GnuPG Implementation with Elliptic Curves

Sergi Blanch i Torné

Director:
Ramiro Moreno Chiral

v 1.0.5beta1

Documentation of the
EccGnuPG 0.1 branch.

4th March 2006

Contents

1 Introduction **4**
1.1 Objectives 4
1.2 Organization and Planning 6

2 ECC **8**
2.1 Introduction to Elliptic Curves 9
2.1.1 Arithmetic over Elliptic Curves 13
2.1.2 Rule P1363 of IEEE 14
2.1.3 NIST Standard 15
2.2 ECElGamal 16
19
2.3 ECDSA 20

3 GnuPG **21**
3.1 Program structure 22
3.1.1 Autotools 24
3.2 ECC Module 26
3.2.1 Needed algorithmics 27
3.2.1.1 EC Key Generation 27
3.2.1.2 EC auxiliary functions to the Key Generation. 28
3.2.1.3 ECElGamal 31
3.2.1.4 ECDSA 32
3.2.1.5 Closed operation between Elliptic Curve Points (over $\mathbb{F}_p$, with projective coordinates) 33
3.2.2 Mathematical functions 39
3.2.2.1 Existing functions 40
3.2.2.2 New Functions 43
3.2.3 Cryptographic functions 45

3.3 Final result 46

4 Conclusions and Future work **48**

A Key uses **49**
A.1 Key generation 49
A.1.1 Program dump 49
A.1.2 Ecc key 53
A.2 Encryption 54
A.2.1 Program dump 54
A.2.2 Ecc encrypted message 56
A.3 Digital signature 57
A.3.1 Program dump 57
A.3.2 Ecc signed message 60

B Execution procedure **61**
B.1 Key generation 62
B.2 Encryption 63
B.2.1 Encrypt 63
B.2.2 Decrypt 64
B.3 Signature 64
B.3.1 Sign 64
B.3.2 Verify 65

Gratitudes

I am specially grateful with Ramiro, he has shown me the thinks that i have capacity to do and he gives me the opportunity to see this. And also to my friends, the ones who have helped me not to stop.

For the support and help that i receive, i need to make an appointment with the people that give me a lot of help to do this project. People as: Damià Castellà, Jose García, Juan Manel Gimeno, Montse Orpina, David Paniagua. Some of them with little contributions, but really effective; others with decisive contributions, that make easy to obtain the results more quickly.

To the David Shaw, Ricard de Benito, Robert J. Hansen, the ones who help me when i ask something in the developers mail list of the GnuPG. To Mikael Mylnikov, a good hacker how find some bugs and he send a really good reports and solutions.

To all the people, that i do not even know, but they share with the community the improvements done over this project.

Chapter 1

Introduction

The modern cryptography has taken advantage of problems with a lot of centuries of existence to protect the modern communications. These problems are usually named One Way Functions (in the future OWF). With the public key cryptography, introduced by W. Diffie and M. Hellman in the year 1976, the innovation in front of symmetric cryptography is the difference and complementary key pair, in an invocate way. This explanation is if you have one key pair not equals between them and they have a complementary relation, one can decrypt the things that the other had encrypt; the fist one is named private key and it is used to decrypt and/or sign, while the other one is named public key and it is used to encrypt and/or verify a sign. The relation between the key pair (public key/private key) is invocate. Like this we can never find the paradox of two different keys, that was synonymous with regard to only one complementary key.

Previously to the appearance of the elliptic curves, the proportional relation in security terms, offered between symmetric and asymmetric key length was really hard. To offer the same strength in security terms, of a symmetric key of 128 bits into an asymmetric key we need a key over 1024 bits. With the elliptic curves, the difference is as well as 192 bits for the asymmetric key into the 128 bits of the symmetric one.

1.1 Objectives

The Elliptic Curve Cryptographic schemes are tested and cryptanalysted for a longtime. This schemes are extensions of the very known modern schemes, but the Elliptic Curve Cryptosystems are mounted over different mathematical bases and, the beginning, ECC were stronger than another

modern schemes.

The ECC due to it flexibility to all knowledge attacks and it little key measure, it is the best scheme for the systems that it have hard limitations of computation and/or bandwidth. Now, this scheme is very extended over hardware systems, for example the smart-cards.

In the software implementations field this scheme has had very small expansion, and it lives in a background. For this reason, the majority of the users of the strong cryptography do not have, in their most extended softwares, of the possibility of Elliptic Curve uses.

Werner Koch began the implementation of a software, based on OpenPGP, that he should set free under GPL license, named GnuPG (Gnu Privacy Guard). This software is the free software option before the commercial software written, some years before, by Phil Zimmermann, named PGP (Pretty Good Privacy). Now, the functionalities of both softwares (GnuPG and PGP) are similar, but GnuPG do not implement any patented algorithm.

The werner's project has gone as far as its maturity as software with the stable versions 1.2.x. This software was created together the developer team: Matthew Skala, Michael Roth, Niklas Hernaeus, Rémi Guyomarch i Werner Koch, and actually maintained by the team: David Shaw, Stefan Ballon, Timo Schulz and Werner Koch, and the contribution of the developers mail list members <gnupg-devel@gnupg.org>.

GnuPG software offers tools of strong cryptography, over modern schemes (except patented), for example the ElGamal scheme for the cipher tool, and the DSA scheme for the sign tool. Moreover, this software has support for RSA cryptographic system and some symmetric Cryptosystems. The supported symmetric systems are: 3DES, CAST5, BLOWFISH, AES, AES192, AES256, TWOFISH. When we use this software in the cipher side, we use hybrid Cryptosystems. That is to say, the program uses a session symmetric key (default AES) and it uses public key scheme to cipher the symmetric key to all the receivers. All the public keys have a list of preferences about the symmetric algorithms to use, and hash or compression systems.

The present project implements the schemes of the public key cryptography with elliptic curves over the free software named GnuPG. The objective, at the users side (as well as the present users of GnuPG, as much as the users of another cryptographic softwares, including the new users that do not have a definitive software chosen now) is to offer the possibility to use a new strong cryptography option over elliptic curve. My option allows all people to have the robustness that this system brings to the public key Cryptosystems.

1.2 Organization and Planning

To organize and make the plannification of this project, i have divided it in two main blocks. The first one, is theory about the elliptic curves, and the second one is about the public software that we use and the union with the own module. In this documentation the made division is the same that first division.

The organization at the project making time have one division more. To simplify the realization of one implementation like this, it is not enough to program, we need to have a background about elliptic curves, and we need an initialization in this. In the documents, we presuppose that the reader has an idea about the mathematical base over the one we move into.

In the first part, destined to learning and documenting about the characteristics and the properties of the elliptic curves, i needed four months. In this period, we did during the realization of the subject named *Theory of the Information and the Codification* (*TIC*)[1]. At this time, i have a deep knowledge about the mathematics tools that the elliptic curves use, and the cryptographic importance of this system, now we could have to go into the search the way to realize this implementation, and with what thinks can do this.

To start a really good, competent and trusted implementation, the first step that we need to do is to search a standard. The standard we need, has been strongly cryptanalysted for the cryptographic community. What we search in this step is protect itself to the probably mistakes, because a lot of people has worked for a long time to find this. In a parallel way to this specification task, we can realize an analysis of the host program. The reason for this analysis is to find the knowledge about the functionalities and the procedures of itself in the way of module insertion task. This task needs to read a lot of programming code, and the following of the execution, but it includes the compilation steps. We can not forget the main task which is the module insertion to the program, and this includes a compilation and an explanation to the program to it have new functions.

To work in this second block of tasks, i have spent a long time of dedication, really a long time. The reason of this time, it is not a special complexity of the ask, fate because the plannification are really spacious.

There are a lot of important things in this second block: the programming language choice to use. The best options are, and they have a heavy weight: C and C++. The second language, and the object oriented programming,

[1] This subject not only study cryptography, but it is an important part

are apparently ideal to work with the data structures and operations that we need. With the operators overloading, we can obtain a similar code as the mathematical writing of the operations. But there are two reasons to tilt the balance to the C language. The first one is the integration of this new module in the GnuPG software, because all the codes of this program are written in this language. It is logical that the addition code will be more refined if this is clearly integrated with the environment. The second reason, is the influence of this project. The *UNIX* community, and this are the expressions used in a response of a query made to the developers of the program, use this language. The C language is in the register of all the computers programmers, if we scorn this language we will exclude the community how can check the module.

Now, it only remains the third block and the last one. This block offers the tangible results, well, it makes things observables same as documentation and code. This point, is the point in which we start programming. Based on the previous blocks, the search of the standard algorithms, we could make a collection of this. But not all of this algorithms we can use, because they can not adapt to the program gratefully, and we need to make modifications over this or remake them. These modifications are always made with a lot of caution in the way of not to generate weakness because we work in a really paranoid science field. We make a compendium of algorithms because later we use them to make the source code, and we make structures in different descendant levels. This descendant levels programming are the best, the clearest and the strongest way to programming. At this point, it only remains the programming work: the translation of the algorithmic language to the C programming language. When we finish the programming, we need to debug the source code.

To realize this last block, we planning to dedicate the last semester of the 2002/03 academic course. But the debugging and the interaction between the module and the main software make some complications to do the last tasks. These complications forced to modify the plannification and made this longer, really longer. The main reason of this delay is the insistence of the programmers manual in this program. The correction of one execution error in this long code is a strong work generally solved with tests and provoked mistakes.

Chapter 2

ECC

The Elliptic Curve Cryptosystem shows a new perspective of work with the classical problems of the modern cryptography. The Elliptic Curve Cryptography expound to change the base over that the cryptosystems work.

In the modern cryptosystems emphasize the *Discrete Logarithm Problem* (in the future DLP), the strength of this depends on a multiplicative group (that is to say, the field without the 0), over this group we make the cryptographic operations. These groups as $\mathbb{F}_q^*$ or $\mathbb{F}_p^*$, in the prime case, are always cyclic. We could expound the same problem over a different group that the classic cyclic group. We replace the number of the cyclic group of a subgroup of points over an elliptic curve, and this subgroup have the same properties that the first one (for example it is cyclic), we could expound the same problem over the new field. Depending on which field we make the definition, we have differences with the operations at the lowest programming level, but the operation interface as the same for all the fields.

The new group with the cryptosystem will work, it would not a multiplicative group, and the DLP have not the basic operation in the inversion of the exponentiation. At the time we base the own operations with the elliptic curves, the product of number is converted to a point addition, and the integer exponentiation is converted to point scalar multiplication. The *Elliptic Curve DLP* (in the future ECDLP) is based on the OWF that says: if you have two random points of the curve, it is a really strong computable problem, how many times on of them are added to himself to obtain the other number.

The most important advantage of the ECDLP is the key size, this size is seriously reduced. This reduction is good because makes better efficiency in systems with limitations of computation and/or bandwidth. This added to

a better strength and the speed up of the cryptosystem setup, contributes with a lot of flexibility to the attacks and the cryptanalysis.

2.1 Introduction to Elliptic Curves

The best way to introduce in the elliptic curves is the definition of them, so to introduce the elliptic curves we need to see their definition. But we need to define something more before it.

Definition 2.1.1 *We can define a projective plan over a field* $\mathbb{F}$ *as:*
Considerate the joint

$$\mathbb{F}^3 - \{(0,0,0)\} = \mathbb{F} \times \mathbb{F} \times \mathbb{F} - \{(0,0,0)\}, \tag{2.1.1}$$

and the next equivalence relation:

$$(x,y,z) \sim (x\prime, y\prime, z\prime) \Rightarrow \exists \lambda \in \mathbb{F}^\star : x = \lambda x\prime, y = \lambda y\prime, z = \lambda z\prime \tag{2.1.2}$$

This relation allows to define a quotient that we name as Projective plan*:*

$$\mathbb{P}_2(\mathbb{F}) = \frac{\mathbb{F} - \{(0,0,0)\}}{\sim} \tag{2.1.3}$$

We denoted one point in the $\mathbb{P}_2(\mathbb{F})$ *plan as:* $[X : Y : Z]$*. And the line with* $Z = 0$*, we named it as* straight line to infinite.

We can make an application between point in the affine plan $\mathbb{A}_2(\mathbb{F}) = \mathbb{F}^2$ *and the projective plan. This reciprocal correspondence is:*

$$\begin{array}{ccc} & f & \\ \mathbb{F}^2 & \longrightarrow & \mathbb{P}_2(\mathbb{F}) \\ (x,y) & \longmapsto & [x:y:1] \end{array} \tag{2.1.4}$$

$$\begin{array}{ccc} & f^{-1} & \\ \mathbb{P}_2(\mathbb{F}) & \longrightarrow & \mathbb{F}^2 \\ [X:Y:Z] & \longmapsto & \begin{cases} \left(\frac{X}{Z}, \frac{Y}{Z}\right) & if \quad Z \neq 0 \\ \neg\exists & if \quad Z = 0 \end{cases} \end{array} \tag{2.1.5}$$

It is an evidence, that the projective straight line to infinite does not exist in the affine plan. And also, if we like to write an affine expression to projective coordinates, we can substitute:

$$\begin{array}{ccc} x & \longmapsto & \frac{X}{Z} \\ y & \longmapsto & \frac{Y}{Z} \end{array} \tag{2.1.6}$$

We remove the divisors and reciprocally, we can write a projective expression to affine coordinates, dividing it to the maximum power of Z and then substituting:

$$\begin{array}{ccc} \frac{X}{Z} & \longmapsto & x \\ \frac{Y}{Z} & \longmapsto & y \end{array} \tag{2.1.7}$$

Definition 2.1.2 *Given an elliptic curve $E/\mathbb{F}$ in $\mathbb{P}_2(\mathbb{F})$ with the Weierstrass Normal Form (WNF)*

$$F(X,Y,Z) = Y^2Z + a_1XYZ + a_3YZ^2 - X^3 - a_2X^2Z - a_4XZ^2 - a_6Z^3, \tag{2.1.8}$$

with $a_1, \ldots, a_6 \in \mathbb{F}$ and without singular points, that is to say, such as the equation system:

$$\left.\begin{array}{c} \frac{\partial F(X,Y,Z)}{\partial X} = 0 \\ \frac{\partial F(X,Y,Z)}{\partial Y} = 0 \\ \frac{\partial F(X,Y,Z)}{\partial Z} = 0 \\ F(X,Y,Z) = 0 \end{array}\right\} \tag{2.1.9}$$

does not have solution.

