Reverse Engineering of Strong Crypto Signatures Schemes

Data	by Evilcry	
05/11/2006		**Published by Quequero**
Where ones sees a limit	Qualche mio...eventuale commento sul tutorial :)))	the others sees an opporunity
Prepare for what you cannot see, expect the unexpected, a waiting game waiting to see......	**WebSite:** **http://evilcry.altervista.org** **E-mail:** evilcry DoT gmail At com **IRC frequentato** **irc.azzurranet.org** **#crack-it on efnet #RET**	http://www.reteam.org
Difficult	**NewBies** () **Intermediate** (X) **Advanced** (X) **Master** ()	

Reverse Engineering of Strong Crypto Signatures Schemes

Written by **Evilcry**.

Introduction

This paper will have the usual classical style of a CryptoReversing Approach, what we are going to talk about is ECC also known as Elliptic Curve Cryptography. After a theorial study we will fly to the most common Secured Software Applications with a touch of Hardware Securityware.

Tools used

- IDA
- Ollydbg
- A good background of math 'n Cryptography

Index

Target

KeygenMe 10 by WiteG

Essay

Basic Intro

Why should a Reverser should study Cryptography?..many people "erroneously" have the bad "abit" to consider these two disciplines as isolated, but as you will see in the Professional or Recreational Reversing: the analysis of the most easy and unknown algorithm is done in the same manner, with the same basic assumptions and concepts for the inveribility. In Cryptography we will deal with most complex mathematical systems and with more "refined" reversing techniques but..as you will see the Resolution Pattern will always be the same.

Today many many Professional Protections base their security upon Crypographic Algorithms of various kinds, and also a big part of the future SSHR (Security-Software/Hardware-Research) will be directed (verso) the realization of **Complex Systems** of crypthographical algorithms, mantained by Purely Mathematical algorithms, and little by little we will need more complex **Algebraic Attacks**.

Actually the research is working on new Full Cryptographic Chips, Hardened USBs and other Hardware that are built to run a specific Crypto Algorithm.

It's necessary to make some other distinctions when we talk about Hardware Cryptography. We have two common systems for the "Efficient" Computation of an Algorithm:

ASIC Devices: **Application Specific Integrated Circuit,** that are specifically built for Maximum Risk Applications, their power is that the Efficiency and algorithm can't be changed after the producion.

FPGA devices: **Field Programmable Gate Arrays**, which contain arrays of computational elements obtained with a restricted set of instructions. These elements are called Logical Blocks and are connected with a set of *Routing Resources* that could be programmed.

In this last period many security studies on FPGA, revealed to us some Complex Attacks that could be performed. I'm talking about *Techniques of FPGA Exploitation* but (un)fortunately this field is not for all :)

Finally..some consideration..there are many yet implemented Algos, such as: AES, RSA and ECC. The most interesting in my opinion are the ECC, because they offer a level of security similar to RSA with truly little KeySizes, and this is a big quality for Hardware devices, ECC are also fast, and are necessary little Certificates! (a foundamental point in mass hardware..PDA, SmartCards, Phones)

Therefore, knowing the possibilities of analysis and reversing on software, we can use the same methods of analysis in order to work on the hardware.

Introduction To Elliptic Curve Cryptography

The Public Key Cryptosystems most used, are ones based on factorization (RSA) and upon the **Discrete Logarithm Problem** (Diffie-Hellman, ElGamal, Schnorr, DES). These Algos make possible, trusted communiations over insecure channels. There are various alternative secure communication systems, one of the is the **Cryptography Based on Elliptic Curves** or more easly **ECC.** This system became Main Stream thanks to the numerous advantages and *flexibility* that ECC offers!. We have many proposed Elliptic Curves for PKC, some of them based on the factorization problem, others on DLP. It's important to talk about the fundamental differences between the two "frameworks". Factorization is essentially an Academical and except little KeySizes there are no big differences with RSA.

More intersting, is ECC based on DLP, because the security of these algorithms depends on the redefinition of the classical algorithms used for common DL problems. This different implementation of classical DLP, drive us to a redefinition of **Exponentiation**, that we can call **Sub-Exponentiation Time**, if applied to the **Resolution of Elliptical Curves**. If we look at more general algorithms specifically built for ECC, we have an **Exponential Time**. Sintetically, aspects of the same agorithm assumes different terms, if referred to the "EFFICACIA" which they have on DLP or ECDLP.

The beautiful story of ECC, begins in 1984 thanks to **Hendrik Lenstra**, who coded a factorization algorithm based on the mathematical proprieties of Elliptic Curves, called **Lenstra Elliptic Curve Factorization**. The true ECC, born in 1985 by **Neal Koblitz** and **Victor Miller** that reimplemented the already known algorithm upon algebrical structures as **Elliptical Curve Math** and **Finite Fields.**.

Basic Math Background - Group Theory

I've decided to make an approximatively complete discussion of ECC, so in this little chapter i've inserted some elements of Group Theory.

An Abelian Group (G,*) is a set G with assigned binary operation:

$$* = G \times G \to G$$

that has the following proprieties:

- **Associativity** *a * (b * c) = (a * b) c* for all the *a, b,c* included in *G*
- **Identity Element**, exists an identity element *e* included in *G* that *a * e = e * a = a* for all the *a* included in *G*
- **Existence of Inverses**, For each *a* exists an element *b* called *Inverse* of*a*, so *a * b = b * a = e*
- **Commutativity**, *a * b = b * a*

There are basiclly two groups, Additive (+) and Multiplicative (*). These two distinctions come from the fact that a group is called additive when his identifying element *a* is 0, while the inverse is *-a*. Multiplicative groups are so called when the identitifying element is 1, and it's inverse is *a^(-1)*. Finally a group is called Finite when G is a finite set, in this case we also have an Order which pratically defines the number of elements of G.

For example, with a fixed prime number p, we can easly build a finite group of order p, $\mathbb{F}_p$ *= {0,1,2,...,p-1},* obtained from a set of integers. For the precedent observations, we can assume two kind of groups *(* $\mathbb{F}_p$*, +)* in other words an additive group of modulus p with identity element equal to 0, and also a group *(* $\mathbb{F}_p$**, *)* where $\mathbb{F}_p$* denotes not-null elements from the set that we have considered and as you can see is a Multiplicative group of order p-1 and identity element 1.

At this point you may think that there exists a group that contains both Additive and Multiplicative operations?..so we have only one mathematical object?..the answer is Yes!! *(* $\mathbb{F}_p$*,+,*)* It is also called Finite Fields. Now you should know that we can define G as a Multiplicative Finite Group of order n and we can also introduce new elements typical of Finite Fields as the the g element, that the most little positive integer given from t and defined as:

$$g^t = 1$$

called Order of g, whose direct and most important consequence is to exist always and to be a divisor of n. Another truly important property of groups is the "chain effect", we can indeed define a set as:

$$\langle g \rangle = \{g^{(i)} : 0 \le i \le t - 1\}$$

in other words the set of all powers of g, which (as you should understanded) is by "itself" a group or better a SubGroup of G, and is called Cyclic SubGroup of G generated by g. For the nitation of Finite Field, all that we have said it's true also for G writted with Additive Rules, or more precisely the order of g is the most little positive divisor t in n, or better:

$$t * q = 0$$

and consequently:

$$\langle g \rangle = \{ i * g : 0 \le i \le t - 1 \}$$

The Added notation t*g assumes the sense of element obtained by adding t copies of g, we can also resume with only one definition: If G has a g element of order n, then G is a Cyclic Group and g will be called Generator of G.

