Task: Implementation of the ElGamal – Zsolt Berecz, Martin Bucko

The brief history and description of the ElGamal

ElGamal encryption was expressed as a public key cryptosystem based on the discrete logarithm problem by Taher ElGamal in 1985. In 1975, Diffie and Hellman introduced a concept for key exchange, in the field of public key cryptography. The ElGamal is based on this key exchange protocol. ElGamal is an encryption is an asymmetric key encryption algorithm that ensures two parties can communicate privately, because of the difficulty of the logarithmic problem hard to crack it.

The algorithm and the steps to implement Elgamal.

There are three steps we must implement, one of that is the key generation, encryption and decryption.

The first step is to select a very large prime number **p**. After that the primitive root of p that is **g** know. A random number which is in the range of 1 < **n** < p-1. We must calculate **e** which is $e = g^a \, mod \, p$. After that we get our public key (p, g, e).

The second step is to encrypt the data. There’s a plain text, which is the message, let’s call it **m**. There’s an important thing that m < p. The sender must select a random integer number, it is not necessary to have it as a prime number in this case, let’s call it **r**. The number must be in the range of 1 < r < p-1. The sender computes two ciphertexts, the name of the variables are **C1** and **C2.**

$C1 = g^b mod \, p \qquad C2 = m * e^b mod \, p$

The ciphertext is: (C1, C2)

The last step is to decrypt. We get as an input the (C1, C2) ciphertexts, and now it’s straightforward to use just simply two formulas to decrypt the messages.

$x = C1^a \, mod \, p \;\; m = C2 * \, x^{p-1} \, mod \, p$

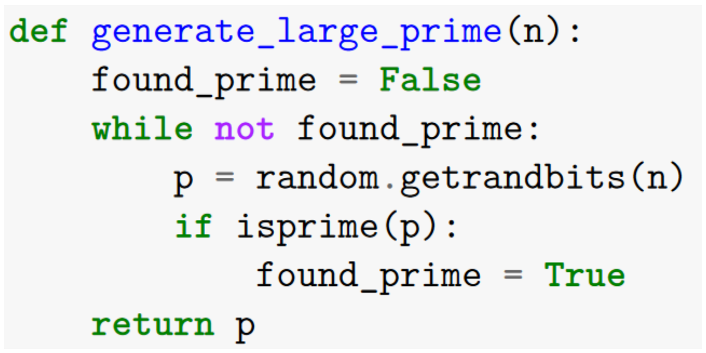
At the end, the m variable is the decrypted message that we want to see. As from the previous paragraphs we can see that ElGamal is simple, straightforward and easy to use and implement. Let’s see in practice.

The implementation of the ElGamal

For the implementation I chose Python, it’s easy to implement and straightforward.

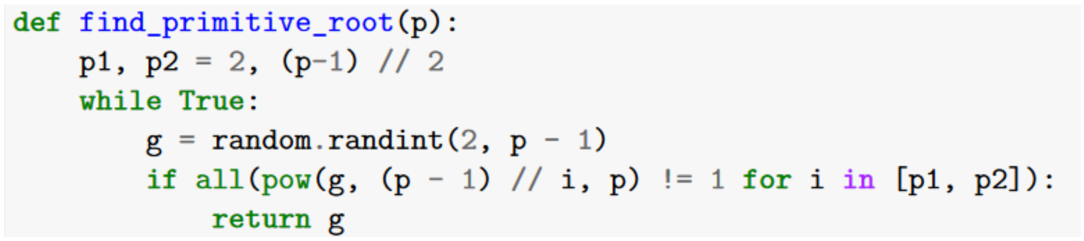
The first and most important step is the Baillie-PSW primality test, that helps to chose a big probably a prime number. With that function we can generate bigger prime numbers for encryption,

We can actually use the basic method of finding a prime number like a for loop that checks the divisibility of the number till the square root of that number, but that is not the best solution for big prime (like) numbers.