The definition 2.1.2 describes the general equation of an elliptic curve, but when we define an elliptic curve to describe the cryptosystem setup, we use the reduced form.

Definition 2.1.3 *If the characteristic of the $\mathbb{F}$ is not 2 or 3 the WNF equation can be reduced to:*

$$Y^2Z = X^3 + aXZ^2 + bZ^3 \tag{2.1.10}$$

and we can change the no-singularity condition to the equivalent like:

$$\Delta = -\left(4a^3 + 27b^2\right) \neq 0 \tag{2.1.11}$$

The 2.1.10 curve cut the straight line to infinite in:

$$\left.\begin{array}{c} Y^2Z = X^3 + aXZ^2 + bZ^3 \\ Z = 0 \end{array}\right\} therefore\ 0 = X \tag{2.1.12}$$

and, $[0:1:0]$ is the point that we name as point at infinite of this elliptic curve; we notate it as: $\mathcal{O}_E = [0:1:0]$

We can write the equations 2.1.2 and 2.1.10 in the affine plan using a conversion:

$$\frac{Y^2}{Z^2} = \frac{X^3}{Z^3} + a\frac{X}{Z} + b \left\{ \begin{array}{ccc} \frac{Y}{Z} & \rightarrow & y \\ \frac{X}{Z} & \rightarrow & z \end{array} \right| \Rightarrow y^2 = x^3 + ax + b \tag{2.1.13}$$

Definition 2.1.4 *As a joint of points:*

$$\left\{(x,y) \in \mathbb{F}^2 : y^2 = x^3 + ax + b, 4a^3 + 27b^2 \neq 0\right\} \cup \{\mathcal{O}_E\} \tag{2.1.14}$$

We can write it as $E(\mathbb{F})$: we can see next time how we need to do to give a characteristic of an abellian group. But before it, some rudiments about finite fields, that they are the interesting thing of this work.

Definition 2.1.5 *We define the finite field as a field that has a finite number of elements.*

The number of elements that belong to this field is the order of the same. The finite fields that we are interested in are now the fields with a primary order or the power of one prime. However, we know that a finite field of q order is the only. Here we have the paths to follow: the fields defined over a big prime (for example: $q = p^1$) or the fields defined over a little prime and a very big exponent (for example: $q = 2^m$).

Definition 2.1.6 *We define the characteristic of the field, as the number over that it made the base of the power operations and the lower integer such as* $\underbrace{1+1+\ldots+1}_{(p} = 0$ *such as p prime.. And we define the extension grade as the exponent of this power operation.*

$$q = p^m, \tag{2.1.15}$$

where p is the characteristic and m is the extension grade.

In the choice of fields that we would do, the *extension grade* is fixed to 1 and the *characteristic* will is a big prime.

Definition 2.1.7 *We define an odd field as a finite field with characteristic a big prime and the extension grade equal to 1: $\mathbb{F}_p$. This odd field contains the elements $\{0, 1, 2, \ldots, p-1\}$.*

Definition 2.1.8 *We define an elliptic curve over an odd field as a modular congruency with this odd number:*

$$y^2 \equiv x^3 + ax + b \ (mod\, p)\,, \tag{2.1.16}$$

where $a, b \in \mathbb{F}_p$. *The joint of points of the elliptic curve is* $E(\mathbb{F}_p) \cup \mathcal{O}_E$, *and the discriminant no zero:*

$$E(\mathbb{F}_p) = \left\{(x,y) \in \mathbb{F}_p^2 : y^2 \equiv x^3 + ax + b \ (mod\, p)\,, 4a^3 + 27b^2 \neq 0\right\} \cup \{\mathcal{O}_E\} \tag{2.1.17}$$

Once we have a clear idea of the description of what is an elliptic curve, we could describe the cyclic subgroup with that after we do the cryptographic work. As the same case appears with the modern cryptosystems, the expounded problems also have operations over a cyclic group, the only one that change and now is different, is the base over that rest the cyclic group.

Definition 2.1.9 *We define the cyclic subgroup of an elliptic curve as the joint of points obtained from a generator point belong the curve:*

$$\langle G \rangle = \{G, 2G, 3G, \ldots, \mathcal{O}_E = nG\}\,, \tag{2.1.18}$$

where n *is the order of the cyclic group.*

$$n = ord(G) = min\{r \in \mathbb{Z}_{>0} : r \cdot G = \mathcal{O}_E\} \tag{2.1.19}$$

We need a big n, *and* $n \approx \#E(\mathbb{F}_p)$, *because* $|\langle G \rangle| \approx |E(\mathbb{F}_p)|$

There exists a relation between the number of points that belong to the elliptic curve and the order of the cyclic subgroup with which we like to base the setup. This value are named *cofactor*, and we represent it with an h.

Definition 2.1.10 *We define the cofactor as a parameter of the cryptosystem setup that related the number of points belong the elliptic curve with the order of the generator subgroup:*

$$h = \frac{\#E/\mathbb{F}_p}{n} \tag{2.1.20}$$

Now, we know what an elliptic curve is, also we know how the cyclic subgroup is generated over that rest the cryptosystem setup, and even we know the relation between the elliptic curve and the own subgroup. In the own cryptosystem that we will define, the cofactor will be fixed to 1, because is easy to generate the subgroup and this is the biggest, as far as possible.

2.1.1 Arithmetic over Elliptic Curves

The main operation that we need to be in present over the field of the elliptic curve is the two points addition. This added operation is closed in the field. If we construct an elliptic curve over the real numbers ($\mathbb{R}$) there exists a graphical view to see the two points addition. Because of the elliptic curve is a three degree function, if we draw one straight line that cutting the curve over this two points always this cut the curve in a third point; at last if we change the sign at this third point we have obtained the result of the point addition operation over the two first points.

There are some special cases. One is when we like to add one point to itself. Then, the straight line, cut the curve only in two points. If you have this situation, you only need to draw a tangent line at the point and find the other cut point, and the inverted. This situation in a point duplication, and we can not forget we have an elliptic curve, that is to say to this curve has not a singular points, and one point only have one tangent line. Then the point duplication is like $-R = P + P = 2P$.

The other special cases, have to do with the point at infinity. If we like to add one point with it inversion, the line that we draw is vertical line in the affine plan, which only cuts the curve in this two points. The operation result is the point at infinity, but we can only represent it into the projective plan. The result of this is $P + (-P) = \mathcal{O}_E$. Making one round more, in the curve there exist one or three points that this points is they and their inverses. There points cut the Y axis and the tangent line in the affine plan, they only cut the curve in this point. The result is $P + P = \mathcal{O}_E$.

When we have correctly defined the basic operation, we can do this more than one time with the same point. That is to say, we can add one point to himself a determined number of times: now we define the product operation, named scalar product between a point and a scalar number. We can define this new operation with a recursive:

$$\begin{aligned} n \cdot P &= P + [(n-1) \cdot P] \\ [0 \cdot P] &= \mathcal{O} \end{aligned} \tag{2.1.21}$$

To define a field, we will need two operations, and now we have these two operations: the point addition and the scalar multiplication. From this two operations we can see other relations operations, one could be the inversion of one point $P = [X : Y : Z]$:

$$-P = [X : -Y : Z] \tag{2.1.22}$$

We can not loose one's sight, but, that Y is an integer module p, and of course $-Y$ is the modular inversion.

There exist a number, the scalar multiplication of this with all the points is interesting to calculate really fast; this number is number 2. Later on we will see in the scalar multiplication could be reduced to do repeatedly the point addition and the point duplication operations.

With these four [1] operations we do not need anything more to realize all the necessary cryptographic operations.

2.1.2 Rule P1363 of IEEE

Facing to generate an implementation of the Elliptic Curve Cryptosystem, we have searched a standard specification that it was studied hardly and cryptoanalysed, for this reduce the probability to it has several mistakes in this specification. Finally the used specification is from the IEEE, exactly the Rule P1363 of 1999. This rule write the specification of a lot of modern cryptographic techniques, as in the encrypt and in the digital signature. Moreover, it include the mathematical primitives that you need to realize and implementation.

The developing work in this rule started in January of 1994 following initiative of the Microprocessor Standards Committee for the developing a new standard for the cryptography. The project starts in 1994 but it do not make real advanced as far as 1998, when the community likes to contribute, and generates a lot of drafts documents until obtain the definitive version. But this final version does not appear as far as 1999, at the the moment that it made public the last draft of the rule, and jump this version to the definitive.

The result of these years of searching and developing, is a complete efficient and strong rule. It describes the procedures for three cryptographic families (IFP, DLP, ECDLP), from the key generation primitives until the low mathematic primitive functions on the last descendant level. The cryptographic family that we work has specifications over two types of fields than you can define the elliptic curve ($\mathbb{F}_p$ i $\mathbb{F}_{2^m}$), and also over the different point representations that you can choose: we have the description of the functions over the affine coordinates type (x, y), and we have the description for the projective coordinates type $[X : Y : Z]$. For this implementation, we have

[1] They are not four, two; the other two are only particular cases and specially favorable to emphasize, but they are not separated operations.

used fields with a big prime for the characteristic to define the elliptic curve over a projective plan $\mathbb{P}_2(\mathbb{F}_p)$.

2.1.3 NIST Standard

The National Institute of Standards and Technology (further on NIST, it makes up in the U.S. Department of Commerce) published the next year of the last P1363 draft apparition, one standard about digital signature. The reference to find this standard is: FIPS PUB 186-2. This standard is not a standard about digital signature over elliptic curves, but it is the base to develop a digital signature system. With regard to the elliptic curves, this document makes reference to another document of the American National standard Institute (further on ANSI) named X9.62, this documents is an alternative to the rule of the IEEE.

The important thing for us in the NIST standard is in the *Appendix 6*: 'Recommended Elliptic Curves for Federal Government Use'. His appendix, has been published at July of 1999 as an independent document. It is specially interesting to see the recommended values for the cofactor (h), they recommend to assign an extremely little value (only accepted values: 1, 2, 4). The explanation for this, is the relation between the number of the point that the curve has and the order of the cyclic subgroup which we use to work. If the cofactor is one, the cyclic subgroup include all the points of the curve, if it is two the subgroup includes half points, and successively. Also it is important to have a small value, because to find a generator point probability of the cyclic subgroup it is great. For the cofactor with value 1, all the points except the *Point at Infinity*, that have order one, have the order n and they are generators, if the cofactor is value 2 the probability to find the generator is 50%.

The next interesting point of the appendix is a table of equivalence between the key size in symmetric algorithms and the key size in elliptic curve cryptosystems, it depends on the two different fields over you can define the curve:

Symmetric ≡ Length	EC Prime Field $\mathbb{F}_p$	EC Binary Field $\mathbb{F}_{2^m}$
80	$\|p\| = 192$	$m = 162$
112	$\|p\| = 224$	$m = 233$
128	$\|p\| = 256$	$m = 283$
192	$\|p\| = 384$	$m = 409$
256	$\|p\| = 521$	$m = 571$

In this table, we need to pay attention in two things. The first is the strange progression that follow the key size of Elliptic Curve Cryptosystem over binary fields. We decide to work with primary fields, and this thing is only to observe, it does not influence in the own work. The next element to pay attention es the $\|p\| = 521$ of the last line, it is NOT a typographical mistake. If you observe the other values, all of the others are divisible per 32 and it is easy to save and use in a data structures of integer arrays of 32 bits, but this is only an accidental thing.

At this time, it only misses to see the setup parameters that the standard recommend. The first recommendation is to use a pseudo-random curves, because efficiency reasons in the calculus of the curve parameter a is fixed to -3. Then the equation of the curve is:

$$E: \ y^2 \equiv x^3 - 3x + b \ (mod\ p) \tag{2.1.23}$$

Due to the complexity of the calculus of all the terms in the cryptosystem setup at the generation moment (we need to remember the large number of parameters that this cryptosystem has, some of them need a really strong calculus (for example: the order of the cyclic subgroup) and it is recommended to precalculate it) was fixed. The standard we view fixes all the parameters of the cryptosystem setup, and less randomized. The random parameters are only the private key and consequently the public key. We have the intention to offer more randomize degree to the setup do not fixate the generator point G, and fixating all the others.

2.2 ECElGamal

In all the consulted standards, rules, and documentation, it does not appear any useful scheme as ElGamal over the Elliptic Curve Cryptosystem. In the standard [P1363], over that we know that it was a cryptoanalysed specification, presents and explain the Diffie-Hellman key exchange, but never use it to encrypt a session key. The DH scheme is valid for on-line communications, but not for offline ones, like mail. And the other issue is the multiple communication to more than one receiver. Let's see the example of the figure 2.2.1. We need to have the N receivers on-line because we make communication with them. And on the other side, we make N ciphered messages to send. The C_{SIM_B} and C_{SIM_C} cipher-text have the same weight than the *plaint text*; and we need to transferee: K_{S_a}, K_{S_b}, K_{S_c}, before.

In the offline schemes, as we need in mail communication, the cryptographic software makes a symmetric key and encrypt the plain text. Then

Figure 2.2.1: Scheme of a DH communication with two different receivers.

Figure 2.2.2: Scheme of a ECElGamal communication with two different receivers.

the symmetric key are ciphered using public key schemes, and obtain N little asymmetric cipher-texts. When the program makes the final cipher-text, it combines at the top a concatenation of all the little asymmetric cipher-texts, with the ciphered data with the symmetric scheme. At the decrypt time, the receiver search for the own identifier [2] decrypt with its secret key to obtain the symmetric one, to finally recuperate the plain text.

In this case, the sent data is: $C_{ASIM_B} \cdot C_{ASIM_C} \cdot C_{SIM}$. There, C_{ASIM_B} are probably a bit bigger than K_{S_b}, but C_{SIM} is only sent one time and it has the same weight than *plaint text*.

Without any standard scheme, we need to create or modify another one. The work that we did was to transform the classical ElGamal scheme over finite fields, over the new elliptic curve scheme. A deep study of the problem, and the searching of one intuitive solution of this scheme, we like to find really near to the classical ElGamal scheme:

Algorithm 2.2.1 *ElGamal encrypt scheme (with $\mathbb{F}_p^*$ group)*

1. Receive *input*
2. Generate k
3. $a = g^k \ (mod\ p)$
4. $b = y^k \ (mod\ p)$ /*$y = g^x$*/
5. $c = b \cdot input$
6. return (a, c)

Attempt to find all the similarities between this schemes, we can obtain the next one:

[2] It is not necessary to include the identifiers, in this case the program test all asymmetric cipher-texts and use the only one that it obtain a coherent symmetric key.

