

**Faculty of Engineering & Technology**  
**Department of Computer Science**

**COMP438: ENCRYPTION THEORY**  
**Diffie-Hellman Enhancement**

---

**Prepared by:**

Obada Tahayna

1191319

Karim Halayqa

1192087

**Instructor:** Mohammad Alkhanafseh

**Section:** 1

**Date:** 18/01/2022

In this paper, we discuss the enhancement of the Diffie-Hellman algorithm done by Aryan Et, and the further enhancement we made on it.

## 1-Finding the primitive root:

A primitive root  $q$  of a prime number  $p$  is a number where the equations  $q$  to the power of  $(0 \rightarrow n-1) \bmod n$  all produce different outputs. Primitive roots are used in cryptography, for instance the Diffie Hellman algorithm, for the reason that computing them is hard, meaning sniffers will have a hard time figuring out the origin of the data they're collecting.

```
14 def primitiveRoot(number):
15     if number < 4:
16         number += 4
17
18     while not isprime(number):
19         number += 1
20
21     for i in range(2, number - 1):
22         x = []
23         found = True
24         for j in range(0, number - 1):
25             prim = (i ** j) % number
26             if prim in x:
27                 found = False
28                 break
29             else:
30                 x.append(prim)
31
32         if found:
33             return i
34     return -1
```

This code represents our implementation of the primitive root calculation method mentioned above, which checks all instances of  $(q \text{ to the power of } (0 \rightarrow (n-1)) \bmod n)$  to extract one with entirely unique outputs.

It checks for errors that might occur, one being the number passed to the function is not a prime number, which in this case, that number is incremented until the nearest prime number is found. Secondly, the function checks if the passed number is less than 4, which negates the chance of it having a primitive root, therefor an incrementation by 4 is placed.

## 2-The Diffie-Hellman algorithm implementation:

At first, a socket connection between a server and a client was established, which enables real messages exchange instead of a simulated one.

Server Code:

```
9      ServerSocket = socket.socket()
10     host = '127.0.0.1'
11     port = 1233
12
```

```
110    try:
111        ServerSocket.bind((host, port))
112    except socket.error as e:
113        print(str(e))
114
115    print('Waiting for a Connection..')
116    ServerSocket.listen(5)
117
118    while True:
119        Client, address = ServerSocket.accept()
120        start_new_thread(threaded_client, (Client,))
```

Client code:

```
9      ClientSocket = socket.socket()
10     host = '127.0.0.1'
11     port = 1233
```

```
80    def main():
81        try:
82            ClientSocket.connect((host, port))
83        except socket.error as e:
84            print(str(e))
85        print('Waiting for connection')
86
```

This way, real key exchange will take place between two sides, resulting in a better, more realistic testing of the algorithm.

Secondly, the Diffie Hellman algorithm was implemented step by step until both sides have their secret shared key.

```
63 def round(connection, shared_key):
64     primRoot = primitiveRoot(shared_key)
65     randomNum = random_number(50)
66
67     publicKey1 = generate_public_key(primRoot, randomNum, shared_key)
68     received_publicKey = exchange_keys(connection, publicKey1)
69
70     sharedKey = generate_public_key(received_publicKey, randomNum, shared_key)
71
72     return sharedKey
```

At the beginning, a primitive root was created from a number passed to the algorithm, which is the prime number (p) that both sides agree on in the beginning. Then, a secure random number is created using the user function (random\_number), which utilizes the library (os) to use its function (urandom) to generate a random byte that, according to its python documentation, is suitable for cryptography use, which is turned into an int using the (int.from\_bytes built-in function). ("This function returns random bytes from an OS-specific randomness source. The returned data should be unpredictable enough for cryptographic applications"[1]).

```
43 # secure random number
44 def random_number(maxN):
45     return (int.from_bytes(os.urandom(1), "big")) % maxN
```

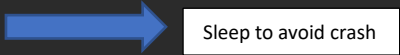
After that, a public key was generated according to the Diffie-Hellman algorithm's public key equation. 4 was added to the equation to avoid the possibility of the key equaling zero, which will result in 0 or -1 outputs for the rest of the algorithm.

```
63 # public key (+ 4 so if it's 0 it doesn't ruin the rest of the algorithm)
64 def generate_public_key(number, power, mod):
65     return (number ** power) % mod + 4
66
```

After each side generates their public key, this key is exchanged between the two sides using these following functions:

Client key exchange function:

```
56 def exchange_keys(send):
57     time.sleep(0.1)
58     socket_send(send)
59
60     return int(socket_receive())
61
```



Server key exchange function:

```
50 def exchange_keys(connection, send):
51     received = int(socket_receive(connection))
52     time.sleep(0.1)
53     socket_send(connection, send)
54
55     return received
```


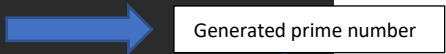
As seen in the functions, the client is the one that initiates the exchange by sending its public key to the server. A (0.1s) sleep timer is added so that the server has a chance to start receiving before the client sends the key, as not doing that can cause a crash in the program. Once the server receives the client's public key, it sleeps for 0.1s to allow the client to open the receiving end, then sends its public key to the client.

Finally, a secret shared key is created according to the Diffie-Hellman's secret key equation.

### 3-The Diffie-Hellman's given enhancement implementation:

From the given enhancement paper, an enhancement suggested adding a second round of the algorithm, doing the same things as the first round, but replacing the prime number that the primitive root is calculated from with the secret shared key produced in the first round.

```
78 round1_key = round(connection, p)
79 round2_key = round(connection, round1_key)
```



The output is a second secret shared key, which is finally used to generate two new shared keys, but this step is postponed until the created enhancement takes place.

## 4-The Diffie-Hellman's created enhancement implementation:

For the further enhancement created, a third round is added, using the result of adding the second shared key to the first one as the base to create the primitive root from.

```
78     round1_key = round(connection, p)
79     round2_key = round(connection, round1_key)
80     ➡ round3_key = round(connection, round2_key + round1_key)
```

After that, the server and the client will create a random number random, then multiply it with the third secret shared key, which results in X for the server and Y for the client. Finally, both sides exchange X and Y, and a final shared key is created by multiplying them on each side, and this key is used to securely exchange messages between the two sides.

Creating X and final key on the server side:

```
83     j = random_number(1000)
84     U = round3_key
85
86     X = U * j
87     Y = exchange_keys(connection, X)
88
89     finalKey = X if Y == 0 else Y if X == 0 else X * Y
```

Creating Y and final key on the client side:

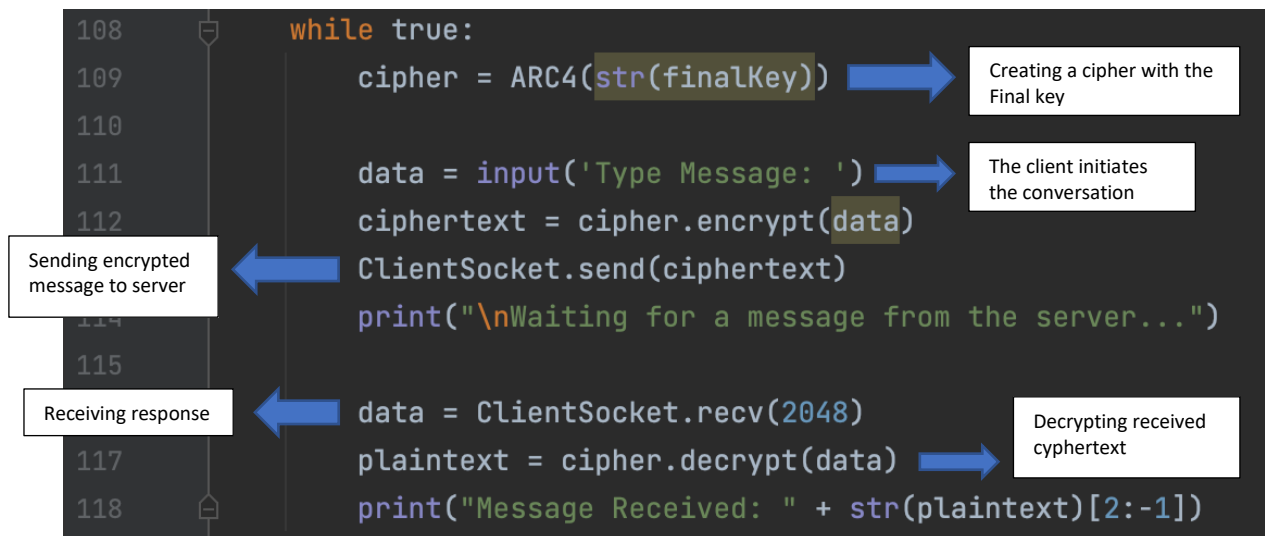
```
98     Y = U * k
99     X = exchange_keys(Y)
100
101     finalKey = X if Y == 0 else Y if X == 0 else X*Y
```

The final key is either if the other is 0, otherwise it's the result of their multiplication.

