A short and gentle

Introduction to Elliptic Curve Cryptography

Written by F. Rienhardt

https://github.com/hazelfazel

Abstract

This is a very basic and simplified introduction into elliptic curve cryptography. I tried to keep things as simple as possible. There are no mathematical proofs or something. This whitepaper hopefully gives you a gentle start into the field of elliptic curve cryptography, without fearing you to hell by using heavy weighted proofs, definitions and cryptic acronyms as usually seen on this topic.

last update: 2020/07/21

Table of Contents

1 Some basics about elliptic curves

In general elliptic curves (ec) combine number theory and algebraic geometry. These curves can be defined over any field of numbers (i.e., real, integer, complex and even Fp). An elliptic curve consists of the set of numbers (x, y), also known as points on that curve, that satisfies the equation: $y^2 = x^3 + ax + b$

Let's say a = 1 and b = 7 then the elliptic curve $y^2 = x^3 + x + 7$ over real numbers looks like this if you plot it:

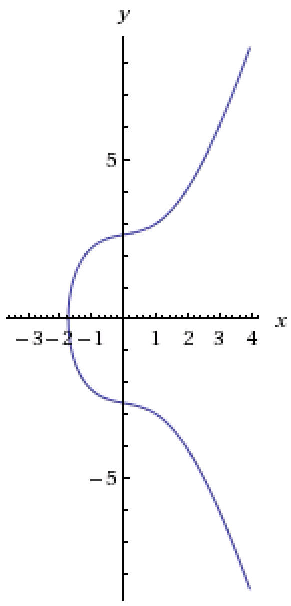

Figure 1: $y^2 = x^3 + x + 7$

The set of all of the solutions to the equation forms the elliptic curve. Changing a and b directly changes the shape of the curve. Small changes in these parameters often result in major changes in the set of (x, y) solutions.

We generally see elliptic curves used over finite fields in cryptography applications where the points (x, y) form an additive group. There an elliptic curve does not looks like the curve in real as above; although the points are not distributed by random and there exists some sort of dividing line that reflects its points.

Like the prime factorization problem in RSA, elliptic curves can be used to define a "hard" to solve problem: Given two points, P and Q, on an elliptic curve, find the integer k, if it exists, such that P = kQ.

In short ECC is "simply" based on the difficulty of solving the Elliptic Curve Discrete Logarithm Problem (ECDLP). ECC was independently formulated in 1985 by the researchers Victor Miller (IBM) and Neal Koblitz (University of Washington).

1.1 Example of an elliptic cure over the Field F_{23}

As a tiny example, consider an elliptic curve over the field F_{23}. With a = 9 and b = 17, the elliptic curve equation is $y^2 = x^3 + 9x + 17$.

For example the point (3, 5) satisfies this equation since:

$5^2 \bmod 23 = 3^3 + 9*3 + 17 \bmod 23$

25 mod 23 = 71 mod 23

2 = 2

The points which satisfy this equation are:

(1, 2), (1, 21), (3, 5), (3, 18), (4, 5), (4, 18), (5, 7), (5 ,16), (7, 3), (7 ,20), (8, 7), (8, 16), (10, 7), (10, 16), (12, 6), (12, 17), (13, 10), (13, 13), (14, 9), (14, 14), (15, 10), (15, 13), (16, 5), (16, 18), (17, 23), (18, 10), (18, 13), (19, 3), (19, 20), (20, 3), (20, 20).

The points can be plotted as follows

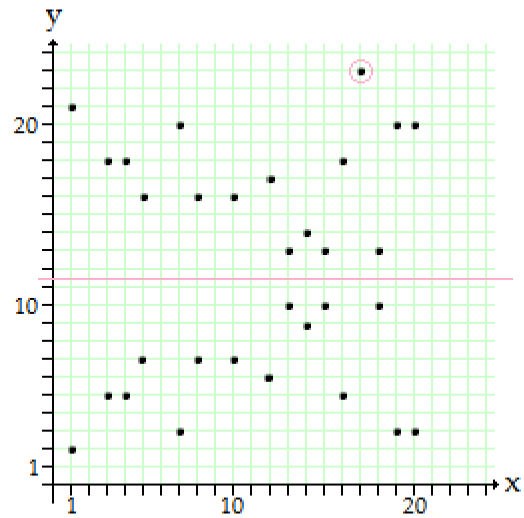

Figure 2: $y^2 \bmod 23 = x^3 + 9x + 17 \bmod 23$

The elliptic curve in Figure 2 does not looks like as tied as in Figure 1 but it is still an elliptic curve satisfying all equitations and rules defined on elliptic curves in general. Note that there are two points for every x value. The set of points defines an additive finit field. Hence each point P on the elliptic curve has its negative point, here $-P = (x_P, (-y_P \bmod 23))$. To do the intended math on such curves we do need some additional operations.

