

Elliptic Curve Builder

This page describes the steps to generate a non-standard or custom domain parameters with Elliptic Curve Builder (ECB) for use in Crypto++. We would use a tool such as ECB because Crypto++ does not implement curve generation (see eccrypto.cpp, near line 485).

ECB, located at <http://www.ellipsa.eu/>, is a elliptical curve utility written by Marcel Martin. Marcel describes ECB as, "... a generator of elliptic curves that are intended for cryptographic use". If problems are encountered with ECB, please contact Marcel directly.

Combining ECB and Crypto++ allows us to use custom curve sizes in Crypto++. Be advised that point counting can be tricky business, so it is usually best to use an approved ANSI, Brainpool, IETF, or NIST curve. Please visit Elliptic Curve Cryptography (http://en.wikipedia.org/wiki/Elliptic_curve_cryptography) and Digital Signature Standard (DSS) (http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf) for more warnings when using non-standard curves.

Finally, if interested in short digital signature schemes, take time to evaluate McEliece Digital Signature Scheme (based on error correcting codes), Quartz (<http://www.cryptosystem.net/quartz>) (based on HFEs), and Pairing Based Cryptography (<http://crypto.stanford.edu/pbc/>). The McEliece cryptosystem predates RSA and produces some of the smallest signatures possible. Quartz is patent encumbered and appears to be poorly supported. For pairings, the PBC Library (<http://crypto.stanford.edu/pbc/>) is offered under GNU licensing and is actively supported by Ben Lynn.

Contents

ECB Toolbar

Initialize the Seed

Generate Curve Parameters

Generate a New Curve

Generate a Point on the Curve

Domain Parameters

Curve

Base Point

Group Order

Subgroup Order

Cofactor

Crypto++ and Domain Parameters

Private and Public Keys

Private Key

Public Key



ECB Toolbar

The ECB toolbar is shown below.



The icons of interest and their actions are explained in the table below. In general, we need to move from the blue lightning bolt to the green lightning bolt. When we receive a result we do not care for, such as a negative coefficient, we click the eraser button and generate another value.

Icon	Function
	Initialize/Reinitialize the random seed
	Create new curve parameters over $F(p)$ or $F(2^m)$
	Create a new curve

	Create a point on the curve
	Erase the curve, parameters, or point

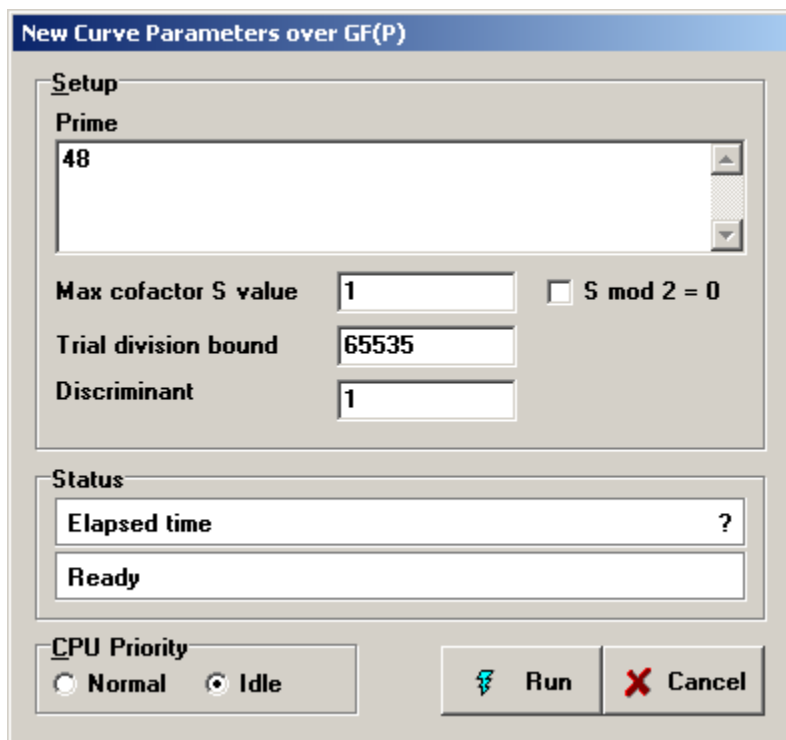
Initialize the Seed

Initialize the seed by clicking the dice.

