Implementation and Benchmarking of Elliptic Curve Cryptography Algorithms

Yulin Ou
yulin_ou@umail.ucsb.edu
Department of Electrical and Computer Engineering
University of California Santa Barbara

June 13, 2017

Abstract

This paper will investigate how to implement the classical Elliptic Curve Cryptography and two variants: Elliptic Curve Discrete Signing Algorithm (ECDSA) and the Elliptic Curve Diffie-Hellman (ECDH) key-exchange. Both algorithm will be implemented in Python programming language. To present a detailed test methodology, this paper will perform test of algorithms based on the same curve representations: Weierstrass curves. Furthermore, this article will evaluate the efficiency of different algorithms in breaking the security of these algorithms.

1 Introduction

Public-key cryptography makes it possible to create digital signatures and do key negotiation, which is essential for todays commercial and governmental online computer systems. Elliptic curve cryptography (ECC) is a popular method to implement public-key cryptography. Compared to other modular arithmetic systems such as the Rivest-Shamir-Adleman algorithm (RSA), ECC has low computational complexity, which provides equivalent security, but only requires smaller keys compared to non-ECC cryptography.
This article will investigate the algorithm of ECC[2] and its variants: Elliptic Curve Diffie-Hellman (ECDH) and Elliptic Curve Discrete Signing Algorithm (ECDSA)[1]. Then implementing ECDH and ECDSA in Python. Finally, comparing the efficiency of two popular algorithms for breaking this discrete logarithm problem.

2 Elliptic curve cryptography

Elliptic curves (EC) can be applied to many mathematical problems. This section will introduce the EC for usage in cryptography. The elliptic curve(E) is presented

as the following: Elliptic curve is the graph given by the equation usually named the short Weierstrass equation:

$$y^2 = x^3 + ax^2 + b, \text{where} 4a^3 + 27b^2 \neq 0$$

Depending on the value of a and b, elliptic curves may assume different shapes on the plane. As it can be easily seen and verified, elliptic curves are symmetric about the x-axis.
Public-key cryptography is based on the intractability of certain mathematical problems. Early public-key systems are secure assuming that it is difficult to factor a large integer composed of two or more large prime factors. For elliptic-curve-based protocols, Elliptic Curve Discrete Logarithm Problem (ECDLP) assumes that finding the discrete logarithm of a random elliptic curve element with respect to a publicly known base point is infeasible. The security of elliptic curve cryptography depends on the ability to compute a point multiplication and the inability to compute the multiplicand given the original and product points. The size of the elliptic curve determines the difficulty of the problem.

1. The **private key** is a random integer d chosen from $\{1, ..., n-1\}$, where n is the order of the subgroup.

2. The **public key** is the point $H = dG$, where G is the base point of the subgroup.

If we know d and G, along with the other domain parameters. Finding H will not spend a lot of time. But if we only know H and G, it will be hard to find the private key d. Since it requires to solve discrete logarithm problem. In the next section, we are going to describe two public-key algorithms based on that: ECDH (Elliptic curve Diffie-Hellman), which is used for encryption, and ECDSA (Elliptic Curve Digital Signature Algorithm), used for digital signing.

3 Elliptic curve Diffie-Hellman

ECDH is a variant of the Diffie-Hellman algorithm for elliptic curves. It basically means that ECDH defines how keys should be generated and exchanged between parties. Its actually a key-agreement protocol. It solves the following problem: two parties (Alice and Bob) want to exchange information securely, while a third party (the Man In the Middle) may intercept them, but may not decode them. This is one of the principles behind Transport Layer Security (TLS). The Diffie-Hellman problem is described below:

1. Alice and Bob generate their own private and public keys. Private key d_A and public key $H_A = d_A G$ for Alice, and private key d_B and $H_B = d_B G$ for Bob. Both Alice and Bob use the same domain parameters: the same base point G on the same elliptic curve on the same finite field.

2. Alice and Bob exchange their public key H_A and H_B over an insecure channel. The Man In the Middle intercept H_A and H_B, but will mot find out neither d_A and d_B without solving the discrete logarithm problem.

