Unit II Fundamentals of Algorithms in Cybersecurity

Assignment

Q1. Explain Data Encryption Standard (DES) and Rivest-Shamir-Adleman (RSA) Algorithms.

Data Encryption Standard (DES)

The Data Encryption Standard (DES) is a symmetric key algorithm used for the encryption of electronic data. It was once the standard encryption technique for many applications and was widely adopted across the globe. DES was developed in the 1970s by IBM and was later adopted by the National Institute of Standards and Technology (NIST) in the United States as a federal information processing standard.

Key Characteristics of DES:

- **Block Cipher:** DES works by encrypting data in fixed-size blocks, typically 64 bits in length.
- **Symmetric Key:** It uses the same secret key for both encryption and decryption. The key length for DES is 56 bits.
- **Feistel Network:** DES is based on a Feistel network structure, which divides the data block into two halves and applies a series of substitution and permutation operations.
- **Round Function:** It consists of 16 rounds of the same function applied to the data block, using a different 48-bit subkey derived from the original 56-bit key for each round.

Data Encryption Standard (DES) is a block cipher with a 56-bit key length that has played a significant role in data security**.** Data encryption standard (DES) has been found vulnerable to very powerful attacks therefore, the popularity of DES has been found slightly on the decline. DES is a block cipher and encrypts data in blocks of size of **64 bits** each, which means 64 bits of plain text go as the input to DES, which produces 64 bits of cipher text. The same algorithm and key are used for encryption and decryption, with minor differences. The key length is **56 bits**.

We have mentioned that DES uses a 56-bit key. Actually, The initial key consists of 64 bits. However, before the DES process even starts, every 8th bit of the key is discarded to produce a 56-bit key. That is bit positions 8, 16, 24, 32, 40, 48, 56, and 64 are discarded.

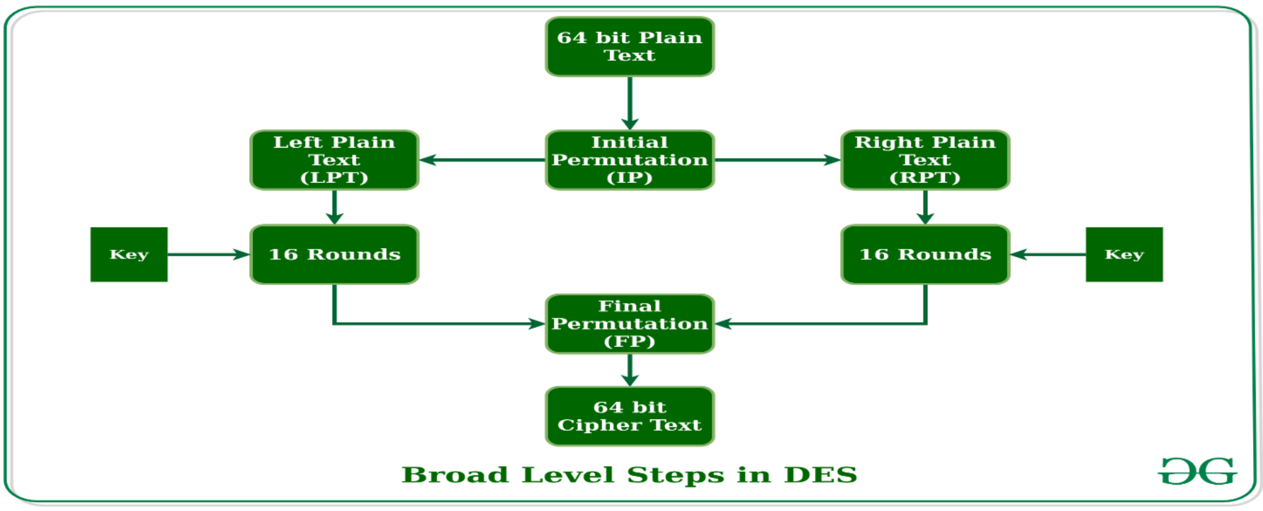
Thus, the discarding of every 8th bit of the key produces a **56-bit key** from the original **64-bit key**.
DES is based on the two fundamental attributes of cryptography: substitution (also called confusion) and transposition (also called diffusion). DES consists

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64

Figure - discording of every 8th bit of original key

of 16 steps, each of which is called a round. Each round performs the steps of substitution and transposition. Let us now discuss the broad-level steps in DES.

- In the first step, the 64-bit plain text block is handed over to an initial Permutation (IP) function.
- The initial permutation is performed on plain text.
- Next, the initial permutation (IP) produces two halves of the permuted block; saying Left Plain Text (LPT) and Right Plain Text (RPT).
- Now each LPT and RPT go through 16 rounds of the encryption process.
- In the end, LPT and RPT are rejoined and a Final Permutation (FP) is performed on the combined block
- The result of this process produces 64-bit cipher text.



Initial Permutation (IP)

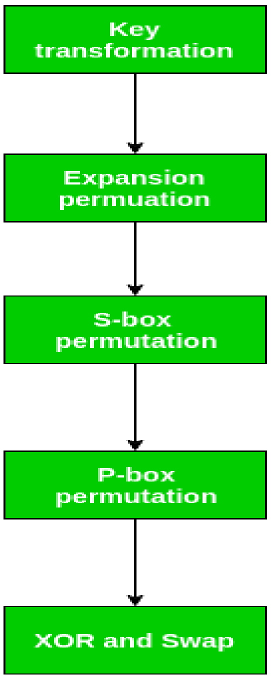
As we have noted, the initial permutation (IP) happens only once and it happens before the first round. It suggests how the transposition in IP should proceed, as shown in the figure. For example, it says that the IP replaces the first bit of the original plain text block with the 58th bit of the original plain text, the second bit with the 50th bit of the original plain text block, and so on.

This is nothing but jugglery of bit positions of the original plain text block. the same rule applies to all the other bit positions shown in the figure.

58	50	42	34	26	18	10	2	60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6	64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1	59	51	43	35	27	19	11	3
61	33	45	37	29	21	13	5	63	55	47	39	31	23	15	7

Figure - Initial permutation table

As we have noted after IP is done, the resulting 64-bit permuted text block is divided into two half blocks. Each half-block consists of 32 bits, and each of the 16 rounds, in turn, consists of the broad-level steps outlined in the figure.



Step 1: Key transformation
We have noted initial 64-bit key is transformed into a 56-bit key by discarding every 8th bit of the initial key. Thus, for each a 56-bit key is available. From this 56-bit key, a different 48-bit Sub Key is generated during each round using a process called key transformation. For this, the 56-bit key is divided into two halves, each of 28 bits. These halves are circularly shifted left by one or two positions, depending on the round.
For example: if the round numbers 1, 2, 9, or 16 the shift is done by only one position for other rounds, the circular shift is done by two positions. The number of key bits shifted per round is shown in the figure.

Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
#key bits shifted	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Figure - number of key bits shifted per round

After an appropriate shift, 48 of the 56 bits are selected. From the 48 we might obtain 64 or 56 bits based on requirement which helps us to recognize that this model is very versatile and can handle any range of requirements needed or provided. for selecting 48 of the 56 bits the table is shown in the figure given below. For instance, after the shift, bit number 14 moves to the first position, bit number 17 moves to the second position, and so on. If we observe the table, we will realize that it contains only 48-bit positions. Bit number 18 is discarded (we will not find it in the table), like 7 others, to reduce a 56-bit key to a 48-bit key. Since the key transformation process involves permutation as well as a selection of a 48-bit subset of the original 56-bit key it is called Compression Permutation.

14	17	11	24	1	5	3	28	15	6	21	10
23	19	12	4	26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40	51	45	33	48
44	49	39	56	34	53	46	42	50	36	29	32

Figure - compression permutation

Because of this compression permutation technique, a different subset of key bits is used in each round. That makes DES not easy to crack.

Step 2: Expansion Permutation

Recall that after the initial permutation, we had two 32-bit plain text areas called Left Plain Text (LPT) and Right Plain Text (RPT). During the expansion permutation, the RPT is expanded from 32 bits to 48 bits. Bits are permuted as well hence called expansion permutation. This happens as the 32-bit RPT is divided into 8 blocks, with each block consisting of 4 bits. Then, each 4-bit block of the previous step is then expanded to a corresponding 6-bit block, i.e., per 4-bit block, 2 more bits are added.

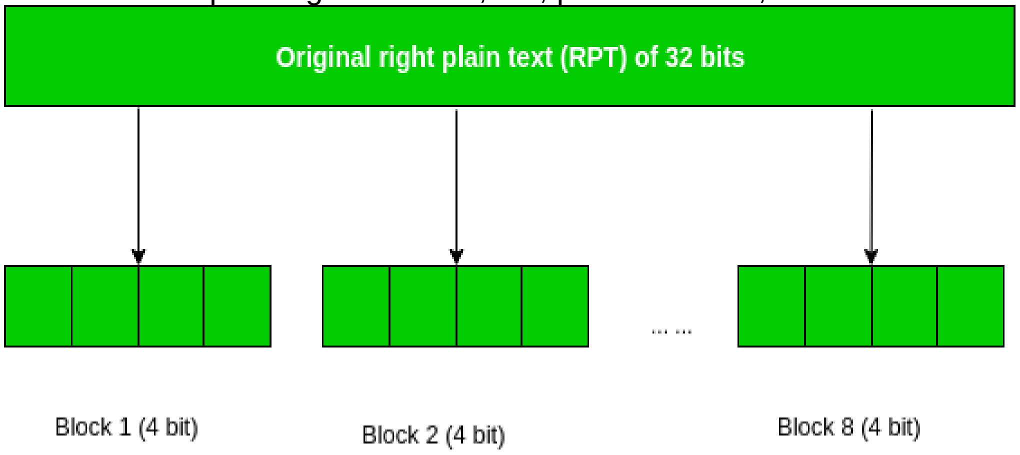


Figure - division of 32 bit RPT into 8 bit blocks

This process results in expansion as well as a permutation of the input bit while creating output. The key transformation process compresses the 56-bit key to 48 bits. Then the expansion permutation process expands the **32-bit RPT** to **48-bits**. Now the 48-bit key is XOR with 48-bit RPT and the resulting output is given to the next step, which is the **S-Box substitution**.

