclick="generate\_random\_number()">generate random number</button> | |  | <br/> | |  | <br/> | |  | BigInt Decimal: | |  | <div id="output"></div> | |  |  | |  | <script src="[biginteger.js](https://titan.csit.rmit.edu.au/~e31435/2025/lab8/biginteger.js)"></script> | |  |  | |  | <script type="text/javascript"> | |  |  | |  | function generate\_random\_number(){ | |  | // display output on the page | |  | document.getElementById('output').innerHTML = random\_bigInt(); | |  | } | |  |  | |  | // generate random number | |  | // | |  | // one output: 160 bits random number (bigInt decimal) | |  | // | |  | // generate random number less than 2: 0 or 1 | |  | // Math.floor(Math.random() \* 2) | |  | // | |  | function random\_bigInt(){ | |  |  | |  | var random\_number = ""; | |  | // generate random bit 160 times | |  | for (i = 0; i < 160; i++) { | |  | random\_number += Math.floor(Math.random() \* 2); | |  | } | |  | // return Decimal bigInt | |  | return bigInt(random\_number,2); | |  | } | |  | </script> | |  |  | |  | </body> | |  | </html> | |  |  | |
|  |
|  |

**a.** 160-bit random number(decimal):

**836204261330556031970824693517419534224995436296**

****

**b.** 160-bit random number(decimal):

**1179801408114423392233118431269275225083686068172**

A screenshot of a computer

AI-generated content may be incorrect.

Implement Sha-1 algorithm

import java.math.BigInteger;

import java.security.MessageDigest;

import java.security.NoSuchAlgorithmException;

public class **Q4sha** {

    public static String sha1(BigInteger number, String name) {

        try {

            MessageDigest md = MessageDigest.getInstance("SHA-1");

            // Convert BigInteger to byte array

            byte[] numberBytes = number.toByteArray();

            byte[] nameBytes = name.getBytes();

            // Concatenate numberBytes + nameBytes

            byte[] combined = new byte[numberBytes.length + nameBytes.length];

            System.arraycopy(numberBytes, 0, combined, 0, numberBytes.length);

            System.arraycopy(nameBytes, 0, combined, numberBytes.length, nameBytes.length);

            // Compute SHA-1

            byte[] hashBytes = md.digest(combined);

            BigInteger hashInt = new BigInteger(1, hashBytes);

            // Convert to 40-character hex string

            String hashtext = hashInt.toString(16);

            while (hashtext.length() < 40) {

                hashtext = "0" + hashtext;

            }

            return hashtext;

        } catch (NoSuchAlgorithmException e) {

            throw new RuntimeException(e);

        }

    }

    public static void main(String[] args) {

        BigInteger a = new BigInteger("836204261330556031970824693517419534224995436296");

        BigInteger b = new BigInteger("1179801408114423392233118431269275225083686068172");

        String firstName = "Joshua"; // Replace with your first name

        String surname = "Cayetano"; // Replace with your surname

        // Compute A = SHA1(a || firstName)

        String A = sha1(a, firstName);

        System.out.println("A = SHA1(a || " + firstName + ") = " + A);

        // Compute B = SHA1(b || surname)

        String B = sha1(b, surname);

        System.out.println("B = SHA1(b || " + surname + ") = " + B);

    }

}

**Result**



**a = SHA1(a || Joshua)** = 901c5425cdaa6eeba5306fd7f2c918a61ef40aaf

**b = SHA1(b || Cayetano)** = 1594067a0a213ff946a71d1fc864a623ea76f7a3

**Note:** We do not convert “Joshua” and “Cayetano” into hex. This is because in the assignment specification (4.3), it states that these are concatenated as they are.

## **4.2)Use a Crypto Library to implement the modular exploentiation algorithm for larger integers and use your implementation to output y=gx(mod p), where x=SHA1(your student ID).**

**Compute y = g^x mod p**

**x** = SHA1(student number)

= 3722151

Use <https://emn178.github.io/online-tools/sha1.html> =

7c177916a41959fcd0964ca183b8e072d593c818

G and P are provided in the assignment specifications. These are decimal values.

* Also highlighted below.

Use the following code in Java, e.g., “**Q4CryptoLibrary**.java”:

import java.math.BigInteger;

public class Q4CryptoLibrary {

    public static void main(String[] args) {

        // Given constants (p and g)

        BigInteger p = new BigInteger(

                "178011905478542266528237562450159990145232156369120674273274450314442865788737020770612695252123463079567156784778466449970650770920727857050009668388144034129745221171818506047231150039301079959358067395348717066319802262019714966524135060945913707594956514672855690606794135837542707371727429551343320695239");

        BigInteger g = new BigInteger(

                "174068207532402095185811980123523436538604490794561350978495831040599953488455823147851597408940950725307797094915759492368300574252438761037084473467180148876118103083043754985190983472601550494691329488083395492313850000361646482644608492304078721818959999056496097769368017749273708962006689187956744210730");

        // SHA-1 of student ID (hex string)

        String xHex = "7c177916a41959fcd0964ca183b8e072d593c818";

        BigInteger x = new BigInteger(xHex, 16); // Convert hex to BigInteger

        // Compute y = g^x mod p (g and p are in decimal, but base doesn't matter for

        // computation)

        BigInteger y = g.modPow(x, p);

        System.out.println("y = g^x mod p (hex): " + y.toString(16));

        System.out.println("y (hex): " + y.toString(16)); // Output in hex

        System.out.println("y (decimal): " + y); // Output in decimal (if needed)

    }

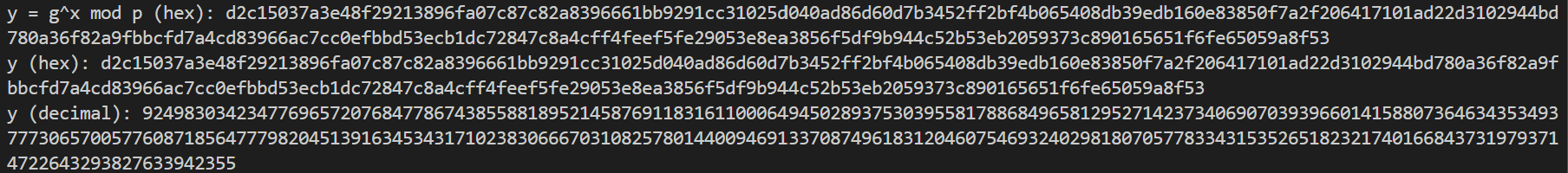
}

**Result (hex)=**

d2c15037a3e48f29213896fa07c87c82a8396661bb9291cc31025d040ad86d60d7b3452ff2bf4b065408db39edb160e83850f7a2f206417101ad22d3102944bd780a36f82a9fbbcfd7a4cd83966ac7cc0efbbd53ecb1dc72847c8a4cff4feef5fe29053e8ea3856f5df9b944c52b53eb2059373c890165651f6fe65059a8f53