Algorithm 2.2.2 *ECElGamal encrypt scheme*

1. Receive *input*
2. Generate k
3. $R = k \cdot G$
4. $Q = k \cdot P$ /*$P = d \cdot G$*/
5. $c = Q_x \cdot input$
6. return (R, c)

These two schemes are virtually the same, and we only need to append one rule because the public key size can be smaller than the symmetric one. This rule is not necessary in without curves schemes, because the relation between the key sizes are so hard. In the case of the elliptic curve scheme, it is an absurd thing but, we could use a bigger symmetric key (over 256 bits) that we encrypt with a public key over 192 bits; if we do not apply this rule, we loose the information at the encrypting time and we can not recuperate the original symmetric key to decrypt.

For this rule, we need to inform to the application in the other side of the communication about if the rule is used (without this the other side can not recuperate the information). For this reason we need to send a bit[3] more of information.

To obtain the decrypt scheme, we proceed as the same way to translate the finite fields classical scheme over elliptic curves:

Algorithm 2.2.3 *Decrypt ElGamal scheme*

1. Receive (a, c)
2. $t_1 = a^x \pmod p$ /*$\left(g^k\right)^x = (g^x)^k$*/
3. $t_2 = t_1^{-1} \pmod p$
4. $output = c \cdot t_2$
5. return $(output)$

Algorithm 2.2.4 *Decrypt ECElGamal scheme*

[3]The value of this bit is true if we apply the rule, or false if we do not need to use.

Figure 2.2.3: General scheme of an hybrid communication.

1. Receive (R, c)
2. $Q = d \cdot R$ /*$d\,(k \cdot G)$*/
3. $Q'_x = Q_x^{-1} \ (mod\ p)$
4. $output\ = c \cdot Q'_x$
5. return $(output)$

Virtually, we also obtained the same result, because the only change that we want to introduce is the inversion of the applied rule in the encrypting time.

2.2.1 IFP ElGamal weakness.[4]

In the section 2.2 is described the hybrid schemes drew in 2.2.3. Because of this, the plaintext that the asymmetric algorithm will encrypt has common properties. The smallest symmetric keys should not be less than 128 bits, until the common biggest schemes are over 256 bits.

The ElGamal scheme described in the algorithm 2.2.1 can be divided in two parts. The first part is related to the DLP or ECDLP and is the time to calculate the exponenciations, in a second time it is used one of the first part results to hide the plaintext. Break one of this parts are fatal. An effective attack is described in [DJN].

With the long keys used over $\mathbb{F}_p$ like 1024 bits or more, exist the problem and it is implemented an attack. But also are hard because factorize the cyphertext can be hard or the list of factors are too long. Over elliptic curves, the keys use less bits, between 192 and 521 if the curves are also defined over a finite field $(E\,(\mathbb{F}_p))$.

The diagram 2.2.4 describe the two parts division about we told before. To find the key to resolve the symmetric encryption it is needed to try to break the 'ecc' box or the 'x' box. Break the 'ecc' box is break the ECDLP and it is known that is a hard task, but not the same for the 'x' box. Factorize the value 'c' of at most 521 bits is possible. If the solution are to prime values,

[4] Mikael Mylnikov report it.

Figure 2.2.4: Hybrid scheme with a detail of the asymmetric.

Figure 2.2.5: Proposal to solve the IFP ElGamal weakness.

the solution are finded but the process is slow. If the factorization concludes with a long list of factors the process to find it is so fast.

With this list of factors, will rest a process to combine it

2.3 ECDSA

In the same way to which we obtain a new scheme virtually equivalent over points on a curve based on the classical finite field scheme, we can find to the digital signature a DSA scheme connected to the elliptic curve over the ECDSA name. But now, we do not need to do this, because this scheme is part of the rule [P1363] of the IEEE institution. About this signature scheme, we could see the algorithm 3.2.11; and the same in the verification that it is in the algorithm 3.2.12.

The sign and verification schemes obtained in the rule of the IEEE keep the NIST standard described in [NIST]. Thanks of this keep an international acknowledged standard, we can say the own algorithmic and the code are ok to the use, and moreover it adapt to all the softwares that other people can implement. The NIST standard that we explain, do not say what do you do to make an elliptic curve digital signature scheme, else it explain about the things that the algorithm need to complain, and after the code, that they lake to follow the DSS standard, about digital signature.

Anyway, and without resting importance to this point, the main use of the NIST standard is in the Appendix 6. In this appendix we have a detailed recommendation about what need to complain all the elliptic curve schemes for a governmental use (it understand in the United States, that is the influence zone for this standard).

Chapter 3

GnuPG

Gnu Privacy Guard is a complete and free[1] program based in the standard OpenPGP. This program has the same quality to the popular program Pretty Good Privacy (PGP is originally developed by Philip Zimmermann el 1990), with one advantage: the present program does not use any patented algorithm (IDEA for example), and this can be used without restrictions. This software is based and we can consider it as a programming language translation of the RFC [2440]. Its version 1.0.0 was liberated in September of 1999 and now we have available the version 1.2.3, but in this project we work with the version 1.2.1. GnuPG is an Open Source Software, and we can use it 'without restrictions'. You can use it freely, modify it, and distribute it, always over the terms of the Gnu General Public License (further on GLP).

GnuPG is a good tool for the secure communication and secure storing of data, and we can use it for encrypt data or for the digital signature of this data. The present stable version is named 1.2.x, big or deep modifications do not exist in these versions; the developers only pretend to polish bugs and fix compatibilities. In the versions 1.3.x we use to developing and contain the new tools that they pretend to append in the next stable branch 1.4.x in a non far future. Finally, there is another developing branch named 1.9.x. this is an experimental branch to join the *Aegypten* project with the *gpg-agent* project, the daemon for *smart-card*, and the gpg for S/MINE. For the requirements of multiple platform support, the POSIX compatibility has an immense importance. The code of the 1.9.x branch are based on the code of the actual developers version 1.3.

[1] free in the direction of this is a open source program

3.1 Program structure

The program structure is not complicated. The code is delivered in multiples directories, according to the content code. Evidently, there is a root directory, in which we can find information about the program and the program environment. Especially we need to know that where the '*configure*' and '*Makefile*' files are, and we need to pay attention to the '*configure.ac*' and '*Makefile.in*' files, but we are going to see it more deeply in section 3.1.1.

The main directories we can have anxiety to see, can be put in order depending on the contained importance. The paramount to consult is the '*./doc/*' directory; there we can find the document after that are placed in the Unix manual accessible with '*$ man gpg*', and we can find a describing file named '*./doc/HACKING*'. It is in this last file where we obtain one guide to visit the program structure and the source code. If anybody makes a modification in the program structure, it is in this file the place they need to reflect its change, and for as will be easy to localize.

Directory Layout	
./	Readme, configure
./scripts	Scripts needed by configure and others
./doc	Documentation
./util	General purpose utility function
./mpi	Multi precision integer library
./cipher	Cryptographic functions
./g10	GnuPG application
./tools	Some helper and demo programs
./keybox	The keybox library (under construction)
./gcrypt	Stuff needed to build libgcrypt (under construction)

From '*./doc/HACKING*'.

The other important directories we like to visit are '*./cipher/*', '*./mpi/*', and '*./g10/*'. In the fist one, '*./cipher/*', the name shows that it contains all the encrypt and decrypt modules. But not only there are the asymmetric key modules, but also there are the symmetric key modules. The other type of modules this directory contains are the digital signature modules, the directory name does not indicate it, but it is evident that modules can not reside in another place. As the same, there are resume functions to obtain the hash of the data that we will like to sign.

We need to pay an especially attention to some files of this directory '*./cipher/*'. The main one which we are interested in is: '*./cipher/pubkey.c*';

in this file there is the data structure that enumerated all the public key algorithms (or isometrics) that the program supports to use. If we does not include the own module in this table, the program do not recognize it, no matter how much the code was included in the compilation. We can observe that the other modules are also in the table, and in the beginning of this are included the header files of that. Then we need to append the information in the table and the '#include "ecc.h"' in the beginning.

Other important files, in spite of we do not describe it (because it will be described in the section 3.1.1, or because the file does not have a direct relation with the own module) are '*./cipher/cipher.c*' and '*./cipher/algorithms.h*'. This files have the same objective than the last one, but for the symmetric and hash algorithms, respectively.

In the own module programming task, we need to arrange basic mathematical operations, because we do not use a conventional data structures, but we need arithmetical functions for big numbers and we need to represent it in special structures. This software facilities it, in a mathematical library named '*Libgcrypt*", but there only included the section to the big integers and this library is named '*Multi-Precision Integers*' (further on MPI).

Once we have this information, we can suppose that this directory is saved the mathematical library in the program: '*./mpi/*'. The only thing that we need to comment about this library, is the existence of the proper subdirectories with the assembler code to the different architectures, because of the efficiency motives the low-level operations are made in this language (evidently, there is one generic implementation, for the non contemplated architectures).

Before going to see the main directory ('*./g10/*'), we need the visit another one that it has not importance for the contained code, but the importance resides in the general level of the program. This directory is '*./include/*'. About the contained files, there are four interesting files, but two of them are really important. The main one to the own intention is: '*./include/cipher.h*'. In this there are the list of all the encryption algorithm that this software supports, add more with this identifier number assigned in the RFC [2440] in the section about constants (p, p.49). The next important file to see is '*./include/mpi.h*', because in this we can find a complete list of the primitives that we can use over the big number mathematical library. Finally, the other two files with an importance to see are '*./include/util.h*' and '*./include/types.h*'. In the first one we use in the debugging time, because it contain the name of functions how call to show information in the execution time. While in the second file, the data structure proper of the program is identified.. There are different data structures thought to avoid

problems with the size of an unsigned integer dependent on the architecture (16, 32, 64 bits).

Now we are going to see the main directory in the program: '*./g10/*'. In this directory, there are only two interesting files for us, but these files are the most necessary files to see the results of the own work. We need to modify more files to the correct performance of the program, but this is because of the interface module-program. The first important file, and probably the most important one, is '*./g10/keygen.c*', what this file contains are the tasks to do when we execute the program to generate a new key with the command '*$ ./gpg –gen-key*'. The modifications did where it to allow the access to the own module with the intention create a new elliptic curve key. The other important file is: '*./g10/g10.c*', we can imagine, because the file name, that this is the main file of the program. It is in this one where are made the decision to go to another one to realize the requested task in the command line is made. Finally, the other changed files are: '*./g10/getkey.c*', '*./g10/keyid.c*', and '*./g10/misc.c*'.

3.1.1 Autotools

In the way to create an executable file with all the codes, and given that there are a lot of source files, we can see that to create manually a *Makefile* is a titanic work. To do this, there exist Gnu tools for the applications development that automatize this task. We talk about the Autotools. This development tool provides the needed and really useful *scripts* files to generate the configuration files and process the compilation option. The use of this tools permits to concentrate ourselves in the code, and less the load to think of the subtle differences between different UNIX[2].

The first tool to use offered in Autotools is the *autoconf*. This one produces the script who prepare the code to adapt itself to the present system. The result of these tools is the shell script '*./configure*', to prepare the compilation. The results also we can see in the named file '*config.h*', that contain the preprocessor macros that it will need depending on the platform. The task of the '*./configure*' script is to comprovate the portability to the present platform that we want to install, depending on the necessities of the '*Makefiles*', header files, and other specific. The installer user does not need to modify any file manually.

[2]We can not forget that with this standard programming we can compile over Windows and MacOS platforms, and in this platforms the differences are bigger.

Figure 3.1.1: Autotools Scheme.

The second tool to make easy the compilation is the *automake*. When the installer wants to use the results of this tool, the code was prepared to the present platform. The task that we need to do now is the compilation and linking of the source and the object files. The tools to do this, generate all the necessary *Makefile*. With the *automake* tool is generated the '*Makefile.in*' (that the *./configure*' script need to create '*Makefile*') from the specification written in the '*Makefile.am*'. We need to remember, there are three different '*Makefile**'. The result of the Autotools application agrees with the GNU standards, and the tool *autoconf* is always necessary for the appropriate use.

There exist a third tool in the Autotools pack named *libtool*. In which i will make an appointment only because it is not used in this program. This tool is used to generate the shared libraries, and not to require neither *autoconf* nor *automake*, but sometimes it interact with themselves.

The main virtues of the Autotools are two. The first one, is the work due to simplifies the portability code in the '*./configure*' file; and the second virtue is to simplifies the compilation in applications with a distributed sources, compiling it in a recursive procedure.

Probably all the big application needs the offered help of the *autotools* which are structured in different subdirectories. We saw in the section 3.1 that own program is not an exception and maintains different directories depending to the source destination. Because of this, we can not have only one *Makefile* we have one per directory and the compilation will due in a recursive form. So, the new module insertion does not affect the *./configure* script, because it prepares the system without importance in the number of modules that have the application.

The *Makefile* that we need to introduce the changes to the code insertion is hosted in: '*./cipher/Makefile.am*'. If we edit it, we can see a list of the sources hosted in this directory, and the work that it only needs to do is to insert the module name to this list. This list is labeled with the variable '*libcipher_a_SOURCES*'. This file is not long, it follows a pattern and almost contains the source list.

When we append the name of the own module to the list, we can use the autotools. Because the modification is in the '*./cipher/Makefile.am*' file, and we do not touch anything of the '*./configure.in*' file[3] we do not need to

[3] In fact, the distribution do not contain this '*./configure.in*' file else contain the '*./con-*

execute '*$ autoconf*', and we can go directly to execute '*$ aclocal*' and '*$ automake -a*'. Because the distribution was created, all the needed files are existing.

The autotools are applied, we have all the needed things to make the new distribution. We can execute '*$ ./configure*' to start the program installation.

3.2 ECC Module

We can start the work and talk about the algorithms and the implemented code. Basically, the work is due to two phases. The first one is to collect and create the needed algorithms for all the operations to realize, and the second task is to translate these algorithmics to a higher level programming language.