Finite Fields Arithmetic

Finite Field Aithmetics is the foundamental basis of each system that uses Elliptical Curves, mainly in cryptography, the correct implementation of all algorithms over Finite Fields, is the most important step to determine the efficiency and security of an ECC System. There are principally three kind of finite fields:

- Prime Fields: formed by prime numbers.
- Binary Fields
- Optimal Extension of the Field

For each of these fields exists growing implementation difficulties in the sorted order that you can see. It's obvious that it is also necessary to implement different algorithms for each kind of Field.

But all algorithms follow one common fundamental concept, the Execution of Aritmethic Operations, INTO and BETWEEN the fields, tecnically working as Mixer/Connector or as Single Operators.

The Fields, are abstractions or better SubSets called F of the various numerical systems that we know. Into fields are principally possible only two operations, Addition and Multiplication and they have exactly the same properties of the Groups.

In other words the possible operations with Fields are four, indeeid we can add Subtraction and Division, as derived operation types the two principal. Subtraction is defined in additive terms as: *a - b = a + (-b),* while the Division is defined in terms of Multiplication, as *a/b= a * b ^ (-1)* as $b \neq 0$ e *b ^ (-1)* inverse element, the element that respects the relation *b * b ^ (-1)=1.*

Finite Prime Fields: Are written as $\mathbb{F}_p$, where it is a prime number > 3, defined also as the modulus of $\mathbb{F}_p$, for each integer *a*, (*a MOD p*) we will have a remainder *r* between *0* and *p*, the inverse operation, necessary to find r is called **Modular Reduction**.

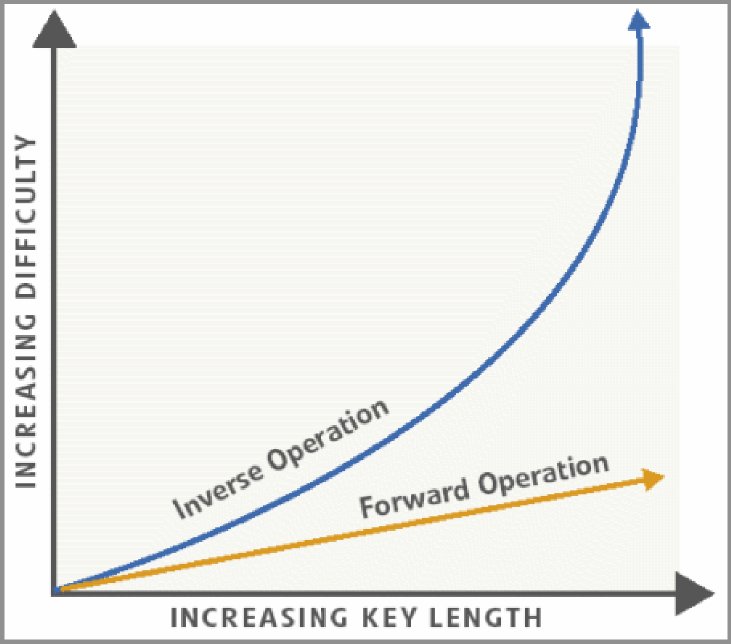
Let's consider for example the field F29, it's elements will be

F29={0,1,2,...,28}

So we can define 4 basical arithmetic operations

- 17+28 = 8 which corresponds to 37 MOD 29 = 8 **(Addition)**
- 17-20 = 26 which corresponds to -3 MOD 29 = 26 **(Subtraction)**
- 17 * 20 = 21 which corresponds to 340 MOD 29 = 21 **(Multiplication)**
- 17^ (-1) = 12 which corresponds to (17*12) MOD 29 = 1 **(Inversion)**

With this little example, we can see how arithmetic field operations basically work and we can also put our attention to an important observation: **the use of finite fields is the principal method to reduce computational complexity in terms of efficiency**, indeed as you should have noticed the simple properties inside an addition (17 + 28 = 8) could have enormous "potential" if used in cryptography!



As you can see in this graph of a generic **Asymmetric** (the case of Elliptic Curve), you should notice that given a good (long) key-length, a computational inversion is too hard, but at the same time, x key-length is easily "workable" by normal computers (forward operation).

Binary Fields: Binary Fields are written as $\mathbb{F}_{2^m}$, and also called Finite Fields of Order 2. They could be cosiderated as a vectorial space m in F2 defined by the elements 0 - 1. From basic notions of Linear Algebra exist: m elements a, that could be defined with a combination of linear independent vectors that originate the a base (m-1). We will consider this "special" Set as a BitString and over this we will define basical arithmetical operations. The Addition corresponds to the XOR between two BitStrings, Multiplication depends on the chosen base. There are many Bases that could be used in $\mathbb{F}_{2^m}$, but the use for computational scopes is reduced because it was discovered that some bases are less efficient that others. The choice could be done between Polynomial Bases and Normal Bases. We will work only with the Polynomial Representation. An irriducible polynome f(z) of degree m is chosen. Irriducible means that f(z) can't be factored as product of polynomes of degree < m. Here are the $\mathbb{F}_{2^m}$ properties:

Let's consider a binary field F2^4, it's elements are 16 polynomes of max degree 3:

0	*z^2*	*z^3*	*z^3+z^2*
1	*z^2+1*	*z^3+1*	*z^3+z^2 +1*

z	*z^2+z*	*z^3+z*	*z^3+z^2 +z*
z+1	*z^2+z +1*	*z^3+z +1*	*z^3+z^2 +z+1*

Possible operations are:

- *(z^3+z^2+1)* + (z^2+z+1) = *z^3+z*
- *(z^3+z^2+1)* - (z^2+z+1) = *z^3+z* (considr that in F2 (-1 = 1))
- *(z^3+z^2+1)* * (z^2+z+1) = *z^2+1* **so** *(z^3+z^2+1)* * (z^2+z+1) = z^5 + z + 1 **FOLLOWS** (z^5 + z + 1) MOD (z^4+z+1) = *z^2+1*
- *(z^3+z^2+1)*^(-1) = z^2 ovvero *(z^3+z^2+1)* * z^2 MOD (z^4+z+1) = 1

The second member of MOD (z^4+z+1) corresponds to f(z)=z^4+z+1

Generalized Discrete Logarithm Problem

Let's now study the Discrete Logarothm: we will also see a practical application of Group Theory. In each system based upon the DL we can find some Paremeters of Public Domain (p, g and q) where p is a common prime number, q a divisor (also prime) of p-1, q have a range [1, p-1] and Order q, so we can say that t = q is the smallest value that verifies the following relation:

g ^t = 1 (mod p)

Now you should see this by other points of view :), indeed if we make some assumptions we can redefine the entire encryption process of DL! suppose indeed that (G,*) is a cyclic multiplicative group of order n that has a generator g, we can "include" the entire algorithm DL into the same G!!. If we consider that Public Domain Parameters are g and n, automatically the private key is an integer x, randomly chosen into the range *[1,n-1]* given by:

y= g^x

The problem to determine x, given g,n and y is defined as **Discrete Logarithm Problem** (DLP) in G, and a DL system based on G *should* be untractable but there are some conditions that make it's efficiency attackable. Every two Cyclic Groups of the same order n, these can be considered operatively as the same groups. In other words, we have two identical boxes with a different (((contents))), as immediate effect we can represent the same object in different forms, as computational consequence we will obtain for each representation different efficiency curves (velocity), independantly from DL or DLP.