**Result (decimal) =**

9249830342347769657207684778674385588189521458769118316110006494502893753039558178868496581295271423734069070393966014158807364634353493777306570057760871856477798204513916345343171023830666703108257801440094691337087496183120460754693240298180705778334315352651823217401668437319793714722643293827633942355



*EXTRA*

1. Don’t convert x to Decimal. This must stay as SHA-1.

X represents my student number in SHA-1. This is converted into Hex. If I use my normal value in decimal, the calculation would fail as it needs to be in SHA-1. This is because BigInteger needs a valid decimal string.

2.Why I shouldn’t convert both P and G into hexadecimal values.

We use BigInteger to store really large numbers. BigInteger internally represents numbers in binary so, hex and decimal base only matters during the result. When calculating g.modPow(x,p), x is converted from hex so the calculation holds.

*Manual Validation of* **y=gx(mod p)**

First, I need to change the values g and p from decimal to hex by using RapidTables. This is because x is a hash value in hex format.

<https://www.rapidtables.com/convert/number/decimal-to-hex.html>

G(decimal)=

174068207532402095185811980123523436538604490794561350978495831040599953488455823147851597408940950725307797094915759492368300574252438761037084473467180148876118103083043754985190983472601550494691329488083395492313850000361646482644608492304078721818959999056496097769368017749273708962006689187956744210730

G(hex) =

F7E1A085D69B3DDECBBCAB5C36B857B97994AFBBFA3AEA82F9574C0B3D0782675159578EBAD4594FE67107108180B449167123E84C281613B7CF09328CC8A6E13C167A8B547C8D28E0A3AE1E2BB3A675916EA37F0BFA213562F1FB627A01243BCCA4F1BEA8519089A883DFE15AE59F06928B665E807B552564014C3BFECF492A

P(decimal)=

178011905478542266528237562450159990145232156369120674273274450314442865788737020770612695252123463079567156784778466449970650770920727857050009668388144034129745221171818506047231150039301079959358067395348717066319802262019714966524135060945913707594956514672855690606794135837542707371727429551343320695239

P(hex)=

FD7F53811D75122952DF4A9C2EECE4E7F611B7523CEF4400C31E3F80B6512669455D402251FB593D8D58FABFC5F5BA30F6CB9B556CD7813B801D346FF26660B76B9950A5A49F9FE8047B1022C24FBBA9D7FEB7C61BF83B57E7C6A8A6150F04FB83F6D3C51EC3023554135A169132F675F3AE2B61D72AEFF22203199DD14801C7

Boxentriq, <https://www.boxentriq.com/code-breaking/modular-exponentiation>

Sub values into formula **y=gx(mod p).** Select “**Use hexadecimal numbers”.**

G(hex) =

F7E1A085D69B3DDECBBCAB5C36B857B97994AFBBFA3AEA82F9574C0B3D0782675159578EBAD4594FE67107108180B449167123E84C281613B7CF09328CC8A6E13C167A8B547C8D28E0A3AE1E2BB3A675916EA37F0BFA213562F1FB627A01243BCCA4F1BEA8519089A883DFE15AE59F06928B665E807B552564014C3BFECF492A

**X(hex)** =

7c177916a41959fcd0964ca183b8e072d593c818

P(hex)=

FD7F53811D75122952DF4A9C2EECE4E7F611B7523CEF4400C31E3F80B6512669455D402251FB593D8D58FABFC5F5BA30F6CB9B556CD7813B801D346FF26660B76B9950A5A49F9FE8047B1022C24FBBA9D7FEB7C61BF83B57E7C6A8A6150F04FB83F6D3C51EC3023554135A169132F675F3AE2B61D72AEFF22203199DD14801C7

Result =

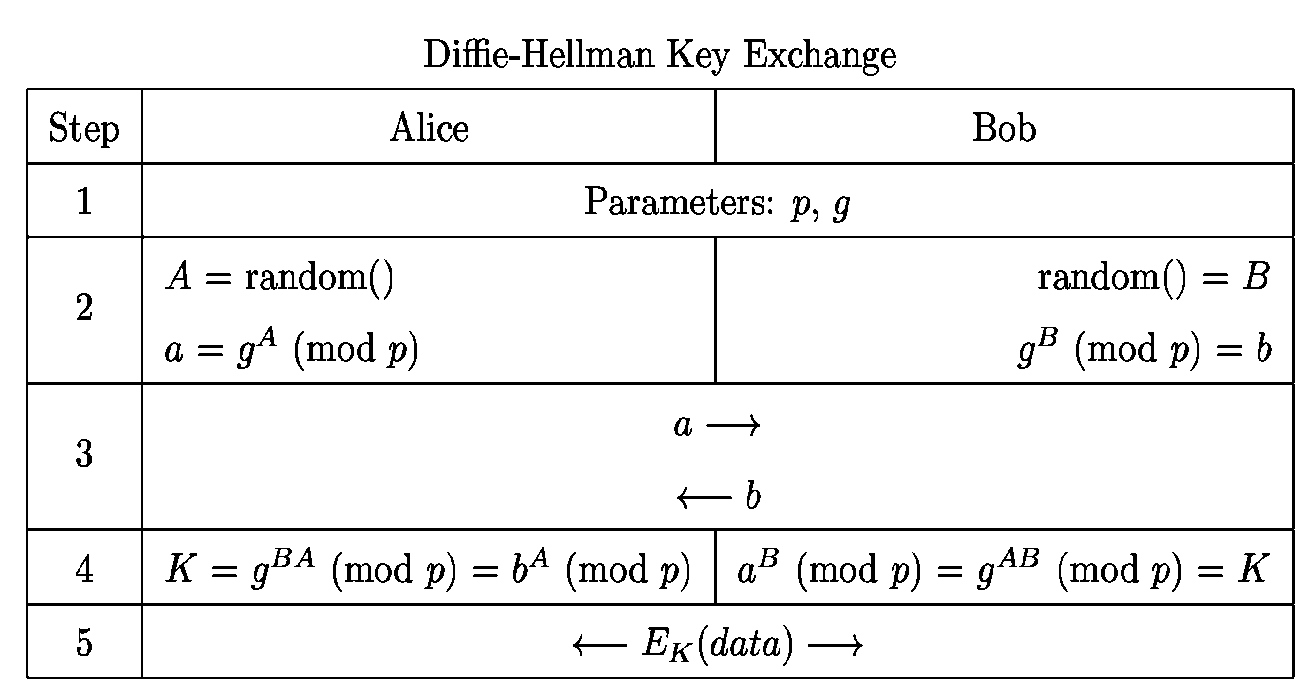
d2c15037a3e48f29213896fa07c87c82a8396661bb9291cc31025d040ad86d60d7b3452ff2bf4b065408db39edb160e83850f7a2f206417101ad22d3102944bd780a36f82a9fbbcfd7a4cd83966ac7cc0efbbd53ecb1dc72847c8a4cff4feef5fe29053e8ea3856f5df9b944c52b53eb2059373c890165651f6fe65059a8f53

