University of the Aegean

Department of Information
and Communication Systems
Engineering

MSc Information & Communication
Systems Security

Building elliptic curves over prime fields having arbitrary small embedding degree

by Athanasios Karakostas
323M/2006020

A Thesis presented to the University of the Aegean
in part fulfilment of the requirements
for the degree of Master of Science
in Information & Communication Systems Security

Supervisor: Dr. E. Konstantinou, Lecturer
Karlovassi, Samos, February 2008

ABSTRACT

A pairing is a function that takes as input two points on an elliptic curve and outputs an element of some multiplicative abelian group. The two pairings that are known at present are the Weil pairing and the Tate pairing. These pairings have recently found numerous applications in the design of cryptosystems. In order to implement such protocols, one needs elliptic curves over which the Weil or Tate pairings can be efficiently implemented. In particular, elliptic curves with sufficiently small embedding degrees are the most proper for such implementations. In this thesis, some well known methods for the generation of such elliptic curves are presented, implemented and evaluated experimentally.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dr. Elisavet Konstantinou, for her comments and support concerning this MSc Thesis, but also for the excellent cooperation over the whole duration of the MSc programme.

0 GENERAL INFORMATION

0.1 List of Figures and Tables

1 INTRODUCTION

1.1 Scope

Consider the Diffie-Hellman protocol that can be used to allow two parties A and B to establish a shared secret by communicating over a channel that is being monitored by an eavesdropper. The Diffie-Hellman protocol can be viewed as a one-round protocol because the two exchanged messages needed in order to complete the secret sharing are independent of each other. The protocol can easily be extended to three parties easily, using a two-round protocol [4]. A natural question to ask is whether there exists a three-party one-round key agreement protocol that is secure against eavesdroppers. This question remained open until 2000, when Joux [18] devised a surprisingly simple protocol that used bilinear pairings[4].

Joux's paper was of great interest to cryptographers, who started investigating further applications of pairings. Pairing-based cryptography is a relatively young area of cryptography that revolves around a certain function with special properties. A pairing is a function that takes as input two points on an elliptic curve and outputs an element of some multiplicative abelian group. Since then, there has been significant activity in the design and analysis of cryptographic protocols using pairings. Pairings have been accepted as an indispensable tool for the protocol designer. There has also been a tremendous amount of work on the realization and efficient implementation of bilinear pairings using the Tate pairing on elliptic curves and more general kinds of abelian varieties. These pairings have recently found numerous applications in the design of cryptosystems, such as identity-based encryption[12,14,17,19,20], identity-based signatures, short signatures[15], group signatures[13,16], non-interactive key distribution or authentication key agreement[4].

The two pairings that are known at present are the Weil pairing and the Tate pairing. In order to implement protocols such as those mentioned, one needs elliptic curves over which the Weil or Tate pairings can be efficiently implemented. In particular, elliptic curves with sufficiently small embedding degrees such as k=3,4 and 6 are the most

proper for such implementations. In this thesis, some well known methods for the generation of such elliptic curves will be presented and will be evaluated experimentally.

1.2 Contribution

Under this Thesis, a primitive software library written in the C Programming language was produced. This software library, which builds on work made previously in the field of elliptic curves and pairing-based cryptography, aims to provide implementations of algorithms that construct elliptic curves of the desired embedding degree. These elliptic curves can then be used on the construction of Weil or Tate pairings in order to implement pairing-based cryptographic protocols.

1.3 Structure of this Thesis

Chapter 2 provides the mathematical basis necessary for the following chapters. In particular, introductory information on Elliptic Curves, Scalar Multiplication of a point of an Elliptic Curve are covered in sections 2.1 and 2.2. Section 2.3 refers to Pell's Equation, a specific type of equation that will appear in almost all algorithms covered later on. Finally, Section 2.4 provides some introductory material on Pairings, the main application field for Elliptic Curves with low embedding degree.

In Chapter 3 the underlying mathematical concept concerning the construction of Elliptic Curves with Embedding Degrees k=3,4 and 6 will be presented. Based on this background, the corresponding algorithms for the construction of such curves are also presented in this chapter. Chapter 4 includes the detailed implementation of the algorithms, as well as a presentation of the measurements taken during the execution of these algorithms and comparison between the different algorithms.

Finally in Chapter 5 the Conclusions of this Thesis are presented.

2 MATHEMATICAL BACKGROUND

In this chapter the mathematical background needed to construct elliptic curves of prescribed embedding degree is set. A short introduction to Elliptic Curves is made, as well as detailed explanation of how the scalar multiplication operation is performed on Elliptic Curves. The Complex Multiplication method, which can be used to construct Elliptic Curves whose order possesses certain properties, is presented next. Pell's equation and the corresponding algorithms that solve it are also presented. This form of equation appears in all algorithms that construct Elliptic Curves having small embedding degree. Finally, a definition of pairings and details about the Tate and Weil pairings is provided.

2.1 Elliptic Curves

Elliptic Curves were first introduced to the world of cryptography in 1985, when a public-key cryptosystem based on Elliptic Curves and the ECDLP problem was proposed independently by V. Miller and N. Koblitz. Their attractiveness lies on the fact that there does not exist up to now an algorithm for solving the ECDLP problem on a properly chosen EC in sub-exponential time. Therefore, similar levels of security to 1024-bit RSA are obtained using an ECC key of only 160-bit. An elliptic curve can be defined over finite fields. When an Elliptic Curve E is defined over F_p, p>3 and prime, the notation $\mathbf{E(F_p)}$ is used. In this case the Elliptic Curve is defined by the parameters $\mathbf{a,b} \in \mathbf{F_p}$ and consists of the points P=(x,y) for $x,y \in F_p$ that satisfy

$$y^2 = x^3 + ax + b$$

together with O (point at infinity). This set of points and a special addition operation define an Abelian group, called the *EC group*. When the Elliptic Curve is defined over F_{2^m}, $\mathbf{E(F_{2^m})}$ is defined by the parameters $\mathbf{a,b} \in \mathbf{F_{2^m}}$, $b \neq 0$, and consists of the points P=(x,y) for $x,y \in F_{2^m}$ that satisfy

$$y^2 + xy = x^3 + ax^2 + b$$

together with O (point at infinity). The Order of a point of a curve is the smallest positive integer r such that $\mathbf{r\ P = O}$. The Curve Order m is the Number of points in E(Fp) or $\mathbf{\#E(Fp)}$. The Curve Order can be computed and must satisfy certain conditions to avoid

known attacks, e.g. by selecting a,b so that the curve order is divisible by large prime. The order of a point cannot exceed the order of the elliptic curve[2].

The expression $t = p + 1 - m$ (which measures the difference between m and p) is called the *Frobenius trace t*. Hasse's theorem states that $|t| \leq 2\sqrt{p}$ which gives upper and lower bounds for m based on p:

$$p + 1 - 2\sqrt{p} \leq m \leq p + 1 + 2\sqrt{p}. \text{ [2]}$$

The security of elliptic curve cryptosystems is based on the difficulty of solving the discrete logarithm problem (DLP) on the EC group. To ensure intractability of solving this problem by all known attacks, the group order m should obey the following conditions:

1. m must have a sufficiently large prime factor (larger than 2^{160}).
2. m must not be equal to p.
3. For all $1 \leq k \leq 20$, it should hold that $p^k \equiv 1 \pmod{m}$.

If the order of an EC group satisfies the above conditions, we call it *suitable*[2].

An Elliptic Curve Cryptosystem is defined by its **ECC Domain Parameters**, which are a

Septuple T = (q,FR,a,b,G,n,h)

consisting of a number q specifying a prime power (q=p or $q=2^m$), an indication FR (field representation) of the method used for representing field elements$\in$ Fq, two field elements a,b $\in$ Fq that specify the equation of the Elliptic Curve E over Fq, a base point $G=(x_G,y_G)$ on E(Fq), a prime n which is the order of G, and an integer h which is the cofactor h=#E(Fq) / n. Since the primary security parameter is n, the ECC key length is defined to be the key length of n. [1]

The fundamental protocols used in Elliptic Curve Cryptosystems are the Elliptic Curve Diffie-Hellman protocol (ECDH) used for key agreement, the Elliptic Curve Digital Signature Algorithm (ECDSA) used for digital signatures, and the Elliptic Curve Authenticated Encryption Scheme. It is beyond the scope of this thesis to analyze these protocols, for which further information is provided in [1].

2.2 Scalar Multiplication

Scalar Multiplication is the central operation used in Elliptic Curve Cryptosystems. In order to understand how Scalar Multiplication works, first Point Addition has to be presented.

The following figure illustrates point addition; In order to add points P and Q of the illustrated Elliptic Curve, we need to calculate the third point of the EC where the line defined by points P and Q crosses the Elliptic Curve, say R. The result of Point Addition P + Q is the point R', which is the symmetrical point to R on the X axis.

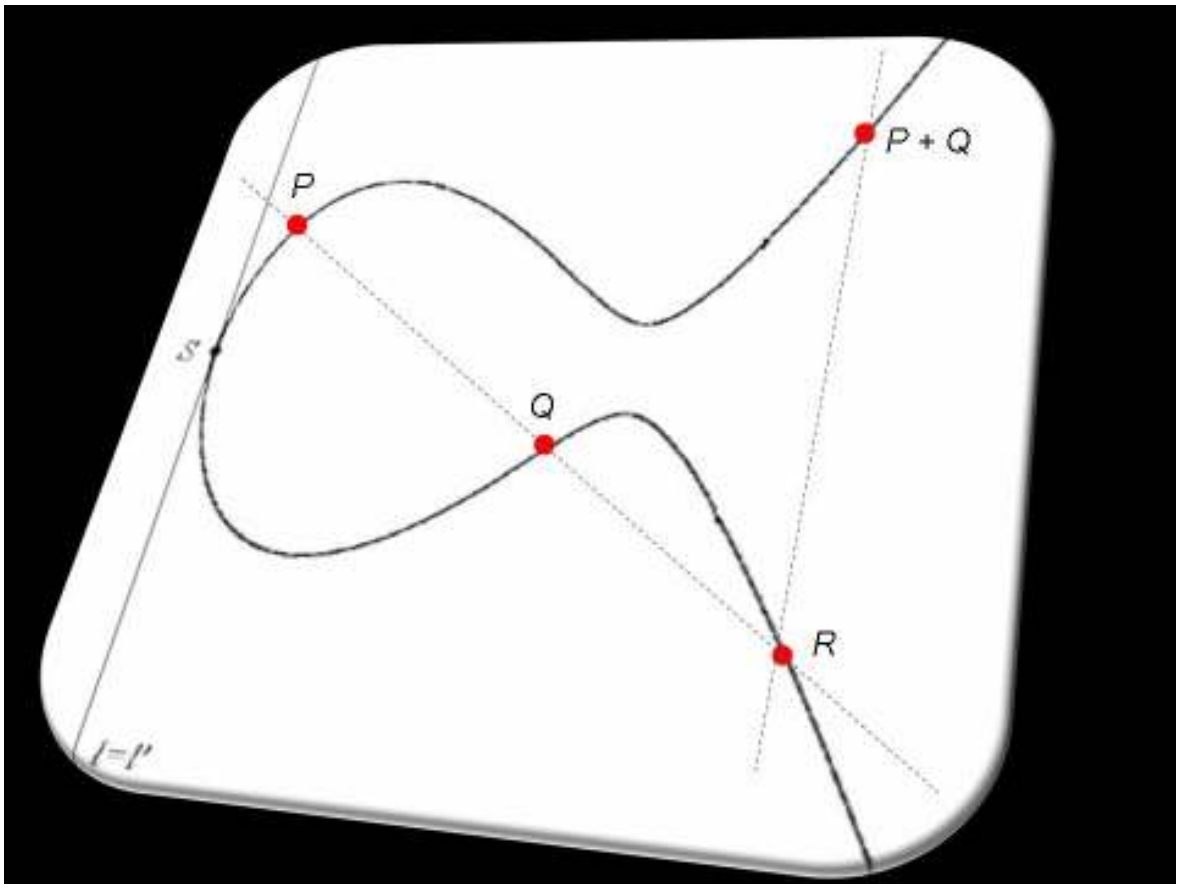


Figure 1 Elliptic Curve Point Addition

Scalar Multiplication is the operation of multiplying an integer number λ to a point P of the Elliptic Curve, or in other words the operation of adding a point P to itself λ times. The operation Q = λ P can be written as Q = P + P + ….. + P (λ times). In order to calculate P + P, the tangent of the Elliptic Curve at point P must be taken and the point R at which this tangent crosses the Elliptic Curve again must be calculated. The symmetrical point R' is the result of the addition P + P or 2P. From this point onwards, 3P, 4P, ….. λP can be calculated as described previously.

