CSCI 458 Network Security & Management

Project: IoT Lightweight Encryption, Our simple version of RSA + PRESENT algorithms

Objective

- Become familiar with a lightweight block cipher for the Internet of Things (IoT) and all its operations
- To implement a simplified version of the PRESENT cipher
- To exchange a symmetric key using a simple version of RSA public key encryption

References:

PRESENT: Read paper "PRESENT: An Ultra-Lightweight Block Cipher" by Bogdanov, L.R. et al. on D2L

THIS IS AN INDIVIDUAL PROJECT
DELIVERABLES

- **You own working Python code**
- **Live demonstration where we will run some test and will explain your code**

PART 1:

PRESENT cipher

In this project, we will implement the PRESENT algorithm. It is considered a lightweight block cipher for embedded devices or sensor nodes in an Internet of Things (IoT) system. PRESENT is usually implemented in hardware using a field programmable gate array (FPGA).

Each block is 64 bits. The key has 80 bits. But we will assume a 64-bit key.

It also applies 32 rounds of encryption to the data, but let's assume just one round for the project.

Figure 1 shows a summary of the PRESENT algorithm. Please try to understand this flowchart. Think about what happens only in the first round:

- We have a 64-bit block of plaintext
- We do a binary XOR with the master key K
- We apply the 4-bit S-Box
- Then there is a permutation layer (where the bit positions are changed)

The "STATE" is the ciphertext output of each previous step.

generateRoundKeys()
for $i = 1$ to 31 **do**
  addRoundKey(STATE,K_i)
  sBoxLayer(STATE)
  pLayer(STATE)
end for
addRoundKey(STATE,K_{32})

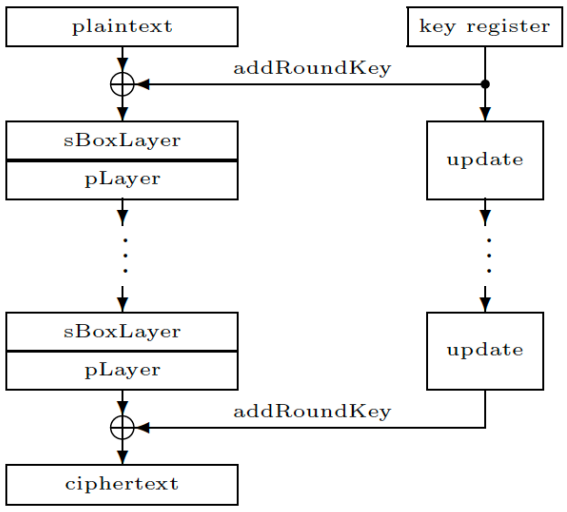

Fig. 1. A top-level algorithmic description of PRESENT.

At each of the 32 rounds, the master key K needs to be updated. So all 32 keys can be generated first (which is the generateRoundKeys() function in Figure 1).

- Implement the first two functions: addRoundKey and sBox4

Task 1: addRoundKey function

Here is how the addRoundKey function is explained in the paper about PRESENT.

> **addRoundKey.** Given round key $K_i = \kappa^i_{63} \ldots \kappa^i_0$ for $1 \leq i \leq 32$ and current STATE $b_{63} \ldots b_0$, addRoundKey consists of the operation for $0 \leq j \leq 63$,
>
> $$b_j \rightarrow b_j \oplus \kappa^i_j.$$

Again, assuming only one round i=1:

- we have K = k_{63}….k_0,
- current state at the first round is your plaintext b_{63}….b_0
- then you do the XOR of b and K. I recommend that you implement this XOR bit by bit (in binary)

Let's assume this 64-bit key (the same size as the block of data).

Assume the block of data is some sensor reading that you got in your embedded device (Table 1 shows some collected from a Capstone project) :
Plaintext = "0x28B4D27B225F8BD8"

Table 1

[0]	unsigned char	0x28 '(' (Hex)
[1]	unsigned char	0xB4 '\xb4' (Hex)
[2]	unsigned char	0xD2 '\xd2' (Hex)
[3]	unsigned char	0x7B '{' (Hex)
[4]	unsigned char	0x22 '"' (Hex)
[5]	unsigned char	0x5F '_' (Hex)
[6]	unsigned char	0x8B '\x8b' (Hex)
[7]	unsigned char	0xD8 '\xd8' (Hex)

And the key is:

K = "0x0123456789ABCDEF"

Note: research on Python how to convert this hexadecimal string to a binary string or binary value. Start with small values, then test the 8-byte (64-bit) key with plaintext.

For simplicity, let's test it locally first (i.e., you can call the functions at the bottom of your file, or in a separate main file). The output should be the "intermediate ciphertext" that we call state. To check if you're doing this addRoundKey function correctly, here is the ciphertext C1 that you should get with the plaintext and key given above.