This is the same as what was calculated in Boxentriq. So the code works.

## **4.3)After randomly genetating 160-bit a and 160-bit b, output (A, gA(mod p)) and (B, gB(mod p)) and the secret key gAB (mod p) established between your VPC (named by your first name) and your data centre (named by your surname) by the Diffie-Hellman key exchange protocol, where A=SHA1(a, your first name) and B=SHA1(b, your surname).**

We don’t need to code here.

What we need to do is use the private keys we generated in 4.1 and sub them into the equations. After 4.1, both the VPC and the Data Centre should have exchanged their public keys, A and B (this will be calculated below). By using the private key values (a and b) and subbing them into equation, **g^ba mod p,** both the VPC and Data Centre should come to generate the output.



**Diffie-Hellman Key Exchange Figure** (X Yi, Chapter 6 Security Protocols Chapter 6.2 SSL)

Important: There is a mix-up between the randomise values. Ideally it should be A and B but in assignment specification, it is written as a and b.

a(Private Key – VPC) = SHA1(a + my first name)

**901c5425cdaa6eeba5306fd7f2c918a61ef40aaf**

b(Private Key – Data Centre) = SHA1(b + my last name)

**1594067a0a213ff946a71d1fc864a623ea76f7a3**

~~a\*b=~~

NOTE: Computationally this is impossible to do as VPC and Data Centre don’t know the others secret private key. Hence it can’t be computed on both sides. This prevents attackers from figuring this out and would have to be done by brute force.

Calculate (A = ga(mod p))

This is the public key value calculated from VPC which is then sent to Data Centre.

We just need to use <https://www.boxentriq.com/code-breaking/modular-exponentiation>.

Click “**Use hexadecimal numbers**”.

**Important:** We need to use the Hex versions of G and P. The specification values are in decimal.

G(hex) =

F7E1A085D69B3DDECBBCAB5C36B857B97994AFBBFA3AEA82F9574C0B3D0782675159578EBAD4594FE67107108180B449167123E84C281613B7CF09328CC8A6E13C167A8B547C8D28E0A3AE1E2BB3A675916EA37F0BFA213562F1FB627A01243BCCA4F1BEA8519089A883DFE15AE59F06928B665E807B552564014C3BFECF492A

a(Private Key – VPC) = SHA1(a + my first name)

**901c5425cdaa6eeba5306fd7f2c918a61ef40aaf**

P(hex)=

FD7F53811D75122952DF4A9C2EECE4E7F611B7523CEF4400C31E3F80B6512669455D402251FB593D8D58FABFC5F5BA30F6CB9B556CD7813B801D346FF26660B76B9950A5A49F9FE8047B1022C24FBBA9D7FEB7C61BF83B57E7C6A8A6150F04FB83F6D3C51EC3023554135A169132F675F3AE2B61D72AEFF22203199DD14801C7

Result= A=

14168869b76b84bf5f91f9f42c66467721a66c99b6a0eab215ffcf010485ccad91ba539a686fc211c7c84306c2f6d226f53a1b0754c55ab36ce5e2777278cce5c697a5d665c8a049d71c48a55a2bdc331af93886e5ee9c3011f193bac3920e7427f9c624a6488214dbc51cee96cfee94887ae198c674aa82fe62a6a66a623505

Calculate (B = gb(mod p))

This is the public key value from calculated from Data Centre and sent to VPC.

We just need to use <https://www.boxentriq.com/code-breaking/modular-exponentiation>.

Click “Use hexadecimal numbers”

G(hex) =

F7E1A085D69B3DDECBBCAB5C36B857B97994AFBBFA3AEA82F9574C0B3D0782675159578EBAD4594FE67107108180B449167123E84C281613B7CF09328CC8A6E13C167A8B547C8D28E0A3AE1E2BB3A675916EA37F0BFA213562F1FB627A01243BCCA4F1BEA8519089A883DFE15AE59F06928B665E807B552564014C3BFECF492A

b(Private Key – Data Centre) = SHA1(b + my last name)

**1594067a0a213ff946a71d1fc864a623ea76f7a3**

P(hex)=

FD7F53811D75122952DF4A9C2EECE4E7F611B7523CEF4400C31E3F80B6512669455D402251FB593D8D58FABFC5F5BA30F6CB9B556CD7813B801D346FF26660B76B9950A5A49F9FE8047B1022C24FBBA9D7FEB7C61BF83B57E7C6A8A6150F04FB83F6D3C51EC3023554135A169132F675F3AE2B61D72AEFF22203199DD14801C7

Result = B

d41c1befcbb5b1671ad81ecf4a279b51766c6433a70850150dc3645cbe77d45cb0e038e707240e7d94c92da5b507bd555e904338c00883c2ed8a5b3a23d846b759f534ba77a1ec7bef8a3dcbd4350ba3c23d1593940d0e2b549499945fb8426f362a69dc1f0e3f6aa189d3919d6a97f8efe99e12fe13f3f215b1aff3c0ba0765

Calculate Shared Key(K)

We just need to use <https://www.boxentriq.com/code-breaking/modular-exponentiation>.

* Click “**Use hexadecimal numbers**”.

When we look at the Diffie-Hellman Key Exchange Figure, K refers to the shared key.