Generate Curve Parameters

For the purposes of this wiki page, we will create an artificially small 48-bit curve over $F(p)$. The 48-bit curve offers a security level of 2^{24} , which is no security at all by any standard.

Generate curve parameters of the desired size by clicking the blue lightning bolt. The left bolt is used to create new curve parameters over $F(p)$, the right bolt is used to create a new curve over $F(2^m)$.



New Curve Parameters over GF(P)

Setup

Prime
48

Max cofactor S value: 1 ☐ S mod 2 = 0

Trial division bound: 65535

Discriminant: 1



Status

Elapsed time: ?

Ready

CPU Priority

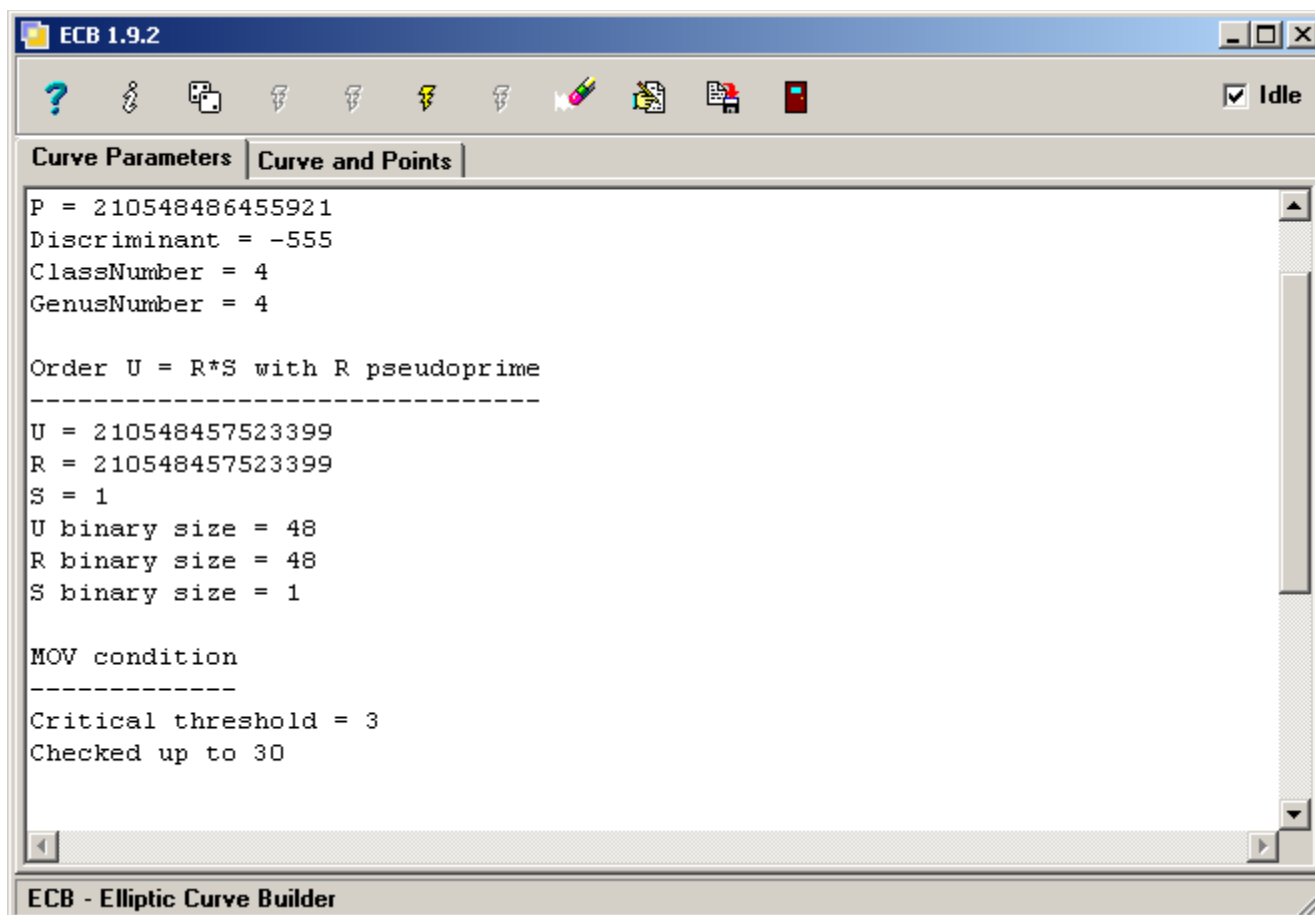
☐ Normal ☒ Idle

 Run  Cancel

The three parameters of interest are:

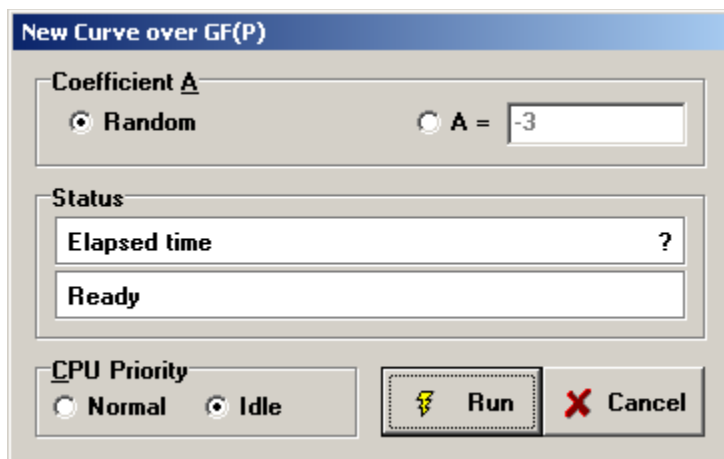
- Prime: 48 (bits)
- Max Cofactor: 1
- Discriminant: 1

The results are shown below.



Generate a New Curve

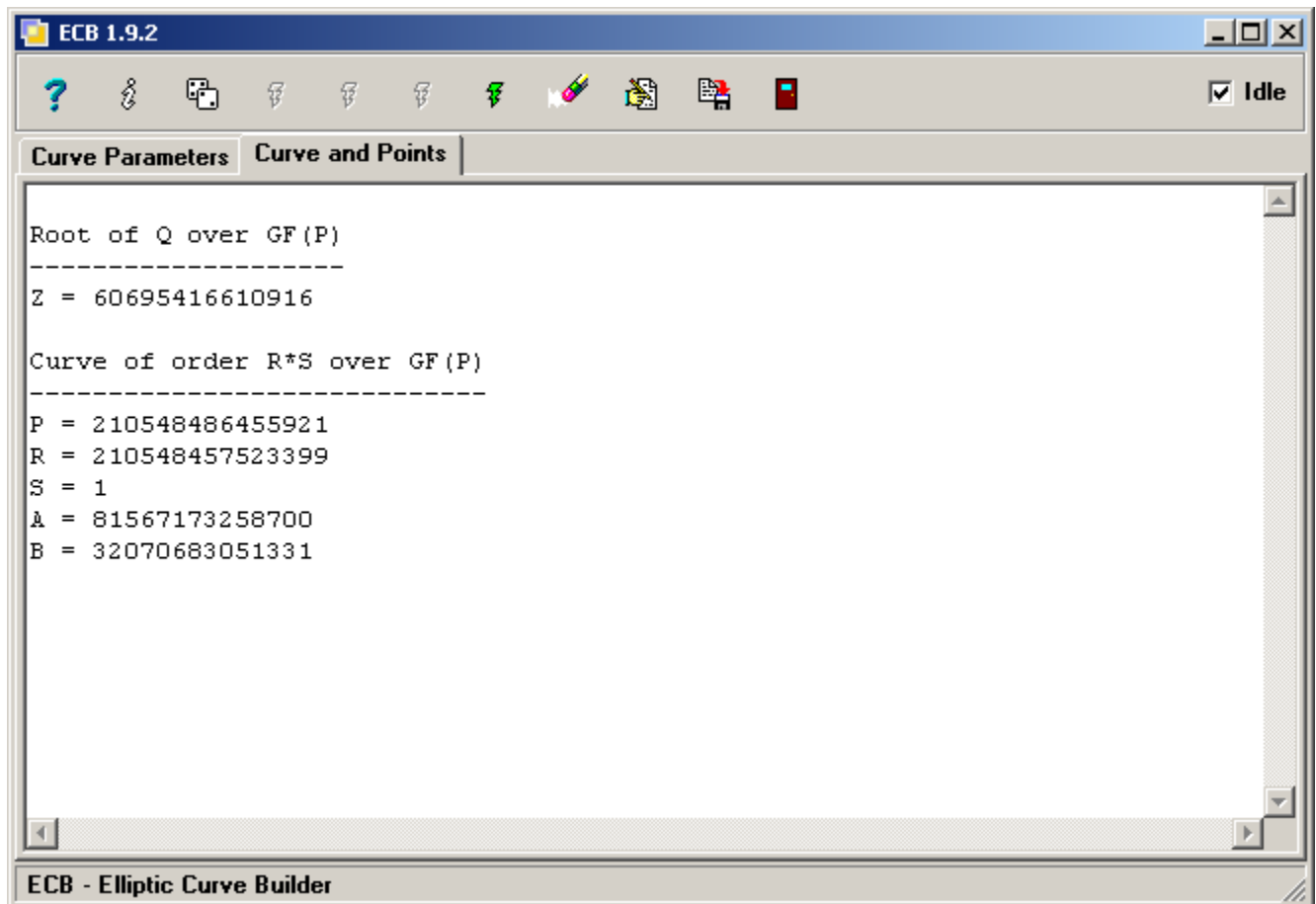
Generate a new curve by clicking the yellow lightning bolt.