The following figure illustrates the Scalar Multiplication Q = λ P:

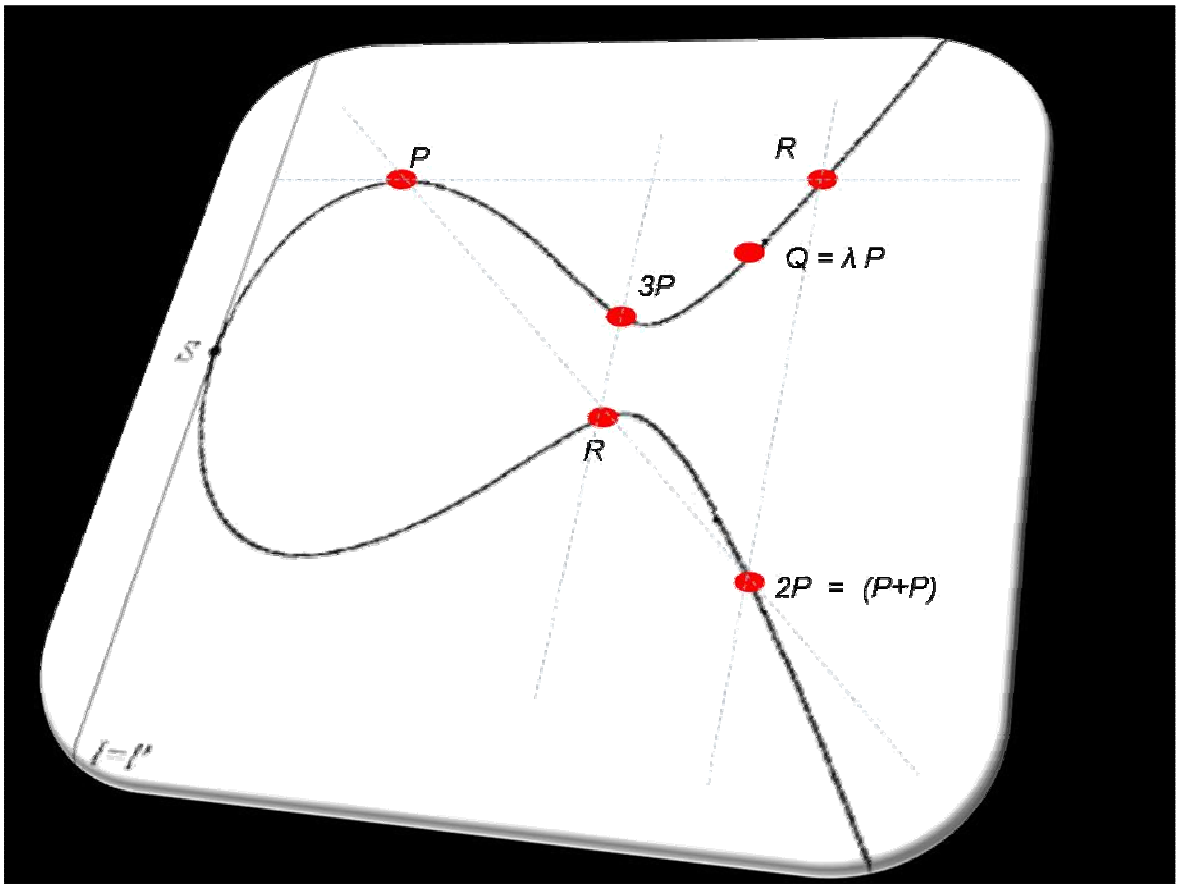


Figure 2 Elliptic Curve Scalar Multiplication

2.3 Complex Multiplication Method

The Complex Multiplication method is a method used to generate ECs whose order possesses certain properties [2]. It is especially useful if a pairing is desired. This method takes as input a number representing the order of the finite field on which the EC will be defined, and from this number determines the parameter *CM Discriminant* ***D.*** The EC is then generated by constructing polynomials based on D and finding their roots.

Two types of polynomials can be used, Hilbert Polynomials and Weber Polynomials. Hilbert polynomials have large coefficients, but their roots can be used directly to construct the EC. They are inappropriate for fast/frequent construction of ECs. The use of Hilbert polynomials in the CM method requires high precision in the arithmetic operations involved in their construction, resulting in a considerable increase of computing resources. This makes them not appropriate for fast and frequent generation of ECs. To overcome the shortcomings of Hilbert polynomials, two alternatives have been recently proposed: either to compute them off-line and store them for subsequent use, or to use Weber polynomials for certain values of *D* and produce the required Hilbert roots from them [2]. Weber polynomials on the contrary, have much smaller coefficients, therefore they are easier and faster to construct. However their roots do not construct EC directly, but instead their roots have to be transformed to Hilbert roots. The Complex Multiplication Method presented in [2] uses Weber polynomials and transforms their roots to the corresponding Hilbert polynomial roots.

The Curve Discriminant Δ and j-invariant can be calculated from a,b. Given j_0, two Elliptic Curves of j-invariant j_0 can be constructed, where the second Elliptic Curve is a twist of the first. j_0 is a root of the Hilbert polynomial which is constructed from a value D calculated from the method. The specific algorithm is analysed extensively in [2] and the corresponding functions have been implemented in ECC-LIB and used in the implementation of the algorithms that construct Elliptic Curves with prescribed embedding degree that are presented in the next chapter.

2.4 Pell's Equation – Generalised Pell Equations

Pell-type equations arise in most of the algorithms analysed in chapter 3 that follows. A **Pell equation** is an equation of the form

$$x^2 - Dy^2 = 1$$

where D, x, y are integers and D is not a square. [5]

We solve such an equation by examining the continued fraction expansion of $\sqrt{D}$. A continued fraction expansion of a real number x is obtained by finding an integer a_0 and positive integers a_1, a_2, ... such that

$$x = a_0 + (1 / (a_1 + 1 / (a_2 + ...)))$$

which we also denote by $x = [a_0, a_1, a_2, ...]$. This sequence of integers can be found by computing the following:

$$P_0 = 0,$$
$$Q_0 = 1$$
$$a_0 = \lfloor \sqrt{D} \rfloor$$
$$P_1 = a_0$$
$$Q_1 = D - a_0^2$$
$$a_n = \lfloor (a_0 + P_n) / Q_n \rfloor$$
$$P_n = a_{n-1}Q_{n-1} - P_{n-1}$$
$$Q_n = (D - P_n^2) / Q_{n-1}$$

One can show for some k we must have $a_{k+1} = 2a_0$, and after this point the a_n sequence begins repeating. That is $\sqrt{D} = [a_0, a_1, ..., a_{k+1}, a_1, ..., a_{k+1}, ...]$.

The convergents are given by

$$p_0 = a_0,$$
$$p_1 = a_0a_1 + 1,$$
$$p_n = a_np_{n-1} + p_{n-2}$$
$$q_0 = 1,$$
$$q_1 = a_1,$$
$$q_n = a_nq_{n-1} + q_{n-2}.$$

These satisfy

$$p_n^2 - Dq_n^2 = (-1)^{n+1}Q_{n+1}$$

It turns out that $(x, y) = (p_k, q_k)$ is the smallest positive integer equation of the Pell equation for odd k, and $(x, y) = (p_{2k+1}, q_{2k+1})$ is smallest for even k. Denote this minimal positive solution by (t, u). Then all positive solutions (x, y) to the Pell equation can be found via

$$x + y\sqrt{D} = (t + u\sqrt{D})^n$$

for all positive integers n. We never need the negative solutions, but these are trivial to find from the positive solutions in any event[5].

The **Generalized Pell Equation** form

$$x^2 - Dy^2 = N$$

(D is not a square, x, y are integers) can be solved by first solving the Pell Equation

$$x^2 - Dy^2 = 1$$

when $N^2 < D$. In this case, using the above method, we can compute the convergents p_n, q_n until the minimal positive solution is found. However, while doing so, we check if $p_n^2 - Dq_n^2 = N/f^2$ for some positive integer f. If so, then we append (fp_n, fq_n) to the list of solutions of the generalized Pell equation.

If no such convergents are found by the time we have reached the minimal positive solution for the Pell equation, then the generalized Pell equation has no solution. Otherwise let (t, u) be the minimal positive solution of the above Pell equation. Then for each (r, s) on the list of solutions we have a family of solutions (x, y) given by

$$(x + y\sqrt{D}) = (r + s\sqrt{D})(t + u\sqrt{D})^n$$

(for all positive integers n). These families account for all positive integer solutions to the generalized Pell equation.

When $N2 \geq D$ there are possibly other fundamental solutions to the generalized Pell equation we must add to the list before generating families of solutions. We can use brute force to find them if the numbers are small enough.

For positive N set

$$L_1 = 0,\ L2 = \sqrt{(N(t-1)/2D)}.$$

For negative N set

$$L1 = \sqrt{(-N/D)},\ L2 = \sqrt{(-N(t+1)/2D)}.$$

For all integers y satisfying $L1 \le y \le L2$ check if there exists any integer x such that $x^2 - Dy^2 = N$. Append any solutions (x, y) to our list. Also append (x,−y) if it does not appear in the family of solutions generated by (x, y)[5].

The abovementioned solutions to the Generalised Pell Equation can be implemented in the form of the following two algorithms from [8]:

Algorithm 1 Pell Equation Solver
Input: $D \in \mathbb{Z}, m \in \mathbb{Z}\backslash\{0\}$; $D > m^2$, D is not a perfect square
Output: all minimal positive solutions (x, y) : $x^2 - Dy^2 = m$

$B_{-1} \leftarrow 0, G_{-1} \leftarrow 1$
$P_0 \leftarrow 0, Q_0 \leftarrow 1, a_0 \leftarrow \lfloor\sqrt{D}\rfloor, B_0 \leftarrow 1, G_0 \leftarrow a_0$
$i \leftarrow 0$
repeat
  $i \leftarrow i + 1$
  $P_i \leftarrow a_{i-1}Q_{i-1} - P_{i-1}$
  $Q_i \leftarrow (D - P_i^2)/Q_{i-1}$
  $a_i \leftarrow \lfloor(P_i + \sqrt{D})/Q_i\rfloor$
  $B_i \leftarrow a_iB_{i-1} + B_{i-2}$
  $G_i \leftarrow a_iG_{i-1} + G_{i-2}$
until $Q_i = 1$ and $i \equiv 0 \pmod 2$
$s \leftarrow 0$
for $0 \le j \le i - 1$ **do**
  if $G_j^2 - DB_j^2 = m/f^2$ for some $f > 0$ **then**
    Output: (fG_j, fB_j)
    $s \leftarrow 1$
  end if
end for
if $s == 0$ **then**
  Output: No solutions exist
end if

Algorithm 1: Pell Equation Solver 1 [8]

Algorithm 2 Pell Equation Solver 2
Input: $D \in \mathbb{Z}$, $m \in \mathbb{Z}\backslash\{0\}$: $D \leq m^2$, D is not a perfect square
Output: all fundamental solutions (x, y) : $x^2 - Dy^2 = m$

1: find a minimal solution, (u, v), to $U^2 - DV^2 = 1$ by using Algorithm 1 with input D, 1.
2: if $m > 0$ then
3: $L_1 \leftarrow 0$, $L_2 \leftarrow \sqrt{m(u-1)/(2D)}$
4: else
5: $L_1 \leftarrow \sqrt{(-m)/D}$, $L_2 \leftarrow \sqrt{(-m)(v+1)/(2D)}$
6: end if
7: for $L_1 \leq y \leq L_2$ do
8: if $m + Dy^2$ is a square then
9: $x \leftarrow \sqrt{m + Dy^2}$
10: if (x, y) and $(-x, y)$ are not in the same class then
11: Output: $(x, y), (-x, y)$
12: else
13: Output: (x, y)
14: end if
15: end if
16: end for

Algorithm 2: Pell Equation Solver 2 [8]

2.5 Pairings

Pairings were introduced in Cryptography by Joux in 2000[18]. Pairings have recently found numerous applications in the design of cryptosystems, such as identity-based encryption, identity-based signatures, short signatures, non-interactive key distribution or authentication key agreement [12,13,14,15,16,17,19,20].