1.2 Adding distinct points P and Q

The negative of the point $P = (x_P, y_P)$ is the point $-P = (x_P, -y_P \bmod p)$. If $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ are distinct points such that P is not -Q, then

$P + Q = R$ where

$s = (y_P - y_Q) / (x_P - x_Q) \bmod p$

$x_R = s^2 - x_P - x_Q \bmod p$ and $y_R = -y_P + s(x_P - x_R) \bmod p$

The value s represents the slope of the line through P and Q.[1]

1.3 Doubling the point P

Provided that y_P is not 0, $2P = R$ where

$s = ((3x_P)^2 + a) / (2y_P) \bmod p$

$x_R = s^2 - 2x_P \bmod p$ and $y_R = -y_P + s(x_P - x_R) \bmod p$

As defined by the quotation of an elliptic curve a is one of the parameters chosen with the elliptic curve itself whereas s represents the so called slope of the line through P and Q.

2 Elliptic curve discrete logarithm

As noted above like the prime factorization problem in RSA, elliptic curves can be used to define a “hard” to solve problem: Given two points, P and Q, on an elliptic curve, find the integer k, if it exists, such that $P = kQ$.

2.1 Example of the Elliptic curve discrete logarithm problem

In the elliptic curve group defined by $y^2 = x^3 + 9x + 17$ over F_{23}, what is the discrete logarithm k of $Q = (4, 5)$ to the base $P = (16, 5)$?

One naive way to find k is to compute multiples of P until Q is found. The first few multiples of P are:

1 division noted by “/” in section 1.2 and section 1.3 represents the division mod p, meaning that it is not the division as known from real numbers. [Wikipedia: http://en.wikipedia.org/wiki/Modulo_operation]: "Given two positive numbers, a (the dividend) and n (the divisor), a modulo n (abbreviated as a mod n) is the remainder of the Euclidean division of a by n. For instance, the expression "5 mod 2" would evaluate to 1 because 5 divided by 2 leaves a quotient of 2 and a remainder of 1."

1P = (16, 5), 2P = (20, 20), 3P = (14, 14), 4P = (19, 20), 5P = (13, 10), 6P = (7, 3), 7P = (8, 7), 8P = (12, 17), 9P = (4, 5)

Since 9P = (4,5) = Q, the discrete logarithm of Q to the base P is k = 9.

In a real application, k would be large enough such that it would be infeasible to determine k with this brute-force approach.

In a cryptographic application the point Q is calculated as a multiple of the starting point P, thus Q = kP. An attacker might know P and Q but finding the integer k is a "hard" problem to solve. There Q (Q = kP) is the public key and k is the private key.

Some additional notes regarding the calculation of kP. Using doubling and adding a distinct point (see section 1.2 and 1.3) it is possible to compute kP. A first step is to get k's binary representation and then add P's n'th doubled value where a bit is set. Example: Let's say k = 13, then its binary representation is 1101 now we can calculate kP using little endian: kP = (1*1P) + (0*2P) + (1*4P) + (1*8P). In an algorithm one starts with 1P and doubles it in each iteration. If bit k[i] in k's binary representation is set, the latest doubled P is added to the result.

2.2 An "Alice and Bob"-example (Diffie-Hellman in ECC)

One of the two (Alice or Bob) defines an elliptic curve and a valid point P and shares the curve information and the point P. Both parties now have the curve and point P.

Now Alice and Bob select a random number each: k_A is Alice's random number and k_B is Bob's random number. Alice calculates $Q_A = k_A P$ and Bob calculates $Q_B = k_B P$. Alice sends Q_A to Bob and Bob sends Q_B to Alice. Alice does not know the value k_B, Bob has randomly chosen, and because the discrete logarithm problem can not be solved fast she is not able to get it out of the public known curve, P and the value Q_B that Bob has sent to her. But she knows her k_A, hence she can multiply Q_B with her k_A - thus $Q_B * k_A = k_B P * k_A = k_A k_B P = Q_{AB}$.