The results are shown below. If **A** or **B** (or both) is negative, Crypto++ will fail to validate the curve because the parameters are not within the interval $[0, p-1]$. This is a Certicom requirement specified in SEC-1, Domain Parameter Validation, Section 3.1.1.2.1.

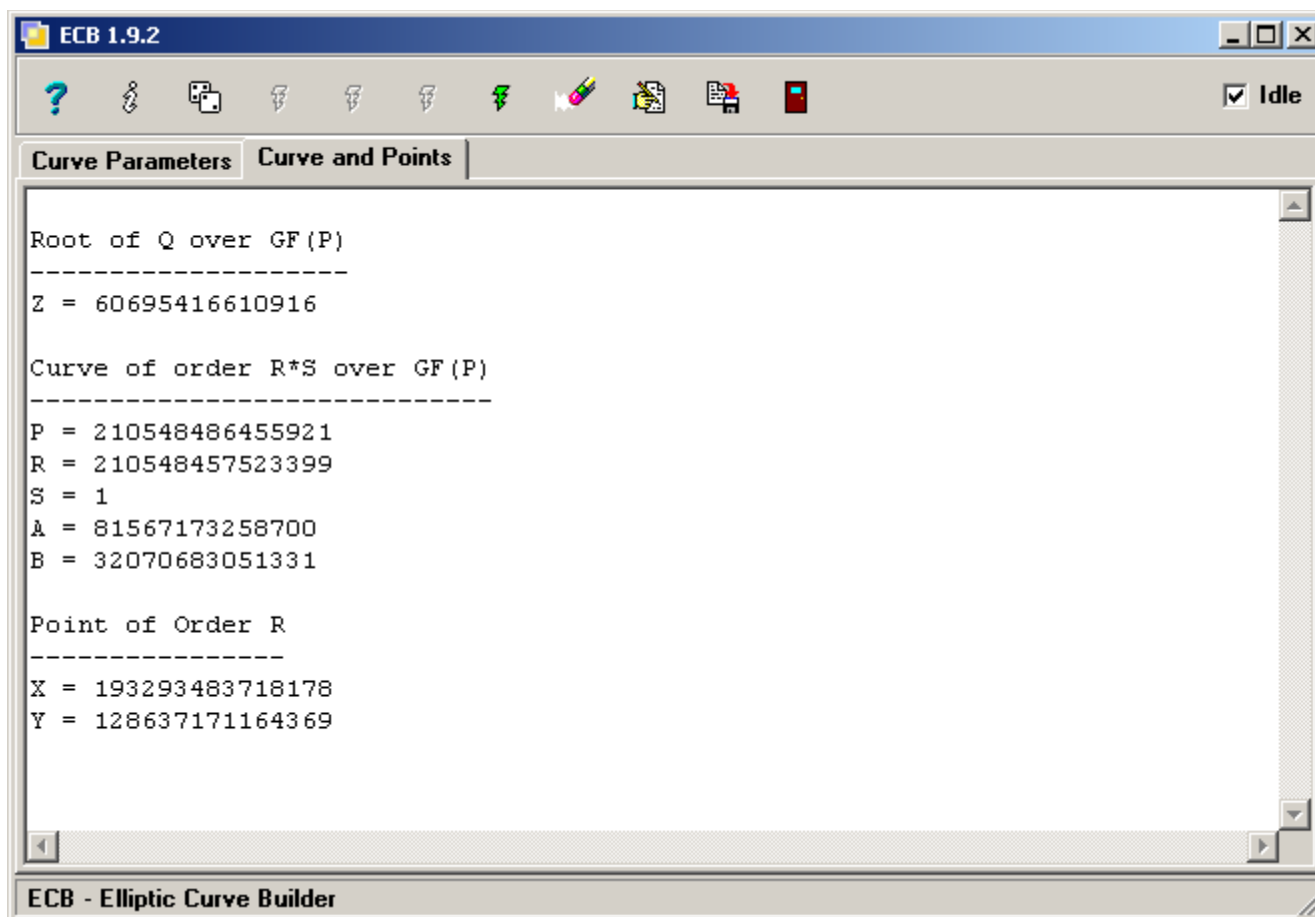
If either parameter **A** or **B** is negative, we can perform one of two actions. First, we can regenerate the curve. Generate new curve parameters by clicking the erase toolbar button and repeating this step. Second, we can perform the following in the source code:

```
Integer a("...");  
Integer b("...");  
  
a %= p;  
b %= p;
```