.

Definition: Bilinear Pairing on (G_1 , G_T) is a map

$$e: G_1 \times G_1 \rightarrow G_T$$

that satisfies the following conditions:

- (**bilinearity**) For all R,S,T $\in G_1$,

  $$e(R+S, T) = e(R,T)\, e(S,T)$$

  and

  $$e(R, S+T) = e(R,S)\, e(R,T)$$

  The bilinearity property means that DLP in G_1 can be reduced to DLP in G_T

- (**non-degeneracy**) $e\,(P,P) \neq 1$

- (**computability**) *e* can be efficiently computed

Two forms exist:

$$e: G_1 \times G_1 \rightarrow G_T,$$

where G_1: group of points on a curve, G_T: subgroup of multiplicative group of a related finite field, and

$$e: G_1 \times G_2 \rightarrow G_T$$

Further separation[3]:

- Type 1: $G_1 = G_2$
- Type 2: $G_1 \neq G_2$ but there exists efficiently computable $\varphi: G_2 \rightarrow G_1$
- Type 3: G1 ≠ G2 and there does not exist an efficiently computable $\varphi: G_2 \rightarrow G_1$

For example:

- G_1 is subgroup of E(Fq)
- G_2 is subgroup of $E(Fq^k)$
- G_T is subgroup of F^*q^k

In this example k is the Embedding Degree

2.5.1 Weil and Tate Pairings

Torsion points: Let K be a finite field of characteristic q, so that K = F_qm, m natural. Let E be an EC defined over K, n=#E(K). Suppose P$\in$E(K) satisfies rP = O, P has order r. Then P is called an r-torsion point, the set of r-torsion points in E(K) is E(K)[r]

The Weil pairing is a bilinear map that takes pairs of elements from E[r] and outputs an r-th root of unity in F_qk.

The Tate pairing is similar, but only the first input is from E(Fq)[r].

Roots and poles: When studying the polynomial f(x)/g(x), the roots of f(x) are the zeroes and the roots of g(x) are the poles.

Weil Pairing

Let E be an EC containing n points over F_q, G be a cyclic subgroup of E(Fq) of order r, with r,q co-prime. Let k be the smallest positive integer so that E(F_qk) contains all of E[r].

Weil Pairing f: E[r] x E[r] → F_qk

For a pair of points P,Q$\in$E[r], choose any R,S $\in$E[F_qk] such that S≠R, P+R, P+R-Q, R-Q

Let f_P be a rational function with divisor $(f_P)=(P+R)^r/(R)^r$
Let f_Q be a rational function with divisor $(f_Q)=(Q+S)^r/(S)^r$

Define $f(P,Q) = (f_P(Q+S) / f_P(S)) / (f_Q(P+R) / f_Q(R))$

Finding explicit expressions for these functions is infeasible, however Miller's Algorithm evaluates these functions at the required points [4]

Tate Pairing

Let E be an EC containing n points over F_q, G be a cyclic subgroup of $E(F_q)$ of order r, with r,q co-prime. Let k be the smallest positive integer so that $r \mid k^q - 1$ and $K = F_qk$.

Weil Pairing $e: E[r] \cap E(K) \times E(K) / r E(K) \rightarrow K^* / K^*r$

Let f_P be a rational function with divisor $(f_P)=(P)^r$

Choose an $R \in E(K)$ such that $R \neq P, P-Q, O, -Q$

Define $f(P,Q) = f_P(Q+R) / f_P(R)$

Weil/Tate and bilinear map

Let E be an EC containing n points over F_q, G be a cyclic subgroup of $E(F_q)$ of order r, embedding degree $k > 1$.

By defining $G_1 = G$, $G_2 = E[r]$, G_T to be the r-th roots of unity in F_qk, the Weil and Tate pairings satisfy the definition of a bilinear map given earlier.

If $k>1$, then both the Tate and Weil pairings may be computed by performing field arithmetic in F_qk.

Arbitrary Embedding Degree: Given any positive integer k, we can construct pairings with embedding degree k, but the subgroup size r will have far fewer bits than q.

3 PRIME ORDER CURVES OF ARBITRARY SMALL EMBEDDING DEGREE k

In this chapter the procedures that need to be followed in order to construct Elliptic Curves having arbitrary small Embedding Degree k is given for the cases k=3,k=4 and k=6. From the mathematical analysis of these problems, the corresponding algorithm that needs to be implemented will emerge.

The following theorem, proven in [8], can be used as a starting point in our analysis:

Let E/Fq be an ordinary elliptic curve defined over a finite field Fq. Let n = #E(Fq) be a prime and k the embedding degree of E.
(i) Suppose q > 64. Then k = 3 if and only if $q = 12 l^2 - 1$ and $t = -1 \pm 6l$ for some $l \in Z$.
(ii) Suppose q > 36. Then k = 4 if and only if $q = l^2 + l + 1$ and $t = -l, l + 1$ for some $l \in Z$.
(iii) Suppose q > 64. Then k = 6 if and only if $q = 4l^2 + 1$ and $t = 1 \pm 2l$ for some $l \in Z$.

The prime order ordinary elliptic curves with embedding degree k = 3, 4, and 6 are completely classified by this theorem. One way of constructing such an elliptic curve E/Fq with trace t is the complex multiplication (CM) method. In this method, given q and t one writes the following CM equation:

$$4q - t^2 = DV^2$$

where D is the square free part of $4q - t^2$. Then any root of the Hilbert class polynomial $H_{-D}(x)$ modulo q gives rise to an elliptic curve E/Fq which has #E(Fq) = q + 1 − t. The CM method is efficient only for small values of D; in practice, we are restricted to $D \leq 10^{10}$. In cryptographic applications it is desirable for q and n to be prime and also log n ≈ log q ≈ 160 for efficiency and security reasons.

3.1 Constructing Elliptic Curves with Embedding Degree k=4, k=6

Let q and n be prime integers. Let E/Fq be an ordinary elliptic curve with embedding degree k = 6, and #E(Fq) = n.

E has an embedding degree k = 6 if and only if $q = 4l^2 + 1$, and $t = 1 \pm 2l$ for some $l \in Z$.

Then $t = 1 \pm 2l$ gives $n = q + 1 - t = 4l^2 \mp 2l + 1$.

The CM equation can be written as

$$4q - t^2 = D'V^2 <=> (6l \pm 1)^2 - 3D'V^2 = -8$$

The above shows that in order to construct an elliptic curve of prime order with embedding degree k = 6 we have to find some special pair of solutions to the following Pell equation:

$$X^2 - DY^2 = -8,\ D > 0,\ D \equiv 0 \pmod 3$$

If (x, y) is a solution to this equation then we have to guarantee that $x \equiv -1 \pmod 6$ or $x \equiv 1 \pmod 6$. In the former case, setting $l = (x + 1)/6$, we must have $q = 4l^2 + 1$ is prime, and $n = 4l^2 - 2l + 1$ is prime. In the latter case, setting $l = (x - 1)/6$, we must have $q = 4l^2 + 1$ is prime, and $n = 4l^2 + 2l + 1$ is prime.

Similarly for k=4, E has an embedding degree k = 4 if and only if $q = l^2 + l + 1$, and $t = -l, l+1$ for some $l \in Z$. Then $t = -l$ gives $n = q + 1 - t = l^2 + 2l + 2$ and $t = l+1$ gives $n = l^2 + 1$. For $t = -l$, the primality of n requires that $l \equiv 1 \pmod 2$. Therefore, we can replace l by $2l' - 1$. For $t = l + 1$, we can replace l by $2l'$ since n is prime and so l is even. In both cases, the CM equation can be written as

$$4q - t^2 = D'V^2 <=> (6l' \pm 1)^2 - 3D'V^2 = -8$$

which is identical to that produced for k=6.

The analysis from this point of the section onwards refers to k=6, the small differences to construct elliptic curves with k=4 are highlighted at the end of the current section.
We have shown that constructing E is reduced to finding some suitable solutions to the Pell equation

$$(3.1.1)\ X^2 - DY^2 = -8,\ D > 0,\ D \equiv 0 \pmod 3,\ D \equiv 1 \pmod 2$$

For efficiency reasons it is essential to keep D small. Therefore, the general strategy is first fixing a small D and then tracing for suitable solutions to the above equation using the algorithms developed previously. In this section we try to find some necessary conditions on D, and also analyze the solution classes of the equation above in order to gain some efficiency in searching for suitable elliptic curves.

If (x, y) is a minimal solution to $X^2 - DY^2 = n$, and (u, v) is a minimal solution to $U^2 - DV^2 = 1$ then all primitive solutions (x_j , y_j) in the class of (x, y) can be generated as follows:

$$(3.1.2)\ x_j + y_j\sqrt{D} = \pm(x + y\sqrt{D})(u + v\sqrt{D})^j \text{ ,where } j \in Z$$

Let m be a nonzero integer, and let D be a positive integer such that D is not a perfect square and $D \equiv 0 \pmod 3$. Then, the sequence $(x_j)_{j\in Z}$ defined as in (3.1.2) and belonging to $X^2 - DY^2 = m$ is periodic modulo 6 with period at most 2. This is because, supposing $j \geq 0$, by expanding (3.2) we can write $x_0 = x$, $y_0 = y$, $x_{i+1} = x_iu+y_ivD$, and $y_{i+1} = x_iv+y_iu$ for $i \geq 0$. Then, using $u^2+v^2D = 1+2v^2D$ and $2D \equiv 0 \pmod 6$, we get:

$$\begin{aligned} x_i &= x_{i-1}u + y_{i-1}vD \\ &= x_{i-2}(u^2 + v^2D) + 2y_{i-2}uvD \\ &\equiv x_{i-2} \pmod 6 \end{aligned}$$

for $i \geq 2$.

If an ordinary elliptic curve E over a prime field with embedding degree 6 is constructible then (3.1.1) must have only primitive solutions and the value D in (3.1.1) must satisfy $D \equiv 9 \pmod{24}$. Also, −2 must be a square modulo D. This is because if E with k = 6 is constructible then there exists some integer l satisfying $12\,l^2 \pm 4l + 3 = D'V^2$, or in other

words, $4l(3l \pm 1) + 3 = D'V^2$, and so $D'V^2 \equiv 3 \pmod 8$. Hence, $D' \equiv 3 \pmod 8$ proving that $D \equiv 9 \pmod{24}$ since $D = 3D'$. Reducing (3.1.1) modulo D proves that -2 must be a square modulo D.

Before giving the searching algorithm we shall summarize the above results:

- D should be fixed such that $0 < D \leq 10^{10}$, D/3 is square free, $D \equiv 9 \pmod{24}$, -2 is a square modulo D.
- Let (u, v) be a minimal solution to $U^2 - DV^2 = 1$. If there is a solution to $X^2 - DY^2 = -8$ then it is enough to find, if exists, only one minimal solution,(x0, y0).
- Let $(x_j , y_j) = \pm(x_0, y_0)(u, v)^j$ be the set of all solutions in the same class as (x, y). It is enough to consider only one of the solutions (x_j , y_j) and $-(x_j , y_j)$.
- If $x_0 \neq \pm 1 \pmod 6$ then there do not exist any suitable solutions (x_j , y_j) for

  $$j \equiv 0 \pmod 2$$

  Similarly, if $x_1 \neq \pm 1 \pmod 6$ then there do not exist any suitable solutions (x_j , y_j) for

  $$j \equiv 1 \pmod 2$$

Algorithm 3 from [8] can be implemented in order to construct Elliptic Curves with Embedding Degree k=6. This algorithm searches through all solutions (x_j , y_j) satisfying $(x_j + y_j\sqrt{D}) = (x + y\sqrt{D})(u + v\sqrt{D})^j$ for $j \geq 0$.

