Diffie-Hellman Algorithm

Criptography

Carmen Azorín Martí

May 15th 2024

1 Introduction

This code written in Python is inspired by the Diffie Hellman Algorithm.
The idea of this algorithm is to create an encryption key that only two people in distance (let's say Alice and Bob) know. To do this, each will be assigned a private integer (unknown to the others) and two public integers.

Alice and Bob have to jointly choose a prime number p and a generator of theirs g. Both numbers will be public. On the other hand, each one chooses any number, which will be their private integer. So, Alice sends to Bob

$$g^a \pmod p$$

being a Alice's private integer. And Bob sends Alice

$$g^b \pmod p$$

being b Bob's private integer. In this way, if everyone raises the integer received to their private integer, they get the key

$$key = (g^a)^b \equiv (g^b)^a \equiv g^{ab} \pmod p$$

But this key is really difficult to calculate if you don't know any of the private integers.

In this report we will define a couple of functions that can be used to obtain the private integers from the public keys and the private key. That is, solve the following equation:

$$h = g^x \pmod p$$

where p and g are the public keys (prime and its generator), and h is the private key. The aim is getting x.

2 Implemented functions

The first function implemented in Pyhton is called *generator()* and receives two integer parameters g and p. This function returns *true* if and only if g is a generator of group $\mathbf{Z_p^*}$.

```
def generator(g,p):
    s = set(range(1,p))
    gen = set()
    for x in s:
        gen.add((g**x)%p)
    if gen == s:
        return True
    return False
```

In general, to show that an element g is a generator of group $\mathbf{Z_p^*}$, you need to show every element in the group is some power of g. In our function,

$$gen = \{g^x \pmod{p} : 1 \leq x \leq p\}$$

gen contains every power of g. We have that: If

$$\mathbf{Z_p^*} = \{0, 1, ..., p-1\} = gen$$

then g is a generator of $\mathbf{Z_p^*}$.

In the previous practice we defined a function to calculate the inverse of a number a modulo b. However, this function was not very efficient. In this practice we are going to define another one using the following Theorem:

Multiplicative Inverse Algorithm. Given two integer $0 < b < a$, consider the Euclidean Algorithm equation which yield $gdc(a, b) = r_j$. Rewrite all of these equations exept the last one, by solving for the remainder:

$$r_1 = a - bq_1$$

r$_2 = b - r_1 q_2$
r$_3 = r_1 - r_2 q_3$
...
r$_{j-1} = r_{j-3} - r_{j-2} q_{j-1} r_j = r_{j-2} - r_{j-1} q_j$
Then, in the last of these equations, replace r_{j-1} with its expression in terms of r_{j-3} and r_{j-2} from the equation immediately above it. Continue this process successively, replacing r_{j-2}, $r - j - 3$,..., until you obtain the final equation

$$r_j = ax + by$$

with x and y integers. In the special case that $gdc(a, b) = 1$, the integer equation reads

$$1 = ax + by$$

Therefore we deduce

$$1 \equiv by \pmod{a}$$

so thar $y$is the multiplicative inverse of b mod a.

This theorem is the inspiration of our recursive function *extended_euclid()* that receives two integers, a and b, and returns 3 elements: the grand common divisor, x and y, that fulfill:

$$gdc = a \cdot x + b \cdot y$$

Let's see the code and observe a simple example of how it works:

```python
def extended_euclid(a, b):
    if a == 0:
        return b, 0, 1
    else:
        gcd, x, y = extended_euclid(b % a, a)
        print(gcd, x, y)
        return gcd, y - (b // a) * x, x
```

For example, suppose we call the function *extended_euclid()* with values $a = 8$ and $b = 11$, the idea is finding the multiplicative inverse of 8 mod 11.