Starting with the algorithmics, we can distinguish five different types depending on the task that they realize. In a first level we can separate three types: key pair generation algorithms, encrypt/decrypt algorithms, and the digital sign/verification algorithms. In a second level, below the last one we can put the auxiliary algorithms to the up-layer. We can divide this level in two types of algorithms: the key pair generation auxiliary, and the algorithms that they do the closed operations between points of the elliptic curve. This second type, can be considered in a third descending level , because it is used for the algorithms of the all other types, as much in the first group as in the second one.

In the implementation way, we use the high level language '**C**' for two reasons. The first one is the used language n the GnuPG program, because the whole of it is written in this language, and the second reason is the council given by *Robert J. Hansen* in the way of the target audience that can hack this code.

The functions, at the same time, and they come from the algorithmics translation, can be divided it in groups. But we make a different division to the implementation division. The first division difference is between mathematical functions and the cryptographycal functions. And because we have a big numbers library in the program, also we make references to the functions that we need in the programming (evidently we do this in the mathematical functions section).

---

figure.ac' we can not execute '*$ autoconf*' without the first.

3.2.1 Needed algorithmics

Here we relate in detail all the algorithms that later we will implement. There was someone's algorithm that it was extracted form the IEEE standard: [P1363], while other algorithms that they are not present in any standard, they are written here at the first time. Without more preambles we can go to see this.

3.2.1.1 EC Key Generation

The first thing that we will do to put on the new public key cryptosystem is to generate a key pair. To generate it, first of all we will decide in which parameters the cryptosystem works, the same as: all the values that it needs to start:

Algorithm 3.2.1 *Setup Generation.*
Input: Prime of big order 'p'(ord (p)). /‖p‖ ∈ {192, 224, 256, 384, 521}*/*
*Output: Parameters of the elliptic curve (E (𝔽p))./*p (𝔽p);a,b (E (a, b));n,h = 1 (ord(G),cofactor);G*/*

1. Select the curve $E(\mathbb{F}_p)$ over a $\mathbb{F}_p$ in the way $p_{min} \leq p \leq p_{max}$.
2. Fixate the values p,a,b,n.
3. Search the generator G of the cyclic subgroup according to 3.2.4.
4. Return p,a,b,n and G.

At the time the setup is generated, we can create a new private key, and after that the public key:

Algorithm 3.2.2 *Generate key pair (`p1363-A.16.9`)*
Input: Parameters p, a, b, n, and h of one elliptic curve (E (𝔽p)), and the generator point G ∈ E (𝔽p).
Output: Public key P, and secret key d.

1. Generate an integer d in the range $0 < d < n$.
2. Calculate $P = d \cdot G$.

3. Return d and P.

Due to possible mistakes (and why not attacks), we need to verify if the received key is ok or on the contrary its key suffers some disturbances (or malicious modifications):

Algorithm 3.2.3 *Key verification*
Input: An elliptic curve ($E(\mathbb{F}_p)$), and the public key P.
Output: Boolean value of acceptation or rejection

1. Comprovate $P \neq \mathcal{O}_E$.
2. Comprovate $n \cdot P = \mathcal{O}_E$.
3. Return 'true' if all comprovation are ok, in other case 'false'.

3.2.1.2 EC auxiliary functions to the Key Generation.

To realize the operation of the 3.2.1.1 section, we need the subfunctions in one descendant level. And now we start to explain it.

The first that we need is called the algorithm 3.2.1. This algorithm at the same time reposes over another one that later we will be able to see.

Algorithm 3.2.4 *Obtain a generator point with a big prime order (`p1363-A.11.3`)*
Input: A big prime (n), an positive integer non divisible by n (h such than $n \nmid h$), an elliptic curve ($E(\mathbb{F}_p)$)
Output: If $\#E(\mathbb{F}_p) = h \cdot n$, return the point $R \in E(\mathbb{F}_p)$ with the order n. Else, error.

1. Generate a random point $P \neq \mathcal{O}_E$.
2. $R \leftarrow h \cdot P$.
3. If $(R = \mathcal{O}_E)$\{goto 1\}.
4. $Q \leftarrow n \cdot R$.
5. If $(Q \neq \mathcal{O}_E)$\{error\}./*invalid order*/
6. Return R.

It is not difficult to generate a random point in an elliptic curve, we only need to find one integer that it can generate a valid coordinate of the curve. Exactly, our knowledge about if the value can proportionate one valid coordinate, or nor, depends on if we can calculate a square root.

Algorithm 3.2.5 *Find a random point in the elliptic curve (`p1363-A.11.1`)*
Input: A prime integer ($p > 3$), the parameters from the elliptic curve($a, b \in \mathbb{F}_p \mid E(\mathbb{F}_p) = \left(y^2 = x^3 + ax + b\right)$)
Output: A random point over the curve ($P \neq \mathcal{O}_E, P \in E(\mathbb{F}_p)$)

1. Generate a random x such that $0 \leq x < p$.
2. $\alpha \leftarrow x^3 + ax + b \ (mod\, p)$.
3. If $(\alpha = 0)\{$Return $P = (x : 0 : 1)\}$.
4. If $(!\exists\,(\beta \leftarrow \sqrt{\alpha}))\{$goto 1$\}$. /*$\beta^2 \equiv \alpha \ (mod\, p)$*/
5. Generate random bit μ.
6. $y \leftarrow (-1)^{\mu}\,\beta \ (mod\, p)$.
7. Return $P = (x : y : 1)$.

The work of finding the square root of a big integer in a finite field is not an easy task. What makes this work especially complicated, is that can not know if this value exists or not into the finite field. To comprovate if this value exist we could calculate a sequence of numbers named *Lucas values*.

Algorithm 3.2.6 *Find, if it exist, the square root of one number module a big prime. (`p1363-A.2.5`)*
Input: A big prime (p), an integer ($g \mid 0 < g < p$)
Output: If exist:$z = \left(\sqrt{g}\right) (mod\, p)$, else -1 (NULL).

1. If $(p \equiv 3 \ (mod\, 4))\{$/*such that $p = 4k + 3$*/
   (a) $z = g^{k+1} \ (mod\, p)\}$
2. if $(p \equiv 5 \ (mod\, 8))\{$/*such that $p = 8k + 5$*/
   (a) $\gamma = (2g)^k \ (mod\, p)$

(b) $i = 2g\left(\gamma^2\right) \pmod p$

(c) $z = g\gamma\left(i - 1\right) \pmod p$

3. If $(p \equiv 1 \pmod 8)\{$/*such that $p = 8k + 1$*/

(a) Generate a random p' such that $0 \leq p' < p$.

(b) Search the Lucas sequence values

i. $V = V_{\frac{p+1}{2}} \pmod p$ i

ii. $Q_0 = Q^{\frac{p-1}{4}} \pmod p$

(c) $z = \frac{V}{2} \pmod p$

(d) If $(z^2 \pmod p = g)$

i. Return z.

(e) If $(Q_0 > 1 \;\&\&\; Q_0 < (p-1))$

i. Return -1.

(f) else, goto 3.a.

Definition 3.2.7 *The formal definition of the Lucas Sequence is:*

$$V_0 = 2;\ V_1 = p;$$
$$V_k = (p \cdot V_{k-1}) - (q \cdot V_{k-2})\ \ for k \geq 2 \tag{3.2.1}$$

Algorithm 3.2.8 *Generate Lucas sequence values (`p1363-A.2.4`)*
Input: A prime (n), two integers (p′ and q′), a positive integer (k)
Output: $V_n \pmod n$ *and* $Q_o^{\lfloor k/2 \rfloor} \pmod n$

1. Initialize: $v_0 = 2$; $v_1 = p'$;$q_0 = 1$; $q_1 = 1$;

2. $k = k_r k_{r-1} \ldots k_1 k_0$/*$k_r = 1$*/

3. From $i = r$, to $i = 0$, down $--1\{$

(a) $q_0 = q_0 \cdot q_1 \pmod n$

(b) If $(k_i = 1)\{$

i. $q_1 = q_0 \cdot q' \pmod n$

ii. $v_0 = v_0 \cdot v_1 - p' \cdot q_0 \ (mod\, n)$/*$t1 = v_0 \cdot v_1$; $t2 = p' \cdot q_0$*/
iii. $v_1 = v_1^2 - 2 \cdot q_1 \ (mod\, n)\}$ /* $t1 = v_1^2$; $t2 = 2 \cdot q_1$*/

(c) Else{

i. $q_1 = q_0$
ii. $v_1 = v_0 \cdot v_1 - p' \cdot q_0 \ (mod\, n)$/*$t1 = v_0 \cdot v_1$; $t2 = p' \cdot q_0$*/
iii. $v_0 = v_0^2 - 2 \cdot q_0 \ (mod\, n)\}$ /* $t1 = v_0^2$; $t2 = 2 \cdot q_0$*/

4. return (v_0, q_0)

At this point, in the algorithmic view, we can have the key pair of the own cryptosystem. We can start to use it.

3.2.1.3 ECElGamal

We saw how we could obtain the encrypt algorithms in the section 2.2, and now explanations are not necessary. It is exactly equal than the last algorithms, but they are equivalent.

Algorithm 3.2.9 *Encrypt ($A \rightarrow B$). (p1363-7.2.1)*
Input: Public key ($pkey_B$) and numeric plain text (z).
Output: Resultant point (R), cipher (c).

1. Generate key session (k)
2. $P = k \cdot pkey_B.Q$ /*$Q_B = d_B \cdot G_B$;$P = k \cdot d_B \cdot G_B$*/
3. $R = k \cdot pkey_B.G$
4. $c = z' \cdot P_x$
5. Return (R,c);

Algorithm 3.2.10 *Decrypt. (p1363-7.2.1)*
Input: Resultant point (R), cipher (c), and private key ($skey_B$)
Output: numeric plain text (z)

1. Received (R,c)

2. $P = skey_B.d \cdot R$ /*$P = d_B \cdot k \cdot G_B$*/
3. $z = c \cdot P_x^{-1} \ (mod\ p)$.
4. Return (z);

3.2.1.4 ECDSA

In the section 2.3 we make reference to the next description algorithms. These schemes are virtually the same than the schemes that we can find without the Elliptic Curve Cryptosystems, the difference is in the operation realized with the public key due to this case is not a number, it is a point (After the operation, we extract the coordinate x).

Algorithm 3.2.11 *Sign ($A \rightarrow B$) (`p1363-7.2.7`)*
Input: Public key ($skey_A$) and message hash (#hash).
*Output: Number pair (r, s)/*such that $0 < r, s < skey_A.n$*/.*

1. Generate session key 'k'.
2. $I \leftarrow k \cdot (skey_A.G)$.
3. $i \leftarrow I_x$.
4. $r \leftarrow i \,(mod\ skey_A.n)$.
5. If $r = 0$: goto 1.
6. $s \leftarrow k^{-1} \cdot (\#hash + (skey_A.d) \cdot r) \,(mod\ skey_A.n)$
7. If $s = 0$: goto 1.
8. Return (r, s).

In the verification sign, we can see all the operations that we can realize with an elliptic curve point. As the same that in the signature procedure, it needs to calculate the scalar multiplication of a point (two times in this case); but here it needs more and add two points.

Algorithm 3.2.12 *Sign verification (p1363-7.2.8)*
Input: Public key ($pkey_A$), message hash ($\#hash$), and number pair (r, s).
Output: Boolean value of acceptation or rejection.

1. Verifies $(r, s) \in [1, (pkey_A.n) - 1]$.
2. $h \leftarrow s^{-1} \, (mod \, pkey_A.n)$.
3. $h_1 \leftarrow (\#hash) \cdot h \, (mod \, pkey_A.n)$.
4. $h_2 \leftarrow r \cdot h \, (mod \, pkey_A.n)$.
5. $Q \leftarrow h_1 \cdot (pkey_A.G) + h_2 \cdot (pkey_A.P)$.
6. If $Q = \mathcal{O}_E$: refuse.
7. $i = Q_x \, (mod \, pkey_A.n)$.
8. If $i = r$ accept; else: refuse.

There are references made to operations in a lower level in all the algorithms. We describe someone, for example the low operation over the field definition, but there are other algorithms that they need, that will be for the times that it is called, or for the possibility of improvement the efficiency and speed up of this.

3.2.1.5 Closed operation between Elliptic Curve Points (over $\mathbb{F}_p$, with projective coordinates)

In this section there are also made in a descent way to make an easy compression, and remark the importance. The first question that you make about one point that you have is if this is Point at infinity, or not. We need to remark, but in what we are interested in is to know is if this is a Point at infinity of the elliptic curve that we use (denoted for $\mathcal{O}_E$), in the case that this point is another Point at infinity (in general, the Point at infinity is denoted for $\mathcal{O}$) or if this is an invalid point out of the curve we receive an error. In spite of this aclaration, the function suppose that the point of the input is a valid elliptic curve point.

Algorithm 3.2.13 *Point at infinity of the curve: $\mathcal{O}_E$*
Input: Valid point of the curve (Q).

Output: Error, If this is not a valid curve point; True, If this is a Point at infinity of the curve (no include all the $\mathcal{O}$ of the projective plan); False, in other case.

1. If $(Q_z = 0)\{$
   (a) If $(Q_y = 0)\{$Return (error);$\}$/* if $Q_x = 0$ invalid point, and if $Q_x \neq 0$ do not belong to the curve*/
   (b) If $(Q_x = 0)$ {Return (true);}
2. Else {Return (false);}

This is, probably, the most used algorithm of this cryptosystem. Continually we are calculating scalar multiplications of the points. This algorithm is extracted from the IEEE rule [P1363] with this, almost, we have a guarantee in efficiency terms at the same that this has a really low probability to contain errors.

Algorithm 3.2.14 *Elliptic scalar multiplication.* (`p1363-A.10.9`)
Input: Scalar integer n, Point of the curve $P = (X : Y : Z) \in E(\mathbb{F}_p)$.
Output: Point of the curve $Q = n \cdot P = (X^\star : Y^\star : Z^\star) \in E(\mathbb{F}_p)$.