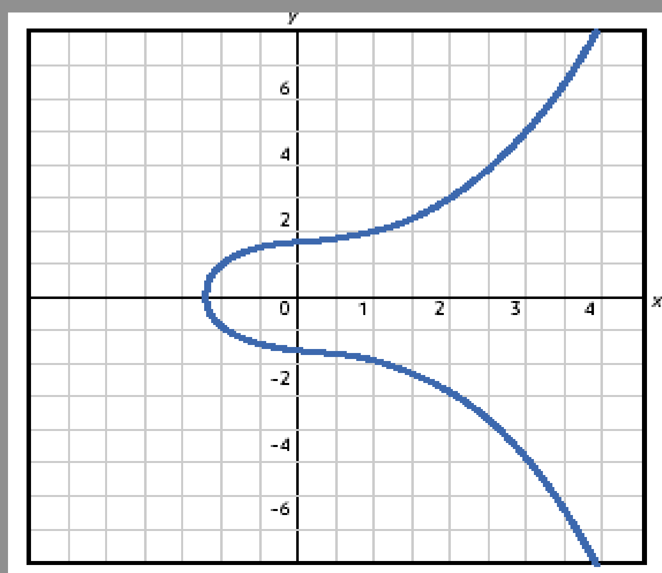
What about DL and ECC?..easy.. we always work with Finite Fields, so researchers rewrite DL into ECC terms :)

What Are Elliptic Curves

An Elliptic Curve is a plane curve given by: ***y^2 = x^3 + ax + b***

The principal property (related to cryptographycal scopes) of this kind of curves is that the Set of the points of this curves formes an Abelian Group, which has as it's Identity element the Infinite. If a curve's coordinates are extracted by a Finite Field sufficiently big, the set of their solutions will form an Abelian Finite Group. You should also remember the DL could be considered as a Set of Finite Cyclic Groups, and the jump to a group of points into an Elliptic Curve is short (if you have clear the previous assumptions) but with a fundamental difference, the Increased Complexity, that is the point of force of ECC.

This is is the plot of an Elliptic Curve obtained by *y^2 = x^3 + ax + b*:



Elliptic Curves in Cryptography

Basically the Elliptic Curve is only the mathematical architecture because its algebraical properties are used to define the elements of the Set from which is computed the Group. Consider a graph-plot obtained in the plain p x p, where p is as usual a prime number, obviously the Field $\mathbb{F}_p$ that we obtain will go to 0 from p-1; so algorithmical operations will converge into thje points that respects the appartenence condition to $\mathbb{F}_p$. Elliptical Curves used in cryptography could be of two classes: the First: $\mathbb{F}_p$ (with p > 3) and the Second with $\mathbb{F}_{2^m}$ In the case of a generic appliation: $\mathbb{F}_q$ (called Extended Optimal Field) where q = p. Into $\mathbb{F}_p$ elements are essentially integers $0 \leq x < p$ derived from Modular Aithmetic operations. The $\mathbb{F}_{2^m}$ applications are the most complex cause the number of possible representations (the same efficiency concept of DL and DLP) as a bitstring of each irreducible polynome f(z) of degree m.

The couple of affine coordinates (x,y) with $x, y \in \mathbb{F}_q$ generates an affine plane $\mathbb{F}_q \times \mathbb{F}_q$, from this specification we can directly obtain the definition of Elliptic Curve E:

An Elliptic Curve E, is the geometrical location of the points of the affine plane, which coordinates satisfy the equation $4a^3 + 27b^2 \neq 0$ *(MOD p)*; which have as Point at Infinity O. In other words the point where the projective plane encounters the line at infinity. In the most *simple* case (p > 3) we will have that the $E(\mathbb{F}_p)$ is the already known *y^2 = x^3 + ax + b*, where a and b belong to $\mathbb{F}_p$. Let's to a little practical example to clarify :)

Consider an Elliptic Curve in F7, you will have as defining equation: *y^2 = x^3 + ax + b*, so the points will be:

E(F7) = {Infinity, (0,2), (0,5), (1,0), (2,3), (2,4), (3,3), (3,4), (6,1), (6,6) }

Now let's consider $E(\mathbb{F}_{2^m})$ whose defining equation could be written as: *y2 + xy = x3 + ax2 + b* where *a* and *b* comes obviously from $\mathbb{F}_{2^m}$ and are constants, with $b \neq 0$ for *O*=(0,0) and in other cases *O*=(0,1); thanks the **Hasse Theorem** over elliptic curves, we can quantify the number of points into an elliptic curve by using the following relation: $(\sqrt{q} - 1)^2 \leq |E(\mathbb{F}_q)| \leq (\sqrt{q} + 1)^2$.

Elliptic Curve Arithmetic

All cryptographical mechanisms are based on the Elliptic Point Arithmetic, that is the fundamental, practical instrument used to attack/implement ECC. As previously said the points of an Elliptic Curve constitute an abelian group $(E(\mathbb{F}), +)$ that has as usual O as point defined at the Infinity, this point has the role of Additive Identity, taken two points $P, Q \in E(\mathbb{F}_q)$ we will have a Third Point defined as P+Q over $E(\mathbb{F}_q)$ obtaining as consequence $P, Q, R \in E(\mathbb{F}_q)$

Points, as previously said are the elements of an Abelian group, so we can define a set of operations defined in G that have O as identity element:

- *P + Q = Q + P*
- *(P+Q) + R = P + (Q + R)*
- *P + O = O + P = P*
- *-P* such as *-P + P = P + (- P)= O*

In light of this property, we can introduce two Fundamental Operations over ECs, that are a truly important part..so open your eyes ;)

Elliptic Curve Addition: We will use two approaches one Geometrical and one Algebric to better understand what is the real meaning of addition over EC. Exists a rule called of Cord and Tangent that allows us to sum two points of an elliptic curve defined as $P, Q \in E(\mathbb{F}_q)$ thanks to which we can obtain a Third Point:

P + Q = R

R is a point of the Elliptic Curve, we can also see (from the Table of Operations) that by defining the negative of *P = (x,y)*, we will have *- P = (x, - y)* for $P \in E(\mathbb{F}_p)$ and *- P = (x,x + y)* for $E(\mathbb{F}_{2^m})$, so we can define the following rules for the addition:

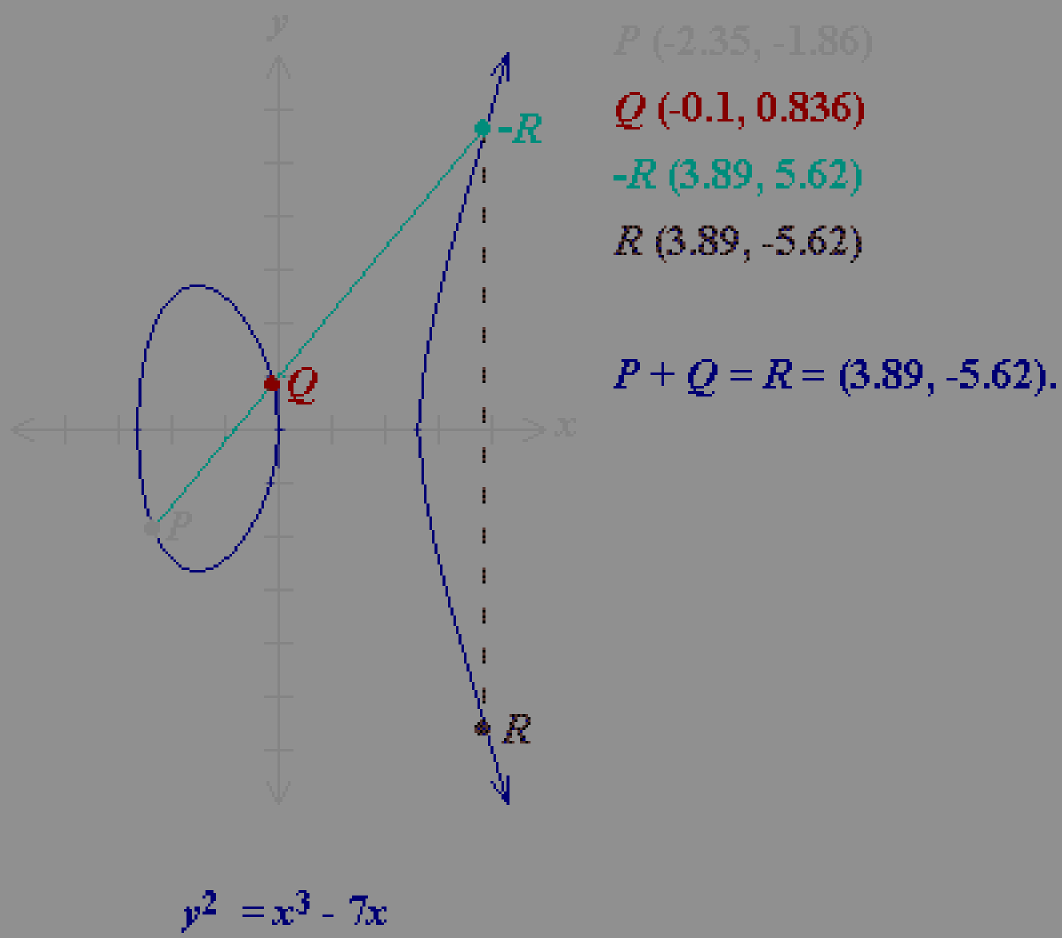
- if *Q = O* then *P + Q = P*
- if *Q = - P* then *P + Q = O*
- if $Q \neq P$ then *P + Q = R*

For this last case we can distinguish between:

- **Prime Fields** : *xR = l^2 - xP - xQ, yR = l(xP - xR) - yP* where *l* = $\lambda = \frac{y_Q - y_P}{x_Q - x_P}$
- **Binary Fields**: *xR = l^2 + l + xP + xQ + a, yR = l(xP + xR) + xR + yP* where *l* = $\lambda = \frac{y_P + y_Q}{x_P + x_Q}$

You can see algebrically $E(\mathbb{F}_q)$ we will obtain a Group with O identity Element..in other words an ECC!

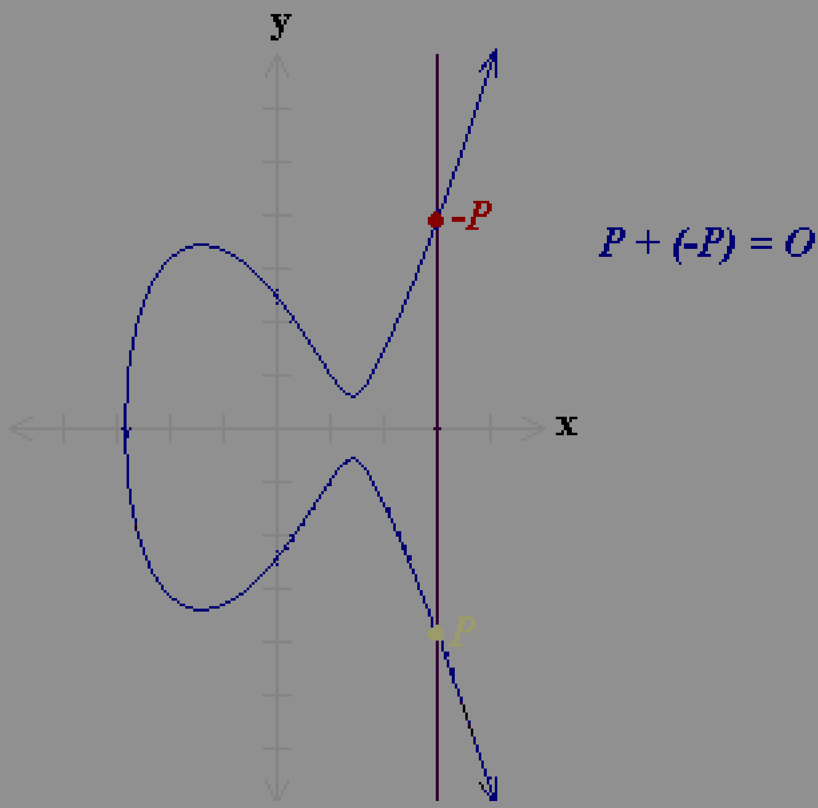
Let's consider now the **Geometrical Approach**. Consider P and Q as two points of E, trace a line that passes from P and Q until it intercepts the Elliptic Curve. Obviously this will reveal a point that Reflected over the x axis will reveal our R(*x3*,*y3*)



This is only one of the possible Addition cases, now we will see the second case:

P* - *P* Addiction**: ***P* + (-*P*) = *O

The jointing P -P is a vertical line that obviously does not "generate" a third point R, so it's extension reaches infinity finding the point O, ***P* + *O* = *P***