Python code for the encryption and decryption in des algorithm:
Hexadecimal to binary conversion

```
def hex2bin(s):
   mp = {'0': "0000",
        '1': "0001",
        '2': "0010",
        '3': "0011",
        '4': "0100",
        '5': "0101",
        '6': "0110",
        '7': "0111",
        '8': "1000",
        '9': "1001",
        'A': "1010",
        'B': "1011",
        'C': "1100",
        'D': "1101",
        'E': "1110",
        'F': "1111"}
   bin = ""
   for i in range(len(s)):
       bin = bin + mp[s[i]]
   return bin

# Binary to hexadecimal conversion


def bin2hex(s):
   mp = {"0000": '0',
        "0001": '1',
        "0010": '2',
        "0011": '3',
        "0100": '4',
        "0101": '5',
        "0110": '6',
        "0111": '7',
        "1000": '8',
        "1001": '9',
        "1010": 'A',
        "1011": 'B',
        "1100": 'C',
        "1101": 'D',
        "1110": 'E',
        "1111": 'F'}
   hex = ""
   for i in range(0, len(s), 4):
       ch = ""
       ch = ch + s[i]
       ch = ch + s[i + 1]
       ch = ch + s[i + 2]
       ch = ch + s[i + 3]
       hex = hex + mp[ch]

   return hex

# Binary to decimal conversion
```

```
def bin2dec(binary):

    binary1 = binary
    decimal, i, n = 0, 0, 0
    while(binary != 0):
        dec = binary % 10
        decimal = decimal + dec * pow(2, i)
        binary = binary//10
        i += 1
    return decimal

# Decimal to binary conversion


def dec2bin(num):
    res = bin(num).replace("0b", "")
    if(len(res) % 4 != 0):
        div = len(res) / 4
        div = int(div)
        counter = (4 * (div + 1)) - len(res)
        for i in range(0, counter):
            res = '0' + res
    return res

# Permute function to rearrange the bits


def permute(k, arr, n):
    permutation = ""
    for i in range(0, n):
        permutation = permutation + k[arr[i] - 1]
    return permutation

# shifting the bits towards left by nth shifts


def shift_left(k, nth_shifts):
    s = ""
    for i in range(nth_shifts):
        for j in range(1, len(k)):
            s = s + k[j]
        s = s + k[0]
        k = s
        s = ""
    return k

# calculating xow of two strings of binary number a and b


def xor(a, b):
    ans = ""
    for i in range(len(a)):
        if a[i] == b[i]:
            ans = ans + "0"
        else:
            ans = ans + "1"
    return ans


# Table of Position of 64 bits at initial level: Initial Permutation Table
initial_perm = [58, 50, 42, 34, 26, 18, 10, 2,
                60, 52, 44, 36, 28, 20, 12, 4,
                62, 54, 46, 38, 30, 22, 14, 6,
                64, 56, 48, 40, 32, 24, 16, 8,
                57, 49, 41, 33, 25, 17, 9, 1,
                59, 51, 43, 35, 27, 19, 11, 3,
                61, 53, 45, 37, 29, 21, 13, 5,
                63, 55, 47, 39, 31, 23, 15, 7]

# Expansion D-box Table
exp_d = [32, 1, 2, 3, 4, 5, 4, 5,
         6, 7, 8, 9, 8, 9, 10, 11,
         12, 13, 12, 13, 14, 15, 16, 17,
         16, 17, 18, 19, 20, 21, 20, 21,
         22, 23, 24, 25, 24, 25, 26, 27,
         28, 29, 28, 29, 30, 31, 32, 1]

# Straight Permutation Table
per = [16,  7, 20, 21,
       29, 12, 28, 17,
       1, 15, 23, 26,
       5, 18, 31, 10,
       2,  8, 24, 14,
       32, 27,  3,  9,
       19, 13, 30,  6,
       22, 11,  4, 25]

# S-box Table
sbox = [[[14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],
         [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],
         [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],
         [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]],

        [[15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],
         [3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],
         [0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],
         [13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9]],

        [[10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],
         [13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],
         [13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],
         [1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12]],

        [[7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],
         [13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],
         [10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],
         [3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14]],

        [[2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],
         [14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],
         [4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],
         [11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3]],

        [[12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],
         [10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],
         [9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],
         [4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13]],

        [[4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],
         [13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],
         [1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],
         [6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12]],

        [[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],
         [1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],
         [7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],
         [2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11]]]

# Final Permutation Table
final_perm = [40, 8, 48, 16, 56, 24, 64, 32,
              39, 7, 47, 15, 55, 23, 63, 31,
              38, 6, 46, 14, 54, 22, 62, 30,
              37, 5, 45, 13, 53, 21, 61, 29,
              36, 4, 44, 12, 52, 20, 60, 28,
              35, 3, 43, 11, 51, 19, 59, 27,
              34, 2, 42, 10, 50, 18, 58, 26,
              33, 1, 41, 9, 49, 17, 57, 25]


def encrypt(pt, rkb, rk):
    pt = hex2bin(pt)

    # Initial Permutation
    pt = permute(pt, initial_perm, 64)
    print("After initial permutation", bin2hex(pt))

    # Splitting
    left = pt[0:32]
    right = pt[32:64]
    for i in range(0, 16):
        #  Expansion D-box: Expanding the 32 bits data into 48 bits
        right_expanded = permute(right, exp_d, 48)

        # XOR RoundKey[i] and right_expanded
        xor_x = xor(right_expanded, rkb[i])

        # S-boxex: substituting the value from s-box table by calculating row and
column
        sbox_str = ""
        for j in range(0, 8):
            row = bin2dec(int(xor_x[j * 6] + xor_x[j * 6 + 5]))
            col = bin2dec(
                int(xor_x[j * 6 + 1] + xor_x[j * 6 + 2] + xor_x[j * 6 + 3] + xor_x[j * 6 + 4]))
            val = sbox[j][row][col]
            sbox_str = sbox_str + dec2bin(val)

        # Straight D-box: After substituting rearranging the bits
        sbox_str = permute(sbox_str, per, 32)

        # XOR left and sbox_str
        result = xor(left, sbox_str)
        left = result

        # Swapper
        if(i != 15):
            left, right = right, left
        print("Round ", i + 1, " ", bin2hex(left),
              " ", bin2hex(right), " ", rk[i])

    # Combination
    combine = left + right

    # Final permutation: final rearranging of bits to get cipher text
    cipher_text = permute(combine, final_perm, 64)
    return cipher_text


pt = "123456ABCD132536"
key = "AABB09182736CCDD"

# Key generation
# --hex to binary
key = hex2bin(key)

# --parity bit drop table
keyp = [57, 49, 41, 33, 25, 17, 9,
        1, 58, 50, 42, 34, 26, 18,
        10, 2, 59, 51, 43, 35, 27,
        19, 11, 3, 60, 52, 44, 36,
        63, 55, 47, 39, 31, 23, 15,
        7, 62, 54, 46, 38, 30, 22,
        14, 6, 61, 53, 45, 37, 29,
        21, 13, 5, 28, 20, 12, 4]

# getting 56 bit key from 64 bit using the parity bits
key = permute(key, keyp, 56)

# Number of bit shifts
shift_table = [1, 1, 2, 2,
               2, 2, 2, 2,
               1, 2, 2, 2,
               2, 2, 2, 1]

# Key- Compression Table : Compression of key from 56 bits to 48 bits
key_comp = [14, 17, 11, 24, 1, 5,
            3, 28, 15, 6, 21, 10,
            23, 19, 12, 4, 26, 8,
            16, 7, 27, 20, 13, 2,
            41, 52, 31, 37, 47, 55,
            30, 40, 51, 45, 33, 48,
            44, 49, 39, 56, 34, 53,
            46, 42, 50, 36, 29, 32]

# Splitting
left = key[0:28]    # rkb for RoundKeys in binary
right = key[28:56]  # rk for RoundKeys in hexadecimal

rkb = []
rk = []
for i in range(0, 16):
    # Shifting the bits by nth shifts by checking from shift table
    left = shift_left(left, shift_table[i])
    right = shift_left(right, shift_table[i])

    # Combination of left and right string
    combine_str = left + right

    # Compression of key from 56 to 48 bits
    round_key = permute(combine_str, key_comp, 48)

    rkb.append(round_key)
    rk.append(bin2hex(round_key))

print("Encryption")
cipher_text = bin2hex(encrypt(pt, rkb, rk))
print("Cipher Text : ", cipher_text)

print("Decryption")
rkb_rev = rkb[::-1]
rk_rev = rk[::-1]
```

```
text = bin2hex(encrypt(cipher_text, rkb_rev, rk_rev))
print("Plain Text : ", text)
```