Algorithm 3 EC parameters with $k = 6$
Input: N, z
Output: EC parameters (q, n, k, D') where q and n are N-bit primes, $k = 6$, and $D' \leq z$ (where $4q - t^2 = D'V^2$)

1: for $0 < D \leq 3z$, $D/3$ square free, $D \equiv 9 \pmod{24}$, -2 is a square modulo D do
2: if $D > 64$ then
3: find a minimal solution, (x_0, y_0), to $X^2 - DY^2 = -8$ by using Algorithm 1 with input D, -8.
4: else
5: find a minimal solution, (x_0, y_0), to $X^2 - DY^2 = -8$ by using Algorithm 2 with input D, -8.
6: end if
7: find a minimal solution, (u, v), to $U^2 - DV^2 = 1$ by using Algorithm 1 with input D, 1.
8: $x_1 \leftarrow x_0 u + y_0 v D$, $y_1 \leftarrow x_0 v + y_0 u$
9: $x \leftarrow x_0$, $y \leftarrow y_0$, $x' \leftarrow x_1$, $y' \leftarrow y_1$
10: if $x_0 \equiv \pm 1 \pmod 6$ then
11: while $|x| \leq 2^{\lceil N/2 \rceil}$ do
12: $l \leftarrow (x \mp 1)/6$
13: if $\lfloor (N-2) \rfloor /2 \leq \log_2 l < \lceil (N-2)/2 \rceil$ then
14: $q \leftarrow 4l^2 + 1$, $n \leftarrow 4l^2 \mp 2l + 1$
15: if q and n are primes then
16: Output (q, n)
17: end if
18: end if
19: $\widetilde{x} \leftarrow x$
20: $x \leftarrow x(2u^2 - 1) + 2yuvD$
21: $y \leftarrow y(2u^2 - 1) + 2\widetilde{x}uv$
22: end while
23: end if
24: if $x_1 \equiv \pm 1 \pmod 6$ then
25: while $|x'| \leq 2^{\lceil N/2 \rceil}$ do
26: $l \leftarrow (x' \mp 1)/6$
27: if $(N-2)/2 \leq \log_2 l < \lceil (N-2)/2 \rceil$ then
28: $q \leftarrow 4l^2 + 1$, $n \leftarrow 4l^2 \mp 2l + 1$
29: if q and n are primes then
30: Output (q, n)
31: end if
32: end if
33: $\widetilde{x}' \leftarrow x'$
34: $x' \leftarrow x'(2u^2 - 1) + 2y'uvD$
35: $y' \leftarrow y'(2u^2 - 1) + 2\widetilde{x}'uv$
36: end while
37: end if
38: end for

Algorithm3: Construction of Elliptic Curve with embedding degree k=6

Returning to the case k=4, the difference in the algorithm is only for lines 14 and 28. These should read $q \leftarrow l^2 + l + 1$ and, say for $t=l+1$, $n \leftarrow l^2 + 1$

3.2 Constructing Elliptic Curves with Embedding Degree k=3

The construction of elliptic curves with embedding degree k = 6 and k=4 was analyzed previously by discussing the set of solutions to the corresponding Pell equation. In this section, a similar analysis is given for embedding degree k = 3.

As mentioned previously, if E is an ordinary elliptic curve defined over a finite field Fq, q is prime, and n = #E(Fq) is prime then E has an embedding degree k = 3 if and only if

$$q = 12l^2 - 1, \text{ and } t = -1 \pm 6l \text{ for some } l \in Z$$

Note that $t = -1 \pm 6l$ gives $n = q + 1 - t = 12l^2 \mp 6l + 1$. The CM equation can be written as

$$4q - t^2 = D'V^2 \Leftrightarrow (6l \pm 3)^2 - 3D'V^2 = 24$$

The above shows that in order to construct an elliptic curve of prime order with embedding degree k = 3 we have to find some special pair of solutions to the following Pell equation:

$$X^2 - DY^2 = 24,\ D > 0,\ D \equiv 0 \pmod 3$$

If (x, y) is a solution to this equation, we have to guarantee that $x \equiv 3 \pmod 6$ and that for $l = (x \pm 3)/6$ we must have $q = 12l^2 - 1$ and $n = 12l^2 \pm 6l + 1$ are primes.

Constructing an elliptic curve E with k = 3 is reduced to finding some suitable solutions to the Pell Equation

$$\textit{(3.2.1)}\ X^2 - DY^2 = 24,\ D > 0,\ D \equiv 0 \pmod 3$$

The analysis of this equation was mostly done by Miyaji, Nakabayashi, and Takano [6]. The summary of their results states that if an ordinary elliptic curve E over a prime field with embedding degree 3 is constructible, then equation (3.2.1) must have only primitive solutions and the value D in (3.2.1) must satisfy $D \equiv 57 \pmod{72}$. Moreover, this

equation has exactly two classes of solutions and if α = (x, y) is a solution to (3.2.1) then α and α′ = (x,−y) represent the two different solution classes.

Now, let (x, y) be a minimal solution to $X^2 - DY^2 = n$, and let (u, v) be a minimal solution to $U^2 - DV^2 = 1$. Recall that all primitive solutions (x_j , y_j) in the class of (x, y) can be generated as follows:

$$x_j + y_j\sqrt{D} = \pm (x + y\sqrt{D})(u + v\sqrt{D})^j \text{ ,where } j \in Z.$$

Before giving the searching algorithm we shall summarize the above results:

- D should be fixed such that $0 < D \leq 10^{10}$, D/3 is square free, $D \equiv 57 \pmod{72}$. Also, 6 must be a square modulo D.

- Let (u, v) be a minimal solution to $U^2 - DV^2 = 1$. If there is a solution to $X^2 - DY^2 = 24$ then it is enough to find, if it exists, only one minimal solution, say (x0, y0).

- Let $(x_j , y_j) = (x_0, y_0)(u, v)^j$ be the set of all solutions as in the same class as (x, y). It is enough to consider only one of the solutions (x_j , y_j) and $-(x_j , y_j)$.

- If $x_0 \neq 3 \pmod 6$ then there do not exist any suitable solutions (x_j, y_j) for $j \equiv 0 \pmod 2$

  If $x_1 \neq 3 \pmod 6$ then there do not exist any suitable solutions (x_j, y_j) for $j \equiv 1 \pmod 2$.

The following algorithm from [8] can be implemented in order to construct Elliptic Curves with Embedding Degree k=3:

Algorithm 4 EC parameters with $k = 3$

Input: N

Output: EC parameters (q, n, k, D) where q and n are N-bit primes, $k = 3$, and $D \leq x$ (where $4q - t^2 = Dy^2$)

1: for $0 < D \leq 32$, $D/3$ square free, $D \equiv 57 \pmod{72}$, 6 is a square modulo D **do**
2: **if** $D > 576$ **then**
3: find a minimal solution, (x_0, y_0), to $X^2 - DY^2 = 24$ by using Algorithm 1 with input D, 24.
4: **else**
5: find a minimal solution, (x_0, y_0), to $X^2 - DY^2 = 24$ by using Algorithm 2 with input D, 24.
6: **end if**
7: find a minimal solution, (u, v), to $U^2 - DV^2 = 1$ by using Algorithm 1 with input D, 1.
8: $x_1 \leftarrow x_0 u + y_0 v D$, $y_1 \leftarrow x_0 v + y_0 u$
9: $x \leftarrow x_1$, $y \leftarrow y_1$, $x' \leftarrow x_1$, $y' \leftarrow y_1$
10: **if** $x_1 \equiv 3 \pmod 6$ **then**
11: **while** $|x| \leq 2^{\lfloor N/4 \rfloor}$ **do**
12: $t_1 \leftarrow (x-3)/6$, $t_2 \leftarrow (x+3)/6$
13: **if** $\lfloor (N-4)/2 \rfloor \leq \log_2 t_1, \log_2 t_2 < \lceil (N-3)/2 \rceil$ **then**
14: $q_0 \leftarrow 12t_1^2 - 1$, $n_1 \leftarrow 12t_1^2 - 6t + 1$, $q_2 \leftarrow 12t_2^2 - 1$, $n_2 \leftarrow 12t_2^2 + 6t_2 + 1$
15: **if** q_1 and n_1 are primes **then**
16: Output (q_1, n_1)
17: **end if**
18: **if** q_2 and n_2 are primes **then**
19: Output (q_2, n_2)
20: **end if**
21: **end if**
22: $\tilde{x} \leftarrow x$
23: $x \leftarrow x(2u^2 - 1) + 2yuvD$
24: $y \leftarrow y(2u^2 - 1) + 2\tilde{x}uv$
25: **end while**
26: **end if**
27: **if** $x_1 \equiv 2 \pmod 6$ **then**
28: **while** $|x'| \leq 2^{\lfloor N/4 \rfloor}$ **do**
29: $t_1 \leftarrow (x'-3)/6$, $t_2 \leftarrow (x'+3)/6$
30: **if** $\lfloor (N-4)/2 \rfloor \leq \log_2 t_1, \log_2 t_2 < \lceil (N-3)/2 \rceil$ **then**
31: $q_1 \leftarrow 12t_1^2 - 1$, $n_1 \leftarrow 12t_1^2 - 6t + 1$, $q_2 \leftarrow 12t_2^2 - 1$, $n_2 \leftarrow 12t_2^2 + 6t_2 + 1$
32: **if** q_1 and n_1 are primes **then**
33: Output (q_1, n_1)
34: **end if**
35: **if** q_2 and n_2 are primes **then**
36: Output (q_2, n_2)
37: **end if**
38: **end if**
39: $\tilde{x}' \leftarrow x'$
40: $x' \leftarrow x'(2u^2 - 1) + 2y'uvD$
41: $y' \leftarrow y'(2u^2 - 1) + 2\tilde{x}'uv$
42: **end while**
43: **end if**
44: **end for**

Algorithm4: Construction of Elliptic Curve with embedding degree k=3

3.3 Constructing Elliptic Curves with Trace of Frobenius t=3

Given the upper bound *UP* >0 on a prime *p*, this algorithm outputs a prime-order elliptic curve *E*/F*p* with *t* =3, or *fail* if such an *E*/F*p* does not exist[6].

1. Choose a positive integer d such that $d \equiv 19 \pmod{24}$.

2. Set $p = dl^2 + dl + (d+9)/4$, $l > 0$.

3. If $p > UP$, then output *fail* and terminate the algorithm. Otherwise goto step 4.

4. If both p and $p-2$ are prime, then goto step 5.

Otherwise goto step 2 and try the next l.

5. Compute the Hilbert class polynomial $P_d(x)$.

6. Solve a root j_0 of $P_d(x) \equiv 0 \pmod{p}$.

7. Construct two elliptic curves E_{j0} and E'_{j0}:

$Ej_0 : y^2 = x^3 + aj_0x + bj_0$, $E'j_0 : y^2 = x^3 + aj_0c^2x + bj_0c^3$,

where $aj_0 = 3j_0 / 1728-j_0 \pmod{p}$,

$bj_0 = 2j_0 / 1728-j_0 \pmod{p}$,

and c is any quadratic non-residue in Fp.