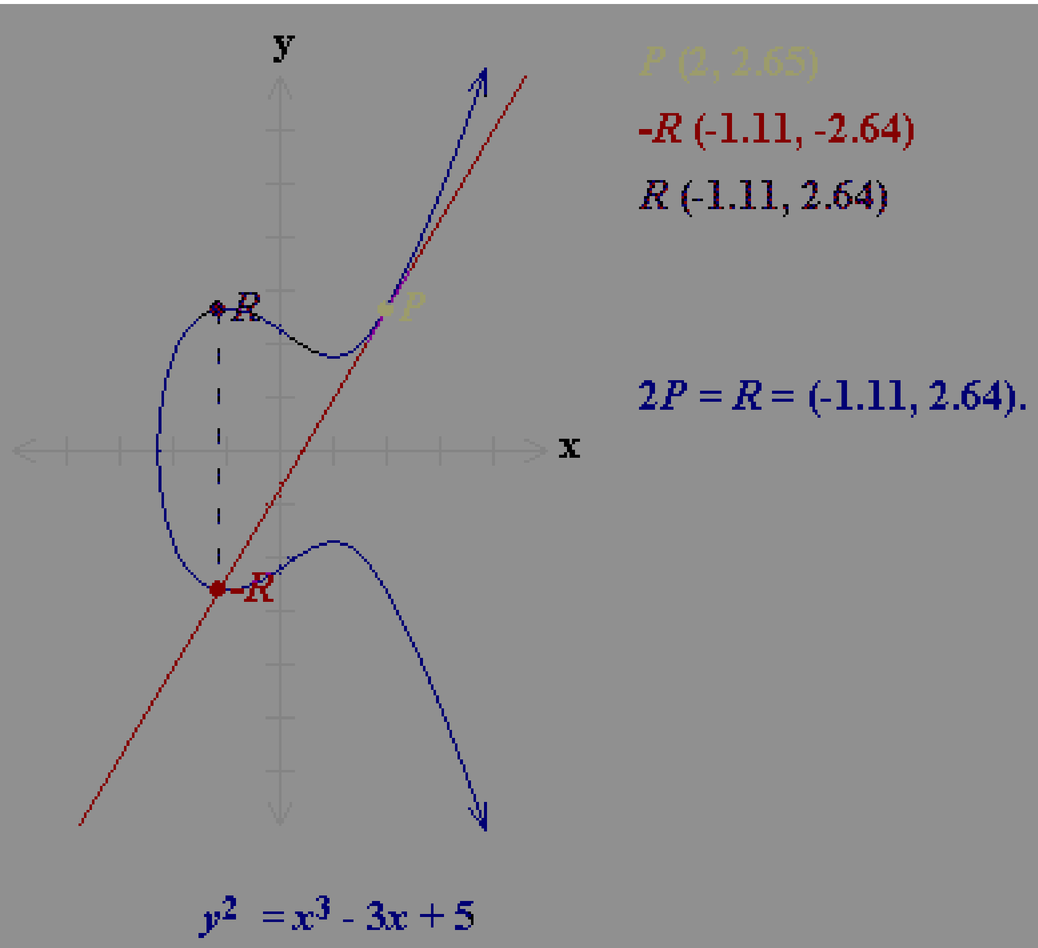
$y^2 = x^3 - 6x + 6$

You will find further references over EC-Addition in the Reverse part

Duplication of *P, 2P*: Duplication can be writted as:

2P = P + P = R

From the geometrical point of view, this is equal to building the tangent to the curve into the point P. It's prosecution will reveal a point that reflected over x will give us R point:

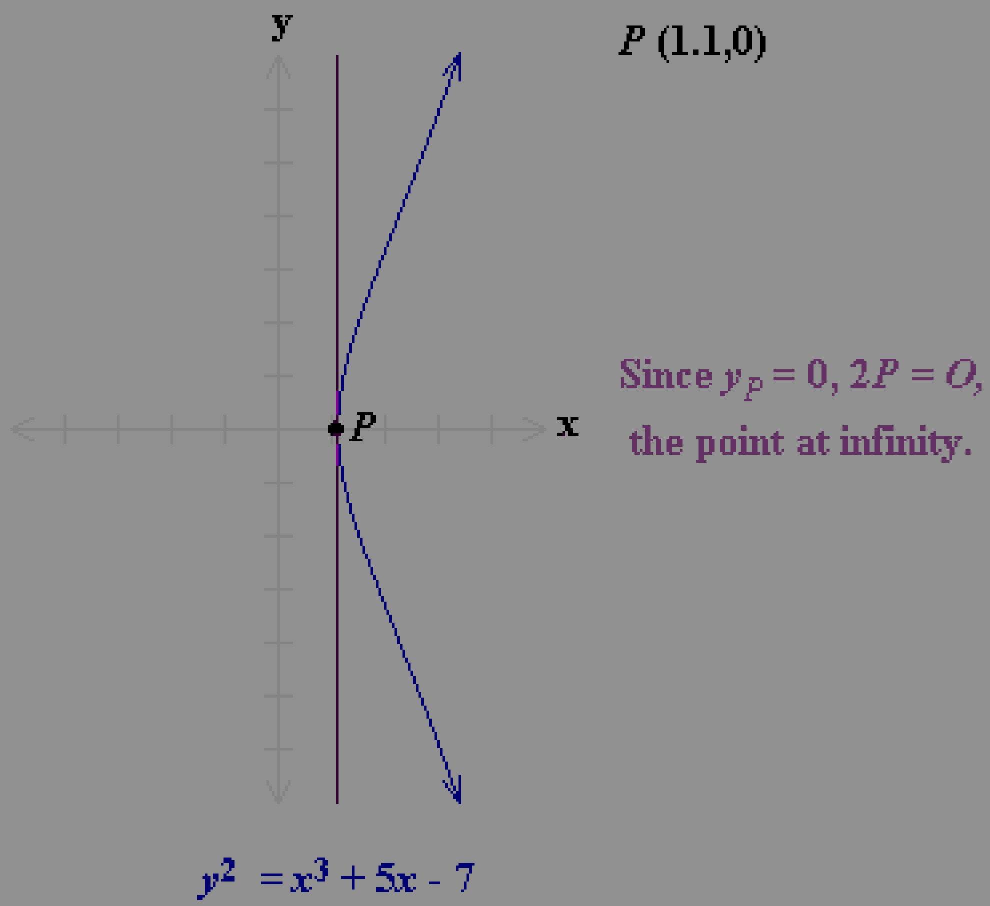


Now come back to some analytical consideration:

- **Prime Finite Field**: *xR = l^2 - 2xP, yR = l(xP - xR) - yP* where *l* = $\lambda = \frac{3x_P^2 + a}{2y_P}$
- **Binary Field**: *xR = l^2 + l+ a, yR = xP^2 + (l + 1) x_R* where *l* = $\lambda = x_P + \frac{y_P}{x_P}$

Duplication of P, if yP = 0: The tangent that passes from P is ALWAYS the vertical if the component yP = 0, consequently if we duplicate a point with this yP, we will obtain a tangent to the entire Elliptic Curve that obviously will never intercept R and only intercept the point O at infinity:

2P = O



If we want to find 3P(always in the case of yP=0) the path is:

3P = 2P + P, where P + O = P

4P = O, where 2P + 2P = O + O = O

5P = P, where 4P + P = 4P + P = O + P

etc..