Output:

```
PS C:\Users\Sashu Akshu> & "C:/Program Files/Python312/python.exe"
"c:/Users/Sashu Akshu/Desktop/myproject/myproject/sample.py"
Encryption
After initial permutation 14A7D67818CA18AD
Round 1   18CA18AD   5A78E394   194CD072DE8C
Round 2   5A78E394   4A1210F6   4568581ABCCE
Round 3   4A1210F6   B8089591   06EDA4ACF5B5
Round 4   B8089591   236779C2   DA2D032B6EE3
Round 5   236779C2   A15A4B87   69A629FEC913
Round 6   A15A4B87   2E8F9C65   C1948E87475E
Round 7   2E8F9C65   A9FC20A3   708AD2DDB3C0
Round 8   A9FC20A3   308BEE97   34F822F0C66D
Round 9   308BEE97   10AF9D37   84BB4473DCCC
Round 10   10AF9D37   6CA6CB20   02765708B5BF
Round 11   6CA6CB20   FF3C485F   6D5560AF7CA5
Round 12   FF3C485F   22A5963B   C2C1E96A4BF3
Round 13   22A5963B   387CCDAA   99C31397C91F
Round 14   387CCDAA   BD2DD2AB   251B8BC717D0
Round 15   BD2DD2AB   CF26B472   3330C5D9A36D
Round 16   19BA9212   CF26B472   181C5D75C66D
Cipher Text:  C0B7A8D05F3A829C
Decryption
After initial permutation 19BA9212CF26B472
Round 1   CF26B472   BD2DD2AB   181C5D75C66D
Round 2   BD2DD2AB   387CCDAA   3330C5D9A36D
Round 3   387CCDAA   22A5963B   251B8BC717D0
Round 4   22A5963B   FF3C485F   99C31397C91F
Round 5   FF3C485F   6CA6CB20   C2C1E96A4BF3
Round 6   6CA6CB20   10AF9D37   6D5560AF7CA5
Round 7   10AF9D37   308BEE97   02765708B5BF
Round 8   308BEE97   A9FC20A3   84BB4473DCCC
Round 9   A9FC20A3   2E8F9C65   34F822F0C66D
Round 10   2E8F9C65   A15A4B87   708AD2DDB3C0
Round 11   A15A4B87   236779C2   C1948E87475E
Round 12   236779C2   B8089591   69A629FEC913
Round 13   B8089591   4A1210F6   DA2D032B6EE3
Round 14   4A1210F6   5A78E394   06EDA4ACF5B5
Round 15   5A78E394   18CA18AD   4568581ABCCE
Round 16   14A7D678   18CA18AD   194CD072DE8C
Plain Text:  123456ABCD132536
```

In conclusion, the Data Encryption Standard (DES) is a block cipher with a 56-bit key length that has played a significant role in data security. However, due to vulnerabilities, its popularity has declined. DES operates through a series of rounds involving key transformation, expansion permutation, and substitution, ultimately producing cipher text from plaintext. While DES has historical significance, it’s crucial to consider more secure encryption alternatives for modern data protection needs.

Rivest-Shamir-Adleman (RSA) Algorithm

RSA is a public-key (or asymmetric) cryptosystem that is widely used for secure data transmission. It was invented in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman of the Massachusetts Institute of Technology. RSA's security is based on the difficulty of factoring large composite numbers.

Key Characteristics of RSA:

- **Public Key Encryption:** RSA allows two parties to communicate securely without having to share a secret key. Each party has a pair of keys: a public key (used for encryption) and a private key (used for decryption).
- **Asymmetric Cryptography:** It uses different keys for encryption and decryption. The encryption key is public, and the decryption key is private.
- **Mathematical Security:** RSA's security is based on the fact that it is very difficult to factor the product of two large prime numbers, a problem that is considered computationally infeasible for large keys.
- **Key Generation:** RSA keys are generated by choosing two large prime numbers, multiplying them to form a composite number, and then calculating the totient of this number to create the keys.

RSA algorithm is an asymmetric cryptography algorithm. Asymmetric actually means that it works on two different keys i.e. **Public Key** and **Private Key.** As the name describes that the Public Key is given to everyone and the Private Key is kept private.

An example of asymmetric cryptography:

1. A client (for example browser) sends its public key to the server and requests some data.
2. The server encrypts the data using the client's public key and sends the encrypted data.
3. The client receives this data and decrypts it.

Since this is asymmetric, nobody else except the browser can decrypt the data even if a third party has the public key of the browser.

The idea! The idea of RSA is based on the fact that it is difficult to factorize a large integer. The public key consists of two numbers where one number is a multiplication of two large prime numbers. And private key is also derived from the same two prime numbers. So if somebody can factorize the large number, the private key is compromised. Therefore encryption strength totally lies on the key size and if we double or triple the key size, the strength of encryption increases exponentially. RSA keys can be typically 1024 or 2048 bits long, but experts believe that 1024-bit keys could be broken in the near future. But till now it seems to be an infeasible task.

Let us learn the mechanism behind the RSA algorithm : >> Generating Public Key:

Select two prime no's. Suppose **P = 53 and Q = 59.**
Now First part of the Public key : n = P*Q = 3127.
We also need a small exponent say **e :**
But e Must be
An integer.
Not be a factor of Φ(n).
1 < e < Φ(n) [Φ(n) is discussed below],

Let us now consider it to be equal to 3.
Our Public Key is made of n and e

>> Generating Private Key:
We need to calculate Φ(n) :
Such that **Φ(n) = (P-1)(Q-1)**
so, Φ(n) = 3016
Now calculate Private Key, **d :**
d = (k*Φ(n) + 1) / e for some integer k
For k = 2, value of d is 2011.

Now we are ready with our – Public Key (n = 3127 and e = 3) and Private Key(d = 2011) Now we will encrypt **"HI"**:
Convert letters to numbers : H = 8 and I = 9
Thus **Encrypted Data c = (89e)mod n**
Thus our Encrypted Data comes out to be 1394
Now we will decrypt **1394** :
Decrypted Data = (cd)mod n
Thus our Encrypted Data comes out to be 89
8 = H and I = 9 i.e. "HI".

The implementation of the RSA algorithm for Encrypting and decrypting using python

```
import random
import math

# A set will be the collection of prime numbers,
# where we can select random primes p and q
prime = set()

public_key = None
private_key = None
n = None

# We will run the function only once to fill the set of
# prime numbers
def primefiller():
    # Method used to fill the primes set is Sieve of
    # Eratosthenes (a method to collect prime numbers)
    seive = [True] * 250
    seive[0] = False
    seive[1] = False
    for i in range(2, 250):
        for j in range(i * 2, 250, i):
            seive[j] = False

    # Filling the prime numbers
    for i in range(len(seive)):
        if seive[i]:
            prime.add(i)
```

```python
# Picking a random prime number and erasing that prime
# number from list because p!=q
def pickrandomprime():
    global prime
    k = random.randint(0, len(prime) - 1)
    it = iter(prime)
    for _ in range(k):
        next(it)

    ret = next(it)
    prime.remove(ret)
    return ret


def setkeys():
    global public_key, private_key, n
    prime1 = pickrandomprime()  # First prime number
    prime2 = pickrandomprime()  # Second prime number

    n = prime1 * prime2
    fi = (prime1 - 1) * (prime2 - 1)

    e = 2
    while True:
        if math.gcd(e, fi) == 1:
            break
        e += 1

    # d = (k*Φ(n) + 1) / e for some integer k
    public_key = e

    d = 2
    while True:
        if (d * e) % fi == 1:
            break
        d += 1

    private_key = d


# To encrypt the given number
def encrypt(message):
    global public_key, n
    e = public_key
    encrypted_text = 1
    while e > 0:
        encrypted_text *= message
        encrypted_text %= n
        e -= 1
    return encrypted_text


# To decrypt the given number
def decrypt(encrypted_text):
    global private_key, n
    d = private_key
    decrypted = 1
    while d > 0:
        decrypted *= encrypted_text
        decrypted %= n
        d -= 1
    return decrypted


# First converting each character to its ASCII value and
# then encoding it then decoding the number to get the
# ASCII and converting it to character
def encoder(message):
    encoded = []
    # Calling the encrypting function in encoding function
    for letter in message:
        encoded.append(encrypt(ord(letter)))
    return encoded


def decoder(encoded):
    s = ''
    # Calling the decrypting function decoding function
    for num in encoded:
        s += chr(decrypt(num))
    return s


if __name__ == '__main__':
    primefiller()
    setkeys()
    message = "jntu sessions"
    # Uncomment below for manual input
    # message = input("Enter the message\n")
    # Calling the encoding function
    coded = encoder(message)

    print("Initial message:")
    print(message)
    print("\n\nThe encoded message(encrypted by public key)\n")
    print(''.join(str(p) for p in coded))
    print("\n\nThe decoded message(decrypted by public key)\n")
    print(''.join(str(p) for p in decoder(coded)))
```