Generate a Point on the Curve

Generate a point on the curve by clicking the green lightning bolt. Note that the point is of *Order R*, meaning it is a subgroup generator.



Domain Parameters

The tuple $\{ p, a, b, G, n, h \}$ is collectively referred to as *Domain Parameters*. Both parties must agree on the domain parameters. The final step to use the ECB generated values is to translate the terms used to describe the domain parameters. The table below offers the translation between ECB and Crypto++.

Term	Some Literature (Crypto++)	Other Literature (ECB)
Modulus	p	P
Coefficient A	a	A
Coefficient B	b	B
Base Point	G	X and Y
Group Order	-	U
Subgroup Order	n	R
Cofactor	h	S
Private Key	x	-
Public Key	Q	-

Curve

The curve is specified by the *Modulus* and *Coefficient A*, and *Coefficient B*.

Base Point

The *Base Point* is also referred to as the generator or subgroup generator; and is sometimes denoted as

- G
- g_x, g_y
- x_g, y_g
- $G(x,y)$

Group Order

Group Order is the number of points on the curve. *Group Order* is often denoted $|E|$, $\#E$, and $\#E(F_p)$ in the literature.

Subgroup Order

Subgroup Order is the number of points in the subgroup. Subgroup order is $\#E/h$.

Cofactor

Cofactor is the number of small additional factors which accompany the large prime factor for the curve. If the cofactor is 1, then the entire group of points is being used (ie, $\#E$). Otherwise, the number of points being used is $\#E/h$.

Certicom's SEC-1 specifies that $h \leq 4$. In addition, all NIST recommended curves have a cofactor of 1, 2, or 4 (refer to Digital Signature Standard (DSS) (http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf), Appendix D, Section 1.1.1).