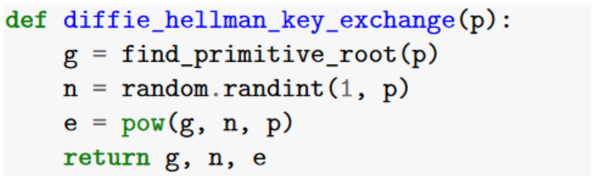
```
def generate_large_prime(n):
    found_prime = False
    while not found_prime:
        p = random.getrandbits(n)
        if isprime(p):
            found_prime = True
    return p
```

The second step is to generate a large prime number that is useful for later, when we encrypt, use the Diffie-Hellman key exchange and decryption as well.
We generate random numbers till it meets the criteria of the Baillie-PSW .



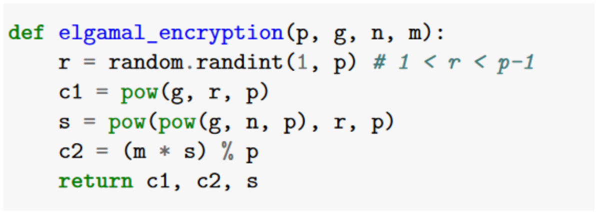
```
def find_primitive_root(p):
    p1, p2 = 2, (p-1) // 2
    while True:
        g = random.randint(2, p - 1)
        if all(pow(g, (p - 1) // i, p) != 1 for i in [p1, p2]):
            return g
```

Finding the primitive root can be easily implemented by Python fortunately, using list compression, using all() is important to check if all elements of the iterable list compression are True. If yes we return the g as the primitive root. If not, it creates a new g value with the random.randint function.



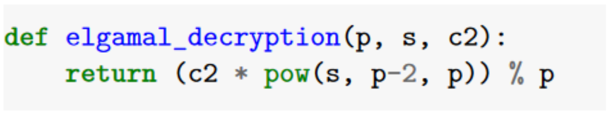
```python
def diffie_hellman_key_exchange(p):
    g = find_primitive_root(p)
    n = random.randint(1, p)
    e = pow(g, n, p)
    return g, n, e
```

For Diffie-Hellman, at first we generate and get and find the primitive root, after with random.randint we generate the integer and what I previously described we generate the e value and return that values.



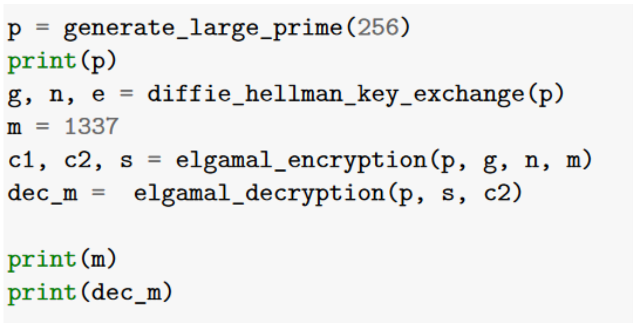
```python
def elgamal_encryption(p, g, n, m):
    r = random.randint(1, p) # 1 < r < p-1
    c1 = pow(g, r, p)
    s = pow(pow(g, n, p), r, p)
    c2 = (m * s) % p
    return c1, c2, s
```

Now the encryption function comes. Here we also generate a random number, and actually creating the C1 and C2 cypher values and s variable with help of the Python the in built pow() function.



```python
def elgamal_decryption(p, s, c2):
    return (c2 * pow(s, p-2, p)) % p
```

The decryption function is more straightforward, for that it is enough to just do a one-liner to decrpyt the message. Here we just need the s the C2 value to decrypt the message, the C1 cypher value is used for the s value. After we got the decrypted message.



```python
p = generate_large_prime(256)
print(p)
g, n, e = diffie_hellman_key_exchange(p)
m = 1337
c1, c2, s = elgamal_encryption(p, g, n, m)
dec_m =  elgamal_decryption(p, s, c2)

print(m)
print(dec_m)
```

Now that, the implementation of the algorithm was done, here it is easy to test it out it works as intended. The p value is to generate the large prime number, the g, n, e values from the key exchange. And the c1, c2 and s values that came from the encryption. The decrypted message in the end. The last two lines are there what was the message before and what is the decrypted message. That show everything went fine.