Output:

```
 S C:\Users\Sashu Akshu> & "C:/Program Files/Python312/python.exe"
"c:/Users/Sashu Akshu/Desktop/myproject/myproject/rsa algorithm.py"
Initial message:
jntu sessions


The encoded message(encrypted by public key)

24438159162487714810145637264785726472649061025115916726 4


The decoded message(decrypted by public key)

jntu sessions
```

Q2. Explain Diffie-Hellman Key Exchange Algorithm with an Example

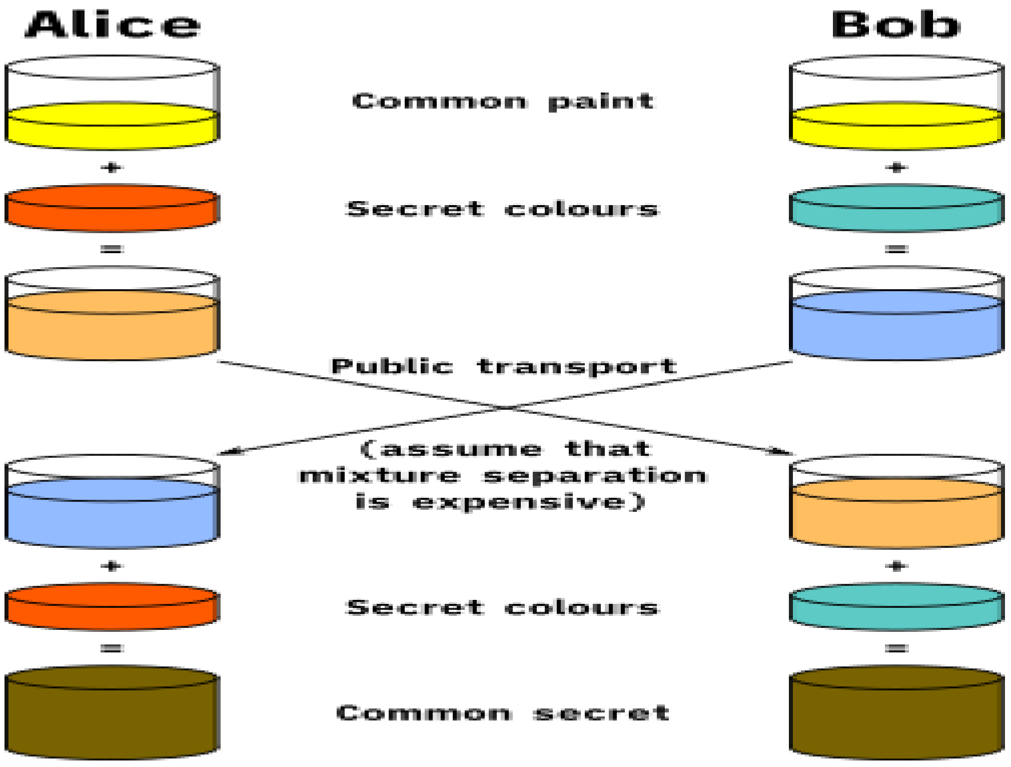
The Diffie–Hellman (DH) Algorithm is a key-exchange protocol that enables two parties communicating over public channel to establish a mutual secret without it being transmitted over the Internet. DH enables the two to use a public key to encrypt and decrypt their conversation or data using symmetric cryptography.

Diffie and Hellman wanted to make Transport Layer Security (TLS), a secure way of computers communicating, more safe to perform. For example, while you can use a password to keep a file safe, if you need to tell the password to somebody there is a risk of the password being seen by third parties. Diffie-Hellman key agreement itself is an *anonymous* (non-*authenticated*) key-agreement protocol: people involved in the trade do not need to prove who they are, but both people need to use their secret keys to fully decrypt the data.

Diffie-Helman is generally explained by two sample parties, Alice and Bob, initiating a dialogue. Each has a piece of information they want to share, while preserving its secrecy. To do that they agree on a public piece of benign information that will be mixed with their privileged information as it travels over an insecure channel. Their secrets are mixed with the public information, or public key, and as the secrets are exchanged the information they want to share is commingled with the common secret. As they decipher the other’s message, they can extract the public information and with knowledge of their own secret, deduce the new information that was carried along. While seemingly uncomplicated in this method’s description, when long number strings are used for private and public keys, decryption by an outside party trying to eavesdrop is mathematically infeasible even with considerable resources.

DH is one of the first practical implementations of asymmetric encryption or public-key cryptography (PKC). It was published in 1976 by Whitfield Diffie and Martin

Hellman. Other contributors who are credited with developing DH include Ralph Merkle and researchers within the United Kingdom's intelligence services (c. 1969).



Example:

Step 1: Alice and Bob get public numbers P = 23, G = 9

Step 2: Alice selected a private key a = 4 and
        Bob selected a private key b = 3

Step 3: Alice and Bob compute public values
Alice: x =(9^4 mod 23) = (6561 mod 23) = 6
        Bob: y = (9^3 mod 23) = (729 mod 23) = 16

Step 4: Alice and Bob exchange public numbers

Step 5: Alice receives public key y =16 and
        Bob receives public key x = 6

Step 6: Alice and Bob compute symmetric keys
        Alice: ka = y^a mod p = 65536 mod 23 = 9
        Bob: kb = x^b mod p = 216 mod 23 = 9

Step 7: 9 is the shared secret.

Implementation: # Diffie-Hellman Code in python

```
def prime_checker(p):

    # Checks If the number entered is a Prime Number or not
    if p < 1:
        return -1
    elif p > 1:
        if p == 2:
            return 1
        for i in range(2, p):
            if p % i == 0:
                return -1
            return 1


def primitive_check(g, p, L):
    # Checks If The Entered Number Is A Primitive Root Or Not
    for i in range(1, p):
        L.append(pow(g, i) % p)
    for i in range(1, p):
        if L.count(i) > 1:
            L.clear()
            return -1
        return 1


l = []
while 1:
    P = int(input("Enter P : "))
    if prime_checker(P) == -1:
        print("Number Is Not Prime, Please Enter Again!")
        continue
    break

while 1:
    G = int(input(f"Enter The Primitive Root Of {P} : "))
    if primitive_check(G, P, l) == -1:
        print(f"Number Is Not A Primitive Root Of {P}, Please Try Again!")
        continue
    break

# Private Keys
x1, x2 = int(input("Enter The Private Key Of User 1 : ")), int(
    input("Enter The Private Key Of User 2 : "))
while 1:
    if x1 >= P or x2 >= P:
        print(f"Private Key Of Both The Users Should Be Less Than {P}!")
        continue
    break

# Calculate Public Keys
y1, y2 = pow(G, x1) % P, pow(G, x2) % P

# Generate Secret Keys
```

```
k1, k2 = pow(y2, x1) % P, pow(y1, x2) % P


print(f"\nSecret Key For User 1 Is {k1}\nSecret Key For User 2 Is {k2}\n")


if k1 == k2:
    print("Keys Have Been Exchanged Successfully")
else:
    print("Keys Have Not Been Exchanged Successfully")
```

Output:

```
PS C:\Users\Sashu Akshu> & "C:/Program Files/Python312/python.exe"
"c:/Users/Sashu Akshu/Desktop/myproject/myproject/# Diffie-Hellman Code.py"

Enter P : 40

Number Is Not Prime, Please Enter Again!

Enter P : 11

Enter The Primitive Root Of 11: 7

Enter The Private Key Of User 1: 5

Enter The Private Key Of User 2 : 3


Secret Key For User 1 Is 10

Secret Key For User 2 Is 10


Keys Have Been Exchanged Successfully
```

Q3. Explain Digital Signature Algorithm (DSA) With an Example.

A Digital Signature is a verification method made by the recipient to ensure the message was sent from the authenticated identity. When a customer signs a check, the bank must verify that he issued that specific check. In this case, a signature on a document acts as a sign of authentication and verifies that the document is authentic. Suppose we have:

- **Alice** is the entity that sends a message or initiates communication.
- **Bob** represents the recipient or receiver of the message.
- **Eve** represents an eavesdropper or adversary who may attempt to intercept or tamper with the communication.