Bob does not know the value k_A, Alice has randomly chosen, and because the discrete logarithm problem can not be solved fast he is not able to get it out of the public known curve, P and the value Q_A that Alice has sent to him. But he knows his k_B, hence he can multiply Q_A with his k_B - thus $Q_A * k_A = k_B P * k_A = k_A k_B P = Q_{AB}$.

Q_{AB}'s x-coordinate is the shared secret and can be used to derivate an encryption key for a symmetric encryption method for example. Because of the discrete logarithm problem an attacker is not able to extract the k out of Alice's $Q_A = k_A P$ neither of Bob's $Q_B = k_B P$, and thus is not able to calculate Q_{AB}. Therefore k_A and k_B are the private parts of the key, while the elliptic curve parameters, P and Q_A/Q_B are public.

3 brainpool example curve domain parameter specification

In this section, a brainpool elliptic curve is specified as an example. ECC brainpool is a consortium of companies and institutions that work in the field of elliptic curve cryptography, who specify and define cryptographic entities in the field of ECC. ECC brainpool also defines elliptic curves that are recommended for cryptographic usage. The following is one such example.
For all brainpool curves,

- an ID is given by which it can be referenced.
- p is the prime specifying the base field.
- a and b are the coefficients of the equation y^2 = x^3 + a*x + b mod p defining the elliptic curve.
- G = (x,y) is the base point, i.e., a point in E of prime order, with x and y being its x- and y-coordinates, respectively.
- q is the prime order of the group generated by G.
- h is the cofactor of G in E, i.e., #E(GF(p))/q.

Here is the brainpoolP192r1

- Curve-ID: brainpoolP192r1
- p = C302F41D932A36CDA7A3463093D18DB78FCE476DE1A86297
- a = 6A91174076B1E0E19C39C031FE8685C1CAE040E5C69A28EF
- b = 469A28EF7C28CCA3DC721D044F4496BCCA7EF4146FBF25C9
- x = C0A0647EAAB6A48753B033C56CB0F0900A2F5C4853375FD6
- y = 14B690866ABD5BB88B5F4828C1490002E6773FA2FA299B8F
- q = C302F41D932A36CDA7A3462F9E9E916B5BE8F1029AC4ACC1
- h = 1

4 ECC in OpenSSL

If I need to manage cryptography stuff quickly I like OpenSSL's elegant simplicity doing lots of crypto operations on the fly just using its console application. Paul Heinlein's howto on OpenSSL[2] gives an excellent introduction into the basic stuff you can do with OpenSSL on the console. Unfortunately the howto lacks an in depth look onto elliptic curves.

If you want to sign a message digest using elliptic curves instead of RSA the following might give you a starting point:

Build up an elliptic curve key:

```
openssl ecparam -out key.pem -name prime192v3 -genkey

openssl ecparam -out key.pem -name brainpoolP320r1 -genkey
```

Check the elliptic curve key:

```
openssl ec -in key.pem -text
```

Extract the public part of your elliptic curve key:

```
openssl ec -in key.pem -pubout -out key2.pem
```

Digest a file and sign it with your elliptic curve key:

```
openssl dgst -ecdsa-with-SHA1 -sign key.pem -out helloworld.txt.ecdsa-sha1 helloworld.txt
```

Verifiy the digest of a file signed with an elliptic curve key:

```
openssl dgst -ecdsa-with-SHA1 -verify key2.pem -signature helloworld.txt.ecdsa-sha1
helloworld.txt
```

If you would like to use more secure hash functions you can sign and verify digital signatures using sha256 or sha384 with elliptic curves, too. Example:

```
openssl dgst -sha256 -sign key.pem -out helloworld.txt.ecdsa-sha256 helloworld.txt
openssl dgst -sha384 -sign key.pem -out helloworld.txt.ecdsa-sha384 helloworld.txt

openssl dgst -sha256 -verify key2.pem -signature helloworld.txt.ecdsa-sha256 helloworld.txt
openssl dgst -sha384 -verify key2.pem -signature helloworld.txt.ecdsa-sha384 helloworld.txt
```

4.1 Make use of your own curves in OpenSSL

OpenSSL comes with great elliptic curve support, but initially it did not support brainpool curves using the named curve parameter. I wanted to use my own curves, especially brainpool curves. So I ended up setting up curve parameters, using

[2] For more details and the how-to see http://www.madboa.com/geek/openssl/

OpenSSL with its ecparam command line option. With ecparam you can easily make use of your own curves by parameter.