3. Alice calculates $S = d_A H_B$ by using her own private key ad Bobs public key, and Bob calculates $S = d_B H_A$ by using his own private key and Alices public key. S is the same for both Alice and Bob.

$$S = d_A H_B = d_A(d_B G) = d_B(d_A G) = d)BH_A$$

However, the Man in Middle only knows H_A and H_B together with the other domain parameters and would not be able to find out the shared secret S. The Diffie-Hellman problem for elliptic curves is assumed to be a "hard" problem. Since it is believed to be spend a lot of time to solve as the discrete logarithm problem. One way to solve the logarithm problem is solving the Diffie-Hellman problem.
Implementing ECDH in Python to compute public / private keys and shared secrets over an elliptic curve. We choose the curve secp256k1, which is also used by Botcoin of digital signatures. Domain parameters are listed below:

- p = 0xffffffff ffffffff ffffffff ffffffff ffffffff ffffffff fffffffe fffffc2f
- a = 0
- b = 7
- x_G = 0x79be667e f9dcbbac 55a06295 ce870b07 029bfcdb 2dce28d9 59f2815b 16f81798
- y_G = 0x483ada77 26a3c465 5da4fbfc 0e1108a8 fd17b448 a6855419 9c47d08f fb10d4b8
- n = 0xffffffff ffffffff ffffffff fffffffe baaedce6 af48a03b bfd25e8c d0364141
- h = 1

Figure 1 shows the result of implementing the python script of ECDH. We generate Alice and Bobsecret keys and compute their public keys respectively. After exchange their public keys, they can calculate the shared secret. Now that Alice and Bob have obtained the shared secret, they can exchange data with symmetric encryption.

```
Curve: secp256k1
Alice's private key: 0xd23db6c44e89d03353d371d2d78bf0788e96a3cfb66bf81e60b533c6eb263844
Alice's public key: (0xad287e650cdc4611ec04df4778e0a369ee8c2b442587131db24e024bc2dfcb3f,
 0x3ea674844fe59c9dd0baa5eded4c058db719ab54f98eae77259ef17e334329e3)
Bob's private key: 0xb597ec1af5fda5904d32aff7d26680e4447a52b8637f1e844bc3bd06c904a701
Bob's public key: (0x78d47fb3af79272a4b339a551f08cd5303ac15e21837546f5db1a073755968b2, 0
x26e442738da9265be8ae2765f51598f48dc4de8ed7b5cf05d851a3c64a94947)
Shared secret: (0x769c54e4a32855312ffef993d7aae188d44c0577da95a45d84c420a9b94ea57e, 0x8b
2df6c3da098aad0e0ff5dcb1f83742186f6ab7d3125997195d92e09feef9e)
```

Figure 1: Implementation of ECDH

4 Elliptic Curve Digital Signature Algorithm

ECDSA is a variant of the Digital Signature Algorithm applied to elliptic curves. ECDSA works on the hash of the message, rather than on the message itself. The choice of the hash function is up to us, but it should be a cryptographically-secure hash function. The hash of the message ought to be truncated so that the bit length of the hash is the same as the bit length of n (the order of the subgroup). The truncated hash is an integer and will be denoted as Z.
The scenario is the following: Alice wants to sign a message with her private key d_A, and Bob wants to validate the signature using Alices public key H_A. Nobody but Alice should be able to valid signature. Everyone should be able to check signature. Figure 2 shows the output of implementing ECDSA in Python.The algorithm performed by Alice is described below:

1. Take a random integer k chosen from $\{1, , n1\}$, where n is still the subgroup order.
2. Calculate the point P=kG, where G is the base point of the subgroup.
3. Calculate the number $r = x_P \mod n$where x_P is the coordinate of P.
4. If r=0, then choose another k and try again.
5. Calculate $s = k^{-1}(z + rd_A) \mod n$, where d_A is Alice's private key and k^{-1} is the multiplicative inverse of k modulo n.
6. If s=0, then choose another k and try again.

```
Curve: secp256k1
Private key: 0x777e1d9aef9c91a72f646d6895e1e159e3bfdd86c7ec02a197b261a2dbe74921
Public key: (0xe58a4c5964e2060639e1524fa1dfedeb0fadba871e9b23b6a37ca1f62ba7179,
0x9d3937a0d89cc2c0f46fd91edb74b79111936d6f522425e6e72ab968f5c7bc16)

Message: b'Hello!'
Signature: (0xd7074bffbe2161a1d9f460e669cbe7500d9653206aff779d7c5ab21067a20ebc,
0x9a8126bacb825491cae4aff1b9c77096e5eab5c3cb0912c1050bb90a7196d4fe)
Verification: signature matches

Message: b'Hi there!'
Verification: invalid signature

Message: b'Hello!'
Public key: (0x47104ba7eddec92b5198d1926bb9ead21c94b920e3924334635c9cb7db617429,
 0xeb53476fcab0761fbb48d99ef4a0de6e713efb6cda8cf11c1bb779b80d1bac9f)
Verification: invalid signature
```

Figure 2: Implementation of ECDSA

5 Attack method

There are two efficient algorithms for computing discrete logarithms on elliptic curve[3]: the baby-step, giant-step algorithm, and Pollardś rho method. The definition of the discrete logarithm problem is: Given two points P and Q, find out the integer x that satisfies the equation $Q = xP$.