* The VPC cannot do g^ba mod p to calculate the shared key as I they don’t know the Data Center’s randomised value (secret key). So, they use the formula, **B^a mod p.**
* The Data Centre cannot do g^ab mod p to calculate shared key. This is because they don’t know the VPC’s private key or randomised value. Instead, they use the formula **A^b mod p.**

**B^a mod p**

B(hex)=

d41c1befcbb5b1671ad81ecf4a279b51766c6433a70850150dc3645cbe77d45cb0e038e707240e7d94c92da5b507bd555e904338c00883c2ed8a5b3a23d846b759f534ba77a1ec7bef8a3dcbd4350ba3c23d1593940d0e2b549499945fb8426f362a69dc1f0e3f6aa189d3919d6a97f8efe99e12fe13f3f215b1aff3c0ba0765

a=

a(Private Key – VPC) = SHA1(a + my first name)

**901c5425cdaa6eeba5306fd7f2c918a61ef40aaf**

P(hex)=

FD7F53811D75122952DF4A9C2EECE4E7F611B7523CEF4400C31E3F80B6512669455D402251FB593D8D58FABFC5F5BA30F6CB9B556CD7813B801D346FF26660B76B9950A5A49F9FE8047B1022C24FBBA9D7FEB7C61BF83B57E7C6A8A6150F04FB83F6D3C51EC3023554135A169132F675F3AE2B61D72AEFF22203199DD14801C7

Result = K =

641ad70dd6cdada4cadcb2755a8bd456d0f5be8ef163562476bd6dd94be039fb1d009cb589d238f7a1d211029136b52a2eafb7af761ac2ecb1d699dbe814f99c2972f126eb2828c087a3487afed62242c7a2c6fe955f7e0527ab4437a40a62a8d5396eac72d22ecb71335fc36ce1aebe1f42bae6b607aa91368b77c2165f9d2

**A^b mod p.**

A(hex)=

14168869b76b84bf5f91f9f42c66467721a66c99b6a0eab215ffcf010485ccad91ba539a686fc211c7c84306c2f6d226f53a1b0754c55ab36ce5e2777278cce5c697a5d665c8a049d71c48a55a2bdc331af93886e5ee9c3011f193bac3920e7427f9c624a6488214dbc51cee96cfee94887ae198c674aa82fe62a6a66a623505

b=

b(Private Key – Data Centre) = SHA1(b + my last name)

**1594067a0a213ff946a71d1fc864a623ea76f7a3**

P(hex)=

FD7F53811D75122952DF4A9C2EECE4E7F611B7523CEF4400C31E3F80B6512669455D402251FB593D8D58FABFC5F5BA30F6CB9B556CD7813B801D346FF26660B76B9950A5A49F9FE8047B1022C24FBBA9D7FEB7C61BF83B57E7C6A8A6150F04FB83F6D3C51EC3023554135A169132F675F3AE2B61D72AEFF22203199DD14801C7

Result = K=

641ad70dd6cdada4cadcb2755a8bd456d0f5be8ef163562476bd6dd94be039fb1d009cb589d238f7a1d211029136b52a2eafb7af761ac2ecb1d699dbe814f99c2972f126eb2828c087a3487afed62242c7a2c6fe955f7e0527ab4437a40a62a8d5396eac72d22ecb71335fc36ce1aebe1f42bae6b607aa91368b77c2165f9d2

Since both values are the same, then calculations are correct and nothing has been compromised.

## **4.4)Can you perform a Man-in-the-Middle Attack to the Diffie-Hellman key exchange protocol? If so, show attacking steps.**

Yes a Man-in-the-Middle attack can be used on Diffie-Hellman key exchange protocol.

A Man-in-the-Middle is when an attacker intercepts communications between 2 parties and can listen between their communications as well as impersonate one of the parties. In this case it would be between the VPC and the Data Centre.

It is possible for an attacker to obtain the values p, g and the public keys A and B (reversed based on assignment specification). However, they cannot compute the shared key (K). This is because they don’t know either of parties (VPC or Data Centres) secret keys, a and b (which is reversed based on assignment specification). The attacker would been to brute force all possible values which will take a long time to do (GeeksforGeeks, 2022).

This was briefly explained in 4.3.

However, it is Diffie Hellman is vulnerable to Man-in-the-middle attacks.

Steps

**1)The public numbers, p and g are generated. P is a prime number while is referred to as the base.**

g=

174068207532402095185811980123523436538604490794561350978495831040599953488455823147851597408940950725307797094915759492368300574252438761037084473467180148876118103083043754985190983472601550494691329488083395492313850000361646482644608492304078721818959999056496097769368017749273708962006689187956744210730

p=

178011905478542266528237562450159990145232156369120674273274450314442865788737020770612695252123463079567156784778466449970650770920727857050009668388144034129745221171818506047231150039301079959358067395348717066319802262019714966524135060945913707594956514672855690606794135837542707371727429551343320695239

**2) Both parties select private numbers.**

The VPC select a random number “a”.

a(Private Key – VPC) = SHA1(a + my first name)

d05f0655fc4917e2d430f0190f1dbbe45139e2ed

The Data Centre selects a random number “b”.

b(Private Key – Data Centre) = SHA1(b + my last name)

d42c6b0f2238a83484c4121575919b9fe7a48a2f

An attacker, called Frank picks 2 random numbers, “c” and “d”

c = 3

d = 5

Note: This is a simplified example. In the real world, larger numbers are used.

**3) The attacker intercepts both parties Public Values**

NOTE: A, B, a, b are mixed up between the lecture slides and assignment specifications.

Frank, intercepts the VPC’s public value A = g^a mod p. This is blocked from reaching the Data Centre. Frank then sends their own public values C = g^c mod p. Frank also intercepts the Date Centres public value, B =g^b mod p. Frank then sends the VPC their own public value D=g^d mod p.

**4) The secret key is computed**

Both the VPC and Data Centre will compute the shared secret key. However, this is based on the values that Frank, the attacker sent by to both parties. VPC will compute K1 = g^da mod p. Data Centre will compute K2 = g^cb mod p.

**5)**

The VPC will use K1 to encrypt a message to Data Centre. However, Frank can decrypt and re-encrypt it using K2 and sends the message to Data Centre. Both parties will not notice anything and will assume that their messages are secure. Frank as an attacker can decrypt, read, modify and re-encrypt all their conversations.

## **4.5)How does IKEv2 overcome the Man-in-the-Middle Attack? Show steps.**

*What it is*

IKEv2 is a VPN protocol that is used to ensure that communications between device is safe by establishing and verifying IPsec connections. This is widely used to keep connections stable even when switching between networks like mobile data and Wi-Fi (NordLayer, 2025).