1. Since $a \neq 0$, the function calls itself with parameters $a = 11 \pmod{8} = 3, b = 8$.

2. Since $a \neq 0$, the function calls itself with parameters $a = 8 \pmod{3} = 2, b = 3$.

3. Since $a \neq 0$, the function calls itself with parameters $a = 3 \pmod{2} = 1, b = 2$.

4. Since $a \neq 0$, the function calls itself with parameters $a = 2 \pmod 1 = 0, b = 1$.

Now, we go backwards:

- 4. Since $a = 0$, the function returns $gdc = 1, x = 0, y = 1$.
- 3. The function returns $gdc = 1, x = 1 - floor(2/1) * 0 = 1, y = 0$.
- 2. The function returns $gdc = 1, x = 0 - floor(3/2) * 1 = -1, y = 1$.
- 1. The function returns $gdc = 1, x = 1 - floor(8/3) * (-1) = 3, y = -1$.

Finally, it returns $gdc = 1, x = -1 - floor(11/8) * 3 = -4, y = 3$
Since $gdc = 1$, the inverse exists and we have that

$$1 \equiv 8 \cdot (-4) + 11 \cdot 3$$

Therefore

$$1 \equiv 8 \cdot (-4) \pmod{11}$$

Since, -4 is not in mod 11, we have this function *euklid()* that calls the function above and returns the inverse mod p:

```
    def euklid(a, p):
    gcd, x, y = extended_euclid(a, p)
    if gcd != 1:
        raise ValueError("Inverse does not exist.")
    return x % p
```

3 Shanks's Baby-Step Giant-Step Algorithm

This is the algorithm we are going to use to solve the equation illustrated in the introduction

$$h = g^x \pmod p$$

Let G be a cyclic group of size n with generator g. Given $h \in G$, the following algorithm computes an m so that $h = g^m$:

1. Compute and store the baby steps values $1, g, g^2, ..., g^{\lfloor \sqrt{n} \rfloor - 1}$.
2. Compute the giant steps $hg^{-i\lfloor \sqrt{n} \rfloor}, i = 1, 2, ...$, and check, for each i, whether the result is on the baby step list. If a match is found, stop; otherwise continue to the next value of i.

Once you have found a match, you have found i and j so that $hg^{-i\lfloor \sqrt{n} \rfloor} = g^j$, so $h = g^{i\lfloor \sqrt{n} \rfloor + j}$.

The idea is to show that there must be some match with $0 \leq i \leq \lfloor \sqrt{n} \rfloor + 1$. To see this, let us suppose that $h = g^m$. Write $m = i\lfloor \sqrt{n} \rfloor + j$, with $0 \leq j \leq \lfloor \sqrt{n} \rfloor - 1$. Then

$$i = \frac{m - j}{\lfloor \sqrt{n} \rfloor} \leq \frac{m}{\lfloor \sqrt{n} \rfloor} \leq \frac{n - 1}{\lfloor \sqrt{n} \rfloor} \leq \frac{n - 1}{\sqrt{n} - 1} = \sqrt{n} + 1$$

so $i \leq \lfloor \sqrt{n} \rfloor + 1$ since i is an integer.

Let's expose the code in Python given p, g and h.

```
p = 1117
g = 6
h = 527

if not generator(g, p):
    print("g is not a generator of  Z p .")
else:
    square_root = int(math.sqrt(p))
    baby_steps = []
    for i in range(0,square_root):
        baby_steps.append(pow(g,i,p))
    i = 1
    giant_step = (euklid(pow(g,square_root*i,p),p)*h)%p
    while not baby_steps.__contains__(giant_step):
        i += 1
        giant_step = (euklid(pow(g,square_root*i,p),p)*h)%p
    pos_baby_step = baby_steps.index(giant_step)
    print(i*square_root+pos_baby_step)
```

The library *math* has been used to calculate the square root. The floor operation is made by function *int*, since since returns the integer part of a decimal number.
If we execute this code, the result is 123. In fact, $x = 123$ fulfills the equation for $p = 1117, g = 6, h = 527$:

$$h = g^x \pmod p$$