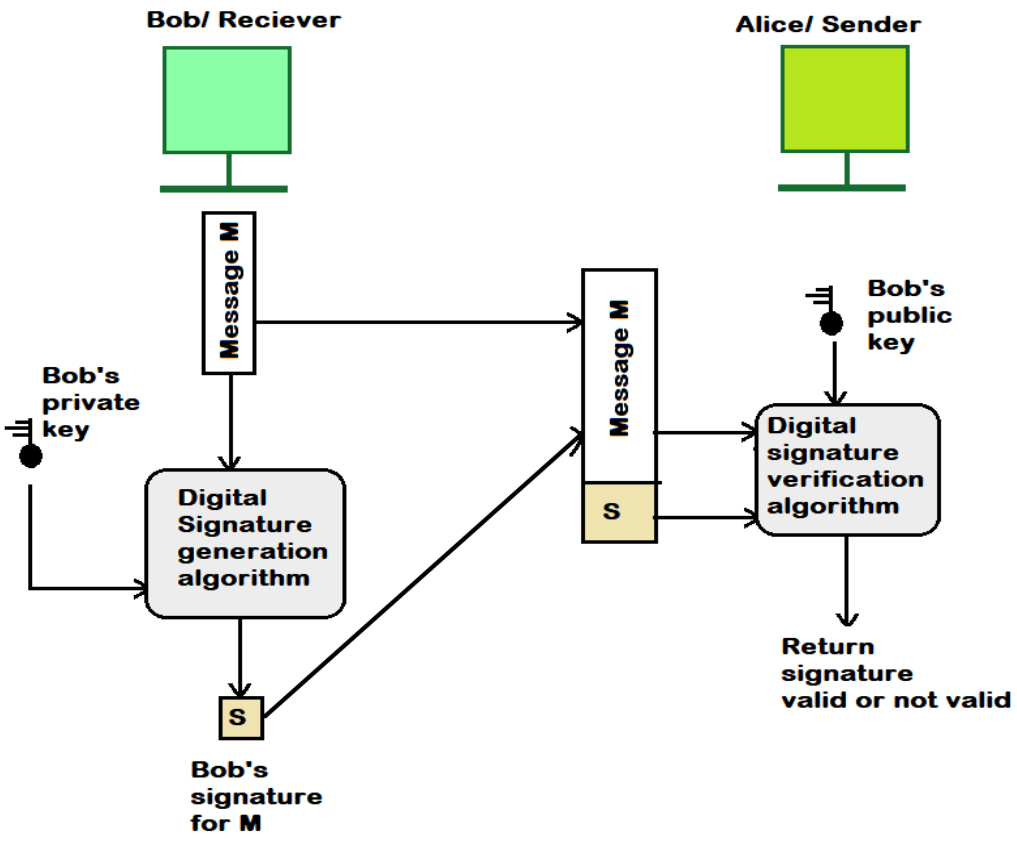
In Public Key cryptography (also known as Asymmetric cryptography), the communication process is as follows:

- Alice encrypts the message using Bob's public key.
- The encrypted message reaches Bob.
- Bob decrypts the message sent by Alice using his private key.

Now, suppose when Alice sends a message to Bob, then Bob will check if the sender is authentic; to ensure that it was Alice who sent the message, not Eve. For this, Bob can ask Alice to sign the message electronically. So we can say that an electronic signature can prove that Alice is authentic and is the one sending the message. We called this type of signature a **digital signature.**

What is Digital Signature?

Digital Signature is a verification method. Digital signatures do not provide confidential communication. If you want to achieve confidentiality, both the message and the signature must be encrypted using either a secret key or a public key cryptosystem. This additional layer of security can be incorporated into a basic digital signature scheme.


Bob/ Reciever
Alice/ Sender
Message M
Bob's private key
Digital Signature generation algorithm
S
Bob's signature for M
Message M
S
Bob's public key
Digital signature verification algorithm
Return signature valid or not valid

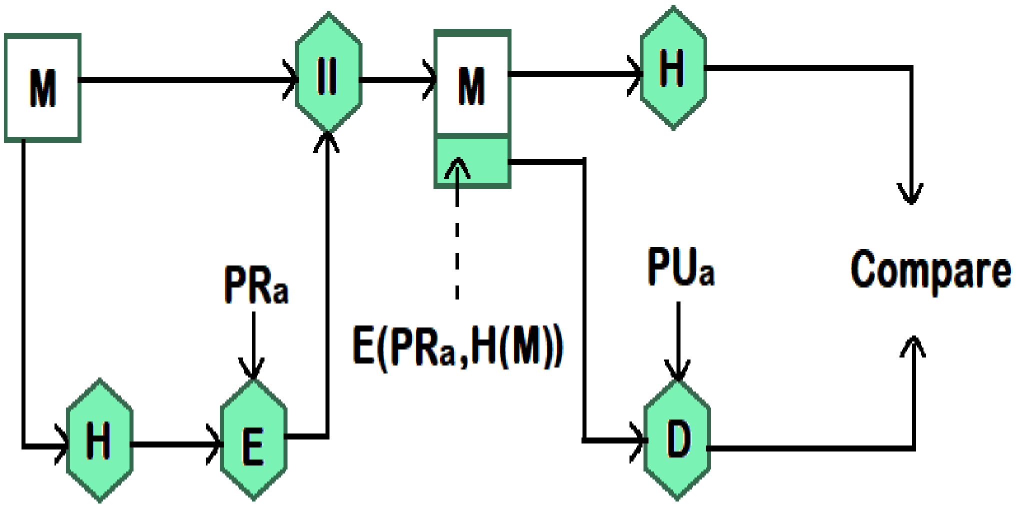
Model of Digital Signature Process

Methods of Digital Signature
These two are standard Approaches to implement the Digital Signature:

- Rivest-Shamir-Adleman (RSA)
- Digital Signature Algorithm (DSA)

Rivest-Shamir-Adleman (RSA)
In the RSA approach, the message that needs to be signed is first fed into a hash function that generates a secure hash code of fixed length. The sender’s private key is then used to encrypt the hash code which makes it signature. The next step involves sending both the signature and the message to the intended receiver. For validation purposes, after receiving the message, the recipient first computes its hash-code. The sender’s public key is applied by recipient to decrypt this already encrypted signature. In case if decrypted signature corresponds to recipient-produced hash code that means that signature would be considered as valid. Since only the sender has access to the private key, only they could have produced a valid signature.
You can refer the below diagram for RSA, here,

- M = Message or Plaintext
- H = Hash Function
- || = bundle the plantext and hash function (hash digest)
- E = Encryption Algorithm
- D = Decryption Algorithm
- PUa = Public key of sender
- PRa = Private key of sender

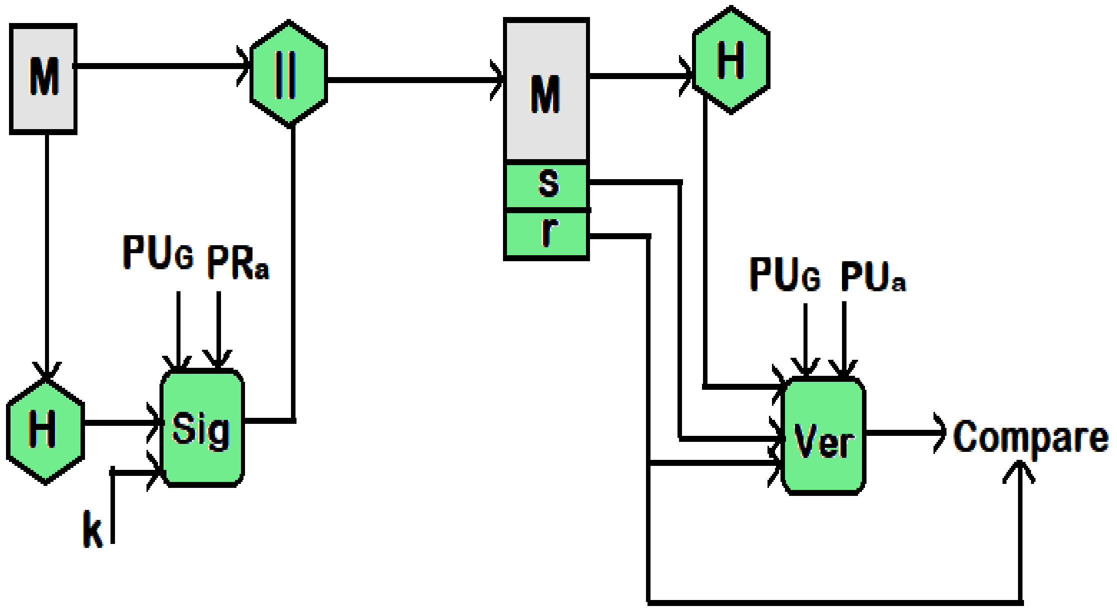


RSA approach

Digital Signature Algorithm (DSA)
The DSA (Digital Signature Algorithm) approach involves using of a hash function to create a hash code, same as RSA. This hash code is combined with a randomly generated number k as an input to a signature function. The signature function depends on the sender’s private key (PRa) as well as a set of parameters that are known to a group of communicating principals. This set can be considered as a global public key (PUG). The output of the signature function is a signature with two

components, s and r. When an incoming message is received, a hash code is generated for the message. This hash code is then combined with the signature and input into a verification function. The verification function depends on the global public key as well as the sender's public key (PUa) which is paired with the sender's private key. The output of the verification function returns a value equal to the signature's component r, if the signature is valid. The signature function is designed in such a way that only the sender, with knowledge of the private key, can produce a valid signature.
You can refer below diagram for DSA, where,

- M = Message or Plaintext
- H = Hash Function
- || = bundle the plantext and hash function (hash digest)
- E = Encryption Algorithm
- D = Decryption Algorithm
- PUa = Public key of sender
- PRa = Private key of sender
- Sig = Signature function
- Ver = Verification function
- PUG = Global public Key



DSA Approach

Primary Termologies

- **User's Private Key (PR):** This key is publicly known and can be shared with anyone. It's used to verify digital signatures created with a corresponding private key.
- **User's Public Key (PU):** A top-secret cryptographic key only possessed by the user is used in DSA algorithm's digital signature generation. As it is, the private key must be kept secret and secure because it proves that a given user is genuine.
- **Signing (Sig):** Signing involves creating a digital signature with the help of a user's private key. In case of DSA, this process requires mathematical operations

to be performed on the message that should be signed using a given private key in order to generate a unique signature for that message.