*How it overcomes Man-in-the-Middle*

Diffie-Hellman has the vulnerability that anyone can send the shared public key values. There is no authentication, no identity verification, no new keys are used, and we do not encrypt the public keys (A and B in this case). IKEv2 solves this problem by introducing the usage of digital signatures or pre-shared keys (PSK). Certificates or PSK’s can be used to verify identities, nonces (random values) can be used to ensure new keys for each session. Finally, after authentication, the public keys are encrypted.

*Steps*

Step 1: Authentication with Digital Signatures or PSKs

We can authenticate either by using certificates or pre-shared keys (PSK). Certificates have both parties sign their shared Key (in this question, A and B) with a private key. The receiver verifies the signature by using the sender’s public certificate (e.g., RSA). With pre-shared keys, both parties use a shared secret key to hash the session data.

HMAC-SHA256 (PSK, nonce of A, nonce of B)

This prevents an attacker from forging signatures or guessing the PSK. This is because the PSK is a long-generated value (e.g. 256 bits). The attacker needs PSK to compute the correct HMAC value to impersonate either party. That would mean that the verification would fail.

Step 2: Nonces for Session Freshness

By itself, Diffie-Hellman can have an attacker use replay attacks if the same sessions keys are used repeatedly. To solve this, both parties generate random numbers or nonce values during a handshake. Session keys/unique keys (i.e. Ephemeral keys) are calculated by nonce\_a +nonce\_b + g^AB. This is only for session only. By using new session keys and nonce values, each session uses unique keys even if long term keys (private keys) are compromised. For example, if the Data Centre and VPC’s private keys (a and b) were compromised, sessions keys would be different as each session has different keys, with old keys discarded after usage.

Step 3: Public key encryption

After authentication the public keys are encrypted using symmetric keys, (SK\_e).

SK\_e = HKDF (g^AB, nonce\_a, nonce\_b, “encryption key)

The attacker can see the ciphertext but cannot see the public key values. In this case the public key values are g^a mod p sent by VPC and g^b mod p sent by Data Centre in 4.3.

Step 4: Key Confirmation

There is no proof that both parties computed the same shared key value(S), g^AB. To solve this, you would use this:

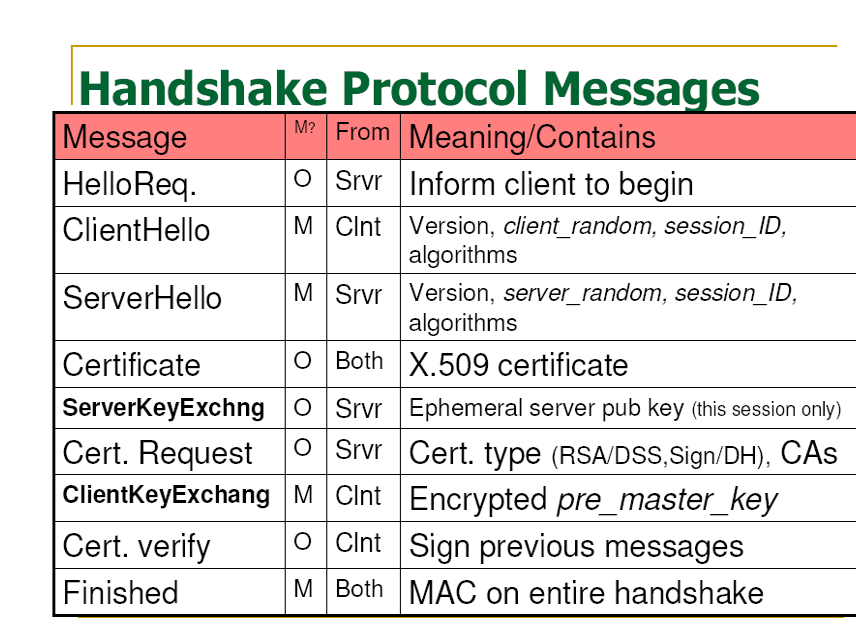
HASH 1= SHA-256 (g^AB + nonce\_a, nonce\_b, +”client”)

HASH 2 = SHA-256 (g^AB + nonce\_a, nonce\_b, +”server”)

“Client” and “server” are fixed strings. We use them to make distinguish the parties hashes. If an attacker tampered with g^AB, the hashes won’t match.

# **Question 5: SSL Hand**

Overview of steps in SSL Handshake Protocol.



**Chapter 6 Security Protocols (X. Yi, 2025)**

1. Client sends “ClientHello” to server

2.Server responds with “ServerHello”, the certificate, ServerKeyExchange, ServerHelloDone.

- ServerKeyExchange is the RSA signature ephemeral RSA key or DH exponent.

3. Client sends to server ClientKeyExchang and ChangeCipherSpec (CCS).

- If RSA used this refers to encrypted pre-master. If DH is used, this refers to the client’s exponent.

4. Server then sends back ChangeCipherSpec (CCS) and is finished.

Generally, this question only requires using the Certificate, ServerKeyExchange and ClientKeyExchange.

The ServerKeyExchange is done in 5.2 by calculating the Ephemeral RSA public Key by using **n\_eph** and my student number in hex. This is used to enable forward secrecy (sessions secure even if long term keys are compromised).

The ClientKeyExchange is done in 5.3 by using **n\_eph** and **e\_eph.** This is to establish a shared secret.

The Certificate will be used in 5.4 to authenticate the server.

## ***5.1) Choose your* ephemeral public key with 1024 bits and set e as the largest prime factor of your student number.**

What I need to be create a temporary RSA key for forward secrecy. The general steps to do this is:

1.Find the largest prime factor of my student number.

2.Select a 1024-bit modulus (**n\_eph)**

3.Compute the ephemeral private key. **d\_\_eph**

1.Find the largest prime factor of my student number.(**e\_eph)**

My student number is 3722151.

I used <https://www.calculatorsoup.com/calculators/math/prime-factors.php> to find the highest prime factor of my student number.

= 3 x 1240717

So **e** = 1240717

This is in decimal.

In HEX it is **12EE8D** by using <https://www.rapidtables.com/convert/number/decimal-to-hex.html?x=1240717>

* We use this value as a public exponent, **e\_eph**

2.Select a 1024 bit modulus **n\_eph**

The ephemeral RSA public key needs to have a key size of 1024 bits. This is calculated with modulus, **n = p \*q.**

This must be independent of the server’s long term key, **n**.

I used this: <https://www.mobilefish.com/services/rsa_key_generation/rsa_key_generation.php#google_vignette>

* Navigate to “Step 1” and select 1024 bits for key size.
* Clicked “Auto generate prime number p and q”.
* Use the values generated below. You will generate new values if you don’t use the already used ones.