8. Output $E \in \{Ej_0, E'j_0\}$ with $\#E(\mathrm{F}p) = p - 2$ and terminate the algorithm.

Algorithm 5: Construction of Prime Order EC with t=3

4 IMPLEMENTATION AND EXPERIMENTAL RESULTS

All the algorithms that are presented in this chapter have been implemented in C, making use of related freely available libraries, and in particular:

- ECC-LIB, developed by E. Konstantinou et al [9]
- Ben Lynn's Pairing-Based Cryptography Library (PBC) [10]
- The GNU Multiple Precision Arithmetic Library (GNUMP) [11]

The algorithms have been grouped into one source file, GenEC.c. The main() function of this program presents the user a simple menu with the available choices of algorithms, as shown in the figure below:

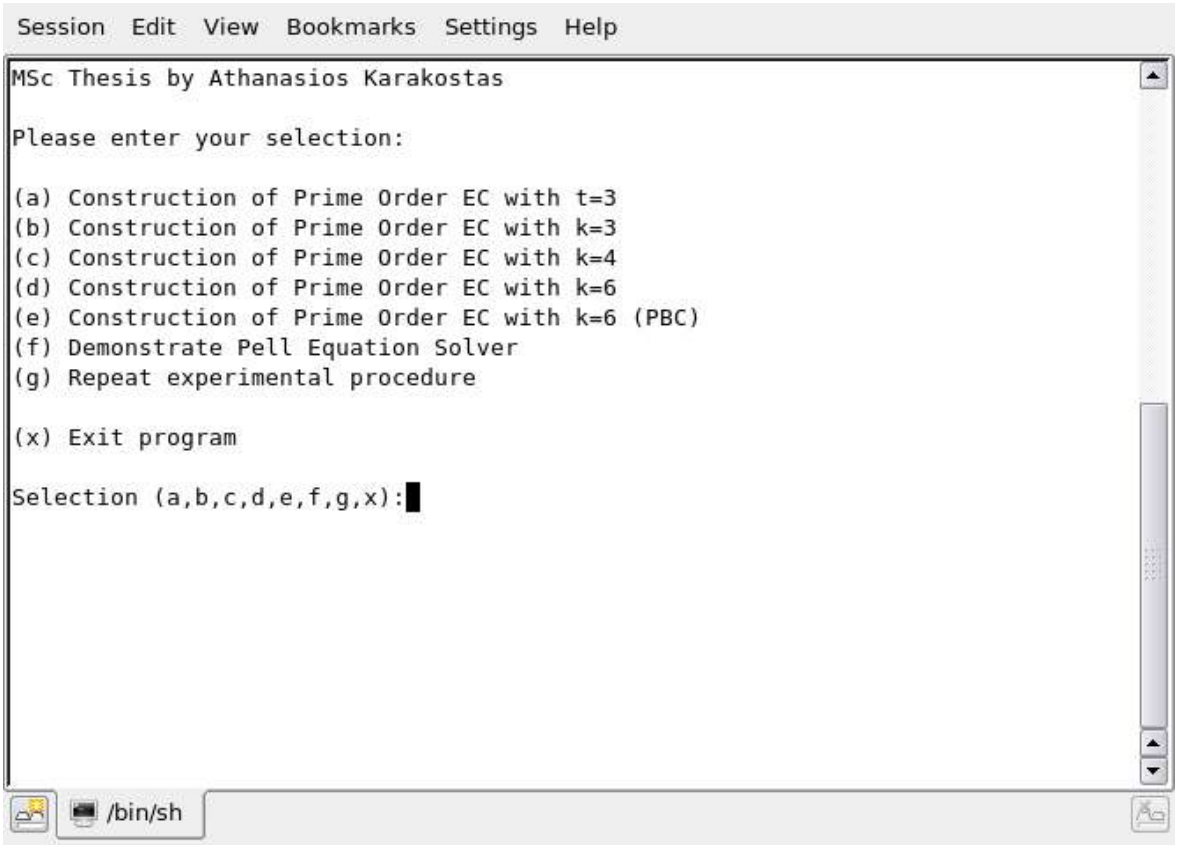


Figure 3 Main menu of GenEC application

4.1 Libraries used

4.1.1 The GNU MP Library

GMP[11] is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating point numbers. There is no practical limit to the precision except the ones implied by the available memory in the machine GMP runs on. The first GMP release was made in 1991. It is continually developed and maintained since. It is distributed under the GNU GPL, which means that the library is free to use, share, and improve, however this licence sets firm restrictions on the use with non-free programs. GMP is part of the GNU project. The main target applications for GMP are cryptography applications and research, Internet security applications, algebra systems, computational algebra research, etc.

There are several categories of functions in GMP. The most used ones are:

1. High-level signed integer arithmetic functions (mpz). There are about 140 arithmetic and logic functions in this category.

2. High-level rational arithmetic functions (mpq). This category consists of about 35 functions, but all signed integer arithmetic functions can be used too, by applying them to the numerator and denominator separately.

3. High-level floating-point arithmetic functions (mpf). This is the GMP function category to use if the C type `double' doesn't give enough precision for an application. There are about 65 functions in this category.

For the purposes of this thesis, the most widely used functions of the GMP library are those of the *mpz* category. Full description for all these functions is provided on the GNU MP manual, which can be obtained from [11].

4.1.2 The PBC Library

The PBC (Pairing Based-Cryptography) library is a free C library (released also under the GNU Public License) based on the GMP library to perform the mathematical operations of the underlying pairing-based cryptosystem. The PBC library is a free portable C library designed to make it easy to implement pairing-based cryptosystems. It provides an abstract interface to a cyclic group with a bilinear pairing, and the programmer does not need to worry about, or even know about elliptic curves. It is built on top of GMP, using the latter to perform arbitrary precision arithmetic on integers, rationals and floats with strong emphasis on portability and speed.

The PBC library is designed to be the backbone of implementations of pairing-based cryptosystems, thus speed and portability are important goals. It provides routines such as elliptic curve generation, elliptic curve arithmetic and pairing computation. Thanks to the GMP library, despite being written in C, execution times are reasonable. According to the author of the library, on a 1GHz Pentium III the execution times are as follows:

- Fastest pairing: 11ms
- Short pairing: 31ms

The PBC library also provides usage examples, such as:

- Boneh-Franklin identity-based encryption
- Hess identity-based signatures
- Joux tripartite Diffie-Hellman
- Paterson identity-based signatures
- Yuan-Li identity-based authenticated key agreement
- Zhang-Kim identity-based blind/ring signatures
- Zhang-Safavi-Naini-Susilo signatures

The main author of the PBC library is Ben Lynn, who is still developing and maintaining it, though many others have also contributed.

For the purposes of this Thesis, the PBC library was used to construct Elliptic Curves having Embedding Degree k=6, using a procedure from the PBC library directly to accomplish this task. Measurements were taken for this procedure in order to compare it to the newly-developed procedures.

4.1.3 The ECC-LIB Library

The ECC-LIB library contains an implementation of the Complex Multiplication method, which is used to obtain the curve parameters given the value of D. The value of D can be obtained in turn given t and q or n and q, which comprise the outputs of the described algorithms for generation of elliptic curves having embedding degree k=3, k=4 or k=6.

The final_hilbert procedure generates the Hilbert Class polynomial for a given value of D, while the zpoly_print procedure of the ECC-LIB can be used to display the generated polynomial on screen. Having generated the polynomial, the procedure myRecurse can be used to obtain a root j_0 of this polynomial.

The ECC-LIB also contains procedures that perform Scalar Multiplication on an elliptic curve's point. These procedures were used in the algorithm described previously for generating an Elliptic Curve having t=3. In particular, the rand_point procedure was used to obtain a random point G of a curve and the procedure point_mult was used to perform scalar multiplication on this point.

It must be noted that the library contains also several other procedures (e.g concerning the generation of Weber polynomials) that were not used in the course of this project, but could be utilised in future versions of the produced library.

4.2 Solving Pell's Equation

Algorithm 1 that was presented in chapter 2.5 was implemented as shown below:

```
//Pell Equation Solver 1, implemented according to K.Karabina thesis
int PellEquationSolver_1(mpz_t *D,mpz_t *m,mpz_t *solution_x,mpz_t *solution_y){
	mpz_t B, preB, prepreB, G, preG, prepreG, P, preP, Q, preQ, A, preA, factor,
G2_DB2, f;
	unsigned long i,j;

	mpz_init(B);
	mpz_init(preB);
	mpz_init(prepreB);
	mpz_init(G);
	mpz_init(preG);
	mpz_init(prepreG);
	mpz_init(P);
	mpz_init(Q);
	mpz_init(preP);
	mpz_init(preQ);
	mpz_init(A);
	mpz_init(preA);
	mpz_init(factor);
	mpz_init(G2_DB2);
	mpz_init(f);

	printf("\nPell Equation Solver 1 input: D=");
	mpz_out_str(stdout, 10, *D);
	printf(", m=");
	mpz_out_str(stdout, 10, *m);
	printf("\n");

	//1
	mpz_set_ui(prepreB,(unsigned)0);
	mpz_set_ui(prepreG,(unsigned)1);

	//2
	mpz_set_ui(preP,(unsigned)0);
	mpz_set_ui(preQ,(unsigned)1);
	mpz_sqrt(preA,*D);
	mpz_set_ui(preB,(unsigned)1);
	mpz_set(preG,preA);

	//3
	i=0;

	//4
	do {
		//5
		i++;

		//6
		mpz_mul(P,preA,preQ);
		mpz_sub(P,P,preP);

		//7
		mpz_mul(factor,P,P);
		mpz_sub(Q,*D,factor);
		mpz_tdiv_q(Q,Q,preQ);

		//8
		mpz_sqrt(factor,*D);
		mpz_add(A,factor,P);
		mpz_tdiv_q(A,A,Q);
```

```
            //9
            mpz_mul(B,A,preB);
            mpz_add(B,B,prepreB);

            //10
            mpz_mul(G,A,preG);
            mpz_add(G,G,prepreG);

            //prepare for next round
            mpz_set(preA,A);
            mpz_set(preP,P);
            mpz_set(preQ,Q);
            mpz_set(prepreB,preB);
            mpz_set(preB,B);
            mpz_set(prepreG,preG);
            mpz_set(preG,G);

      //11
      } while ( !((mpz_cmp_ui(Q,(unsigned)1)==0)  && (i % 2 == 0)));

      /*Due to memory constraints, proposed algorithm is not easy to implement.
        Instead, the B and G parameters will be calculated again using the same
algorithm
        for this part - same code as above that was used to determine the limit i*/

      //1
      mpz_set_ui(prepreB,(unsigned)0);
      mpz_set_ui(prepreG,(unsigned)1);

      //2
      mpz_set_ui(preP,(unsigned)0);
      mpz_set_ui(preQ,(unsigned)1);
      mpz_sqrt(preA,*D);
      mpz_set_ui(preB,(unsigned)1);
      mpz_set(preG,preA);

      for(j=1;j<i;j++){ //j<=i;;

            //6
            mpz_mul(P,preA,preQ);
            mpz_sub(P,P,preP);

            //7
            mpz_mul(factor,P,P);
            mpz_sub(Q,*D,factor);
            mpz_tdiv_q(Q,Q,preQ);

            //8
            mpz_sqrt(factor,*D);
            mpz_add(A,factor,P);
            mpz_tdiv_q(A,A,Q);

            //9
            mpz_mul(B,A,preB);
            mpz_add(B,B,prepreB);

            //10
            mpz_mul(G,A,preG);
            mpz_add(G,G,prepreG);

            //Bj, Gj are now calculated!! We can continue according to the algorithm
            mpz_mul(factor, B, B);
            mpz_mul(factor,factor,*D);
            mpz_mul(G2_DB2, G, G);
            mpz_sub(G2_DB2, G2_DB2, factor);
```

```
            //can we find an f!=0 so that G^2 - D*B^2 = m / f^2?
            // G^2 - D*B^2 = m / f^2 <=> f^2 = m / (G^2 - D*B^2)
            mpz_tdiv_q(factor, *m, G2_DB2);

            if(mpz_cmp_ui(factor,(unsigned)0) > 0) {
                  //a square root (and possibly an f) exist
                  mpz_sqrt(f,factor); /*if f>0 and m/f^2=G^2-D*B^2, f is found!*/

                  //verify f is correct
                  mpz_mul(factor,f,f);
                  mpz_tdiv_q(factor, *m ,factor);

                  if (mpz_cmp(factor, G2_DB2)==0) {
                        //f is found!
                        mpz_mul(*solution_x,f,G);
                        mpz_mul(*solution_y,f,B);

                        mpz_clear(B);
                        mpz_clear(preB);
                        mpz_clear(prepreB);
                        mpz_clear(G);
                        mpz_init(preG);
                        mpz_init(prepreG);
                        mpz_clear(P);
                        mpz_clear(Q);
                        mpz_clear(preP);
                        mpz_clear(preQ);
                        mpz_clear(A);
                        mpz_clear(preA);
                        mpz_clear(factor);
                        mpz_clear(G2_DB2);
                        mpz_clear(f);

                        return EC_SUCCESS;
                  }
            }

            //prepare for next round
            mpz_set(preA,A);
            mpz_set(preP,P);
            mpz_set(preQ,Q);
            mpz_set(prepreB,preB);
            mpz_set(preB,B);
            mpz_set(prepreG,preG);
            mpz_set(preG,G);
      }

      //if we made it here, we were not successful
      mpz_clear(B);
      mpz_clear(preB);
      mpz_clear(prepreB);
      mpz_clear(G);
      mpz_init(preG);
      mpz_init(prepreG);
      mpz_clear(P);
      mpz_clear(Q);
      mpz_clear(preP);
      mpz_clear(preQ);
      mpz_clear(A);
      mpz_clear(preA);
      mpz_clear(factor);
      mpz_clear(G2_DB2);
      mpz_clear(f);

      return EC_FAILURE;
}
```