1. If $(n = 0 \;||\; Z = 0)\{$Return (1:1:0)$\}$
2. Establish
   (a) $X^\star \leftarrow X$
   (b) $Z^\star \leftarrow Z$
   (c) $Z_1 \leftarrow 1$
3. If $(n < 0)\{$ goto 6$\}$/*$(-n) \cdot P = n \cdot (-P)$*/
4. Establish
   (a) $k \leftarrow n$
   (b) $Y^\star \leftarrow Y$
5. goto 8
6. Establish $k \leftarrow (-n)$

7. $Y^\star \leftarrow -Y \ (mod\ p)$. /*$\Rightarrow \mathbb{F}_p$*/
8. If $(Z^\star = 1)\{X_1 \leftarrow X^\star, Y_1 \leftarrow Y^\star\}$ Else$\{X_1 \leftarrow X^\star/(Z^\star)^2, Y_1 \leftarrow Y^\star/(Z^\star)^3$
9. $h = 3k = h_l h_{l-1} \ldots h_1 h_0$ /*$h_l = 1$*/
10. $k = k_l k_{l-1} \ldots k_1 k_0$
11. For $i = (l-1)$ downto 1{
    (a) $(X^\star : Y^\star : Z^\star) \leftarrow 2\,(X^\star : Y^\star : Z^\star)$
    (b) If $(h_i = 1 \,\&\&\, k_i = 0)\{(X^\star : Y^\star : Z^\star) \leftarrow (X^\star : Y^\star : Z^\star) + (X_1 : Y_1 : Z_1)\}$
    (c) If $(h_i = 0 \,\&\&\, k_i = 1)\{(X^\star : Y^\star : Z^\star) \leftarrow (X^\star : Y^\star : Z^\star) + (-(X_1 : Y_1 : Z_1))\}$
12. Return $(X^\star : Y^\star : Z^\star)$

The next function operates between two points and we only use it once in the signature verification. But we need this function to have a really efficiency. We need to pay extremely attention in the way that this algorithm returns to an inexistent point in the projective plan ($[X : Y : Z]$), this only occurs in an error case, and we use this notation to destacate it.

Definition 3.2.15 *Point addition between equal or different points, described in [Men93]:*

1. $P = (x_1, y_1) = [X_1 : Y_1 : Z_1]$
   $-P = (x_1, -y_1) = [X_1 : -Y_1 : Z_1]$
   $Q = (x_2, y_2) = [X_2 : Y_2 : Z_2]$
   $Q \neq -P$
   $P + Q = (x_3, y_3) = [X_3 : Y_3 : Z_3]$

2. $x_3 = \lambda^2 - x_1 - x_2$
   $y_3 = \lambda\,(x_1 - x_3) - y_1$

3. $\lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & if \quad P \neq Q \\ \frac{3x_1^2 + a}{2y_2} & if \quad P = Q \end{cases}$,

   but in projective coordinates: $\lambda = \begin{cases} \frac{Y_2 Z_1 - Y_1 Z_2}{X_2 Z_1 - X_1 Z_2} & if \quad P \neq Q \\ \frac{3X_1^2 Z_2 + a Z_1^2 Z_2}{2 Y_2 Z_1^2} & if \quad P = Q \end{cases}$

Algorithm 3.2.16 *Point addition over the curve.* (`p1363-A.10.5`)
Input: Points to add $P_0 = (X_0 : Y_0 : Z_0), P_1 = (X_1 : Y_1 : Z_1) \mid (P_0 \neq \mathcal{O}) \wedge (P_1 \neq \mathcal{O}) \wedge (P_0, P_1 \in E(\mathbb{F}_p))$, *and the elliptic curve at both belong* $E(\mathbb{F}_p)$.
Output: Point $P_2 = P_1 + P_0 = (X_2 : Y_2 : Z_2) \in E(\mathbb{F}_p)$./*Possible return (0:0:0)!!!*/

1. $T_1 \leftarrow X_0$
2. $T_2 \leftarrow Y_0$
3. $T_3 \leftarrow Z_0$
4. $T_4 \leftarrow X_1$
5. $T_5 \leftarrow Y_1$
6. If $Z_1 \neq 1$\{
   (a) $T_6 \leftarrow Z_1$
   (b) $T_7 \leftarrow T_6^2$
   (c) $T_1 \leftarrow T_1 \cdot T_7$
   (d) $T_7 \leftarrow T_6 \cdot T_7$
   (e) $T_2 \leftarrow T_2 \cdot T_7$
7. $T_7 \leftarrow T_3^2$
8. $T_4 \leftarrow T_4 \cdot T_7$
9. $T_7 \leftarrow T_3 \cdot T_7$
10. $T_5 \leftarrow T_5 \cdot T_7$
11. $T_4 \leftarrow T_1 - T_4$
12. $T_5 \leftarrow T_2 - T_5$
13. If $(T_4 = 0)$\{
    (a) If $(T_5 = 0)$\{Return (0:0:0)\}
    (b) Else \{Return (1:1:0)\}\}
14. $T_1 \leftarrow 2 \cdot T_1 - T_4$

15. $T_2 \leftarrow 2 \cdot T_2 - T_5$
16. If $Z_1 \neq 1\{$
    (a) $T_3 \leftarrow T_3 \cdot T_6$
17. $T_3 \leftarrow T_3 \cdot T_4$
18. $T_7 \leftarrow T_4^2$
19. $T_4 \leftarrow T_4 \cdot T_7$
20. $T_7 \leftarrow T_1 \cdot T_7$
21. $T_1 \leftarrow T_5^2$
22. $T_1 \leftarrow T_1 - T_7$
23. $T_7 \leftarrow T_7 - 2 \cdot T_1 \begin{cases} (23a) & T_6 \leftarrow 2T_1 \\ (23b) & T_7 \leftarrow T_7 - T_6 \end{cases}$
24. $T_5 \leftarrow T_5 \cdot T_7$
25. $T_4 \leftarrow T_2 \cdot T_4$
26. $T_2 \leftarrow T_5 - T_4$
27. $T_2 \leftarrow T_2/2$ [4]
28. $X_2 \leftarrow T_1$
29. $Y_2 \leftarrow T_2$
30. $Z_2 \leftarrow T_3$
31. Return $P_2 = (X_2 : Y_2 : Z_2)$.

We call the next algorithm several times and we need a really efficient algorithm to calculate the double of one point, or in other words: the scalar multiplication of a point with the scalar 2. The algorithms written here have importance in the calculate efficiency and/or in the repetitive use; in it case this is there for both motives.

[4] see A.2.4 of [P1363]; shift

Algorithm 3.2.17 *Duplicate a point. (p1363-A.10.4)*
Input: Point to duplicate $P_1 = (X_1 : Y_1 : Z_1) \in E(\mathbb{F}_p)$*, and the curve at it belong* $E(\mathbb{F}_p)$.
Output:Point $P_2 = 2P_1 = (X_2 : Y_2 : Z_2) \in E(\mathbb{F}_p)$.

1. $T_1 \leftarrow X_1$
2. $T_2 \leftarrow Y_1$
3. $T_3 \leftarrow Z_1$
4. If $(T_2 = 0 \mid\mid T_3 = 0)\{$ Return $P_2 = (1 : 1 : 0)\}$
5. If $(a = p - 3)\{$
   (a) $T_4 \leftarrow T_3^2$
   (b) $T_5 \leftarrow T_1 - T_4$
   (c) $T_4 \leftarrow T_1 + T_4$
   (d) $T_5 \leftarrow T_4 \cdot T_5$
   (e) $T_4 \leftarrow 3 \cdot T_5\}$
6. Else$\{$
   (a) $T_4 \leftarrow a$
   (b) $T_5 \leftarrow T_3^2$
   (c) $T_5 \leftarrow T_5^2$
   (d) $T_5 \leftarrow T_4 \cdot T_5$
   (e) $T_4 \leftarrow T_1^2$
   (f) $T_4 \leftarrow 3 \cdot T_4$
   (g) $T_4 \leftarrow T_4 + T_5\}$
7. $T_3 \leftarrow T_2 \cdot T_3$
8. $T_3 \leftarrow 2 \cdot T_3$
9. $T_6 \leftarrow T_2^2$ /*Original:$T_2 \leftarrow T_2^2$*/
10. $T_5 \leftarrow T_1 \cdot T_6$ /*Original: $T_5 \leftarrow T_1 \cdot T_2$*/
11. $T_5 \leftarrow 4 \cdot T_5$

12. $T_1 \leftarrow T_4^2$
13. $T_1 \leftarrow T_1 - 2 \cdot T_5 \begin{cases} (13a) & T_7 \leftarrow 2 \cdot T_5 \\ (13b) & T_1 \leftarrow T_1 - T_7 \end{cases}$
14. $T_2 \leftarrow T_6^2$ /*Original: $T_2 \leftarrow T_2^2$*/
15. $T_6 \leftarrow 8 \cdot T_2$ /*Original: $T_2 \leftarrow 8 \cdot T_2$*/
16. $T_5 \leftarrow T_5 - T_1$
17. $T_5 \leftarrow T_4 \cdot T_5$
18. $T_2 \leftarrow T_5 - T_6$ /*Original: $T_2 \leftarrow T_5 - T_2$*/
19. $X_2 \leftarrow T_1$
20. $Y_2 \leftarrow T_2$
21. $Z_2 \leftarrow T_3$
22. Return $P_2 = (X_2 : Y_2 : Z_2)$.

Finally, the most called algorithm is the inversion point. This is extremely efficient, because we only need to calculate the modular inversion of one component, but we can not forget that we work in a finite field, and we have not negative values; it is not enough to change the sign, but also we need to calculate the modular inversion of the Y coordinate.

Algorithm 3.2.18 *Point inversion.* *(p1363-A.10.8)*
Input: Point of the curve $P = (X : Y : Z) \in E(\mathbb{F}_p)$.
Output: Point of the curve $Q = -P = (X : -Y : Z) \in E(\mathbb{F}_p)$.

1. $Y \leftarrow -Y$.
2. Return $(X : Y : Z)$.

3.2.2 Mathematical functions

Once all the algorithmics are described, we can go further and start the implementation. To materialize the new module, we use some functions from the big numbers library and functions which came from the own algorithmics. It is for this reason that the description has been separated in two blocs.

3.2.2.1 Existing functions

The used functions from Libgcrypt, in the subgroup offered to work with big modular integers, we use in the implementation are:

1. mpi_alloc(unsigned nlimbs):
   This is the first function that we see to create data structures of the type big integer. This call over one variable initialize it, with the indicated longitude in the parameter.

2. mpi_alloc_secure(unsigned nlimbs):
   To realize the same function than the last but for variables that they will contain hidden values, and we need to protect to the third person view, we use this function to initialize it and maintain in secure memory.

3. mpi_set(MPI w, MPI u):
   In this case, the result of the function call is to establish the same structure of one variable to another, we can see this function similar to the copy.

4. mpi_set_secure(MPI a):
   The name of this function can send to a wrong path, because it does not realise the same function to the last, else it establish the secure condition to variables that do not have this since the creation.

5. mpi_set_ui(MPI w, ulong u):
   This a really useful function, because it serves to generate structured values in the big number mathematic library from normal unsigned integers.

6. mpi_alloc_set_ui(unsigned long u):
   The same action than the last one, only with the difference in the returned value, because now we do not recuperate it in the parameter else the function return the integer structure.

7. mpi_free(MPI a):
   The action of this function over one variable is the same than a destructor of the data structure, and free the reserved memory.

8. mpi_normalize(MPI a):
   When you use during a lot of time one variable this can have incongruences in the stile of use more memory than the necessary. The call to this function check the variable and normalize this use of memory.

9. mpi_invm(MPI y, MPI x, MPI n):
   Simple modular inversion: $y = x^{-1} \, (mod \, n)$.

10. mpi_cmp(MPI a, MPI b):
    The code of this function are complex, because the data structure, but we can resume it saying: $\begin{cases} 0 & a = b \\ \neq 0 & a \neq b \end{cases}$. Becareful to the equal values comparison because it return 'false'.

11. mpi_cmp_ui (MPI u, unsigned long v):
    It do the same action than the last one (an it is useful to understand the performance of the last), but the comparison are realized between one big integer and one normal unsigned integer.

12. mpi_copy(MPI a):
    We make reference to this function in another function, it do a simply copy, return this value a.

13. mpi_mul(MPI w, MPI u, MPI v):
    This is the classical multiplication function, but for big integers: $w = u \cdot v$.

14. mpi_mulm(MPI w, MPI u, MPI v, MPI m):
    Exactly the same than the last, but return the result module an integer: $w = u \cdot v \, (mod \, m)$.

15. mpi_add(MPI w, MPI u, MPI v):
    At the same than the multiplication function, it simply do two big integer addition.

16. mpi_Adam(MPI w, MPI u, MPI v, MPI m):
    Performance of the modular big numbers addition: $w = u+v\,(mod\,m)$.

17. mpi_add_ui(MPI w, MPI u, ulong v):
    This function are extremely useful to little increments of one big integer, because add to this big number one unsigned normal integer. We need to put attention in the result, because it is not modular, if you need this you can calculate it apart.

18. mpi_sub(MPI w, MPI u, MPI v):
    The same than the multiplication or addition operation, but now is the torn of the subtraction.

19. mpi_subm(MPI w, MPI u, MPI v, MPI m):
    Modular Subtraction: $w = u - v\,(mod\,m)$.

20. mpi_sub_ui(MPI w, MPI u, ulong v):
    Also useful function for little decrements.

21. mpi_fdiv_r(MPI rem, MPI dividend, MPI divisor):
    This is the module operation.

22. mpi_fdiv_q(MPI quot, MPI dividend, MPI divisor):
    Division of two big numbers.

23. mpi_fdiv_qr(MPI quot, MPI rem, MPI dividend, MPI divisor):
    The union of the two last, it calculate the division and the rest or module.

24. mpi_powm(MPI res, MPI base, MPI exp, MPI m):
    Modular Power: $res = base^{exp} \, (mod \, m)$.

25. mpi_rshift(MPI x, MPI a, unsigned n):
    Shift function. $x = a/2^n$, useful for multiples to two divisions.

26. mpi_fromstr(MPI val, const char *str):
    Convert one string that it have a number in hexadecimal representation to a big integer, useful to introduce manually a big number, but with the limitation of only support to hexadecimal representation.

27. mpi_getbyte(MPI a, unsigned idx):
    Return the byte in the position 'idx' of the big integer; the first byte is the byte that contain the first '1' more heavy; if *idx* do not corresponding to any byte return -1 (useful to mark when the round algorithm are finished, and it rounded all the bytes).