All is in hexadecimal with a bit size of 512.

Prime number p =

eeb010d23b95dab208023fee98c6da670485fe516d4f79d1cfc9361aaeed57181e3d4f3f0c422035bc3214d9402c75deba5ed7a7388f759942faa57ada62a081

Prime number q =

b4c99444ee1c3afb43db78675076ca295a8ea081e70087c8cea8838e88554390c06a9491c9cc05ef2e10cac09bc4e3de8ea5ed298b8a435035e7f2bcbbd22a29

Euler's phi(n) function - phi(n) = (p - 1) \* (q - 1) = **λ(n\_eph)** =

a88fbe4f15762a9bcdbf8d4176e5b44c779a4bb8259c0db363c1d10b45dc35b2bcd4d2b01c37e906168a807a217260c395c9f86af9b059c044894215c18729c5f6290974b42227081aff0babfb655b7ec194db78e6feb92caec6b6fd9860ac941ef2b497113bc01ae4ad20071056bf9be5b13ca5c0c0584f0244112b76be1400

3.Compute the ephemeral private key. **d\_\_eph**

The ephemeral key is a temporary key that we use that is discarded after the session. Even if the server’s long term private key is exposed, past sessions remain secure.

We use <https://www.boxentriq.com/code-breaking/modular-multiplicative-inverse>

**Ephemeral private key (d\_\_eph)** = **e\_eph** ^-1 mod lambda (**n\_eph**)

We enter the following values:

Number a = **e\_eph** =

**12EE8D**

Modulo = **λ(n\_eph)**

IMPORTANT: Make sure to select “ Use hexadecimal numbers”.

**Result = d\_\_eph =**

15fe3d460e57d28f345294762d5cf122139501acd45171565607445b4f9eb9b59f6a460fe7ac66200baeefe40ea727a543cae22b92c6f628335e80f1e79bad3e21e849b17e2c78f968f9c9e64d121303e87b5d631b4b2b47e9ccb38a6b30daa0191619d433af2ef6bed270fb39653c4818ebf36a38105974cf4fc045c2261045

Why we do these calculations:

This value is the server’s ephemeral private exponent to decrypt the ClientKeyExchange message.

Generally, the Client will send the ClientKeyExchange.

-ClientKeyExchange = **PreMasterSecret**^**e\_eph** mod **n\_eph**

The Server can decrypt this value by reversing the values

- **PreMasterSecret** = ClientKeyExchange ^**e\_eph** mod **n\_eph**

The server can then derive the session keys.

Verification

**(e\_eph** \* **d\_eph) mod λ(n\_eph) = 1**

This is to check if inverse is correct i.e. equal 1.

<https://www.boxentriq.com/code-breaking/big-number-calculator>

1.Calculate **(e\_eph** \* **d\_eph)**

* Make sure numercal system is “Hexadecimal”, arithmetic “Standard”.
* Click “a \* b”

Number a = **e\_eph** =

**12EE8D**

Number b = **d\_eph** =

15fe3d460e57d28f345294762d5cf122139501acd45171565607445b4f9eb9b59f6a460fe7ac66200baeefe40ea727a543cae22b92c6f628335e80f1e79bad3e21e849b17e2c78f968f9c9e64d121303e87b5d631b4b2b47e9ccb38a6b30daa0191619d433af2ef6bed270fb39653c4818ebf36a38105974cf4fc045c2261045

Result = **(e\_eph** \* **d\_eph) =**

=

1a05ec8ebe3185a31c95307d0d3cc58c3de736ec7874ea04bbd0af1f9237979d858b687c9aa440f26dccff0692dfc428b3f50dc1366721f42908b37f033817a2c9113cfee7930587fe2873d79bac496e873f64dbc7e3ef5b2225cb4651c53d46b050e2e1b243ea27fae1ab09bd2d8f7defb7833b9c2c65e6271add895679876f11c01

2.Calculate **(e\_eph** \* **d\_eph) mod λ(n\_eph)**

Number a =**(e\_eph** \* **d\_eph) =**

=

1a05ec8ebe3185a31c95307d0d3cc58c3de736ec7874ea04bbd0af1f9237979d858b687c9aa440f26dccff0692dfc428b3f50dc1366721f42908b37f033817a2c9113cfee7930587fe2873d79bac496e873f64dbc7e3ef5b2225cb4651c53d46b050e2e1b243ea27fae1ab09bd2d8f7defb7833b9c2c65e6271add895679876f11c01

Number b = **λ(n\_eph)**

a88fbe4f15762a9bcdbf8d4176e5b44c779a4bb8259c0db363c1d10b45dc35b2bcd4d2b01c37e906168a807a217260c395c9f86af9b059c044894215c18729c5f6290974b42227081aff0babfb655b7ec194db78e6feb92caec6b6fd9860ac941ef2b497113bc01ae4ad20071056bf9be5b13ca5c0c0584f0244112b76be1400

Result = 1

D\_eph is correct.

That means this is the inverse of e\_eph. Without this the server cannot decrypt the ClientKey Exchange message, leading to handshake failure. If d\_eph is incorrect, an attacker could calculate it and this would break forward secrecy, leading to past sessions being decrypted.

## ***5.2) What is the ServerKeyExchange message (Hex numbers) in Figure 5?***

Here we send the server’s ephemeral RSA public key to the client.

The public key is donated as = (n\_eph, e\_eph)

(n\_eph, e\_eph )-> n\_eph ||e\_eph

**n\_eph**=

This is calculated as **n = p \* q.**

This is from 5.1.

Prime number p =

eeb010d23b95dab208023fee98c6da670485fe516d4f79d1cfc9361aaeed57181e3d4f3f0c422035bc3214d9402c75deba5ed7a7388f759942faa57ada62a081

Prime number q =

b4c99444ee1c3afb43db78675076ca295a8ea081e70087c8cea8838e88554390c06a9491c9cc05ef2e10cac09bc4e3de8ea5ed298b8a435035e7f2bcbbd22a29

Use <https://www.boxentriq.com/code-breaking/big-number-calculator>

* Select Numerical system “Hexadecimal”, Arithmetic “Standard” and after inserting values, select “a\*b”.