- **Verifying (Ver):** Verifying is the process of verifying whether or not a digital signature has been forged using its corresponding public key. In DSA, this involves comparing the messages hash against the verification value through mathematical operations between two binary strings – one representing an encrypted data and another one representing plain-text original message.

Steps to Perform DSA

The Digital Signature Algorithm (DSA) is a public-key technique (i.e., assymetric cryptography) and it is used to provide only the digital signature function, and it cannot be used for encryption or key exchange.

The Steps to perform the Digital Signature Algorithm can be broadly divided into:

- Global Public-Key Components
- User’s Private Key
- User’s Public Key
- Signing
- Verifying

1. Global Public-Key Components

There are three parameters that are public and can be shared to a set of users.

- A prime number **p** is chosen with a length between 512 and 1024 bits such that q divides (p – 1). So, p is prime number where $2L-1 < p <2L$ for $512<= L<=1024$ and L is a multiple of 64; i.e., bit length of between 512 and 1024 bits in increments of 64 bits.
- Next, an N-bit prime number **q** is selected. So, q is prime divisor of (p – 1), where $2N-1 < q < 2N$ i.e., bit length of N bits.
- Finally, g is selected to be of the form h(p-1)/q mod p, where h is an integer between 1 and (p – 1) with the limitation that g must be greater than 1. So, g is = h(p – 1)/q mod p, where h is any integer with $1 < h < (p – 1)$ such that h(p-1)/q mod p > 1.

If a user has these numbers, then it can selects a private key and generates a public key.

2. User’s Private Key

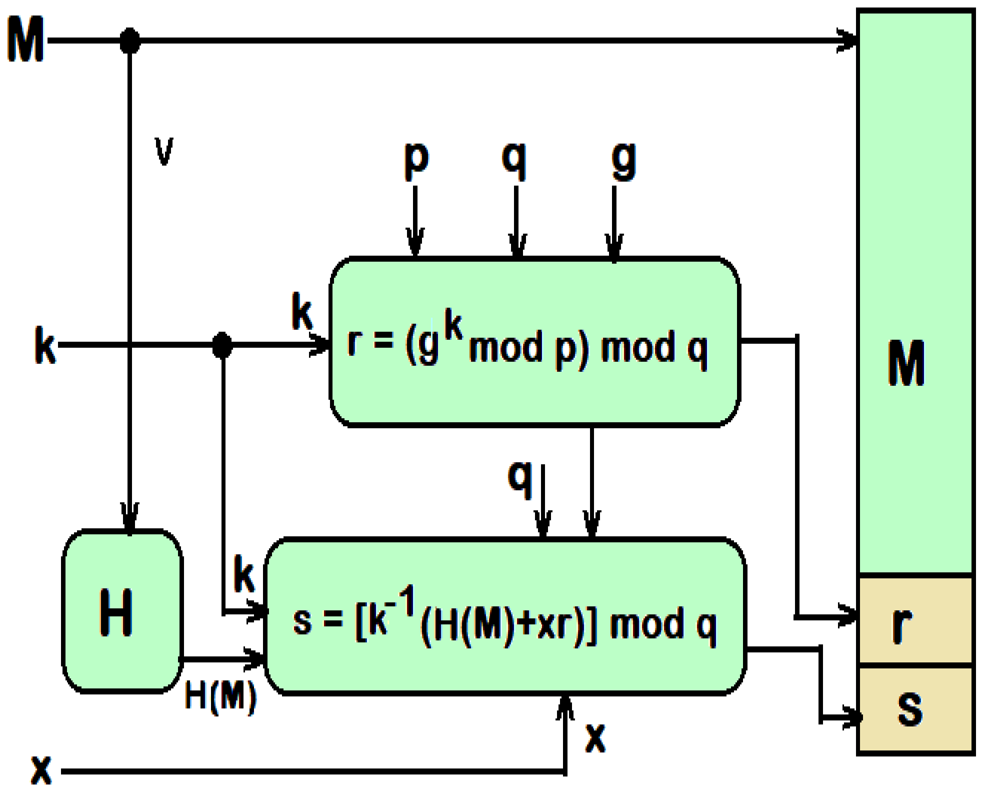
The private key x should be chosen randomly or pseudorandomly and it must be a number from 1 to (q – 1), so x is random or pseudorandom integer with $0 < x < q$.

3. User’s Public Key

The public key is computed from the private key as y = gx mod p. The computation of y given x is simple. But, given the public key y, it is believed to be computationally infeasible to choose x, which is the discrete logarithm of y to the base g, mod p.

4. Signing

If a user want to develop a signature, a user needs to calculates two quantities, r and s, that are functions of the public key components (p, q, g), the hash code of the message H(M, the user’s private key (x), and an integer k that must be generated randomly or pseudo randomly and be unique for each signing. k is generated randomly or pseudo randomly integer such that $0<k < q$.



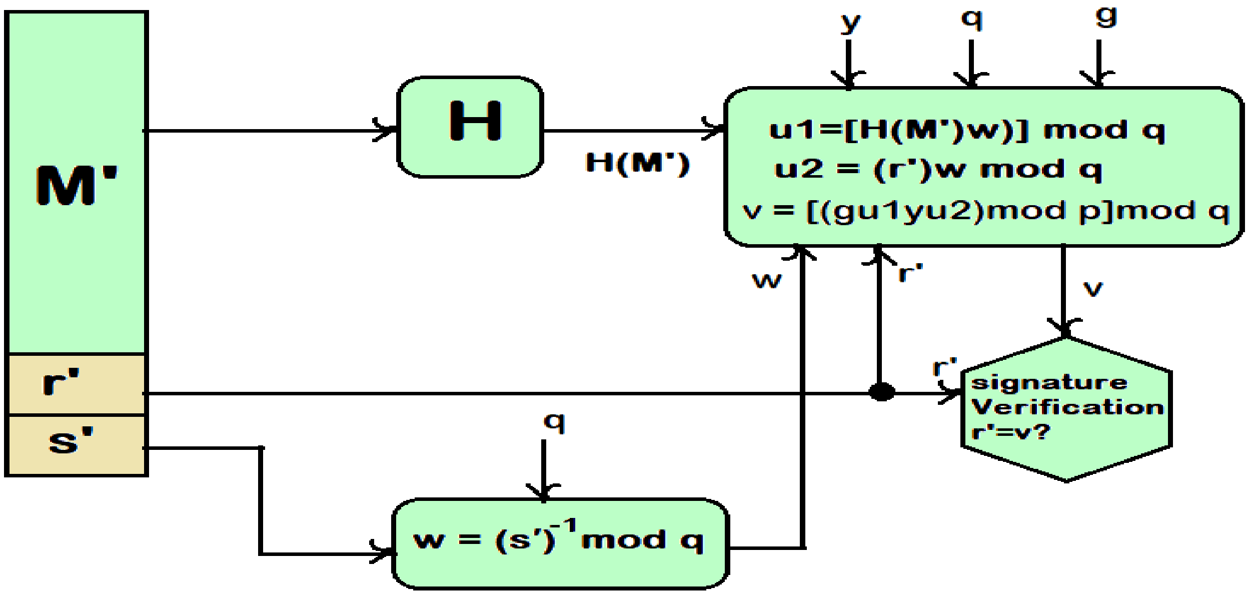
Signing

5. Verification

Let M, r′, and s′ be the received versions of M, r, and s, respectively.
Verification is performed using the formulas shown in below:

- w = (s′)-1 mod q
- u1 = [H(M′)w] mod q
- u2 = (r′)w mod q
- v = [(gu1 yu2) mod p] mod q

The receiver needs to generate a quantity v that is a function of the public key components, the sender's public key, and the hash code of the message. If this value matches the r value of the signature, then the signature is considered as valid.
TEST: v = r′



Verification

Now, at the end it will test on the value r, and it does not depend on the message or plaintext as, r is the function of k and the three global public-key components as mentioned above. The multiplicative inverse of k (mod q) when passed to the function that also has as inputs the message hash code and the user's private key. The structure of this function is such that the receiver can recover r using the incoming message and signature, the public key of the user, and the global public key.
It is given that there is difficulty in taking discrete logarithms, it is not feasible for an attacker to recover k from r or to recover x from s. The only computationally demanding task in signature generation is the exponential calculation gk mod p. Because this value does not depend on the message to be signed, it can be computed ahead of time. Indeed, a user could precalculate a number of values of r to be used to sign documents as needed. The only other somewhat demanding task is the determination of a multiplicative inverse, k-1.

Services

- **Message Authentication:** A secure digital signature scheme, like a secure conventional signature (one that cannot be easily copied) can provide message authentication (also referred to as data-origin authentication). Bob can easily confirm that the plaintext/message is sent by Alice as Alice's public key is used for verification and the Alice's public key woult not verify the signature signed by Eve's private key. Hence, A digital signature provides message authentication.
- **Message Integrity:** When we sign a whole message, its integrity remains intact because if the message changes, we won't get the same signature. Nowadays, digital signature methods use a special function called a hash function in both signing and verifying to ensure the message's integrity.
- **Nonrepudiation:** If Alice signs a message and later claims she didn't, can Bob provide evidence that she did? For example, if Alice instructs a bank (Bob) to transfer $10,000 to Ted's account and then denies sending the message, Bob needs to keep the signed message and use Alice's public key to recreate it. However, this approach may not work if Alice changes her keys or disputes the

authenticity of the file. A solution is involving a trusted third party. This trusted party can authenticate messages and prevent Alice from denying them. In this setup, Alice sends her message along with her identity, Bob's identity, and her signature to the trusted center. The center verifies the message's authenticity and timestamps it before creating its own signature. This process ensures that if Alice denies sending the message later, the center can provide evidence to settle the dispute. Encryption can also be added for confidentiality. Thus, nonrepudiation is achievable through a trusted party.