28. mpi_test_bit(MPI a, unsigned idx):
    Return the value of the bit in the position 'idx', and idx are moved from left to right.

3.2.2.2 New Functions

Here are described that functions that belong directly to the new module. This functions appeared in the algorithmics section, but these are not a cryptographic functions. These cryptographic functions will appear in the section 3.2.3. We can divide these functions in four blocks: one with the random function, the second with the operations between elliptic curve points, the third with the auxiliary operations of the second block, and finally the fourth block with the function used to work with the special data structures of points and elliptic curves.

1. gen_k

2. escalarMult

3. sumPoints
4. duplicatePoint
5. invertPoint

6. genBigPoint
7. genPoint
8. existSquareRoot
9. lucas
10. pointAtInfinity
11. gen_bit
12. gen_y_2

13. point_copy
14. point_free
15. curve_copy
16. curve_free

Entrance in the first group, that only have one random function. It is not necessary a really specific function to generate random values over elliptic curves, and we do not realize any modification towards the function existing in the equivalent module without elliptic curves ('*./cipher/elgamal.c/*').

In the second group, almost corresponding with the algorithmics section 3.2.1.5, and only subtracting the function described in the algorithm 3.2.13, that we move it to the next group. The main function described is, without any doubt, the point scalar multiplication; but we can not rest importance to the point addition function. In a secondary side we have the function to duplicate points, that are commented in the algorithm 3.2.17, and obtain the importance from the performance of the scalar multiplication can be reduced to a point addition and duplication. The last function of this block is the inversion of one point, this inversion is realized in the modular inversion of the coordinate Y.

In the third block, we can find the moved function from the last block, it is joined to the functions described in the section 3.2.17. We do not need to say anything about this function because these are a translations of the algorithm. The last two are worthy to an especial mention because they are new. The first, '*gen_ bit*', is used to decide of the two coordinates we choose at the point generation time. The second, '*gen_ y_ 2*', it is used to calculate the right side of the elliptic curve equation, as a previous step of the square root search.

Finally, it only remains the fourth block. In this, the functions have a poor importance and we can call them *macros*. This block is implemented to make easy the programming and the reading of the code. In the way to avoid, to copy the components of point number one to one, or the same in the elliptic curve structure, we design separated functions to do this. In the same way of the copy for point and elliptic curves, we can also do it with the free structure memory of the unused data.

3.2.3 Cryptographic functions

At this point, only remains the function that we like to work. To make easy the description they are also divided in group the same as in the section 3.2.2.2, but at this time it only has two block. In deep this are repeated functions, but this is necessary because the second block is the interface between the module with the public key section of the program.

1. generateCurve
2. generateKey
3. testKeys
4. doEncrypt
5. decrypt
6. sign
7. verify

8. ecc_generate
9. ecc_check_secret_key

10. ecc_encrypt
11. ecc_decrypt
12. ecc_sign
13. ecc_verify
14. ecc_get_nbits
15. ecc_get_info

To follow the same procedure, the first function that it is called is: *ecc_generate*. This call does the same with *generateKey* and reorganize the parameters to unify it to the standard form of the program. The *generateKey* function starts to generate an elliptic curve with *generateCurve*. This function, firstly, only copies the values advised in [NIST], and we talk about this in the section 2.1.3. After, we can append more randomize to the setup generation, with the use of the function *genBigPoint* that we saw in the section 3.2.2.2. The last thing to do with a new key is to test it; and we can do this with the function *testKeys* (this generate a random value to encrypt and decrypt to compare the result; also it can test the signature, doing it with the value and verify it.).

We can not confuse the functions *ecc_check_secret_key* and *testKeys*, because the first one makes the test under the program request (the user in the last term) and the second is used for the module to detect problems in the key generation.

Indirectly, we talked about the functions *doEncrypt*, *decrypt*, *sign*, and *verify*; because we use them or test the new key. But also, this are used since the interface functions: *ecc_encrypt*, *ecc_decrypt*, *ecc_sign*, i *ecc_verify*.

About the two last interface functions, we can say that are independents to the module. Their functions (*ecc_get_nbits* i *ecc_get_info*) have own use for the program to know with which key size and with which algorithm they work, respectively.

3.3 Final result

Once we finish all the task of the project, we can see with perspective, the obtained results. In the start time, we made some milestones to manage, and one idea about the appearance of the final result. Although the project have a huge complexity, at a first sight due to the necessity to know in deep

cryptology to do the implementation, result apparently easier. This easier view is apparently, because we need to use a really large time to do it.

Anyway, the results always compensate the effort. The resulted module complains with the initial specifications, and it is adapted to the host software. The bug search always is opened, because we can never say that the program is 100% secure. But we base it on long time cryptoanalized standard that one security apport in the algorithmic base hardly has errors. Anyway, it is always disposable that third person analize it module in the way to search, find and correct hidden bugs; not only cryptographyc bugs, also writting programming bugs.

The main weakness of the implemented module, is the setup. This is not a big problem, do not offer less security if this is not randomized, because the security is in the secret key integer. But in the side to make more difficult the cryptanalysis, it is useful to append randomize to this setup. doing this the cryptoanalyzer needs to make the specific calculates to all the specific keys.

Chapter 4

Conclusions and Future work

Now the project is finished. We can think about the beginning expounding that we made over this, and see it the work is finished in the little distance of the wanted results. The first thing to see is this module is available to all the people that like to use strong cryptography, and we cover a hardship in the cryptographic software for encryption and sign in general use. In this softwares we put the capacity to use elliptic curve cryptosystem. In this meaning the user does not need to change over the old cryptosystem to this, also the user can choose one cryptosystem more if he likes.

Anyway, there is a long way to walk. The finalized module is completely external to the program, but depends on the mathematic library. One task to do in the future is to move the mathematical function to this library after the line of the first descendant level. Doing this, the mathematical elliptic curve functions belong to this. The module win more light, because it only contains cryptographic functions.

We talked in the section 3.3 about its necessity to offer more randomize to the setup, and with this way we can obtain a stronger keys, with all the attempts to cryptoanalize need to start to the zero point all the time.

In the elliptic curve cryptosystem view, this module uses one mathematical base over that we define the curves are $\mathbb{F}_p$, but this is not the only one base. There exist the field for characteristic 2 and a degree extension m, represented as $\mathbb{F}_{2^m}$. The advantage of these fields is that we can make the calculus for a binary polynomials.

In the last step to realize a cryptographic implementation really strong, we could jump to the hyperelliptic curve cryptosystem. This cryptosystem is stronger than the elliptic curves. There is a long way to program modern and strong cryptosystems over public and general softwares.

Appendix A

Key uses

A.1 Key generation

A.1.1 Program dump

[sergi@hyperion sergi]$ gpg –gen-key –debug 4
gpg: s'estan llegint opcions de '/home/sergi/.gnupg/gpg.conf'
gpg (GnuPG) 1.2.1-ecc; Copyright (C) 2002 Free Software Foundation, Inc.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions. See the file COPYING for details.

gpg: AV´S: esteu usant memòria insegura!
gpg: si us plau, visiteu http://www.gnupg.org/faq.html per a més informació
Trieu quin tipus de clau voleu:
(1) DSA i ElGamal (predeterminat)
(2) DSA (només signar)
(5) RSA (només signar)
(8) ECDSA ECElGamal
Trieu: 8
Experimental: Research Project.
gpg: ECC get info.
gpg: DBG: PUBKEY_USAGE_SIG=1 PUBKEY_USAGE_ENC= 2
gpg: DBG: PUBKEY_USAGE_SIG|PUBKEY_USAGE_ENC= 3
gpg: DBG: use= 3
About to generate a new ECELG ECDSA keypair.
minimum keysize is 192 bits
default keysize is 224 bits

higher keysizes are 256, 384, 521 bits
What keysize do you want? (224) 384
La grandària sol*cdot*licitada és 384 bits
Especifiqueu el temps de validesa de la clau.
0 = la clau no caduca
<n> = la clau caduca als n dies
<n>w = la clau caduca a les n setmanes
<n>m = la clau caduca als n mesos
<n>y = la clau caduca als n anys
Indiqueu la validesa de la clau (0) 0
Key no caduca en absolut
És correcte? (s/n)s

Necessiteu un ID d'usuari per a identificar la vostra clau; el programa
construeix la id de l'usuari amb el Nom, Comentari i Adreça electrónica
d'aquesta forma:
"Heinrich Heine (Der Dichter) <heinrichh@duesseldorf.de>"

Nom i cognomsno384
Adreça electrónica:
Comentari:
Heu triat l'usuari:
"no384"

Canvia (N)om, (C)omentari, (E)mail o (O) d'acord / (Q) ixo
Cal una contrasenya per a protegir la clau secreta.

No voleu contrasenya: és una mala idea!
Bé. Si voleu canviar-la més endavant,
useu aquest programa amb l'opció" --edit-key".

Cal generar molts bits aleatoriament. És bona idea fer alguna altra cosa
(teclejar, moure el ratolí, usar els discos) durant la generació de
nombres primers; aço dóna oportunitat al generador de nombres aleatoris
d'aconseguir prou entropia.

gpg: generation p= FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFF0000000000000000FFFFFFFF
gpg: generation a= -3
gpg: generation b= B3312FA7E23EE7E4988E056BE3F82D19181D9C6EFE8

```
141120314088F5013875AC656398D8A2ED19D2A85C8EDD3EC2AEF
gpg: generation n= FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFC7634D81F4372DDF581A0DB248B0A77AECEC196ACCC52973
gpg: Gx ok
gpg: Gy ok
gpg: Gz ok
gpg: generation Gx= AA87CA22BE8B05378EB1C71EF320AD746E1D3B628B
A79B9859F741E082542A385502F25DBF55296C3A545E3872760AB7
gpg: generation Gy= 3617DE4A96262C6F5D9E98BF9292DC29F8F41DBD28
9A147CE9DA3113B5F0B8C00A60B1CE1D7E819D7A431D7C90EA0E5F
gpg: generation Gz= 1
gpg: Setup generated

gpg: DBG: choosing a random x of size 384..+++++
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: Testing key.
gpg: DBG: choosing a random k of 178 bits
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: input=F80A60815D587A17FF9D6DA1A8E3F8ABEC5693B23BF0C7D0E1D5
5C38B519FD91F8DFC2CBB98F2C94
gpg: DBG: doEncrypt: end.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: output=F80A60815D587A17FF9D6DA1A8E3F8ABEC5693B23BF0C7D0E1D
55C38B519FD91F8DFC2CBB98F2C94
gpg: ECELG operation: encrypt, decrypt ok.
gpg: DBG: choosing a random k of 178 bits.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: ?is a Point At Infinity.
gpg: False:It isn't a point at Infinity.
gpg: Verification: Accepted.
```

```
gpg: ECDSA operation: sign, verify ok.

gpg: [ecc] p= FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFEFFFFFFFF0000000000000000FFFFFFFF
gpg: [ecc] a= -3
gpg: [ecc] b= B3312FA7E23EE7E4988E056BE3F82D19181D9C6EFE814112
0314088F5013875AC656398D8A2ED19D2A85C8EDD3EC2AEF
gpg: [ecc] Gx= AA87CA22BE8B05378EB1C71EF320AD746E1D3B628BA79B9
859F741E082542A385502F25DBF55296C3A545E3872760AB7
gpg: [ecc] Gy= 3617DE4A96262C6F5D9E98BF9292DC29F8F41DBD289A147
CE9DA3113B5F0B8C00A60B1CE1D7E819D7A431D7C90EA0E5F
gpg: [ecc] Gz= 1
gpg: [ecc] n= FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
C7634D81F4372DDF581A0DB248B0A77AECEC196ACCC52973
gpg: [ecc] Qx= E6FC74150DD9E656112843814DEFDE27712BD2925BA4BD7
5EAEDB904B97088E563032957BEE71284B9B725A10545EC41
gpg: [ecc] Qy= 57C95A98C6B95BFF20D574A55F8DBB6621B27FC6C10FAB3
C465FF62E6D94ADFD0F8D0C8936203FA696F7F22B3CBD39DB
gpg: [ecc] Qz= E3072E4F8B92D23A3B40E2C607253F81C1B9D7928A8356E
09A1AE6DE6CB0212F9263A9F57E22EC327B6EF129D846F435
gpg: [ecc] d= F06C3C4DF9E43C17614805DFB6A5181098646F0DF4DC2887
33D3C6C362C5B1B5B47578424FDB05853537AD979135BDAB
gpg: ECC key Generated.
gpg: No hauríeu d'usar algoritmes experimentals!
gpg: DBG: pubkey_sign: algo=103
gpg:    data:1FFFFFFFFFFFFFFFFFFFF003021300906052B0E03021A050004
1460241B9DAF0EA2541222B396E1DB5B4D9BD0C32F
gpg: ECC sign.
gpg: DBG: choosing a random k of 178 bits
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: ECC sign END.
gpg:    sig:736685714A0CC4330574B598B5257D761D03C6E0D2E5DA3867D
77EDDDDBD1840C811877846F8AC6F6498895E09E7644
gpg:    sig:2E05AB58B6814404DC1649C7ED3FC69C45EAC2A919AFD3A85ED
69451807A9F42E1CD6F47319F9F691AF5F5BFEB46315C
gpg: ECC verify.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: Calculating an scalar Multiple.
```

gpg: Scalar Multiple calculated.
gpg: ?is a Point At Infinity.
gpg: False:It isn't a point at Infinity.
gpg: Verification: Accepted.
gpg: ECC verify END.
gpg: ECC sign.
gpg: DBG: choosing a random k of 178 bits.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: ECC sign END.
gpg: sig:331049FC86BDB239A2B4391D0DAA591EF1104C4B0EB54D96777
9E3498F89309D6D5E2D5F08028881740B7E2C118DA82B
gpg: sig:9923D92476CC9ECC6479D89A773CDE17073C6F94560D52094EF
BCF14B05F5BC1AD1B26A188445332C39AFD746D6745B3
gpg: ECC verify.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: ?is a Point At Infinity.
gpg: False:It isn't a point at Infinity.
gpg: Verification: Accepted.
gpg: ECC verify END.
claus pblica i secreta creades i signades.
clau marcada com confiada absolutament.