**Result = n = n\_eph**

a88fbe4f15762a9bcdbf8d4176e5b44c779a4bb8259c0db363c1d10b45dc35b2bcd4d2b01c37e906168a807a217260c395c9f86af9b059c044894215c18729c799a2ae8bddd43cb566dcc401e4a3000f20a97a4c3b4ebac74d3870a6cfa3473cfd9a9867e749e63fceefffa0ec4819592eb6017684da11387b26a9630cf2dea9

e\_eph =

**12EE8D**

**n\_eph ||e\_eph** =

a88fbe4f15762a9bcdbf8d4176e5b44c779a4bb8259c0db363c1d10b45dc35b2bcd4d2b01c37e906168a807a217260c395c9f86af9b059c044894215c18729c799a2ae8bddd43cb566dcc401e4a3000f20a97a4c3b4ebac74d3870a6cfa3473cfd9a9867e749e63fceefffa0ec4819592eb6017684da11387b26a9630cf2dea9**12EE8D**

## ***5.3) If Pre\_Master\_Secret is SHA384(your real email address), where the hash function is SHA384 (***[*https://emn178.github.io/online-tools/sha384.html*](https://emn178.github.io/online-tools/sha384.html)***), what is the ClientKeyExchange message (Hex number) in Figure 5?***

What we needed to do is calculate **Pre\_Master\_Secret**. This will then be sent to the server.

This is by using [**https://emn178.github.io/online-tools/sha384.html**](https://emn178.github.io/online-tools/sha384.html) with my student email address.

Email: [s3722151@student.rmit.edu.au](mailto:s3722151@student.rmit.edu.au)

Output:

53fa1474e84361268e6acb63cd62dd0e6839bd474d1de4d290bf176c395dba19df014ccae91483cddd6e0b80c1886729

* This is a big-endian integer for RSA encryption.

We then use <https://www.boxentriq.com/code-breaking/modular-exponentiation>

ClientKeyExchange is pre-encrypted Pre\_Master\_Secret.

This is to calculate ***(Pre\_Master\_Secret)^e mod n***

* Make sure to select “**Use hexadecimal numbers”.**

We enter the following values:

**Number a** = Pre Master Secret (PMS) = Email =

53fa1474e84361268e6acb63cd62dd0e6839bd474d1de4d290bf176c395dba19df014ccae91483cddd6e0b80c1886729

**Exponent** = **e\_eph**

12EE8D

* This is a hex number.
* For forward secrecy it is important to use ephemeral RSA key we generated in 5.1.

**Modulo** = **n\_eph** =

a88fbe4f15762a9bcdbf8d4176e5b44c779a4bb8259c0db363c1d10b45dc35b2bcd4d2b01c37e906168a807a217260c395c9f86af9b059c044894215c18729c799a2ae8bddd43cb566dcc401e4a3000f20a97a4c3b4ebac74d3870a6cfa3473cfd9a9867e749e63fceefffa0ec4819592eb6017684da11387b26a9630cf2dea9

**Result = ClientKeyExchange message =**

67e323c283b7feb7f06edddf3fa1179b6b2a918ea1df42ed8453f1144bf77cba723bf2344e5d2efd7e13ba31b7a58a8b7b0e90f32af8c00640e59d5d9d8eab9528672326660b19da89266757d10d42e2a211e1520b935228df7abed093b38038dbb411e4c74db5beca6ab3a9a9af3c5e493a6fe28578099c15d8ee9aae6be7ae

*Extra explanation*

We will only use 10001 to authenticate the server via the certificate in 5.4.

*Why this works?*

Forward secrecy: When we use Boxentriq, PreMasterSecret(my email) is encrypted with ephemeral key (**e\_eph, n\_eph)** not server’s long-term key.

* That means **d\_eph** (temporary ephemeral key) is discarded after the session. Even if private key **d** (Assignment specification, 32e…) is compromised, past sessions remain secure.
* **d** id only used for authentication not encryption of session keys.

Server decryption: Only the server (with **d\_eph)** can decrypt the **PreMasterSecret.**

* **PreMasterSecret** is a 384-hash value encrypted by the client. This is by using **e\_eph** and **d\_eph.**

## ***5.4) Analyse client authentication and server authentication of the handshake protocol.***

In this step, we confirm if the client talks to the server. We do this in 2 ways.

1)Server Authentication

2)Client Authentication

Server Authentication

The server’s RSA certificate is signed by a trusted Certificate Authority (CA). The client then checks this certificate and validates this.

Client Authentication

This is an optional step which we don’t need do. Generally, the client signs a hash of the handshake by using the private key. My calculations focus on sever authentication and ephemeral key exchange (5.1 – 5.3).

*How we do Server Authentication*

1.Certificate Validation

The server’s certificate contains a 2048 bit RSA public key (n, e). **N** and **e** is from the assignment specification. A client check’s this by:

* Looking at the CA’s signature over the server’s certificate. This is by using the CA’s public key.
* There is also verification through a certificate chain. This is a hierarchical chain that links the server’s certificate back to a trusted Root Authority (CA). In my case, AWS Certificate Manager acts as trusted CA. How it works is The Server Certificate issues the certificate to the domain. This certificate contains (**n, e**=10001) which is signed by Intermediate CA.
* The client also goes over the metadata. This will contain values such as the domain name and expiry date. The client needs to check if the domain works and the CA is still valid and not expired. This is done by Online Certificate Stats Protocol (OCSP).

2. Proof of Private Key Ownership

This is critical for handshake. During this time, the Server proves it holds d (assignment specs). This is through:

* Signing the ServerKeyExhcange message. Within this, the RSA value, **n\_nph||e\_eph** from 5.2.
* The signing is done by:
  + 1.Calculating the hash handshake of messages. (E.g., Sha-256 of ClientHello +ServerHello + **n\_nph||e\_eph.**
  + 2. This has is then encrypted alongside the long term private key, d.
* Client Verification: The client then decrypts the signature using the server’s public key (n,e=10001) and compares it to the its own hash computation.
* This is essential as otherwise an attacker could create their own **n\_nph||e\_eph,** enabling man-in-the-middle attack.

3.Link to Forward Secrecy

The ServerKeyExchange contains my ephemeral RSA key. If the long term key **d** was known by an attacker, past sessions remain safe.

* Session keys are calculated from the PreMasterSecret which is encrypted with e\_enph.
* D\_eph was a temporary key and discarded after use (5.1).

4.Why e is only used for authentication?

The question uses 2 RSA keys.

The long-term key (n, e = 10001, d): This signs the handshake for authentication.

Ephemeral key (**n\_nph, e\_eph, d\_eph)** : These are used to encrypt the PreMasterSecret to ensure forward secrecy.