Pratical Elliptic Curve Operation Samples

With this last part, you reach a little basical reverser point of view of ECC :), let's see two pratical samples of how EC operations work over $\mathbb{F}_p$:

Elliptic Curve Addition: let's consider the point *P*(4,7) e *Q*(13,11), ***P + Q = (X3,Y3)*** that will be determinated as follows:

X3 = ((11 - 7) / (13 - 4))^ 2 - 4 - 13 =

= 3^2 - 4 - 13 = -8 =

= **15** MOD 23

For y:

Y3 = 3(4-15) - 7 = -40 =

6 MOD 23

and finally *R* = (15,6)

Elliptic Curve Point Doubling: let's consider the same point *P*(4,7), ***2P = (X3,Y3)*** that will be determinated as follows:

X3 = (check the Lambda for Point Doubling)

((3*4^2 + 1) / (14))^2 - 8 =

=15^2 - 8 =

= 217 =

= **10** MOD 23

Y3 = 15(4 - 10) - 7 =

= -97 =

=**18** MOD 23

Our R, will be **R = (10, 18)**

Obviously the 23 comes from the field that we chose, F23 ;)

.

Fundamental Features of EC, and pratical ECC generation

Order of the Group: Considering the curve $E(\mathbb{F}_q)$**,** thanks to the **Hasse Theorem over Elliptic Curves** we can know the number of points of E included O, here the fundamental relation of Hasse:

$$|\sharp E(\mathbb{F}_p) - p - 1| < 2\sqrt{p}$$

Computationally, this is resumed into the **Schoff** algorithm known also as **SEA** (**Schoof-Elkies-Atkin**).

The knowledge of the number of points of E is fundamental for cryptography, in terms of implementability and of security, consider indeed a sum P+P+P+... sufficiently big, and remember that we're working on Finite Fields, easly we can reach the point O and this is NOT a good thing, because:

*a*P = b*P* for some *a, b* with *b > a* this demostrates the existence of ***c*P = O*** where ***c = b - a*** !!!

ECC Generation

This is nothing more than the Order of the Group, that as you should understand CAN divide the order of the group itself! So it's truly important to generate an ECC with a precise order, and we can do this by using Schoof algorithm (in the previous example obviously this algorithm is used indirectly ;)) or with more complex algorithms, as **Complex Multiplication** or the more immediate **Theorem of Weil**, instead if we're working on Hardware devices we can use the systems of **Point Counting** as **AGM** (**Arithmetic Geometric Mean**) and **SST** (**Satoh-Skjernaa-Taguchi**) that are the most performant at spreading especially over Binary Fields.

Other important conditions for a good ECC is the **MOV Condition** (**Menezes Okamoto e Vanstone**), without this you can almost reduce the complexity in a logarithmic form! :)

MOV Condition: Suppose we have an ECC over GF(q) and a point fixed in F, firstly we check that the first T terms of the sequence *(q, q^2, q^3...,q^T*) are different from *1 mod Prime_Order of F*. T usually is chosen as

T = log2(q)/8

For the entire attack procedure, you can search on CACR website.

Also remember that a field of q elements should have an order different from q or we will have an **Anomalous Elliptic Curve,** and that's truly insecure!.

Elliptic Curves Cryptographic Applications

And now finally we reached the pratical and most interesing part, what could be the uses of our ECC:

- **Use strictly crypographycal**
- **Integer Factorization**
- **Primality Proving**

The fundamental assumpion of EC Cryptography is, *how readapt a certain algorithm based over group theory with groups based EC Theory*. Here some of the most common ECC Algorithms:

- **ECDSA**
- **ECDH Elliptic Curve Diffie-Hellman**
- **ECMQV basato su MQV**
- **ECIES**
- **EC-KDSA**

Basic ECC Architecture Considerations

Every implementation of EC applied on crypto algorithms has a basic architecture common to all algorithms, the fundamendal structure is the **Domain Parameters** procedure.

Domain Parameters: This structure defines the elliptic curve E in function of the **Field** a **Basic Point G** and the **Order n** of the field.

Usually is pointed out as : **D = (q,FR, S,a,b, P,n,h)**

- *q* the order of the Field chosen
- *FR* (Field Representation) denotes the representation used for the elements of $\mathbb{F}_q$
- *S* (Seed) in the case the EC needs a PRNG
- *a,b* the two coefficients of the defining equation
- *P* org *G* is the **Base Point** or **Generator** (is a cyclic SubGroup), to be cryptographically usable the Order of *G* needs to be the lesser prime number not negative such that $nG = O$.
- *n* is the order *n* of the Base Point
- *h* is the **Cofactor**

Because 'h' is the order of $E(\mathbb{F}_q)$, for the **Lagrange Theorem** $h = \frac{|E|}{n}$, into cryptographycal operation the cofactor needs to be $h \leq 4$ and usually is used $h = 1$.

The Domain Parameters are the most vulnerable elements of an ECC, and are the first parameters that an hypothetical attacker will go to check, it's important from the security point of view that Domain Parameter respects the prescriptions of **NIST** and **SECG;** the most classical attacks are the *Pohlig-Hellman* and *Pollard's rho,* strictly dependent from the number of points of an elliptic curve.

Key Pairs: Are the keys used into ECC, and are strictly related to Domain Parameters, the **Public Key** is chosen casually by selecting a point Q into the group *<P>* generated from the point *P*, while the **Private Key** *d* is extracted from the following relation:

d = LOGp (G)

The elliptic curve discrete logarithm problem and ECC Attacks

The intractability of the DLP is of fundamental importance for the security of every ECC algorithm. We talked briefly about the fact that the problems connected to the DLP are the same for DLP over EC, which is called **ECDLP**, and can be defined as:

Given a curve E defined over $\mathbb{F}_q$ *and a point* $P \in E(\mathbb{F}_p)$ *of order n, and a point Q appartaining to <P>, we have to find the integer I included in the range [1,n-1] so that Q = l P. This integer i it's indeed the Discrete Logarithm of Q in base P denoted l = LOGp (Q).*

So you understand how important the chosen Domain Parameters are as necessity to resist against all attacks based on ECDLP. The most known algorithm for ECDLP resolution is the **Exhaustive Search**, that computes the sequence of points *P, 2P, 3P* until Q is found, the principal disavantage of this algorithm is the low velocity, indeed the Running Time is approximately of *n,* so an *n* choised sufficiently big it's a good countermeasure against this kind of attack. Other important attacks are **Pohlig-Hellman** and **Pollard's rho** that have an Exponential Running Time, precisely of *O(Sqrt(p)),* where *p* is a prime number sufficiently big, to be protected against this attack is necessary to choose an n divisible for p, such as step-*Sqrt(p)* which will be unusable cause the increase of computational time.

The basic mechanism of Pollard's Rho it's easy, we have to following random steps (better defined as **Random Walk**) until *ax, ay, bx ,by* are found:

*= ax*B + bx*A == ay*B + by*A*

*= ax*l*A + bx*A == ay*l*A + by*A*

*= (ax-ay)*l*A == (by-bx)*A*

*= (ax-ay)*l == (by-bx) mod NP*

= l == (by-bx)(ax-ay)^(-1) mod NP*

(if you're confused with letters just refer to the last crackme analysis of this paper)

There are also other category of attacks, called **Isomorphism Attacks** that try to reduce ECDLP to DLP, most known are **Weil and Tate Pairing Attacks,** these kind of attacks can be used only in presence of **Anomalous Prime Fields**.