gpg: ECC get nbits.
pub 384E/0710ECCB no384
Empremta digital = 0BEA 9994 43AC 8CB5 69E8 7B8D 6197 E242 0710 ECCB

secmem usage: 2048/2496 bytes in 10/14 blocks of pool 2528/16384
[sergi@hyperion sergi]$

A.1.2 Ecc key

```
---BEGIN PGP PUBLIC KEY BLOCK---
Version:  GnuPG v1.2.1-ecc (SunOS)
mQGcBD+NE4JnAYD/////////////////////////////////////v////8A
AAAAAAAAAP////8AAgMBgLMxL6fiPufkmI4Fa+P4LRkYHZxu/oFBEgMUCI9QE4da
```

```
xlY5jYou0Z0qhcjt0+wq7wGAqofKIr6LBTeOscce8yCtdG4dO2KLp5uYWfdB4IJU
KjhVAvJdv1UpbDpUXjhydgq3AX42F95KliYsb12emL+Sktwp+PQdvSiaFHzp2jET
tfC4wApgsc4dfoGdekMdfJDqDl8AAQEBgP////////////////////////////
/8djTYH0Ny3fWBoNskiwp3rs7BlqzMUpcwGAvA3jdAqwZloJafqBqRclX6FNS3X0
fyZ3RmmlsDIfYdanz6rq8jVvku2ugCoFKBHQAX0cIBZXYvThPHFAU5zFRCGGf4er
FBcr0l316UrZuytHLXtA98xsUr6xMIVJ2W/4oo8BfiTFMN2I2XvYeEtM5C7nhLOV
10akiOk/hVOlLURvkbDs09fImBhMv6FFE8f3djk9frQFbm8zODSIkQQTZwIAGQUC
P40TggQLBwMCAxUCAwMWAgECHgECF4AACgkQLyg4Lsr9a2/cKQGA4kYoXM6sjWyn
I1Rlnyz4azqrSDyluGakKDHiHURtaWqdTy+n2JIFSraiwIGuPKnrAYDuJQy+d1id
YHjKla5FtBU5m6gz6+Yl2TXt+hzhzDXPoeMhfIKy+1ueHVGc9UT9mfM=
=LzAn
---END PGP PUBLIC KEY BLOCK---
```

A.2 Encryption

A.2.1 Program dump

Sample of executing an encryption:

bash-2.05$ gpg -a –debug 4 -r no384 -e encrypt.version
gpg: reading options from '/home/sblanch/.gnupg/gpg.conf'
gpg: WARNING: using insecure memory!
gpg: please see http://www.gnupg.org/faq.html for more information
gpg: ECC get info.
gpg: ECC verify.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: ?is a Point At Infinity.
gpg: False:It isn't a point at Infinity.
gpg: Verification: Accepted.
gpg: ECC verify END.
gpg: DEK is: 8F 02 2B F9 77 A7 D2 4D 2B 89 8B 68 BE AF 9C 6D
gpg: Experimental algorithms should not be used!
gpg: ECC get nbits.
gpg: DBG: pubkey_encrypt: algo=103
gpg: pkey:FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFEFFFFFFFF0000000000000000FFFFFFFF

```
gpg: pkey:3
gpg: pkey:B3312FA7E23EE7E4988E056BE3F82D19181D9C6EFE81411203140
88F5013875AC656398D8A2ED19D2A85C8EDD3EC2AEF
gpg: pkey:AA87CA22BE8B05378EB1C71EF320AD746E1D3B628BA79B9859F74
1E082542A385502F25DBF55296C3A545E3872760AB7
gpg: pkey:3617DE4A96262C6F5D9E98BF9292DC29F8F41DBD289A147CE9DA3
113B5F0B8C00A60B1CE1D7E819D7A431D7C90EA0E5F
gpg: pkey:1
gpg: pkey:FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC7634
D81F4372DDF581A0DB248B0A77AECEC196ACCC52973
gpg: pkey:BC0DE3740AB0665A0969FA81A917255FA14D4B75F47F26774669A
5B0321F61D6A7CFAAEAF2356F92EDAE802A052811D0
gpg: pkey:1C20165762F4E13C7140539CC54421867F87AB14172BD25DF5E94
AD9BB2B472D7B40F7CC6C52BEB1308549D96FF8A28F
gpg: pkey:24C530DD88D97BD8784B4CE42EE784B395D4E6A488E93F8553A52
D446F91B0ECD3D7C898184CBFA14513C7F776393D7E
gpg: data:290E0AE92B19A7BD5FB3E8248B6159C91680B50BD8C997C968A8B
00078F022BF977A7D24D2B898B68BEAF9C6D080F
gpg: ECC encrypt.
gpg: DBG: choosing a random k of 178 bits
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: DBG: doEncrypt: end.
gpg: resarr[0]=AED3A94B5F11B9F31BD0799BD342ED4EB845A020E58E
EED06B66BDCC9A13DD8350A55A692196814DB28EC061FEF52C35
gpg: resarr[1]=51123177BE7008BB7C98E5ED3D7236C0766AAE4DF0DB
CF73C3CC03AA079570C9D67FB1551CD9701B35B2A366684E78A9
gpg: resarr[2]=5920360B36FCF0A3F08331E57AF1B85F07921C2C0D33
17B0F6B018839B3CC5A8FBB8AAAE7420A6F9C9BFCD3404B45FBE
gpg: resarr[3]=1E288A33D93E90DA830AC4F0CA5D2CF4919B4733BF0E
30980ED8D088B46D3A32F1CF45EF5C6C018D2839B7BFDDFC5F3836DAE87C949EF4C9288
9C493BBC7E7E5837CAB5DFCCA87D69692BB3ED4CA161BC0C1B3F4AC93FFE5FFCB9E3778
B30
gpg: resarr[4]=0
gpg: DBG: ECC doEncrypt end.
gpg: ECC encrypt END.
gpg: encr:AED3A94B5F11B9F31BD0799BD342ED4EB845A020E58EEED06B66B
DCC9A13DD8350A55A692196814DB28EC061FEF52C35
```

gpg: encr:51123177BE7008BB7C98E5ED3D7236C0766AAE4DF0DBCF73C3CC0
3AA079570C9D67FB1551CD9701B35B2A366684E78A9
gpg: encr:5920360B36FCF0A3F08331E57AF1B85F07921C2C0D3317B0F6B01
8839B3CC5A8FBB8AAAE7420A6F9C9BFCD3404B45FBE
gpg: encr:1E288A33D93E90DA830AC4F0CA5D2CF4919B4733BF0E30980ED8D
088B46D3A32F1CF45EF5C6C018D2839B7BFDDFC5F3836DAE87C949EF4C92889C493BBC7
E7E5837CAB5DFCCA87D69692BB3ED4CA161BC0C1B3F4AC93FFE5FFCB9E3778B30
gpg: encr:0
secmem usage: 2112/2752 bytes in 5/8 blocks of pool 3040/16384
bash-2.05$

A.2.2 Ecc encrypted message

Sample of plain text:

```
gpg (GnuPG) 1.2.1-ecc
Copyright (C) 2002 Free Software Foundation, Inc.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions.  See the file COPYING for details.
Home:  ~/.gnupg
Supported algorithms:
Pubkey:  RSA, RSA-E, RSA-S, ELG-E, DSA, ELG, ECELG & ECDSA
Cipher:  3DES, CAST5, BLOWFISH, AES, AES192, AES256, TWOFISH
Hash:  MD5, SHA1, RIPEMD160
Compress:  Uncompressed, ZIP, ZLIB
```

The same text but now are ciphered:

```
---BEGIN PGP MESSAGE---
Version:  GnuPG v1.2.1-ecc (SunOS)
hQEDAy8oOC7K/WtvZwGArtOpS18RufMbOHmbOOLtTrhFoCDlju7Qa2a9zJoT3YNQ
pVppIZaBTbKOwGH+9Sw1AX9REjF3vnAIu3yY5eO9cjbAdmquTfDbz3PDzAOqB5Vw
ydZ/sVUc2XAbNbKjZmhOeKkBf1kgNgs2/PCj8IMx5XrxuF8HkhwsDTMXsPawGIOb
PMWo+7iqrnQgpvnJv8OOBLRfvgLxAeKIoz2T6Q2oMKxPDKXSzOkZtHM78OMJgO2N
CItGO6MvHPRe9cbAGNKDm3v938Xzg22uh8lJ7OySiJxJO7x+flg3yrXfzKh9aWkr
s+1MoWG8DBs/Ssk//l/8ueN3izAAANLA4gHqaNARg/9orwRNAKpnBIqWuWGnN3bi
d4GMef3QlN/x9UV/BY6O/nlz7PSwRle9uO7F8u8y4UCUjPuI6TbGe4+SdjxC8U6p
```

```
n7f66CAreRU/a1sG2C7bLJ1QOlaimOexDxt3NpLXSQqWGXis6AfwKyEAwW9mqTEm
J2b6f4MLjlvCMsHpxxyZ7vqPaTlYj8mP1JXpZxO23oBMhwQ58z9VRfKwiITkKXWO
3Y+yboptMEuCvCW6JRHRkRwzOf3MQO4PaBhvohBtQJO3EI8r+2OltdlqFCwl1UCX
/HURSguZpQZlpOg5ymCHOihryIs7KG2vWQaSScJzuavP7+wPlrEXotQvvThDN9uF
D8HEItzpcjpNI4+cRMkDDnT3nl9bq+DXQln8Yeb9JPJJSS4xqbza/UIRWAq+uInk
3SBfesaTMJ9xulVvIPTDJSe1PaT+AGs8ICKPO+gCC95XfwwWcEtJOlEQv/O2OGdA
n/otX9BK93AKYKj8SxvKIaZ1orK8Z8TcNdJXT/tEv3ZzgTohGproBHIqAt7hWOkH
1/zQ5uWHkX1J21A=
=EGh4
---END PGP MESSAGE---
```

A.3 Digital signature

A.3.1 Program dump

Sample of executing a digital signature

bash-2.05$ gpg -a –debug 4 -b gnupg-1.2.1-ecc.diff
gpg: reading options from '/home/sblanch/.gnupg/gpg.conf'
gpg: WARNING: using insecure memory!
gpg: please see http://www.gnupg.org/faq.html for more information
gpg: ECC get info.
gpg: ECC verify.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: ?is a Point At Infinity.
gpg: False:It isn't a point at Infinity.
gpg: Verification: Accepted.
gpg: ECC verify END.
gpg: Experimental algorithms should not be used!
gpg: DBG: pubkey_sign: algo=103
gpg: skey:FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFEFFFFFFFF0000000000000000FFFFFFFF
gpg: skey:3
gpg: skey:B3312FA7E23EE7E4988E056BE3F82D19181D9C6EFE81411203140
88F5013875AC656398D8A2ED19D2A85C8EDD3EC2AEF
gpg: skey:AA87CA22BE8B05378EB1C71EF320AD746E1D3B628BA79B9859F74

1E082542A385502F25DBF55296C3A545E3872760AB7
gpg: skey:3617DE4A96262C6F5D9E98BF9292DC29F8F41DBD289A147CE9DA3
113B5F0B8C00A60B1CE1D7E819D7A431D7C90EA0E5F
gpg: skey:1
gpg: skey:FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC7634
D81F4372DDF581A0DB248B0A77AECEC196ACCC52973
gpg: skey:BC0DE3740AB0665A0969FA81A917255FA14D4B75F47F26774669A
5B0321F61D6A7CFAAEAF2356F92EDAE802A052811D0
gpg: skey:1C20165762F4E13C7140539CC54421867F87AB14172BD25DF5E94
AD9BB2B472D7B40F7CC6C52BEB1308549D96FF8A28F
gpg: skey:24C530DD88D97BD8784B4CE42EE784B395D4E6A488E93F8553A52
D446F91B0ECD3D7C898184CBFA14513C7F776393D7E
gpg: skey:E017EAEA6263B4697B5C847445FB51B342BB6501F96D915C197C1
CE6BE0B6501121236B2301CCFD18E778EF22C25BF0B
gpg: data:1FFFFFFFFFFFFFFFFFFFF003021300906052B0E03021A05000414
BB6C70BC93A70165C56A3E9B6DA714EF4B2530CF
gpg: ECC sign.
gpg: DBG: choosing a random k of 178 bits
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: ECC sign END.
gpg: sig:F01C5381A2AD9E526E8EB235D815798323E0BCBD9AE0608A07922E
0150AFC99DF6D92D0440C4379DF7642C9E09951B23
gpg: sig:1B5ADA16F3AEC326A632C70B256DD471F10068ACB14A6F3017EF37
BE4A39FB5AEF3E3F2D5AB8218C3A29E6064DD16CA1
gpg: ECC verify.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: ?is a Point At Infinity.
gpg: False:It isn't a point at Infinity.
gpg: Verification: Accepted.
gpg: ECC verify END.
gpg: ECC verify.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: ?is a Point At Infinity.

gpg: False:It isn't a point at Infinity.
gpg: Verification: Accepted.
gpg: ECC verify END.
secmem usage: 1472/1600 bytes in 3/5 blocks of pool 1600/16384
bash-2.05$

Sample of executing a digital signature verification

bash-2.05$ gpg –debug 4 –verify gnupg-1.2.1-ecc.diff.asc
gpg: reading options from '/home/sblanch/.gnupg/gpg.conf'
gpg: WARNING: using insecure memory!
gpg: please see http://www.gnupg.org/faq.html for more information
gpg: ECC get info.
gpg: Signature made Wed Oct 22 11:06:25 2003 CEST using
ECELG & ECDSA key ID CAFD6B6F
gpg: ECC verify.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: ?is a Point At Infinity.
gpg: False:It isn't a point at Infinity.
gpg: Verification: Accepted.
gpg: ECC verify END.
gpg: ECC verify.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: ?is a Point At Infinity.
gpg: False:It isn't a point at Infinity.
gpg: Verification: Accepted.
gpg: ECC verify END.
gpg: ECC verify.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: Calculating an scalar Multiple.
gpg: Scalar Multiple calculated.
gpg: ?is a Point At Infinity.

gpg: False:It isn't a point at Infinity.
gpg: Verification: Accepted.
gpg: ECC verify END.
gpg: Good signature from "no384"
secmem usage: 1408/1408 bytes in 2/2 blocks of pool 1408/16384
bash-2.05$

A.3.2 Ecc signed message

Sample of digital signature over the diff file.

```
---BEGIN PGP SIGNATURE---
Version:  GnuPG v1.2.1-ecc (SunOS)
iHcDBQA/lkZMLyg4Lsr9a29nArtsAYDwHFOBoq2eUm6OsjXYFXmDI+C8vZrgYIoH
ki4BUK/JnfbZLQRAxDed92QsngmVGyMBfRta2hbzrsMmpjLHCyVt1HHxAGissUpv
MBfvN75KOfta7z4/LVq4IYw6KeYGTdFsoQ==
=OFej
---END PGP SIGNATURE---
```

Appendix B

Execution procedure

In this Appendix, we would look what the code flow when we launch a command. In this way, there are the main part of the program and some satellite modules, almost all of this interactuate with the mathematical library. When the user call the program, with know parameters, the program need to scan the parameters that the user write after the command. We could use some parameter at the same time, but without conflicting commands because the program control it. When the program scan the parameters fix some program environment variables to make the orders in the way that the user like. For example, if you like to obtain the results in an '*ascii armor*' you need to say this to the program with the parameter '`--armor`' combined with other main orders; or if you like to change the default home directory, you use the '`--homedir` *`directory`*'. All of this named parameters are an execution option but it does not a commands in the program. The main parameters in the program, to modify specially the control flow, are the commands like '`--gen-key`', '`--sign`', '`--verify`', '`--encrypt`', '`--decrypt`',... This is the main command parameters that we need.