*Extra*

Important: e =10001(hex value) is only used for authentication, not for encryption.

We don’t do calculations in this question as server authentication relies on Certificate Authorities.

## ***5.5) Analyse the forward security of the handshake protocol.***

*What is Forward Secrecy?*

This is a feature to protect session key keys from being compromised even when the server’s private key is (d) is exposed. How this works is for each session, a unique session key (PreMasterSecret) is used. That means that past communications remain secure and only data encrypted with future sessions after being compromised are at risk.

*How my steps were able to do Forward Secrecy*

1.Ephemeral Key Generation

At step 5.1 the server generates ephemeral keys, **e\_eph, n\_eph** and **d\_eph.** E\_eph is the public exponent used to encrypt PreMasterSecret. N\_eph is the modulus that is part of ephemeral public key. D\_eph is used to decrypt the PreMasterSecret. This is deleted after the session. At the start of each session, new ephemeral keys are used ( e\_eph, n\_eph, d\_eph). After using these values, d\_eph is deleted. By deleting d\_eph, past session is immune to future compromises of **d** (long term key). That is because even if an attacker gained **d,**  they cannot use d\_eph as it is deleted which means they cannot decrypt the PreMasterSecret. They also cannot get d\_eph as the attack needs **p** and **q** to do this which would be very large numbers which will take time to compute.

2.Key exchange (5.2 – 5.3)

The client encrypts PreMasterSecret (based on SHA-384 my student email) using **e\_eph** and **n\_eph**. The only way that PreMasterSecret can be decrypted is with **d\_eph**.

3.Long Term Keys purpose (d)

The long-term key is only used to sign the ServerKeyExchange from the Server, proving it is them. This should never encrypt PreMasterSecret to ensure forward secrecy.

The long-term key (d) is only used for signing (authentication).

* **Long-term key** = (n, e, d)
* Note: We don’t need to calculate it in previous steps.
* N(assignment specification)= 
* E = 10001
* d **=**  modular inverse of **e** mod λ(n)

Even if the long-term key (d) was exposed, PreMasterSecret values cannot be decrypted as **d\_eph** no longer exists (session over). You also need e\_eph.

This enables forward secrecy.

Without Ephemeral Keys, PreMasterSecret would be encrypted with server’s long-tem **e (10001).** If an attacker had **d,** they would have access to all sessions.

# References

vlogize (June 2024), ‘Decrypting AES CBC with PKCS5Padding in Java’ [video]**,** *vlogize*, YouTube, accessed on 22 April 2024. [**http://youtube.com/watch?v=ZSY-t95kaIY**](http://youtube.com/watch?v=ZSY-t95kaIY)

tutorial plus (2024), ‘Java AES Encryption and Decryption’ [video], *tutorial plus*, YouTube, Accessed on 22 April 2024. <https://www.youtube.com/watch?v=LtUU8Q3rgjM>

Computerphile (2021), ‘TLS Handshake Explained – Computerphile’ [video], *Computerphile,* YouTube, Accessed on 22 April 2024.<https://www.youtube.com/watch?v=86cQJ0MMses>

Practical Networking (2022), ‘Diffie-Hellman Key Exchange - the MAGIC that makes it possible - Cryptography - Practical TLS’ [video], Practical Networking, YouTube, Accessed on 22 April 2024.<https://www.youtube.com/watch?v=KXq065YrpiU>

Michael Grieco (2022), ‘C/C++ Math Library - 19 - Keccak/SHA3’ [video], *Michael Grieco*, YouTube, Accessed on 22 April 2024. <https://www.youtube.com/watch?v=9yWlIG0UkOI>

Vmware (2025), *Perfect Forward Secrecy Definition,* vmware website, accessed on 9 May 2025. <https://www.vmware.com/topics/perfect-forward-secrecy>

Chemejon (6 December 2021), *SHA-3 Explained in Plain English,* chemejon website, accessed 9 May 2021. <https://chemejon.wordpress.com/2021/12/06/sha-3-explained-in-plain-english/>

TrutheTable (2025), *Truth Table Generator – Boolean Table & Logic Gates, Trueth Table Generator,* accessed on 9 May 2025. <https://truthtablegen.com/>

W3schools (2025), Java Math pow() Method, *w3schools,* accessed on 9 May 2025. <https://www.w3schools.com/java/ref_math_pow.asp>

Nordlayer (2025), What is IKEv2 VPN protocol, *Nordlayer,* accessed on 9 May 2025. [**https://nordlayer.com/learn/vpn/ikev2/**](https://nordlayer.com/learn/vpn/ikev2/)

GeeksforGeeks (2022), Man in the Middle attack in Diffie-Hellman Key Exchange,*Geeksforgeeks, accessed on 9 May 2025.* <https://www.geeksforgeeks.org/man-in-the-middle-attack-in-diffie-hellman-key-exchange/>

GeeksforGeeks (2022), SHA-1 Hash,*Geeksforgeeks, accessed on 9 May 2025.* <https://www.geeksforgeeks.org/sha-1-hash-in-java/>

GeeksforGeeks (2022), Java Program to Convetr Binary to Hexadecimal,*Geeksforgeeks, accessed on 9 May 2025.* <https://www.geeksforgeeks.org/java-program-to-convert-binary-to-hexadecimal/>

LeventOzturk (N.a), ONLINE SHA-3 Keccak CALCULATOR - CODE GENERATOR, *LeventOxturk*, accessed on 9 May 2025. <https://leventozturk.com/engineering/sha3/>

Visualise Keccak (N.a), Keccak Algorithm Visualizer, *Visualise Keccak*, accessed on 9 May 2025. <https://visualizekeccak.com/theory>

Visualise Keccak (N.a), Keccak Simulation, *Visualise Keccak*, accessed on 9 May 2025. <https://visualizekeccak.com/simulation>

Yi, X (2025), Cloud Security 2025 SHA1 HASH DEMO, *RMIT University,*  accessed on 9 May 2025. <https://titan.csit.rmit.edu.au/~e31435/2025/lab7/sha1demo.html>,

Fang, X (n.a), Fig 2 – uploaded by Xin Fand, *Research Gate*, accessed on 9 May 2025. <https://www.researchgate.net/figure/Terminology-used-in-Keccak_fig2_280113206>

Yi, X (2025), ‘Chapter 4 Authentication Standards 4.3 Hash Function’ [slides], RMIT University, Melbourne

Yi, X (2025), ‘Chapter 6 Security Protocols Chapter 6.2 SSL’, [slides], RMIT University, Melbourne