Advantages of DSA

Authentication: At some point, digital signatures ensure strong identity authentication for the sender. The recipient can be sure that the message or document was signed by the purported signatory.

- **Integrity**: Digital signatures ensure the integrity of the content. If something is altered in the content after the signature is made, then it becomes invalid with respect to verifying the content.
- **Non-Repudiation:** A digital signature gives non-repudiation, meaning the sender cannot disclaim his creation of that document post factum. Most relevant in legal and contractual issues.
- **Efficiency:** Digital signatures make the process of signing electronic and automate it, giving way to fast online transactions free from the need of manual verification, paperwork, and a physical signature.
- **Security:** As long as the whole digital signing process is well organized, digital signatures may prove to be secure. Cryptographic public key cryptography and hashing algorithms prevent unauthorized parties from forging digital signatures.
- **World Acceptance:** Such a mechanism (digital signatures) to represent the conclusion of the related transaction in case of legal or contractual terms is known and widely accepted all over the world.
- **Timestamping:** Timestamping would also make another secure layer against replay attacks and against the freshness of the signature.
- **Cost Savings:** The digital signing process discontinues the need for transporting documents, thereby saving on costs to be done with printing, courier services, and manual handling.

Disadvantages of DSA

- **Key Management Complexity:** Cryptographic keys that are used for signing documents must be properly managed. Generating, storing, and distributing keys in a secure manner are all complicated procedures that need to be attended to, and revocation has to be handled carefully.
- **Infrastructure Dependence:** Digital signatures are built on a secure and reliable infrastructure of Public Key Infrastructure (PKI) and Certificate Authorities. If the infrastructure is compromised or becomes unavailable, it may compromise trust in digital signatures.
- **Legal and Regulatory Challenges:** Although many people are increasingly using digital signatures, there might still be legal and regulatory challenges in some places. It will be very important to observe local laws and standards.
- **Initial Setup Costs:** A proper setup of an extensive digital signature system may include the cost of obtaining certificates for digital certificates, putting in place safety measures, and training of the users.
- **Offline Usability:** In the event of not having access to the signer's private key, digital signatures are found to be challenged. Solutions of hardware tokens and secure elements add to the complexity.

- **User Education:** Education of the proper application and value of digital signatures is necessary in order that the users should be educated in use. The correct measures to be taken against vulnerability, as well as being aware of any possible threat, are important in successful implementation.
- **Vulnerability to Key Compromise:** Private keys need to be safeguarded from unauthorized access since one compromised private key can initiate fraudulent signatures.

Q4. Explain the Following Types of One-time Password (OTP) Algorithms with Examples:
a . Time-based OTP (TOTP)
b. HMAC-based OTP (HOTP)

One Time Password (OTP) algorithm in Cryptography

Authentication, the process of identifying and validating an individual is the rudimentary step before granting access to any protected service (such as a personal account). Authentication has been built into the cyber security standards and offers to prevent unauthorized access to safeguarded resources. Authentication mechanisms today create a double layer gateway prior to unlocking any protected information. This double layer of security, termed as two factor authentication, creates a pathway that requires validation of credentials (username/email and password) followed by creation and validation of the **One Time Password (OTP)**. The OTP is a numeric code that is randomly and uniquely generated during each authentication event. This adds an additional layer of security, as the password generated is fresh set of digits each time an authentication is attempted and it offers the quality of being unpredictable for the next created session. The two main methods for delivery of the OTP is:

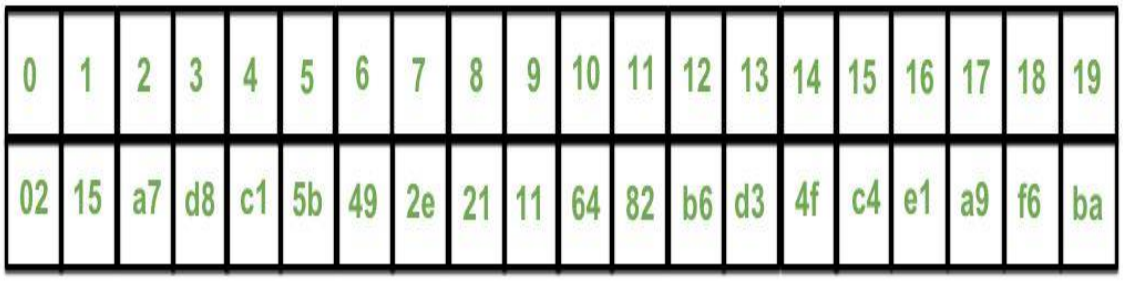
1. **SMS Based:** This is quite straightforward. It is the standard procedure for delivering the OTP via a text message after regular authentication is successful. Here, the OTP is generated on the server side and delivered to the authenticator via text message. It is the most common method of OTP delivery that is encountered across services.
2. **Application Based:** This method of OTP generation is done on the user side using a specific smartphone application that scans a QR code on the screen. The application is responsible for the unique OTP digits. This reduces wait time for the OTP as well as reduces security risk as compared to the SMS based delivery.

The most common way for the generation of OTP defined by The Initiative For Open Authentication (OATH) is the **Time Based One Time Passwords (TOTP)**, which is a Time Synchronized OTP. In these OTP systems, time is the cardinal factor to generate the unique password. The password generated is created using the current time and it also factors in a secret key. An example of this OTP generation is the Time Based OTP Algorithm (TOTP) described as follows:

1. Backend server generates the secret key
2. The server shares secret key with the service generating the OTP
3. A hash based message authentication code (HMAC) is generated using the obtained secret key and time. This is done using the cryptographic SHA-1 algorithm. Since both the server and the device requesting the OTP, have access to time, which is obviously dynamic, it is taken as a parameter in the algorithm. Here, the Unix timestamp is considered which is independent of time

zone i.e. time is calculated in seconds starting from January First 1970. Let us consider “0215a7d8c15b492e21116482b6d34fc4e1a9f6ba” as the generated string from the HMAC-SHA1 algorithm.

4. The code generated is 20 bytes long and is thus truncated to the desired length suitable for the user to enter. Here dynamic truncation is used. For the 20-byte code “0215a7d8c15b492e21116482b6d34fc4e1a9f6ba”, each character occupies 4 bits. The entire string is taken as 20 individual one byte string.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
02	15	a7	d8	c1	5b	49	2e	21	11	64	82	b6	d3	4f	c4	e1	a9	f6	ba

We look at the last character, here a. The decimal value of which is taken to determine the offset from which to begin truncation. Starting from the offset value, 10 the next 31 bits are read to obtain the string “6482b6d3″. The last thing left to do, is to take our hexadecimal numerical value, and convert it to decimal, which gives 1686288083. All we need now are the last desired length of OTP digits of the obtained decimal string, zero-padded if necessary. This is easily accomplished by taking the decimal string, modulo 10 ^ number of digits required in OTP. We end up with “288083” as our TOTP code.

5. A counter is used to keep track of the time elapsed and generate a new code after a set interval of time
6. OTP generated is delivered to user by the methods described above.

Apart from the time-based method described above, there also exist certain mathematical algorithms for OTP generation for example a one-way function that creates a subsequent OTP from the previously created OTP. The two factor authentication system is an effective strategy that exploits the authentication principles of “something that you know” and “something that you have”.The dynamic nature of the latter principle implemented by the One Time Password Algorithm is crucial to security and offers an effective layer of protection against malicious attackers. The unpredictability of the OTP presents a hindrance in peeling off the layers that this method of cryptography has to offer.

Example:

we’ll create a simple One Time Password (OTP) algorithm using Python’s built-in ‘secrets' module. The OTP algorithm will generate a random one-time password, which will be used as a secure authentication token for a user.

Explanation: The OTP algorithm will use a secret key (a random string) to generate the one-time password. The ‘secret’ key should be kept secure and not shared with others. The secrets module provides a strong source of randomness to generate the key securely.

We’ll use the ‘secrets.token_hex()' function to generate a random secret key and the ‘secrets.choice()' function to create a random OTP by choosing characters randomly from a predefined set.