At the first execution of the 'gpg' command without parameters, it create the home directory with the default name '∼/.gnupg' and the files '∼/.gnupg/options', in the second execution it creates the void keyrings '∼/.gnupg/pubring.gpg' and '∼/.gnupg/secring.gpg'.

In the executions with parameters, the program scan it and find in a list, that are in the file 'g10/g10.c'. This is the first file that we execute in the runtime. In less words, this file are used to know and jump to the other module files, but not for all options; for example, in the '`--version`' it perform the main thinks in this file, but calling function that are in other ones.

In a chronological order, we can follow the execution of '`--gen-key`' in the first case, and after try to do the same with '`--encrypt`' and '`--sign`', and the opposite commands '`--decrypt`' and '`--verify`', respectively.

For all options and commands, we need to perform the main function. This function precess all parameters and activate something like flags. For the command[1] parameters it function call '`set_cmd()`', and for the options modify a datastructure named '`opt`', described in '`g10/options.h`'. The function '`set_cmd()`' control the commands combination, like when the user like to encrypt and sign with only one shell program call. After this establishment, the program control the conflicts and overriding, also between commands or between options.

Here we have the three different cases that we like to follow: '`aKeygen`', '`aEncr/aDecrypt`', '`aSign/aVerify`' [2]. When the control come in one branch of the case, then call to the specific function.

B.1 Key generation

To come to this branch, we suppose that the user launch a command in a shell like '`$ gpg --key-gen`'. As the same as all the programs, it start with the main function in '`g10/g10.c:main()`'. This function have a data structure to recognize the command call '`--gen-key`', it establish the label variable to '`aKeygen`', and call the function '`g10/keygen.c:generate_keypair()`'. At this point starts a travel in the code source to the destination the own new module, but is in this function when the program ask to the user which type of key like to create (at this time all options continue in the same way). The last named function call some functions to perform the code, and one of this is '`g10/keygen.c:proc_parameter_file()`'. At the next time it call '`g10/keygen.c:do_generate_keypair()`', and continue to '`g10/keygen.c:do_create()`'.

At this function we find different execution options, separated according to the type of key that the user like to create. Remember that the program knew which algorithm we like to use, because it ask it before.

The first modification in the source code is at this time, because the program go to different ways, and we need to create a new one. To create this new way, we make the function '`g10/keygen.c:gen_ecc()`' based on the

[1] There are differences in the translation of equivalent arguments, like '`-a`' and '`--armor`', but the result is the same.

[2] There exist a case named '`aSignEncr`', that is really similar to '`aSign`' but with and special function parameters to encrypt first the source information.

Figure B.1.1: Scheme of the execution procedure in key generation.

existed '`g10/keygen.c:gen_elg()`'. In the next steps we jump to another file of the source, now we are at '`cipher/pubkey.c:pubkey_generate()`' that all the algorithms call, and try to find in a table of functions interface to the public key modules in that we insert a section. This table translate the call from '`cipher/pubkey.c:pubkey_table[i].generate`' to '`cipher/ecc.h:ecc_generate()`'. Now we finally are in the new module code an this function is in the module interface that call '`cipher/ecc.c:generateKey()`' that have all the needed code to generate an Elliptic Curve Keypair.

B.2 Encryption

In the cipher way, we have two opposite operations to do: Encrypt and Decrypt. The user call this commands with the parameters '`$ gpg --encrypt`' and '`$ gpg --decrypt`', respectively but it is not allow to call both at the same time.

B.2.1 Encrypt

Now we suppose that the use call in the command like the order '`$ gpg --encrypt`', and the program do the same than always: perform the '`g10/g10.c:main()`' function. The label variable of the command are '`aEncr`' in this case[3], and the first thing that it do is call the function '`g10/encode.c:encode_crypt()`'. The way to go to the ecc module is really near in this case, this function in one step call '`g10/encode.c:write_pubkey_enc_from_list()`', but because of the name is not trivial to know that this second function after call '`cipher/pubkey.c:pubkey_encrypt()`'. At this point, we jump to the cipher modules directory and we are in the global public key functions; we remember that in the key generation, it use a similar function in this file. Evidently, at this point only rest to find the module interface in the *translation name function table* with the function '`cipher/pubkey.c:pubkey_table[i].encrypt`'. This translation guide to the function in the own module interface named '`cipher/ecc.h:ecc_encrypt()`' that prepare all the structs to call the private module function '`cipher/ecc.c:doEncrypt()`'.

[3]there are other options to encrypt like '`aEncrFiles`' or '`aSignEncr`' but the differences are really little, and it is not necessary to make an special mention.

Figure B.2.1: Scheme of the execution procedure in encryption.

Figure B.2.2: Scheme of the execution procedure in decryption.

B.2.2 Decrypt

In the way to restore the encrypted data, we use the opposite command '`$ gpg --decrypt`', and the program start the execution like always. When the program translate the parameter to the command label variable, named '`aDecrypt`', call the function '`g10/decrypt.c:decrypt_message()`'. In this function, the program control the type of data that it have; for example, if the data have an ascii armor. Then call '`g10/mainproc.c:proc_packets()`' to prepare the memory structure and call '`g10/mainproc.c:do_proc_packets()`'. At this time, the program need to know if the data are encrypted with a symmetric algorithm or with a public key one, in the own case is a pubkey and call '`g10/mainproc.c:proc_pubkey_enc()`'. After make different way for the public key algorithms, the program need to know which public key algorithm use, and we can insert the '`PUBKEY_ALGO_ECC`' in the control point, and continue to the '`g10/pubkey_enc.c:get_session_key()`' and '`g10/pubkey_enc.c:get_it()`' just after, to go to the new module. Now we are in the last points, in the call to the function '`cipher/pubkey.c:pubkey_decrypt()`', that access to the translation table '`cipher/pubkey.c:pubkey_table[i].decrypt`', to know the interface function '`cipher/ecc.h:ecc_decrypt()`', and finally arrive to the own programmed function '`cipher/ecc.c:decrypt()`'.

B.3 Signature

To make the digital signature and verified if this is correct and nobody make any modification, the user can use the command parameters '`$ gpg --sign`' and '`$ gpg --verify`', now we like to have a description of this procedures.

B.3.1 Sign

The procedure to sign a document is really similar to encrypt in the words of the execution procedure. The first thing is the parameter translation to the command label variable '`aSign`', and call the function '`g10/sign.c:sign_file()`'.

Figure B.3.1: Scheme of the execution procedure in the digital signature.

Figure B.3.2: Scheme of the execution procedure in the signature verification.

The next steps are prepare the blocks and write the sign, and the program do this first with the function '`g10/sign.c:write_signature_packets()`' and then with '`g10/sign.c:do_sign()`'. Finally we are in front the modules interface, because of the call to '`cipher/pubkey.c:pubkey_sign()`', and the table '`cipher/pubkey.c:pubkey_table[i].sign`' that translate the call to the module interface function '`cipher/ecc.h:ecc_sign()`'. As the same than in the other commands, at this point only rest to call to the own programmed '`cipher/ecc.c:sign()`'.

B.3.2 Verify

In the last case of the sign verification, there are a lot of functions to call. The parameter translation of the shell command '`$ gpg --verify`', is '`aVerify`'. The first function to perform is '`g10/verify.c:verify_signatures()`', but the next functions are in the '`g10/mainproc.c`' and it make a lot of jumps into this file ('`proc_signature_packets()`', '`do_proc_packets()`', '`add_onepass_sig()`', '`release_list()`', '`proc_tree()`', '`check_sig_and_print()`', '`do_check_sig()`', '`check_key_signature2()`'). After the last one, the program jump to the cipher code section with the call to the function '`cipher/sig-check.c:signature_check2()`' and the function '`cipher/sig-check.c:do_check()`'. Finally we are at the modules inteface with the call '`cipher/pubkey.c:pubkey_verify()`', and the translation in the table '`cipher/pubkey.c:pubkey_table[i].verify`', and go to the module calling '`cipher/ecc.h:ecc_verify()`'. The function in the own module to verify one signature, that it is called from the module interface is '`cipher/ecc.c:verify()`'.

Figure B.3.3: Global scheme of the execution procedure in this work.

Bibliography

[P1363] IEEE P1363/D13 (Draft Version 13) Standard Specifications for Public key Cryptography. 1999 November 12.

[NIST] FIPS PUB 186-2, Digital Signature Standard (DSS), U.S.Department of Commerce/National Institute of Standards and Technology. 2000 January 27.

[Auto98] Autoconf:Creating Automatic Configuration Scripts. Edition 2.13, for Autoconf version 2.13. December 1998

[822] Standard for the format of arpa internet text messages (Obsolete: see 2822). 1982 August.

[1991] PGP Message Exchange Formats. 1996 August.

[2015] MIME Security with Pretty Good Privacy (PGP) (Obsolete: see 3156). 1996 October.

[2440] OpenPGP Message Format. 1998 November

[2822] Internet Message Format. 2001 April.

[3156] MIME Security with OpenPGP. 2001 August.

[3278] Use of Elliptic Curve Cryptography (ECC) Algorithms in Cryptographic Message Syntax (CMS). 2002 April.

[PKCS#1] PKCS#1 v2.1: RSA Crptography Standard. RSA Laboratories. 2002 June 14.

[mov97] Alfred J, Menezes, Paul C.van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, first edition, 1997. http://cacr.math.uwaterloo.ca/hac/

[WZ99] M.Wiener and R. Zuccherato, *Faster attacks on elliptic curve cryptosystems*, Selected Areas in cryptography, Lecture notes in Computer Science, n1556 (1999), Springer-Verlag, 190-200.

[PH78] S. Pohlig and M. Hellman, *An improved algorithm for computing logarithms over GF(p) and its cryptographic significance*, IEEE Transactions on Information Theory, n24 (1978), 106-110.

[OW99] P. Van Oorschot and M. Wiener, *Parallel Collision search with cryptanalytic applications*, Journal of Cryptology, n12 (1999), 1-28.

[Mill86] V. Miller, *uses of elliptic curves in cryptography*l Advances in Cryptology - Crypto '85, Lecture Notes in Computer Science, n218 (1986), Springer-Verlag, 417-426.

[Silv99] J. Silverman and J. Suzuki, *Elliptic curve discrete logarithms and the index calculus*, advances in Cryptology - Asiacrypt '98, Lecture Notes in Computer Science, n1514 (1999), Springer-Verlag, 110-125.

[Silv00] J. Silverman, *The Xedni calculus and the elliptic curve discrete logarithm problem*, Designs, codes and Cyptography, n20 (2000), 5-40.

[Men95] Alfred Menezes, *Elliptic curve Cryptosystems*, CryptoBytes - The Technical Newsletter of RSA Laboratories, volumen 1, number 2, Summer 1995, 1-4.

[Xedni99] M. j. jacobson, N. Koblitz, J. H. Silverman, A. Stein, E. Teske, *Analysis of the Xedni Calculus Attack*, 1999.

[ECDSA] D. Johnson, A. Menezes, S. Vanstone, *The elliptic Curve Digital Signature Algorithm (ECDSA)*. Dept. of Combinatorics & Optimization, University of Waterloo, Certicom, Canada.

[Men93] Alfred Menezes, *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, 1993

[Rami02] Manuel Rami Perat, *Factorización con curvas elípticas. Variantes del algoritmo de Lenstra.* TFC (director: Josep Maria Miret Biosca), EUP UdL, Gener 2002.

[Cosi01] Ana Cosialls Abillá, *Criptosistemas tipo el ElGamal con curvas elípticas.* TFC (directora: Magda Valls Marsal), EUP UdL, Febrer 2001.

[toolstut] Eleftherios Gkioulekas *Learning the GNU development tools.* 1998-07-29. Edition 1

[autoconf] Mark Galassi *Autoconf Tutorial*

[toolsman] David MacKenzie and Ben Elliston *Autoconf, Creating Automatic Configuration Scripts*. December 1998. Edition 2.13, for Autoconf version 2.13

[DJN] Dan Doneh, Antoine Joux, Phong Q. Nguyen *Why Textbook ElGamal and RSA Encryption are Insecure*. Extended Abstract.

[DHAES] Michel Abdalla, Mihir Bellare, Phillip Rogeway *DHAES: An Exncryption Scheme Based on the Diffie-Hellman Problem*. September 1998. Submission to IEEE P1363a.

[EPOC] Tatsuaki Okamoto, Shigenori Uchiyama, Eiichiro Fujisaki *EPOC: Efficient Probabilistic Public-Key Encryption*. November 1998. Submission to IEEE P1363a.

[EPOC3] Tatsuaki Okamoto, David Pointcheval *EPOC-3: Efficient Probabilistic Public-key Encryption - v3*. May 2000. Submission to P1363a.

[P1363a] Tatsuaki Okamoto *Proposal to IEEE P1363a*. February 2000.

[SEC1] Dan Boneh *Review ofSEC1: Elliptic Curve Cryptography*.