Homomorphic property of the ElGamal and implementation

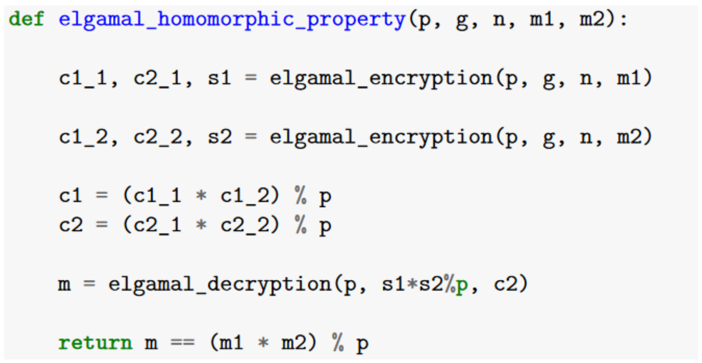
What is the homomorphic property?

The homomorphic property of the ElGamal encryption scheme refers to its ability to perform operations on ciphertexts that correspond to operations on the plaintexts. So that means if we multiply to ciphertext together and then decrypt that. We can describe that as E() that is ElGamal encryption and m is the message. Then E(m1) and E(m2) is going to be:

$$E(m1) * E(m2) = E(m1 * m2)$$

With that property, which is useful, we can make ElGamal much more mature and secure, it helps such as electronic voting. It allows operations to be performed without decrypting the data.

The implementation of the homomorphic property



```python
def elgamal_homomorphic_property(p, g, n, m1, m2):

    c1_1, c2_1, s1 = elgamal_encryption(p, g, n, m1)

    c1_2, c2_2, s2 = elgamal_encryption(p, g, n, m2)

    c1 = (c1_1 * c1_2) % p
    c2 = (c2_1 * c2_2) % p

    m = elgamal_decryption(p, s1*s2%p, c2)

    return m == (m1 * m2) % p
```

The implementation is also straightforward, that we have two messages one p value for the prime number, and the same g and n value from the Diffie-Hellman key exchange. We use m1 and m2 to encrypt the data and after that we perform the multiplication for the c1 and c2 values that are encrypted. After that, we decrypt the value and in the return, we compare that the m decrypted message exactly the same as the (m1 * m2) % p, with that we can prove that the homomorphic property is working for the ElGamal encryption.

Conclusion

The ElGamal encryption algorithm is a powerful public key encryption scheme that was first proposed by Taher ElGamal in 1985. It is based on the difficulty of computing discrete logarithms over finite fields.

One of the main strengths of the ElGamal encryption algorithm is that it provides perfect forward secrecy. This means that even if an attacker were to compromise the private key, they would not be able to decrypt messages that were sent prior to the compromise.

Another strength of ElGamal is that it is relatively easy to implement and is widely used1. However, one weakness of ElGamal is that it is relatively slow compared to other encryption algorithms such as AES. Additionally, the size of the ciphertext is larger than the size of the plaintext, which can be problematic for certain applications1.

The ElGamal encryption scheme ensures confidentiality by relying on the Discrete Logarithm Problem, making it computationally infeasible for an attacker to recover the shared secret without knowledge of the private key.

In terms of its homomorphic properties, the ElGamal encryption scheme is multiplicatively homomorphic. This means that if you multiply two ciphertexts together, and then decrypt the result, it’s as if you multiplied the original plaintexts. This property is useful in many cryptographic protocols, such as secure multi-party computation and electronic voting.

In conclusion, while the ElGamal encryption scheme may have some limitations, such as its speed and the size of the ciphertext, its strengths, including its perfect forward secrecy and its homomorphic properties, make it a valuable tool in the field of cryptography.