Similarly, Algorithm 2 was implemented as shown below:

```
//Pell Equation Solver 2, implemented according to K.Karabina thesis
int PellEquationSolver_2(mpz_t *D,mpz_t *m,mpz_t *solution_x,mpz_t *solution_y){
	mpz_t u, v, L1, L2, y, x, factor;
	int res;

	mpz_init(u);
	mpz_init(v);
	mpz_init(L1);
	mpz_init(L2);
	mpz_init(x);
	mpz_init(y);
	mpz_init(factor);

	printf("\nPell Equation Solver 2 input: D=");
	mpz_out_str(stdout, 10, *D);
	printf(", m=");
	mpz_out_str(stdout, 10, *m);
	printf("\n");

	//1
	mpz_set_ui(factor,(unsigned)1);
	res = PellEquationSolver_1(D, &factor, &u, &v);

	if (res==EC_FAILURE) {
		printf("\nPell Equation Solver 1 returned failure for D=");
		mpz_out_str(stdout, 10, *D);
		printf("\n");
		return EC_FAILURE;
	}

	//2
	if (mpz_cmp_ui(*m, (unsigned)0) > 0){
		//3
		mpz_set_ui(L1,(unsigned)0);
		mpz_sub_ui(L2,u,(unsigned)1);
		mpz_mul(L2,L2,*m);
		mpz_mul_ui(factor,*D,(unsigned)2);
		mpz_tdiv_q(L2,L2,factor);
		mpz_sqrt(L2,L2);
	} else {
		//5
		mpz_neg(L1,*m);
		mpz_tdiv_q(L1,L1,*D);
		mpz_sqrt(L1,L1);

		mpz_neg(L2,*m);
		mpz_add_ui(factor,v,(unsigned)1);
		mpz_mul(L2,L2,factor);
		mpz_mul_ui(factor,*D,(unsigned)2);
		mpz_tdiv_q(L2,L2,factor);
		mpz_sqrt(L2,L2);
	}
	//7 - cannot be done with for loop due to mpz_t usage
	mpz_set(y,L1);
	while(mpz_cmp(L2,y) >= 0) { //while L2 >= y >= L1
		mpz_mul(factor,y,y);
		mpz_mul(factor,factor, *D);
		mpz_add(factor,factor, *m);
		//now factor = m + d*(y^2)

		//8
		if (mpz_perfect_square_p(factor) != 0) {
			mpz_sqrt(x,factor);
			mpz_set(*solution_x, x);
			mpz_set(*solution_y, y);
```

```
                printf("\nPell Equation Solver 2 output: x=");
                mpz_out_str(stdout, 10, *solution_x);
                printf(", y=");
                mpz_out_str(stdout, 10, *solution_y);
                printf("\n");

                mpz_clear(u);
                mpz_clear(v);
                mpz_clear(L1);
                mpz_clear(L2);
                mpz_clear(x);
                mpz_clear(y);
                mpz_clear(factor);

                return EC_SUCCESS;
            }
            //increase y
            mpz_add_ui(y,y,(unsigned)1);
        }
        //if we made it here, we were not successful
        mpz_clear(u);
        mpz_clear(v);
        mpz_clear(L1);
        mpz_clear(L2);
        mpz_clear(x);
        mpz_clear(y);
        mpz_clear(factor);
        return EC_FAILURE;
}
```

A sample of the execution of the above algorithms is given below. The corresponding Pell equation is $x^2 – 95y^2 = 5$, and a pair of roots is correctly calculated as (10,1):

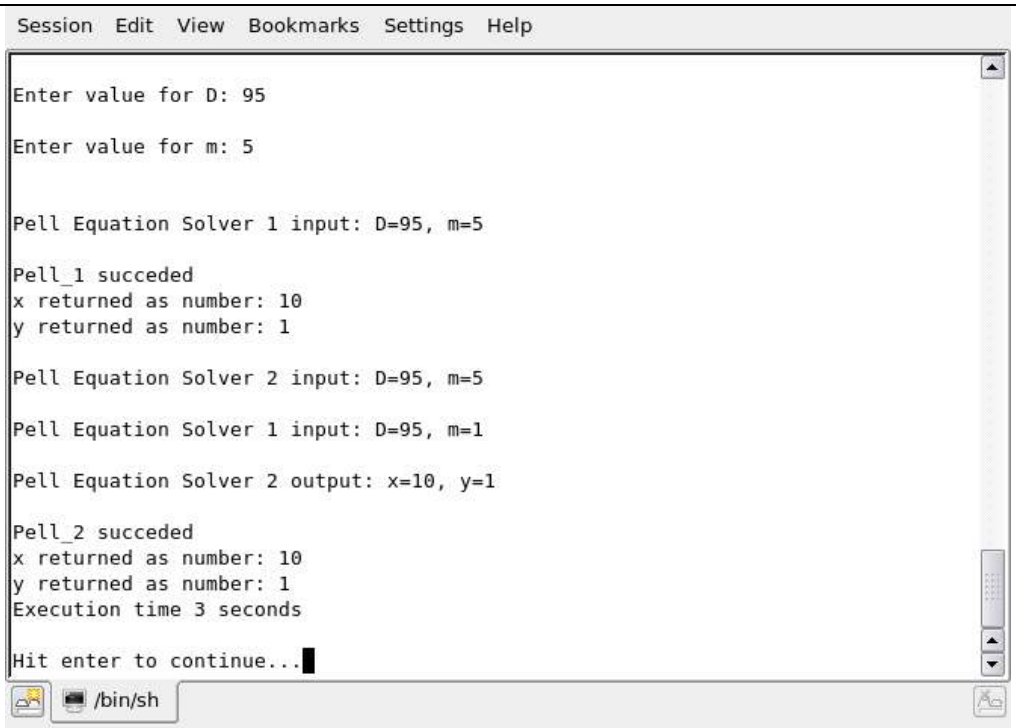


Figure 4 Solving Pell's Equation

4.3 Constructing Elliptic Curves with Trace of Frobenius t=3

Algorithm 3 from section 3.1 has been implemented as ConstructPrimeOrderEC. Its source code is shown below, the comments in the source code indicate which step of the algorithm is performed by each block of code:

```
int ConstructPrimeOrderEC (mpz_t UP /*mpz_t* value*/) {
	/*variable declarations*/
	mpz_t d,p,p_minus_2,l,a,b,c,c2,c3,factor;
	mpz_t G[2];
	mpz_t H[2];
	long dPnew,i;
	long root_size = 0;

	mpz_t Pnew[POLY_SIZE];
	mpz_t j0[POLY_SIZE];
	mpz_t curv1[2];
	mpz_t curv2[2];

	for (i = 0; i < POLY_SIZE; i++) {
		mpz_init(Pnew[i]);
		mpz_init(j0[i]);
	}

	/*mpz_t initializations*/
	mpz_init(d);
	mpz_init(p);
	mpz_init(p_minus_2);
	mpz_init(l);
	 mpz_init(factor);/*general purpose variable*/
	mpz_init(a);
	mpz_init(b);
	mpz_init(c);
	mpz_init(c2);
	mpz_init(c3);
	mpz_init(curv1[0]);
	mpz_init(curv1[1]);
	mpz_init(curv2[0]);
	mpz_init(curv2[1]);
	mpz_init(G[0]);
	mpz_init(G[1]);
	mpz_init(H[0]);
	mpz_init(H[1]);

	/*Step 1: Generate d */
	/*QUESTION: How many bits for d?*/
	 Generate_19_Mod_24_Number(&d, 160);
	printf("d returned as number: "); mpz_out_str(stdout, 10, d); printf("\n");

	/*REMOVE THE NEXT TWO LINES TO AVOID D=259*/
	mpz_set_ui(d,(unsigned)259);/* d=259, could be 43,67,91,115,...*/
	printf("...for faster execution, d=");
	mpz_out_str(stdout, 10, d);
	printf (" will be used to execute algorithm steps\n");

	/*start with l=0*/
	mpz_set_ui(l,(unsigned)0);
	do { /*repeat until p,p-2 are prime*/
		/*Step 2: Calculate p = d*l*l + d*l + (d+9)/4 */
		/*increase l by 1*/
		mpz_add_ui(l,l,(unsigned)1);
		/*printf("l set to: "); mpz_out_str(stdout, 10, l); printf("\n");*/

		mpz_add_ui(p,d,(unsigned)9);/* p=d+9*/
```

```
		mpz_cdiv_q_ui(p,p,(unsigned)4);/* p=(d+9)/4*/

		/*Find d*l*/
		mpz_mul(factor,d,l);
		mpz_add(p,p,factor);/* p=dl + (d+9)/4*/

		/*Find d*l*l*/
		mpz_mul(factor,factor,l);

		mpz_add(p,p,factor);/* p=d*l*l + dl + (d+9)/4 */
		/*printf("p calculated: "); mpz_out_str(stdout, 10, p); printf("\n");*/

		if (mpz_cmp(p,UP)>0) { /*if p>UP*/
			/*clear all mpz_t variables*/
			mpz_clear(d);
			mpz_clear(p);
			mpz_clear(p_minus_2);
			mpz_clear(l);
			mpz_clear(factor);
			mpz_clear(a);
			mpz_clear(b);
			mpz_clear(c);
			mpz_clear(c2);
			mpz_clear(c3);
			mpz_clear(G[0]);
			mpz_clear(G[1]);
			mpz_clear(H[0]);
			mpz_clear(H[1]);
			mpz_clear(curv1[0]);
			mpz_clear(curv1[1]);
			mpz_clear(curv2[0]);
			mpz_clear(curv2[1]);
			for (i = 0; i < POLY_SIZE; i++) {
				mpz_clear(Pnew[i]);
				mpz_clear(j0[i]);
			}

			return EC_FAILURE;
		}

		mpz_sub_ui(p_minus_2,p,(unsigned)2);
	} while((MillerRabinPrimalityTesting(&p,100) == 0)
||(MillerRabinPrimalityTesting(&p_minus_2,100) == 0));

	printf("l set to: "); mpz_out_str(stdout, 10, l); printf("\n");
	printf("p   calculated: "); mpz_out_str(stdout, 10, p); printf("\n");
	printf("p-2 calculated: "); mpz_out_str(stdout, 10, p_minus_2); printf("\n");

	/*Step 5: Compute the Hilbert Class Polynomial Pd(x)*/
	printf("Calling Hilbert Polynomial calculator function...\n");
	final_hilbert(mpz_get_ui(d), Pnew, &dPnew);
	printf("Dpnew: %d\n",dPnew);
	printf(" The Hilbert polynomial is:\n");
	zpoly_print(dPnew, Pnew);

	/*Step 6: Solve a root j0 of Pd(x)≡0 (mod p)*/
	//returns always 0,temporarily j0[0]=1726
	myRecurse(dPnew, &p, Pnew, j0, &root_size);
	//Recurse(dPnew, &p, Pnew, j0, &root_size);
	//mpz_set_ui(j0[0],(unsigned)1726);

	printf("Number of roots returned: %d\n",root_size);
	for(i=0;i<root_size;i++) {
		printf("Root j[%d] calculated: ",i); mpz_out_str(stdout, 10, j0[i]);
printf("\n");
	}
```

```
        printf("Using Root j[0]: "); mpz_out_str(stdout, 10, j0[0]); printf("\n");

        /*Step 7: Construct two elliptic curves Ej0 and E'j0*/
        //calculate 1728-j0
        mpz_ui_sub(factor, (unsigned)1728, j0[0]);

        //a=3*j0 / factor mod p
        mpz_mul_ui(a, j0[0],(unsigned)3);
        mpz_div(a, a, factor);
        mpz_mod(a, a, p);

        //b=2*j0 / factor mod p
        mpz_mul_ui(b, j0[0],(unsigned)2);
        mpz_div(b, b, factor);
        mpz_mod(b, b, p);

        printf("a calculated: "); mpz_out_str(stdout, 10, a); printf("\n");
        printf("b calculated: "); mpz_out_str(stdout, 10, b); printf("\n");

        /*Calculate a suitable value c*/
        /*c must be a quadratic non-residue*/
        /*QUESTION: How many bits for c?*/
        Generate_Quadratic_Non_Residue_in_Fp(&c, &p, 20);
        printf("c calculated: "); mpz_out_str(stdout, 10, c); printf("\n");
        //calculate a*c^2, store to c2
        //calculate b*c^3, store to c3
        mpz_mul(c2,c,c);
        mpz_mul(c3,c2,c);
        mpz_mul(c2,c2,a);
        mpz_mul(c3,c3,b);
        mpz_mod(c2,c2,p);
        mpz_mod(c3,c3,p);

        /*Step 8: Output E such that E □ {Ej0, E'j0} with #E(Fp) = p-2 and terminate
algorithm*/
        printf("Constructed two curves:\n");
        printf("Ej0: y^2 = x^3 + ");
        mpz_out_str(stdout, 10, a);
        printf ("x + ");
        mpz_out_str(stdout, 10, b);
        printf("\n");
        mpz_set(curv1[0],a);
        mpz_set(curv1[1],b);

        printf("and\n");
        printf("E'j0: y^2 = x^3 + ");
        mpz_out_str(stdout, 10, c2);
        printf ("x + ");
        mpz_out_str(stdout, 10, c3);
        printf("\n");
        mpz_set(curv2[0],c2);
        mpz_set(curv2[1],c3);

        printf("modulo ");
        mpz_out_str(stdout, 10, p);
        printf("\n");

        /*for which of the two is it that (p-2)*G = O ?*/
        //First try Ej0, resulting point is H
        printf("Ej0: scalar multiplication of random G by p-2 results in:\n");
        rand_point(curv1,&p,G);
        point_mult(curv1,G,&p_minus_2,H,&p);
        print_point(H);

        //Then try E'j0, resulting point is H
        printf("E'j0: scalar multiplication of random G by p-2 results in:\n");
        rand_point(curv2,&p,G);
```

```
	point_mult(curv2,G,&p_minus_2,H,&p);
	print_point(H);

	printf("The correct curve is the one that results in point O\n");

	/*clear all mpz_t variables*/
	mpz_clear(d);
	mpz_clear(p);
	mpz_clear(p_minus_2);
	mpz_clear(l);
	 mpz_clear(factor);
	mpz_clear(a);
	mpz_clear(b);
	mpz_clear(c);
	mpz_clear(c2);
	mpz_clear(c3);
	mpz_clear(G[0]);
	mpz_clear(G[1]);
	mpz_clear(H[0]);
	mpz_clear(H[1]);
	for (i = 0; i < POLY_SIZE; i++) {
		mpz_clear(Pnew[i]);
		mpz_clear(j0[i]);
	}
	mpz_clear(curv1[0]);
	mpz_clear(curv1[1]);
	mpz_clear(curv2[0]);
	mpz_clear(curv2[1]);

	printf("SUCCESS!\n");
	return EC_SUCCESS;
}
```