The Elliptic Curve Digital Signature Algorithm

The ECDSA is the corrisponding DSA over EC, I chose ECDSA because is the most diffused in SW Protections, it's also the most standardized (**ANSI X9.62, FIPS 186-2, IEEE 1363- 2000** e l' **ISO/IEC 15946-2**). Let's study the algorithm step by step:

ECDSA signature generation
Input: D = (q,FR, S,a,b, P,n,h)*;* Private Key *d;* message *m.* **Output:**(r,s) 1. Select *k* pertaining to *r [1,n-1].* 2. Computes *kP = (x1, y1),* next converting *x1* into an integer *X1.* 3. Computes *r = X1 mod n*. If *r = 0,* reselect *k* 4. *e = H(m).* 5. Computes *s = k^(-1) * (1(e +d*r)) mod n*. If *s = 0* recomputes *k.*

ECDSA signature verification

Input: D = (q,FR, S,a,b, P,n,h); Public Key *Q*; message *m*, Signature *(r,s)*.

Output: Accept / Refuses Signature

1. Verify that *r* and *s* are integers included into the range *[1,n-1]*, in case the condition is FALSE return Rejected Signature.
2. $e = H(m)$.
3. Computes $w = s^{(-1)} \bmod n$.
4. Computes $u1 = ew \bmod n$ and $u2 = rw \bmod n$.
5. $X = u1P + u2Q$.
6. If $X = Infinity$ return Rejected Signature.
7. Converts the coord *x1* of *X* into an integer *X1*.
8. Computes $v = x1 \bmod n$.
9. If $v = r$ then Signature is Valid, else Rejected Signature.

H() is a generic hash() function which usually is used in *SHA* or *SHA-1*.

A Reverse Engineering Approach

Now the Reversing part!, as target I chose crackme 10 of WiteG which implements ECDSA Signature. We will go directly on the ECC part without other techical explainations of the cm itself.

This crackme implements ECDSA by using MIRACL, it's important to say that truly professional software will never use MIRACL because with few functions you have a fully working ECC Architecture (i'm talking of .NET and CryptoApi).

Let's suppose that we already know that our crackme is based on ECC, so first thing to focus is the Input parameters that the algorithm will receive. We have 4 EditBoxes, Name and three containing serial (probably the three parameter of every common ECC ;)).

We know also that the second fundamental step of every ECC, is the **Domain Parameter Generation**

```
miracl *mip;

mip = mirsys(100,10);

mip->IOBASE=16; //The basis is switched to 16, Name and Serial will be in the form 0..9..A...F

secp160r1_a=mirvar(0); //Coeff a

secp160r1_b=mirvar(0); //Coeff b

secp160r1_p=mirvar(0); //Base Point or Generator

secp160r1_n=mirvar(0); //Order of the Base Point

secp160r1_x=mirvar(0); //Coord x

secp160r1_y=mirvar(0); //Coord y

cinstr(secp160r1_a, "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7FFFFFFC");

cinstr(secp160r1_b, "1C97BEFC54BD7A8B65ACF89F81D4D4ADC565FA45");

cinstr(secp160r1_p, "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7FFFFFFF");

cinstr(secp160r1_n, "0100000000000000000001F4C8F927AED3CA752257");

cinstr(secp160r1_x, "4A96B5688EF573284664698968C38BB913CBFC82");

cinstr(secp160r1_y, "23A628553168947D59DCC912042351377AC5FB32");
```

As you can see our D will be *D(a,b,n,x,y)*, there is something different from the canonical D, does not appear the coord *x* and *y*, but don't worry you 'll see in a bit what this mean..now let's study the true core algorithm:

```
unsigned long lName,lSNX,lSNR,lSNS,snLSB,i,j;
```

```
lName = GetDlgItemTextA(hDlg, EDIT_NAME, szName, 0x40); //lName will
contain the name

lSNX = GetDlgItemTextA(hDlg, EDIT_X, szSNX, 0x40); //X = first serial
inserted

lSNR = GetDlgItemTextA(hDlg, EDIT_R, szSNR, 0x40); //R = second serial
inserted

lSNS = GetDlgItemTextA(hDlg, EDIT_S, szSNS, 0x40); //S = third serial
inserted
```

Ogni curva ellittica necessita di un'inizializzazione:

```
ecurve_init(secp160r1_a,secp160r1_b,secp160r1_p,MR_PROJECTIVE);

epoint* pointG = epoint_init(); //Initialize a point of ECC called G

epoint* pointH = epoint_init(); //Initialize a point of ECC called H

epoint* pointJ = epoint_init(); //Initialize a point of ECC called J

epoint_set(secp160r1_x,secp160r1_y,0,pointG);
```

The first function initializes an Elliptic Curve E of the kind y^2 =x^3 + ax + b mod p, in other words the classical curve that take the name of **Weierstrass's Mode**l, the fourth parameter (`MR_PROJECTIVE`) specifies if we want to use Affine or Projective coordinates.

```
hashing(szName, lName, e1); // e1 = H(Name) review the Signature
Verification

lstrcat(szName, szTag);

lName = lstrlen(szName);

hashing(szName, lName, e2); // e2 = H(Name) this time only the Name that is
united to the static Tag

cinstr(x, szSNX); // x will contain the first serial

cinstr(r, szSNR); // r will contain the second serial
```

```
cinstr(s, szSNS); // s will contain the third serial

snLSB = remain(r,2);
```

This easy piece of code produces two messages e1 and e2, using an hashing function H(). The function remain() instead divides a big number with an integer (2 in our case), obtaining the Integer. Remember the division r/2, necessary to give us the bits of the **Point Compression.**

```
if ((compare(r,secp160r1_n)<0) && (compare(s,secp160r1_n)<0) && (epoint_set
(x,x,snLSB, pointH) == TRUE))

if (point_at_infinity(pointH)==FALSE)
```

And now we landed on the first Signature Verification, the checked condition are three, if (r < n) and (s<n) and if the point H finally pertain to the curve (LSB is referred to the Point Compression resolution) then the execution can continue:

First Check:

```
xgcd(s,secp160r1_n,s,s,s); // s = s^(-1) mod secp160r1_n (s can be the w of
the point 3 of SignVer proc ;))

mad(e1,s,s,secp160r1_n,secp160r1_n,u1); // u1= s*e1 mod secp160r1_n

mad(r,s,s,secp160r1_n,secp160r1_n,u2); // u2= s*r mod secp160r1_n

ecurve_mult2(u2,pointH,u1,pointG,pointJ); // J = u1*G + u2*H
```

This piece of code should sounds you familiar, indeed is the procedure of Signature Verification that we have seen in the points 3 - 4 - 5, the only difficulty may be in the different letters used, but it's only necessary a bit of attention. The next control is again on the point H, that as you should understood needs to be different from Infinity.

```
epoint_get(pointJ,x,x); // x = J.x determine the x component of the point J

if ((compare(x,z)!=0) && (compare(x,r)==0))
```

it's foundamental that xJ is equal between the two messages

Second Check:

```
mad(e2,s,s,secp160r1_n,secp160r1_n,u1); // u1= s*e2 mod secp160r1_n
```

```
mad(r,s,s,secp160r1_n,secp160r1_n,u2); // u2= s*r mod secp160r1_n

ecurve_mult2(u2,pointH,u1,pointG,pointJ); // J = u1*G + u2*H
```

The second check is again performed over H, and finally is again founded xJ

```
if ((compare(x,z)!=0) && (compare(x,r)==0))
```

If also this check is passed the Signature is Correct!