All you have to do is calling OpenSSL with ecparam specifying your favorite elliptic curve parameters, specified as described in RFC3279. As an example I show how things look like, if you are using the brainpool curve P256r1 (a.k.a. brainpoolP256r1). The curve is given as:

```
Prime:                      0x00A9FB57DBA1EEA9BC3E660A909D838D726E3BF623D52620282013481D1F6E5377
A:                          0x7D5A0975FC2C3057EEF67530417AFFE7FB8055C126DC5C6CE94A4B44F330B5D9
B:                          0x26DC5C6CE94A4B44F330B5D9BBD77CBF958416295CF7E1CE6BCCDC18FF8C07B6
Generator (uncompressed): 0x048BD2AEB9CB7E57CB2C4B482FFC81B7AFB9DE27E1E3BD23C23A4453BD9ACE32
                            62547EF835C3DAC4FD97F8461A14611DC9C27745132DED8E545C1D54C72F046997
Order:                      0x00A9FB57DBA1EEA9BC3E660A909D838D718C397AA3B561A6F7901E0E82974856A7
Cofactor:                   0x01
```

Encode the parameters above in the format as specified in RFC3279 and save the file as brainpoolP256r1.asn1:

```
asn1=SEQUENCE:ecparams
[ecparams]
no=INTEGER:0x01
prime_field=SEQUENCE:prim
coeff=SEQUENCE:coeffs
generator=FORMAT:HEX,OCTETSTRING:048BD2AEB9CB7E57CB2C4B482FFC81B7AFB9DE27E1E3BD23C23A4453BD9AC
E3262547EF835C3DAC4FD97F8461A14611DC9C27745132DED8E545C1D54C72F046997
primeord=INTEGER:0x00A9FB57DBA1EEA9BC3E660A909D838D718C397AA3B561A6F7901E0E82974856A7
cofac=INTEGER:0x01
[prim]
whatitis=OID:prime-field
prime=INTEGER:0x00A9FB57DBA1EEA9BC3E660A909D838D726E3BF623D52620282013481D1F6E5377
[coeffs]
A=FORMAT:HEX,OCTETSTRING:7D5A0975FC2C3057EEF67530417AFFE7FB8055C126DC5C6CE94A4B44F330B5D9
B=FORMAT:HEX,OCTETSTRING:26DC5C6CE94A4B44F330B5D9BBD77CBF958416295CF7E1CE6BCCDC18FF8C07B6
```

Call OpenSSL as follows:

```
openssl asn1parse -genconf brainpoolP256r1.asn1 -out brainpoolP256r1.der
```

You can cross check that the elliptic curve parameters are proper by calling

```
openssl ecparam -inform DER -in brainpoolP256r1.der -check
```

checking elliptic curve parameters: ok

```
-----BEGIN EC PARAMETERS-----
MIHgAgEBMCwGByqGSM49AQECIQCp+1fboe6pvD5mCpCdg41ybjv2I9UmICggE0gd
H25TdzBEBCB9Wgl1/CwwV+72dTBBev/n+4BVwSbcXGzpSktE8zC12QQgJtxcbOlK
S0TzMLXZu9d8v5WEFilc9+HOa8zcGP+MB7YEQQSL0q65y35XyyxLSC/8gbevud4n
4eO9I8I6RFO9ms4yYlR++DXD2sT9l/hGGhRhHcnCd0UTLe2OVFwdVMcvBGmXAiEA
qftX26Huqbw+ZgqQnYONcYw5eqO1Yab3kB4OgpdIVqcCAQE=
-----END EC PARAMETERS-----
```

Now you can use this curve to perform crypto on it like you would do with named curves. For example generate an elliptic curve key:

```
openssl ecparam -inform DER -in brainpoolP256r1.der -out brainpoolP256r1.key.pem -genkey
```

5 Further reading / References

- Harper, G., Menezes, A., and S. Vanstone, "Public-Key Cryptosystems with Very Small Key Lengths", Advances in Cryptology -- EUROCRYPT '92, LNCS 658, 1993.
- http://tools.ietf.org/html/rfc5639