4.4 Constructing Elliptic Curves with Embedding Degree k=3

The corresponding source code for Algorithm 4 is shown below:

```
//Construction of k=3 curves, implemented according to K.Karabina thesis
void Construct_k3_curve (long N) {

	mpz_t
D_mpz,i_mpz,factor,square,x0,y0,x1,y1,x,y,xx,yy,pre_x,pre_xx,u,v,exp_high,exp_low;
	mpz_t l1,l2,q1,q2,n1,n2,limit;
	long D,i,comp_low,comp_high;
	int D_mod_72_is_57,D_third_square_free, six_is_square_mod_D;
	int res;

	void clear_mpz(){
		mpz_clear(D_mpz);
		mpz_clear(i_mpz);
		mpz_clear(factor);
		mpz_clear(square);
		mpz_clear(x0);
		mpz_clear(y0);
		mpz_clear(x1);
		mpz_clear(y1);
		mpz_clear(x);
		mpz_clear(y);
		mpz_clear(xx);
		mpz_clear(yy);
		mpz_clear(pre_x);
		mpz_clear(pre_xx);
		mpz_clear(u);
		mpz_clear(v);
		mpz_clear(l1);
		mpz_clear(l2);
		mpz_clear(q1);
		mpz_clear(q2);
		mpz_clear(n1);
		mpz_clear(n2);
		mpz_clear(exp_high);
		mpz_clear(exp_low);
		mpz_clear(limit);
	}

	//start of procedure
	mpz_init(D_mpz);
	mpz_init(i_mpz);
	mpz_init(factor);
	mpz_init(square);
	mpz_init(x0);
	mpz_init(y0);
	mpz_init(x1);
	mpz_init(y1);
	mpz_init(x);
	mpz_init(y);
	mpz_init(xx);
	mpz_init(yy);
	mpz_init(pre_x);
	mpz_init(pre_xx);
	mpz_init(u);
	mpz_init(v);
	mpz_init(l1);
	mpz_init(l2);
	mpz_init(q1);
	mpz_init(q2);
	mpz_init(n1);
	mpz_init(n2);
```

```
	mpz_init(exp_high);
	mpz_init(exp_low);
	mpz_init(limit);

	printf("Construction of EC with k=3, q and n are %d-bit primes...\n",(N+1)/2);
	mpz_ui_pow_ui(limit,(unsigned)2,(unsigned)((N+1)/2)); //limit = 2^(N/2)
	printf("The limit for %d bits is: ",(N+1)/2); mpz_out_str(stdout, 10, limit);
printf("\n");

	for(D=1;D<z;D++) {
		/*if (D % 1000==0) {
			printf(".");
		}*/

		mpz_set_ui(D_mpz,(unsigned)D);
		mpz_set_ui(factor,(unsigned)(D/3));

		D_mod_72_is_57 = (D % 72 == 57);

		//D is not good if it is divisible by 3
		AND the quotient is a perfect square
		D_third_square_free = (D%3!=0) ||
		((D%3==0)&&(mpz_perfect_square_p(factor)==0));

		//6 must be a square mod D (there exists a number x for which x^2=6mod D)
		six_is_square_mod_D = 0;
		for(i=1;i<D;i++) {
			mpz_set_ui(i_mpz,(unsigned)i);
			mpz_powm_ui(square,i_mpz,(unsigned)2,D_mpz); //square=i^2 mod D
			if (mpz_cmp_ui(square,(unsigned)6)==0) {
				//printf("6 is %d^2 mod %d\n",i,D);
				six_is_square_mod_D = 1;
			}
		}

		mpz_set_ui(factor,(unsigned)D);
		if (D_mod_72_is_57 && D_third_square_free &&
		    six_is_square_mod_D && (mpz_perfect_square_p(factor)==0)){
			//printf("Found a D:%d\n",D);

			//2
			mpz_set_ui(factor,(unsigned)24);
			if (D > 576) {
				//3
				res = PellEquationSolver_1(&D_mpz, &factor, &x0, &y0);
				if (res = EC_FAILURE) {
					printf("\n D>576: Pell_1 failed\n");
					clear_mpz();
					return;
				}/* else {
					printf("\nPell_1 succeded\n");
					printf("x0 returned as number: ");
					 mpz_out_str(stdout, 10, x0); printf("\n");
					printf("y0 returned as number: ");
					mpz_out_str(stdout, 10, y0); printf("\n");

				}*/
			} else {
				//5
				res = PellEquationSolver_2(&D_mpz, &factor, &x0, &y0);
				if (res = EC_FAILURE) {
					printf("\n D<576: Pell_2 failed\n");
					clear_mpz();
					return;
				} /*else {
					printf("\nPell_2 succeded\n");
					printf("x0 returned as number: ");
```

```
                mpz_out_str(stdout, 10, x0); printf("\n");
                printf("y0 returned as number: ");
                mpz_out_str(stdout, 10, y0); printf("\n");

        }*/
}

//7
mpz_set_ui(factor,(unsigned)1);
res = PellEquationSolver_1(&D_mpz, &factor, &u, &v);
if (res = EC_FAILURE) {
        printf("\n u,v: Pell_1 failed\n");
        clear_mpz();
        return;
} /*else {
        printf("\nPell_1 succeded\n");
        printf("u returned as number: ");
        mpz_out_str(stdout, 10, u); printf("\n");
        printf("v returned as number: ");
        mpz_out_str(stdout, 10, v); printf("\n");
}*/

//8
mpz_mul(factor,x0,u);
mpz_mul(x1,y0,v);
mpz_mul(x1,x1,D_mpz);
mpz_add(x1,x1,factor);

mpz_mul(factor,x0,v);
mpz_mul(y1,y0,u);
mpz_add(y1,y1,factor);

//9
mpz_set(x,x0);
mpz_set(y,y0);
mpz_set(xx,x1);
mpz_set(yy,y1);

//10
mpz_mod_ui(factor,x0,(unsigned)6);

if(mpz_cmp_ui(factor,(unsigned)3)==0){
        mpz_abs(factor,x);
        while(mpz_cmp(limit,factor) >= 0) {
                //12
                //l1=(x-3)/6
                mpz_sub_ui(l1,x,(unsigned)3);
                mpz_tdiv_q_ui(l1,l1,(unsigned)6);
                //l2=(x+3)/6
                mpz_add_ui(l2,x,(unsigned)3);
                mpz_tdiv_q_ui(l2,l2,(unsigned)6);

                //13
                comp_low=(N-5)/8; //-5 to round down
                comp_high=(N-2)/2; //-2 to round up
                //comp_low < log2 l1 is the same as saying
                2^comp_low is less than l1

mpz_ui_pow_ui(exp_low,(unsigned)2,(unsigned)comp_low);

mpz_ui_pow_ui(exp_high,(unsigned)2,(unsigned)comp_high);

                if((mpz_cmp(exp_low,l1) <= 0) &&
                (mpz_cmp(l2,exp_high) < 0)) {
                        //14

                        mpz_pow_ui(q1,l1,(unsigned)2);
```

```
            mpz_mul_ui(q1,q1,(unsigned)12);
            mpz_sub_ui(q1,q1,(unsigned)1);

            mpz_pow_ui(n1,l1,(unsigned)2);
            mpz_mul_ui(n1,n1,(unsigned)12);
            mpz_add_ui(n1,n1,(unsigned)1);
            mpz_mul_ui(factor,l1,(unsigned)6);
            mpz_sub(n1,n1,factor);

            mpz_pow_ui(q2,l2,(unsigned)2);
            mpz_mul_ui(q2,q2,(unsigned)12);
            mpz_sub_ui(q2,q2,(unsigned)1);

            mpz_pow_ui(n2,l2,(unsigned)2);
            mpz_mul_ui(n2,n2,(unsigned)12);
            mpz_add_ui(n2,n2,(unsigned)1);
            mpz_mul_ui(factor,l2,(unsigned)6);
            mpz_add(n2,n2,factor);

            if( MillerRabinPrimalityTesting (&q1,20)
             && MillerRabinPrimalityTesting (&n1,20)) {
                  printf("\nSuccess!\n");
                  printf("k : 3\n");
                  printf("D : %d\n",D);
                  printf("n : ");
                  mpz_out_str(stdout, 10, n1);
                  printf("\n");
                  printf("q : ");
                  mpz_out_str(stdout, 10, q1);
                  printf("\n");
                  mpz_powm_ui(factor,q1,(unsigned)3,n1);
                  printf("q^3 mod n=");
                  mpz_out_str(stdout, 10, factor);
                  printf("\n");

                  clear_mpz();
                  return;
            }

            if( MillerRabinPrimalityTesting (&q2,20)
             && MillerRabinPrimalityTesting (&n2,20)) {
                  printf("\nSuccess!\n");
                  printf("k : 3\n");
                  printf("D : %d\n",D);
                  printf("n : ");
                  mpz_out_str(stdout, 10, n2);
                  printf("\n");
                  printf("q : ");
                  mpz_out_str(stdout, 10, q2);
                  printf("\n");
                  mpz_powm_ui(factor,q2,(unsigned)3,n2);
                  printf("q^3 mod n=");
                  mpz_out_str(stdout, 10, factor);
                  printf("\n");

                  clear_mpz();
                  return;
            }
      }
      //22
      mpz_set(pre_x,x);
      //23-24
      mpz_pow_ui(factor,u,(unsigned)2);
      mpz_mul_ui(factor,factor,(unsigned)2);
      mpz_sub_ui(factor,factor,(unsigned)1);

      mpz_mul(x,x,factor);
```

```
                        mpz_mul(y,y,factor);

                        mpz_mul(factor,u,v);
                        mpz_mul_ui(factor,factor,(unsigned)D);
                        mpz_mul(factor,factor,y);
                        mpz_mul_ui(factor,factor,(unsigned)2);
                        mpz_add(x,x,factor);

                        mpz_mul(factor,u,v);
                        mpz_mul(factor,factor,pre_x);
                        mpz_mul_ui(factor,factor,(unsigned)2);
                        mpz_add(y,y,factor);
                }
        }

        //27
        mpz_mod_ui(factor,x1,(unsigned)6);
        if(mpz_cmp_ui(factor,(unsigned)3)==0){
                mpz_abs(factor,xx);
                while(mpz_cmp(limit,factor) >= 0) {
                        //29
                        //l1=(x'-3)/6
                        mpz_sub_ui(l1,xx,(unsigned)3);
                        mpz_tdiv_q_ui(l1,l1,(unsigned)6);
                        //l2=(x'+3)/6
                        mpz_add_ui(l2,xx,(unsigned)3);
                        mpz_tdiv_q_ui(l2,l2,(unsigned)6);

                        //30
                        comp_low=(N-5)/8; //-5 to round down msg:was /2
                        comp_high=(N-2)/2; //-2 to round up
                        //comp_low < log2 l1 is the same as saying
                        2^comp_low is less than l1

mpz_ui_pow_ui(exp_low,(unsigned)2,(unsigned)comp_low);

mpz_ui_pow_ui(exp_high,(unsigned)2,(unsigned)comp_high);

                        if((mpz_cmp(exp_low,l1) <= 0) &&
                        (mpz_cmp(l2,exp_high) < 0)) {

                                //31
                                mpz_pow_ui(q1,l1,(unsigned)2);
                                mpz_mul_ui(q1,q1,(unsigned)12);
                                mpz_sub_ui(q1,q1,(unsigned)1);

                                mpz_pow_ui(n1,l1,(unsigned)2);
                                mpz_mul_ui(n1,n1,(unsigned)12);
                                mpz_add_ui(n1,n1,(unsigned)1);
                                mpz_mul_ui(factor,l1,(unsigned)6);
                                mpz_sub(n1,n1,factor);

                                mpz_pow_ui(q2,l2,(unsigned)2);
                                mpz_mul_ui(q2,q2,(unsigned)12);
                                mpz_sub_ui(q2,q2,(unsigned)1);

                                mpz_pow_ui(n2,l2,(unsigned)2);
                                mpz_mul_ui(n2,n2,(unsigned)12);
                                mpz_add_ui(n2,n2,(unsigned)1);
                                mpz_mul_ui(factor,l2,(unsigned)6);
                                mpz_add(n2,n2,factor);


                                if( MillerRabinPrimalityTesting (&q1,20)
                                 && MillerRabinPrimalityTesting (&n1,20)) {
                                        printf("\nSuccess!\n");
```

```
                                        printf("k : 3\n");
                                        printf("D : %d\n",D);
                                        printf("n : ");
                                        mpz_out_str(stdout, 10, n1);
                                        printf("\n");
                                        printf("q : ");
                                        mpz_out_str(stdout, 10, q1);
                                        printf("\n");
                                        mpz_powm_ui(factor,q1,(unsigned)3,n1);
                                        printf("q^3 mod n=");
                                        mpz_out_str(stdout, 10, factor);
                                        printf("\n");

                                        clear_mpz();
                                        return;
                                }

                                if( MillerRabinPrimalityTesting (&q2,20)
                                 && MillerRabinPrimalityTesting (&n2,20)) {
                                        printf("\nSuccess!\n");
                                        printf("k : 3\n");
                                        printf("D : %d\n",D);
                                        printf("n : ");
                                        mpz_out_str(stdout, 10, n2);
                                        printf("\n");
                                        printf("q : ");
                                        mpz_out_str(stdout, 10, q2);
                                        printf("\n");
                                        mpz_powm_ui(factor,q2,(unsigned)3,n2);
                                        printf("q^3 mod n=");
                                        mpz_out_str(stdout, 10, factor);
                                        printf("\n");

                                        clear_mpz();
                                        return;
                                }
                        }

                        //39
                        mpz_set(pre_xx,xx);
                        //40-41
                        mpz_pow_ui(factor,u,(unsigned)2);
                        mpz_mul_ui(factor,factor,(unsigned)2);
                        mpz_sub_ui(factor,factor,(unsigned)1);

                        mpz_mul(xx,xx,factor);
                        mpz_mul(yy,yy,factor);

                        mpz_mul(factor,u,v);
                        mpz_mul_ui(factor,factor,(unsigned)D);
                        mpz_mul(factor,factor,yy);
                        mpz_mul_ui(factor,factor,(unsigned)2);
                        mpz_add(xx,xx,factor);

                        mpz_mul(factor,u,v);
                        mpz_mul(factor,factor,pre_xx);
                        mpz_mul_ui(factor,factor,(unsigned)2);
                        mpz_add(yy,yy,factor);
                }
        }
    }

    printf("\nNo suitable EC with k=3 found, try increasing the value of z...\n");
    clear_mpz();
}
```