Crypto++ and Domain Parameters

To verify that Crypto++ uses **n** as the *Subgroup Order*, examine eccrypto.h near line 135. There does appear to be a slight deviation in that the *Cofactor* is saved in `m_k`, and not `m_h`.

```
template <class EC>
class DL_GroupParameters_EC : public ...
{
    ...

    OID m_oid;           // set if parameters loaded from a recommended curve
    Integer m_n;          // order of base point
    mutable Integer m_k;  // cofactor
};
```

Crypto++ does not store the *Group Order*. Crypto++ uses the *Cofactor* and *Subgroup Order* to calculate the *Group Order*. We can verify this in pubkey.h near line 600:

```

class DL_GroupParameters : public CryptoParameters
{
    ...

    Integer GetGroupOrder() const {return GetSubgroupOrder()*GetCofactor();}
};

```

Private and Public Keys

Given the domain parameters, it is possible to generate a private key and derive a public key. The private key, which is an integer, is referred to as **x** in Crypto++ and FIPS literature; and **d** in Certicom literature. The private key is also referred to as the private exponent. Finally, The public key **Q** is an element on the curve such that **Q = xG**.

Though the private key element is sometimes referred to as an exponent, it is readily apparent from **Q = xG** that the group operation is multiplication. The reasons are pedantic and find their origins in RSA and the exponent associated with a RSA private key.

Crypto++ does not provide constructors which can be used for initialization. Instead, Crypto++ provides overloads on the *Initialize* method for both the public key and private key.

Private Key

There are four overloads for private keys. Two of the overloads accept a random number generator as a parameter. The two PRNG overloads cause Crypto++ to generate the private exponent **x** or **d**. The remaining two private key overloads are used to load a previously generated private key (that is, a private exponent is not generated). Always validate a private key using level 3, which is the strongest validation available.

The Crypto++ code to initialize a private key (and generate the exponent) using ECB generated domain parameters is as follows.

```

Integer p("210548486455921");
Integer a("81567173258700");
Integer b("32070683051331");
Integer gx("67205560863566");
Integer gy("128637171164369");
Integer n("210548457523399");

ECP curve( p, a, b );
ECP::Point g( gx, gy );

ECDSA<ECP, SHA1>::PrivateKey pdf
privateKey.Initialize( prng, curve, g, n );

bool result = privateKey.Validate( prng, 3 );
if( !result ) { ... }

```

To retrieve the private exponent, perform the following:

```

const Integer& x = privateKey.GetPrivateExponent();

```


To load a private key given **x**, perform the following:

```
ECP curve( p, a, b );
ECP::Point g( gx, gy );

ECDSA<ECP, SHA1>::PrivateKey privateKey;
privateKey.Initialize( curve, g, n, x );

bool result = privateKey.Validate( prng, 3 );
if( !result ) { ... }
```

Public Key

For public keys, two overloads are provided. In addition, Crypto++ provides *MakePublicKey* which can be called on a private key to generate a public key. As with a private key, validate the public key using level 3.

The public key is derived from the private key as follows.

```
ECDSA<ECP, SHA1>::PrivateKey privateKey;
...

ECDSA<ECP, SHA1>::PublicKey publicKey;
privateKey.MakePublicKey(publicKey);

bool result = publicKey.Validate( prng, 3 );
if( !result ) { ... }
```

We can also use *AssignFrom* to derive the public key from the private key.

```
ECDSA<ECP, SHA1>::PrivateKey privateKey;
...

ECDSA<ECP, SHA1>::PublicKey publicKey;
publicKey.AssignFrom(privateKey);

bool result = publicKey.Validate( prng, 3 );
if( !result ) { ... }
```

To retrieve the public key element **Q**, perform the following:

```
const ECP::Point& q = publicKey.GetPublicElement();
const Integer& qx = q.x;
const Integer& qy = q.y;
```

```
const Integer& qx = publicKey.GetPublicElement().x;
const Integer& qy = publicKey.GetPublicElement().y;
```

To load a public key given **Q**, perform the following:

```
ECP curve( p, a, b );
ECP::Point g( gx, gy );
ECP::Point q( qx, qy );

ECDSA<ECP, SHA1>::PublicKey publicKey;
publicKey.Initialize( curve, g, n, q );
```

```
bool result = publicKey.Validate( prng, 3 );  
if( !result ) { ... }
```

Retrieved from "http://www.cryptopp.com/w/index.php?title=Elliptic_Curve_Builder&oldid=14268"

This page was last edited on 24 March 2017, at 00:32.

Content is available under [Crypto++ license](#) unless otherwise noted.