Python code

```
import secrets
```

```
# Function to generate a random secret key
def generate_secret_key():
    return secrets.token_hex(16)  # 16 bytes (32 hex characters)

# Function to generate a One Time Password (OTP) using the secret key
def generate_otp(secret_key, length=6):
    # Defining the characters allowed in the OTP
    allowed_characters = "0123456789"

    # Generating a random OTP using the secret key and allowed characters
    otp = ''.join(secrets.choice(allowed_characters) for _ in range(length))

    return otp

# Example usage
if __name__ == "__main__":
    # Generate a random secret key (this should be kept secure)
    secret_key = generate_secret_key()

    # Simulate sending the OTP to the user
    otp = generate_otp(secret_key)

    # Simulating user input for OTP verification
    user_input = input("Please enter the received OTP:")
```

```
# Verify the OTP entered by the user

if user_input == otp:

    print("OTP verification successful. Access granted!")

else:

    print("OTP verification failed. Access denied!")
```

Output:

```
PS C:\Users\Sashu Akshu> & "C:/Program Files/Python312/python.exe" "c:/Users/Sashu Akshu/Desktop/myproject/myproject/otp generator.py"

Please enter the received OTP:12345

OTP verification failed. Access denied!
```

Explanation of the above code

1. The ‘generate_secret_key()' function generates a 16-byte (32 hexadecimal characters) random secret key using ‘secrets.token_hex()'. You can adjust the length if needed, but 16 bytes is considered secure.
2. The ‘generate_otp()' function takes the secret key and an optional length argument (default is 6) to specify the length of the OTP. It creates an OTP by randomly choosing characters from the string “0123456789” (numbers only) and returns the OTP.
3. In the example usage, we generate a random secret key using ‘generate_secret_key()'. This key should be kept secure and not shared.
4. We simulate sending the OTP to the user by calling ‘generate_otp(secret_key)' and storing the OTP in the variable ‘otp'.
5. We ask the user to input the received OTP and store it in the variable ‘user_input'.
6. Finally, we compare the user-inputted OTP with the generated OTP. If they match, the user is granted access, otherwise, access is denied.

HMAC(Hash based Message Authentication Code):

HMAC (Hash-based Message Authentication Code) is a type of a message authentication code (MAC) that is acquired by executing a cryptographic hash function on the data (that is) to be authenticated and a secret shared key. Like any of the MAC, it is used for both data integrity and authentication. Checking data integrity is necessary for the parties involved in communication. HTTPS, SFTP, FTPS, and other transfer protocols use HMAC. The cryptographic hash function may be MD-5, SHA-1, or SHA-256. Digital signatures are nearly similar to

HMACs i.e they both employ a hash function and a shared key. The difference lies in the keys i.e HMACs use symmetric key (same copy) while Signatures use asymmetric (two different keys).

History

Processes and decisions pertinent to business are greatly dependent on integrity. If attackers tamper this data, it may affect the processes and business decisions. So while working online over the internet, care must be taken to ensure integrity or least know if the data is changed. That is when HMAC comes into use.

Applications

- Verification of e-mail address during activation or creation of an account.
- Authentication of form data that is sent to the client browser and then submitted back.
- HMACs can be used for Internet of things (IoT) due to less cost.
- Whenever there is a need to reset the password, a link that can be used once is sent without adding a server state.
- It can take a message of any length and convert it into a fixed-length message digest. That is even if you got a long message, the message digest will be small and thus permits maximizing bandwidth.

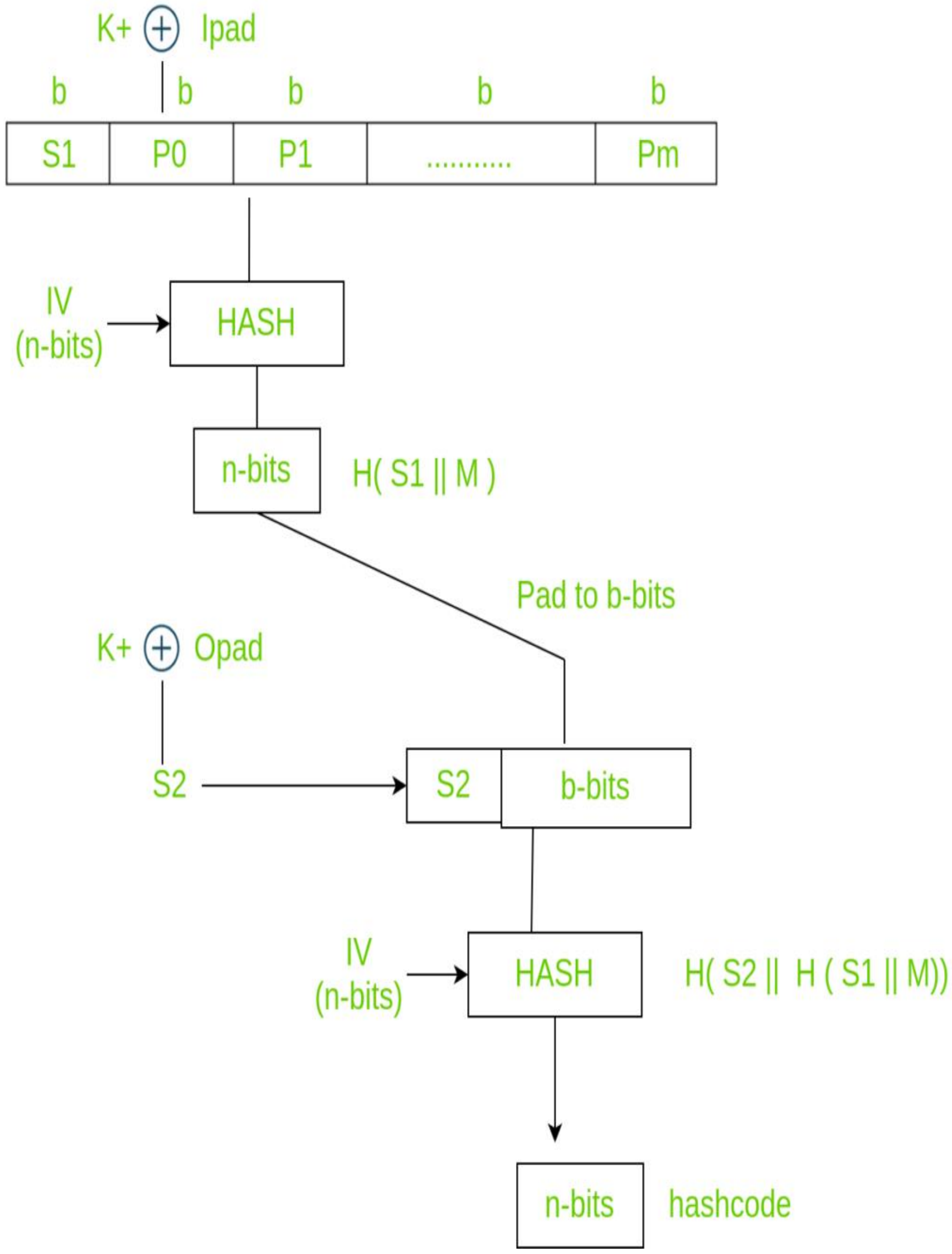
Working of HMAC

HMACs provides client and server with a shared private key that is known only to them. The client makes a unique hash (HMAC) for every request. When the client requests the server, it hashes the requested data with a private key and sends it as a part of the request. Both the message and key are hashed in separate steps making it secure. When the server receives the request, it makes its own HMAC. Both the HMACS are compared and if both are equal, the client is considered legitimate.

The formula **for HMAC:**

```
HMAC = hashFunc(secret key + message)
```

There are three types of authentication functions. They are message encryption, message authentication code, and hash functions. The major difference between MAC and hash (HMAC here) is the dependence of a key. In HMAC we have to apply the hash function along with a key on the plain text. The hash function will be applied to the plain text message. But before applying, we have to compute S bits and then append it to plain text and after that apply the hash function. For generating those S bits we make use of a key that is shared between the sender and receiver.



Using key K (0 < K < b), K+ is generated by padding O’s on left side of key K until length becomes b bits. The reason why it’s not padded on right is change (increase) in the length of key. b bits because it is the block size of plain text. There are two predefined padding bits called ipad and opad. All this is done before applying hash function to the plain text message.

```
ipad - 00110110

opad - 01011100
```

Now we have to calculate S bits

K+ is EXORed with ipad and the result is S1 bits which is equivalent to b bits since both K+ and ipad are b bits. We have to append S1 with plain text messages. Let P be the plain text message.

S1, p0, p1 upto Pm each is b bits. m is the number of plain text blocks. P0 is plain text block and b is plain text block size. After appending S1 to Plain text we have to apply HASH algorithm (any variant). Simultaneously we have to apply initialization vector (IV) which is a buffer of size n-bits. The result produced is therefore n-bit hashcode i.e H(S1 || M).

Similarly, n-bits are padded to b-bits And K+ is EXORed with opad producing output S2 bits. S2 is appended to the b-bits and once again hash function is applied with IV to the block. This further results into n-bit hashcode which is H(S2 || H(S1 || M)).

Summary:

1. Select K.
   If K < b, pad 0’s on left until k=b. K is between 0 and b (0 < K < b)
2. EXOR K+ with ipad equivalent to b bits producing S1 bits.
3. Append S1 with plain text M
4. Apply SHA-512 on (S1 || M)
5. Pad n-bits until length is equal to b-bits
6. EXOR K+ with opad equivalent to b bits producing S2 bits.
7. Append S2 with output of step 5.
8. Apply SHA-512 on step 7 to output n-bit hashcode.

Advantages

- HMACs are ideal for high-performance systems like routers due to the use of hash functions which are calculated and verified quickly unlike the public key systems.
- Digital signatures are larger than HMACs, yet the HMACs provide comparably higher security.
- HMACs are used in administrations where public key systems are prohibited.

Disadvantages

- HMACs uses shared key which may lead to non-repudiation. If either sender or receiver’s key is compromised then it will be easy for attackers to create unauthorized messages.