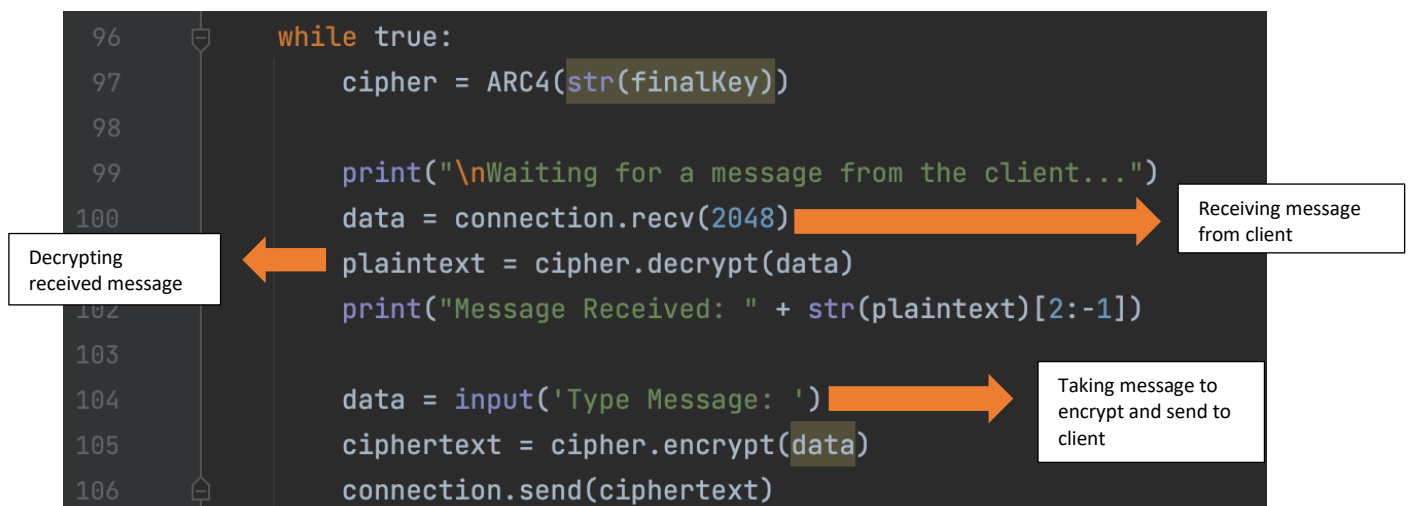
## 5-Using the enhanced Diffie-Hellman in message exchange:

RC4 was selected as an encryption algorithm to use the key generated by the Diffie-Hellman enhanced algorithm to exchange messages on both sides.

### Client-side RC4 utilization:



### Server-side RC4 utilization:



## 5-Output:

### Server algorithm implementation output:

```
Run: server x client1 x
/Users/kareemhalayka/PycharmProjects/DiffieHellmanEnhanced/venv/bin/python /Users/kareemhalayka/PycharmProjects/DiffieHellmanEnhanced/server.py
Waiting for a Connection..
X: 860
Y: 774
Final Key: 665640
```

### Client algorithm implementation output:

```
Run: server x client1 x
/Users/kareemhalayka/PycharmProjects/DiffieHellmanEnhanced/venv/bin/python /Users/kareemhalayka/PycharmProjects/DiffieHellmanEnhanced/client1.py
Waiting for connection
X: 860
Y: 774
Final Key: 665640
```

As it can be observed, both parties have the same final key without ever exchanging it.

### Client-Server messages exchange using RC4 and enhanced Diffie-Hellman:

```
-----| Conversation starts here |-----
Type Message: | Client
```

```
-----| Conversation starts here |-----
Type Message: Hello server! Client
Waiting for a message from the server...
```

```
-----| Conversation starts here |-----
Waiting for a message from the client...
Message Received: Hello server!
Type Message: Server
```

```
-----| Conversation starts here |-----
Waiting for a message from the client...
Message Received: Hello server!
Type Message: Hey Client! Server
Waiting for a message from the client...
```



This implementation portrays a real message exchange using the enhanced Diffie-Hellman key exchange algorithm and RC4.

Reference:

- [1] <https://docs.python.org/3/library/os.html>
- [2] <https://pypi.org/project/arc4/>