C1 = (you should get this: 0x2997971cabf42637)

Test the decryption as well (it should be the same operation). Check if you can recover the plaintext

Task 2: Four-bit S-Box

This is the S-box layer defined in the PRESENT algorithm.
For the encryption. every letter x is a 4-bit hexadecimal value in Table 2 which is replaced with the value S(x).

sBoxlayer. The S-box used in PRESENT is a 4-bit to 4-bit S-box $S : \mathbb{F}_2^4 \rightarrow \mathbb{F}_2^4$. The action of this box in hexadecimal notation is given by the following table.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Table 2 – 4-bit S-Box for the PRESENT algorithm

For decryption you should use the inverse S-box (the inverse of Table 1), as below:

c	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2
$S^{-1}(c)$	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

- Test your S-Box using the following 64-bit plaintext:

Plaintext = "0x28B4D27B225F8BD8"

First test your S-box encryption and decryption locally (at the end of the functions file or in a main file). You should get this ciphertext:

Ciphertext = _____

- Test the decryption too (use the inverse S-Box on Ciphertext) to see if you get the plaintext.

Task 3: addRoundKey + 4-bit S-box

Apply multiple encryption function before to your message P given key K:

C1 = addRoundKey(K, P)
C2 = sBox(C1)

Let's test in your S-Box using the following 64-bit plaintext:

Assume the block of data is some sensor reading that you got in your embedded device (the figure below shows collected from a Capstone project):

Plaintext = "0x28B4D27B225F8BD8"

[0]	unsigned char	0x28 '(' (Hex)
[1]	unsigned char	0xB4 '\xb4' (Hex)
[2]	unsigned char	0xD2 '\xd2' (Hex)
[3]	unsigned char	0x7B '{' (Hex)
[4]	unsigned char	0x22 '"' (Hex)
[5]	unsigned char	0x5F '_' (Hex)
[6]	unsigned char	0x8B '\x8b' (Hex)
[7]	unsigned char	0xD8 '\xd8' (Hex)

And the key is:

K = "0x0123456789ABCDEF"

C1 = __(you should get: 0x2997971cabf42637)

C2 = __ you should get: 0x6eeded54f8296abd)_____

Make sure to test the decryption in the reverse order:

inv_sBox
addRoundKey

Permutation Layer – pLayer

Here you should move the position of your bits. Just be careful with the following:

- in a Python string or array, the position [0] is the first one (the most significant bit, or first bit on the left)
- in the permutation box, bit 0 (i=0) is the last bit in your array (the least significant bit, on last bit on the right)

pLayer. The bit permutation used in PRESENT is given by the following table. Bit i of STATE is moved to bit position $P(i)$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P(i)$	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P(i)$	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P(i)$	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P(i)$	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

Now apply multiple encryption functions before to your message P given key K:

C1 = addRoundKey(K, P)
C2 = sBox(C1)
C3 = pLayer(C2)

Make sure to test the decryption in reverse order:

inv_pLayer
inv_sBox
addRoundKey

PART 2:

RSA Algorithm

Public key cryptography uses two different keys - one being public and the other being private. It is computationally hard to deduce the private key from the public key. Mathematically, the process is based on the "trap-door" one-way function concept.

A 4-bit block RSA

Let us implem-ent a simple python program to show us how the public-key algorithm called RSA works. To generate the two keys, choose two random large prime numbers p and q. For simplicity of calculation in our lab, let's assume small prime numbers such as;

$p = 47$
$q = 7\ 1$

Then calculate: $n = p \times q$
And calculate: $\varphi(n) = (p-1) \times (q-1)$

Let's assume your public key is made of these two numbers:
Public key = $[e, n]$
Private key = $[d, n]$
where $e = 97$ and $d = 1693$.

Assume your plaintext is: '**0x012324B10AFBECDD'**. Convert each hexadecimal value of your hexadecimal string to a number m_i.

For each number m_i, encrypt it by doing the following:
$c_i = E_{PUB}(m_i) = (m_i^e)\ modulus\ n$

Print the final ciphertext $C = [c_0, c_1, c_2, \dots, c_n]$ (You can print a list of values)

Then, decrypt your ciphertext C by doing the following for each number c_i:
$D_{PRIV}(c_i) = (c_i^d)\ modulus\ n$

This should give you back m_i.

def cipherRSA(data, e, n)

Use the RSA key to share a symmetric key K in your UDP client/server application

Open your client and server application with the code that you did in Lab 9 (polyalphabetic cipher). Last week, we had to give the same key K to the client and server. Then, both client and server used the polyalphabetic cipher to send encrypted chat messages.

The difference is that in this project we will use a hybrid crypto system: RSA and PRESENT

Please add the following steps to your Python code from last week:

1 – The client creates a session key K. For instance K ="tbs" (Note: The server does not know this key yet).

2 – The client has the server's RSA public key (e,n) that you generated in Task 2. The server has both the RSA public key (e,n) and the private key (d,n).

3 – The client encrypts the session key K with the public key (e,n) using the RSA algorithm

4 – When the server receives the encrypted message, it decrypts K using its private key (d,n). Now the server has K.

5 – Now the client sends the 64-bit hexadecimal sensor data to the server encrypted with the session key K and the PRESENT cipher, and the server decrypts it using K.