Finally we are in front of a Signature Verification based on ECDSA, our task is to find a Signature(*r*,*s*) and the Private Key (called in this case x) for the two messages *e1*, *e2* that have two hashes of 160 bits.

In conclusion we find ourselves of with one Signature Verification based on ECDSA, therefore to find one Signature (r, s) and the private key (called in this case x) for two messages e1, e2 having two hashes of 160 bits.

There are two checks: one for each message but computationally are equals for both messages, we can indeed see that the kernel of our check is the point J:

$\mathbf{J} = u1*G + u2*H =$

$= w*e*G + w*r*H =$

$= (w*e + w*r*d)*G$

Where $w=s^{(-1)} \bmod n$ and $H = d*G$

The J is equal to the already known *X* used in the canonical scheme:

$\mathbf{X} = u1P + u2Q$

And finally we discover that G and H are the famous *P* and *Q* :)

As you can see, the true control is indeed to verify the abscisses of the obtained points, that need to be identical for the two messages, may look a complicated but remember the basic concept of ECC **Everything is a Point**!

J(x, y) and it's projecion *J(x, -y)* have indeed the same x, so let's build the Resolving Equation:

*w*e1 + w*r*d = -w*e2 - w*r*d*

*= w(e1+e2) = -2*w*r*d*

*d = (n-2)^(-1) * (e1+e2) * r^(-1) mod n*

As you can see these are truly easy operations, the only thing is that is necessary is a little bruteforce for::

*H(r, y1)=k*G*

*J(x, y2)=d*G*

And magically our x is given by:

x = Lsb(y)

This little "trick" may be truly useful in many many protection schemes.

ECDLP and Schoof Solving, for Security Patterns

Now let's study some weakness of Domain Parameters, that make possible the Pollard's Rho / Polhig-Hellman attack to solve DLP.

The resolution of DLP needs necessarly these parameters:

D(P,Q,a,b,p,np)

Obviously an ECDLP attack makes sense if Domain Paramenters are sufficiently little or badly implemented. In other rare cases DLP can be solved with an easy Bruteforce, as in this case:

```
[...]

004013D0 push eax

004013D1 push 0Ch

004013D3 mov [esp+0A0h+var_58], 0

004013D8 mov esi, ecx

004013DA call bytetobig ;Build a Bignumber A, using some letters that cames
```

```
from a serial

[...]

004013E7 push ecx

004013E8 push ebx

004013E9 push ebp

004013EA push edi

004013EB call powmod ;27AB8CB1F847BBBC412CAA33^A mod
C3CEAB06781ECF3B69EA2103

[...]

0040140C push eax

0040140D push ecx

0040140E call _compare ;Powmod == 7DC79E80D9CBBD7DB291643C
```

This piece of code is taken from a crackme truly easy but in the same time you can see the hint to use DLP Braking, a BigNumber A is built using sone characters of the Serial and finally with a Powmod **27AB8CB1F847BBBC412CAA33^A mod C3CEAB06781ECF3B69EA2103**, it compares the result obtained with **7DC79E80D9CBBD7DB291643C**, consequently we have to found the correct value of A, for this we have to use DLP.

We are in the case of the order of **27AB8CB1F847BBBC412CAA33** over the field **F (C3CEAB06781ECF3B69EA2103)** is **C3CEAB06781ECF3B69EA2102 = 2*3*1D*78B*46FA51*89C040BCD81E05** so have sense to use Pollard's Rho and il Polhig-Hellman in combo.

Let's see another case:

```
004013F0 push ebp

004013F1 push esi ;Serial length

004013F2 call inttobig ;Transform the lenght of the serial into BigNum
```

```
[...]

004013FD push ebp

004013FE push edi

004013FF call powmod ;27AB8CB1F847BBBC412CAA33^B mod
C3CEAB06781ECF3B69EA2103

[...]

00401426 push edx

00401427 push eax

00401428 call _compare ;Powmod == 4A2BEE4544261D982D959675
```

The algorithm generates a BigNumber B that contains the length of Serial and as usual executes a Powmod **27AB8CB1F847BBBC412CAA33^B mod C3CEAB06781ECF3B69EA2103** and next step is the comparison with **4A2BEE4544261D982D959675**, what we can do in this case?....Polhig-Hellman.. no..it's not necessary B is a BigNumber, but it's dimension is truly little (it expresses only the lenght of the Serial) and consequently B can be founded with a basical Bruteforce.

After obtaining A and B, two BigNums *X1* and *X2* are generated, and next by using the curve $y^2 = x^3 + x$ into the field **F(ACC00CF0775153B19E037CE879D332BB)** and with A e B, are determined:

$P = X1*A + X2*B$

$c = X2 - P.x$

next by checking that *c* begins with "TMG-" and finally by jointing *c* with Name we will have our serial:

For the resolution *X1* and *X2* are arbitrarly choised:

$P = X1*A + X2*B$

$X2' = c + P.x$

It's now necessary to find *X1'* to satisfy the following **X1*A + X2*B == X1'A + X2'*B** the equivalence criteria is based on the concept that we have to obtain the same abiscis of P, and also that A and B are of the same order, so we have to compute *l* so that $l*A = B$, in other words we have to solve the ECDLP. With Schoof we can also know that the Curve have **ACC00CF0775153B19E037CE879D332BC** points and consequently thanks to **Lagrange Theorem** we have the order of A and B, as:

566006783BA8A9D8CF01BE743CE9995E =
2*3*7*D*D*7F*D3*1DF75*5978F*1F374C47*5F73FD8D3

..and as a fairy tale the good old Pohlig Hellman will guide us :P, it's hint is to compute the ECDLP as **F(2), F(3), F(7), F(D^2), F(7F), F(D3), F(1DF75), F(1F374C47), F(5F73FD8D3)** which is better than computing the entire BigNumber.

fonded *l* = **1212121255555ABCDEFABCDEF9999999** follows that:

*X2*A + X1*B == X2'A + X1'*B =*

*= X2*l*B + X1*B == X2'*l*B + X1'*B*

*= X2*l*B + X1*B == X2'*l*B + X1'*B*

*= X2*l + X1 == X2'*l + X1' mod NP*

ed infine..

*X1' = X2*l + X1 - X2'*l mod NP*

Note finali

A big regard to all **UIC**: Quequero, AndreaGeddon, LonelyWolf, Alfa62, ZaiRoN, LittleLuk, **RET**: Devine9, Black-eye, BigS, Aimless, Haldir, ^Daemon^ and last but not least my Cattina^_^) for all others not mentioned here..sorry :)

Disclaimer