(a) Baby-step, giant-step

The consideration behind Baby-step, giant-step algorithm is that we can always write any integer x as $x = am + b$, where a, m and b are three arbitrary integers. With this in mind, we can rewrite the equation for the discrete logarithm problem as follows:

$$Q - amP = bP$$

. Contrary to the brute-force attack, which forces us to calculate all the points xP for every x until we find Q, we will calculate fewer value before we find a correspondence. The algorithm works as follows:

1. Calculate $m = \lceil \sqrt{n} \rceil$
2. For every b in 0,,m, calculate bP and store the results in a hash table.
3. For every a in 0,,m:
   - Calculate amP;
   - Calculate $Q - amp$;
   - Check the hash table and look if there exist a point bP such that $QamP = bP$;
   - If such point exists, then we have found $x = am + b$.

Initially we calculate the points bP with little (i.e. "baby") increments for the coefficient $b(1P, 2P, 3P, ...)$. Then in the second part of the algorithm, we calculate the points amP with huge (i.e. "giant") increments for $am(1mP, 2mP, 3mP, ...)$.Once a match is found, calculating the discrete logarithm is a matter of rearranging terms. This algorithm has both time and space complexity $O(\sqrt{n})$. It's still exponential time, but much better than a brute-force attack.

(b) Pollardś rho

Pollard's rho is another algorithm for computing discrete logarithms. It has the same asymptotic time complexity $O(\sqrt{n}$ of the baby-step giant-step algorithm, but its space complexity is $O(1)$. Baby-step giant-step can't be used in practice, because of the huge memory requirements. Compare with Baby-step giant-step, Pollard's rho requires very few memory. With Pollard's rho, we need to solve a different problem: Given P and Q, find the integers, a, b , A and B such that $aP + bQ = AP + BQ$.Once the four integers are found, we can find out $x = (a-A)(B-b)^{-1} \mod n$. The principle of Pollardś rho is simple: we define a pseudo-random sequence of (a,b) pairs. This sequence of pairs can be used to generate the sequence of points $aP + bQ$. Since

both P and Q are elements of the cyclie subgroup, the sequence of points $aP + bQ$ is cyclic too. FInally, we will find a pair (a,b)and another distinct pair (A,B) such that $aP + bQ = AP + BQ$.

(c) Comparison of Different Attacking Algorithms

Figure 3 shows the output of implementing ECDSA in Python.Obviously, the brute-force method is tremendously slow if compared to the others two algorithms. Baby-step giant-step is the faster, while Pollard's rho is more than three times slower than baby-step giant-step. Then examining he number of steps: brute force used 5193 steps on average for computing each logarithm. Baby-step giant-steps and Pollard's rho used 152 steps and 138 steps respectively, two numbers very close to the $\sqrt{10331} = 101.64$.

```
Curve order: 10331
Using bruteforce
Computing all logarithms: 100.00% done
Took 3m 29s (5180 steps on average)
Using babygiantstep
Computing all logarithms: 100.00% done
Took 0m 8s (152 steps on average)
Using pollardsrho
Computing all logarithms: 100.00% done
Took 0m 24s (136 steps on average)
```

Figure 3: Comparison of brute-force, baby-giant-step and pollardsrho

6 Conclusions

Elliptic Curve Cryptography (ECC) is one of the most powerful but least understood types of cryptography in wide use today. The aim for this article is to investigate the algorithms and performance of ECDH and ECDSA. It's important to be bear in mind that algorithms can be greatly optimized in many ways such as improvements in hardware. An approach seems impractical today does not mean that this approach will not work in the future. And in the future, there may exists better algorithms. For example, a quantum algorithm , Shor's algorithm, is capable of computing discrete logarithms in polynomial time.

References

[1] Olav Wegner Eide. Elliptic curve cryptography-implementation and performance testing of curve representations. 2017.

[2] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. Guide to elliptic curve cryptography. 2006.

[3] ÇK Koç. Cryptographic engineering (2009).