Cryptographic Engineering: Assignment 4 Prelab

In this assignment, you implement one of the most important cryptography algorithms which is used inside Transport Layer Security (TLS) protocol in the real world applications. This algorithm computes a shared secret key between two parties using a set of public parameters, so they can communicate securely using the generated shared secret key, utilizing a symmetric cryptography protocol such as AES.

Exponentiation

In the previous assignment, you used exponentiation to compute inversion using Fermat's Little Theorem. Algorithm 1 shows exponentiation using square-and-multiply algorithm. In inversion, the exponent is fixed $(p-2)$. In this assignment, you are going to do exponentiation with a variable exponent. Therefore, the if statement (Line 4) is **needed** because the condition of each iteration is unknown. However, using such if statements will lead to side channel and fault injection attacks.

To make the code operate in constant time, you can compute both $t_1 = b^2 \mod p$ and $t_2 = t_1 \cdot a \mod p$, then

- If $e_i = 0$, then $b = t_1$ which makes $b = b^2 \mod p$
- If $e_i = 1$, then $b = t_2$ which makes $b = b^2 \cdot a \mod p$

Now, you compute the if statement condition independent of e_i. However, for completely secure design, if statements should be avoided. A direct masking approach, like the one used in Assignment 3, is not possible. However, a select function can be implemented without if statements to solve this problem.

Algorithm 1: Modular exponentiation using square-and-multiply algorithm.

Input : a, p, $e = (e_{k-1} \dots e_1 e_0)$ with $e_{k-1} = 1$
Output: $b = a^e \mod p$

```
1 b ← a
2 for i ← k − 2 downto 0 do
3 |   b ← b^2 mod p
4 |   if e_i = 1 then
5 |   |   b ← b · a mod p
6 return b
```

Select Function

A selection function has 4 parameters r, a, b, and `option` and implements the following

- If `option` $= 0$, then $r = a$
- If `option` $= 1$, then $r = b$

To implement this function without if statements, a masking approach is used. First compute `mask` $= 0 -$ `option` which sets all bits to 1 when `option` $= 1$ and all bits to 0 when `option` $= 0$. Then, compute $d = a$ `xor` b which sets different bits to 1 and similar bits to 0. Next, compute $g =$ `mask and` d which sets it to d when `option` $= 1$ and to 0 when `option` $= 0$. Finally, compute $r = a$ `xor` g which is basically $r = a$ `xor` $d = a$ `xor` a `xor` $b = b$ when `option` $= 1$ and $r = a$ `xor` $0 = a$ when `option` $= 0$.

Diffie-Hellman

The algorithm you are going to implement in this assignment is called Diffie-Hellman key-exchange algorithm and it is proposed by Diffie and Hellman in 1976. Each communication party has two keys, a private key and a public key. The private key is generated and stored at each party's local machine, while the public key is published publicly over the public media.

In this assignment, you will use the arithmetic functions which you developed in Assignment 3 to construct a real world cryptography protocol over the group $\mathbb{Z}_p^* = \mathbb{Z}_p - \{0\}$ where $p = 2^{255} - 19$. We also need to define another public parameter g, which is our generator. g is chosen such that $g^0, g^1, \ldots, g^{p-1}$ will generate group $\mathbb{Z}_p^*$. This public parameter is given in the source code.

Key Generation

Key generation is the first function required in Diffie-Hellman. Key generation generates the secret key and public key.

The secret key sk is a random number usually between 0 (inclusive) and p (exclusive) with all values must having equal probability to be generated. However, in computer machines, which work in binary, this task is very expensive and probabilities are not guaranteed to be equal. A random number between 0 (inclusive) and power of 2 (exclusive) with equal probability can easily be generated by computers. Choosing a range larger than p will lead to certain values in $\mathbb{Z}_p^*$ having double probability of being chosen compared to other values. Therefore, the range chosen must be smaller than p. The recommended approach is to generate 256 random bits. Then clear bits 0, 1, 2 and 255 and set bit 254. This will give unique values in $\mathbb{Z}_p^*$ with equal probability excluding 0 which we don't like to work with because $a^0 = 1$ which is the identity of the group.

A function called `RANDOM_DATA(ptr,size)` is provided to you to generate random values using the recommended approach in Linux which is basically reading data from the file `/dev/urandom`.

```
FILE *frandom = fopen("/dev/urandom", "r");
fread(ptr, size, 1, frandom);
fclose(frandom)
```

However, testing that your functions work correctly is impossible with random values. Therefore, the random values will be read from the test bank file instead which are not random. A Macro `REAL_RANDOM` is used to switch between reading real random values for real implementations and test bank values for testing. `#define REAL_RANDOM 1` reads random values while `#define REAL_RANDOM 0` reads test bank values.

The public key pk is generated from the generator and secret key by performing $pk = g^{sk} \mod p$ which is a simple exponentiation.

Shared Secret Generation

Shared secret generation is the second function required in Diffie-Hellman. The shared secret ss is generated from a public key and a secret key by performing $ss = pk^{sk} \mod p$.

Diffie-Hellman Algorithm

Two parties Alice and Bob want to generate a shared secret using Diffie-Hellman. The algorithm works as follows:

- Alice generates her pair of secret key $sk_A = a$ and public key $pk_A = g^a$ using the key generation function. Similarly, Bob generates his pair of secret key $sk_B = b$ and public key $pk_B = g^a$ using the key generation function. This can be done ahead of time as no party is required to communicate with the other party.

- Alice and Bob exchange their public keys pk_A and pk_B through a public medium (Example: Internet).
- Alice generates her shared secret ss_A using her secret key $sk_A = a$ and Bob's pubic key $pk_B = g^b$ using the shared secret generation function. Similarly, Bob generates his shared secret ss_B using his secret key $sk_B = b$ and Alice's pubic key $pk_A = g^a$ using the shared secret generation function.

The two shared secrets are guaranteed to be equal.

$$ss_A = (g^b)^a = g^{ba} = g^{ab} = (g^a)^b = ss_B$$