Efficient Elliptic Curve Diffie Hellman

For this final, I decided to implement the Elliptic Curve Diffie Hellman key exchange protocol. I am utilizing python for its extensive cryptography standard library and default bigint number system. While this functionality exists in other languages, the default behavior of python makes my code much simpler.

In this form of cryptography, values are represented as points on some curve defined by domain parameters A,B,P,G,N, and H. A and B define the curve through the equation $y^2 = x^3 + Ax + B$ (where $4A^3 + 27b^2 \neq 0$). P defines the modulus for all operations, so all coordinate values are in the range $0 <= x <= P-1$. G is the generator point whose cyclic group is of size N. H is the cofactor, which tells us how many of the points on the curve fall in the cyclic group of G. From these parameters we can define two operations for points on the curve. Firstly, we have the operation of addition. To add two points on the curve, we draw a line between then, and create a new point where that line intersects the curve. We then negate the y coordinate of the new point to get our final addition result. Next, we have the doubling operation. In this operation, we take in a single point P and find its ‘double.’ To do this, we draw the tangent line to the curve at the point P, then negate the y coordinate of the intersection to get our final result.

The core of elliptic curve cryptography is the point multiplication trapdoor function. Point multiplication is the process by which we take a point P on our curve and add it to itself K times. For example $P' = P * K = P_0 + P_1 + P_2 \ldots + P_{K-1}$. Doing this Naively takes exponential time as the bit size of K gets larger $(O(\log(K)^2)$, but we can use an algorithm known as ‘double and add,’ which is similar to ‘square and multiply’ to exponentially speed up our point multiplication. With this optimization, we can decrease the runtime of point multiplication to linear on the size of K $(O(\log(K))$. This means we can very quickly find the point $P' = P_0 + P_1 + P_2 \ldots + P_{K-1}$, even when K is very large. On the other hand, an attacker is not able to quickly find K given P’. Assuming the attacker has not solved the elliptic curve discrete logarithm problem, the attacker is forced to calculate P’ for all possible values of K in the range $0 <= K <= N-1$. N is chosen to be very large such that his computation is not feasible. SECP192K1, A medium strength elliptic curve has an N value of 6277101735386680763835789423061264271957123915200845512077, which is 6 Billion quadrillion quadrillion.

I spent around 3 hours researching and implementing an elliptic curve diffie hellman key exchange. I used the aforementioned SECP192K1, which is an SEC recommended curve, and was able to successfully complete the entire task. After deriving the shared secret point, I also generated a 256 bit key which can be used for symmetric encryption.

Below is the Jupyter notebook containing the code and results.

ecdh

April 15, 2022

1 ECDH

This program simulates the elliptic curve Diffie Hellman key exchange protocol. This program uses the curve 'SECP192K1'

Sources: SEC 2: Recommended Elliptic Curve Domain Parameters Elliptic Curve Cryptography

[ ]:
```
class Point():
    def __init__(self,x,y, inf=False) -> None:
        self.x = x
        self.y = y
        self.inf = inf
    def __repr__(self) -> str:
        return "Infinity" if self.inf else f"Point({self.x}, {self.y})"
```

[ ]:
```
def inv(x: int):
    return pow(x, -1, P) ### Beautiful extended euclid, TY python

def add(p: Point,q: Point):
    if p.inf :
        return q
    if q.inf:
        return p
    if p.x == q.x:
        return Point(0,0,True)
    s = ((p.y - q.y) * inv((p.x - q.x))) % P
    x = (s** 2 - (p.x + q.x)) % P
    y = (s * (p.x - x) - p.y) % P
    return Point(x,y)

def double(p: Point):
    if p.x == 0:
        return Point(0,0,True)
    s = int(((3 * (p.x ** 2)) + A) * inv(2 * p.y)) % P
    x = int(s** 2 - (p.x * 2)) % P
    y = int(s * (p.x - x) - p.y) % P
    return Point(x,y)
```

```python
def mult(p: Point, k: int): #double and add ~~ square and multiply
    current = p
    result = Point(0,0,True)
    while k:
        if k & 1:
            result = add(result, current)
        current = double(current)
        k = k >> 1
    return result
```

[ ]:
```python
import secrets
import hashlib

## SECP192K1
A = 0x000000000000000000000000000000000000000000000000
B = 0X000000000000000000000000000000000000000000000003
P = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFEE37
G = Point(0x04DB4FF10EC057E9AE26B07D0280B7F4341DA5D1B1EAE06C7D,␣
 ↪0x9B2F2F6D9C5628A7844163D015BE86344082AA88D95E2F9D)
N = 0xFFFFFFFFFFFFFFFFFFFFFFFE26F2FC170F69466A74DEFD8D

#Bob Initial
Beta = secrets.randbelow(N)
BobPublic = mult(G, Beta)
print(f"Bob's Private key: {Beta}")
print (f"Bob's public key: {BobPublic}")


#Alice Initial
Alpha = secrets.randbelow(N)
AlicePublic = mult(G, Alpha)
print (f"Alice's private key: {Alpha}")
print (f"Alice public key: {AlicePublic}")

#Bob sends BobPublic to Alice, and Alice sends AlicePublic to Bob
```

```
Bob's Private key: 3945862639310893485759080664415460008390603909207623410445
Bob's public key:
Point(878296211905532047611311614104022565670265235945129097939,
1419003335040752837732819551671792517708432192983673224558)
Alice's private key: 5859487053533843463773314974543126959829008016559297189308
Alice public key:
Point(5307732700349788847027568308535485745914904531261843410902,
4840226664934380822411372726518640819870985863255946033337)
```

```
[ ]: #Bob Later
     PBob = mult(AlicePublic, Beta)
     #Alice Later
     PAlice = mult(BobPublic, Alpha)

     BobHasher = hashlib.sha256()
     BobHasher.update(str(PBob.x).encode('utf8'))
     BobEncryptionKey = BobHasher.hexdigest()

     print(f"{BobEncryptionKey=}")

     AliceHasher = hashlib.sha256()
     AliceHasher.update(str(PAlice.x).encode('utf8'))
     AliceEncryptionKey = AliceHasher.hexdigest()
     print(f"{AliceEncryptionKey=}")
```

```
BobEncryptionKey='25ac4eed2fdb640a9c1e1812ff1e7f24e12fd2cabea88a10073bbfd42e8938
a3'
AliceEncryptionKey='25ac4eed2fdb640a9c1e1812ff1e7f24e12fd2cabea88a10073bbfd42e89
38a3'
```