Example of the execution of the specific algorithm is illustrated below:

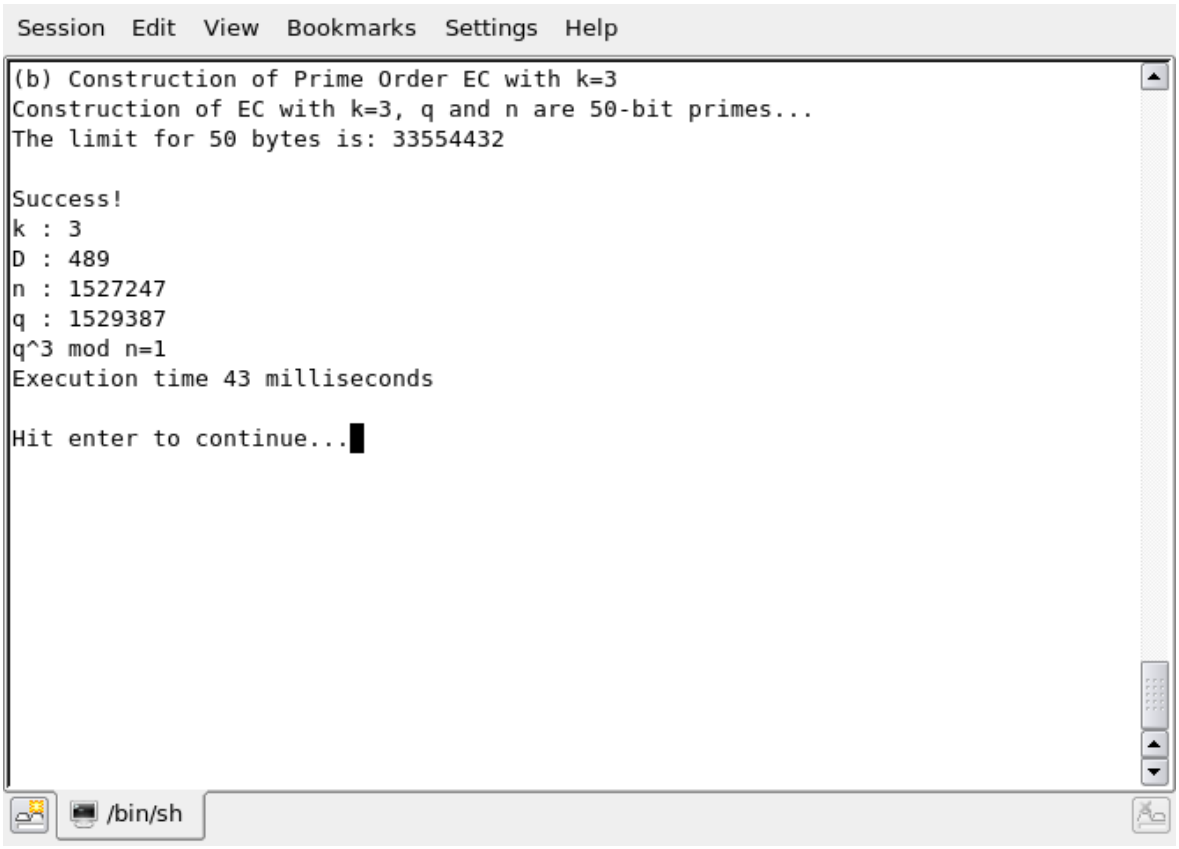


Figure 5 Construction of Elliptic Curve with Embedding Degree k=3

4.5 Constructing Elliptic Curves with Embedding Degree k=4

The corresponding source code for Algorithm 4 is shown below:

```
void Construct_k4_curve (long N) {

	mpz_t D_mpz,i_mpz,factor,square,x0,y0,x1,y1,x,y,
	      xx,yy,pre_x,pre_xx,u,v,exp_high,exp_low;
	mpz_t l1,q1,n1,limit;
	long D,i,comp_low,comp_high;
	int D_mod_24_is_9,D_third_square_free, minus_two_is_square_mod_D;
	int res;

	void clear_mpz(){
	      mpz_clear(D_mpz);
	      mpz_clear(i_mpz);
	      mpz_clear(factor);
	      mpz_clear(square);
	      mpz_clear(x0);
	      mpz_clear(y0);
	      mpz_clear(x1);
	      mpz_clear(y1);
	      mpz_clear(x);
	      mpz_clear(y);
	      mpz_clear(xx);
	      mpz_clear(yy);
	      mpz_clear(pre_x);
	      mpz_clear(pre_xx);
	      mpz_clear(u);
	      mpz_clear(v);
	      mpz_clear(l1);
	      mpz_clear(q1);
	      mpz_clear(n1);
	      mpz_clear(exp_high);
	      mpz_clear(exp_low);
	      mpz_clear(limit);
	}

	//start of procedure
	mpz_init(D_mpz);
	mpz_init(i_mpz);
	mpz_init(factor);
	mpz_init(square);
	mpz_init(x0);
	mpz_init(y0);
	mpz_init(x1);
	mpz_init(y1);
	mpz_init(x);
	mpz_init(y);
	mpz_init(xx);
	mpz_init(yy);
	mpz_init(pre_x);
	mpz_init(pre_xx);
	mpz_init(u);
	mpz_init(v);
	mpz_init(l1);
	mpz_init(q1);
	mpz_init(n1);
	mpz_init(exp_high);
	mpz_init(exp_low);
	mpz_init(limit);

	printf("Construction of EC with k=4, q and n are %d-bit primes...\n",(N+1)/2);
	mpz_ui_pow_ui(limit,(unsigned)2,(unsigned)((N+1)/2)); //limit = 2^(N/2)
	printf("The limit for %d bits is: ",(N+1)/2); mpz_out_str(stdout, 10, limit);
```

```
printf("\n");

for(D=1;D<z;D++) {
	/*if (D % 1000==0) {
		printf(".");
	}*/

	mpz_set_ui(D_mpz,(unsigned)D);
	mpz_set_ui(factor,(unsigned)(D/3));

	D_mod_24_is_9 = (D % 24 == 9);
	//D is not good if it is divisible by 3
	AND the quotient is a perfect square
	D_third_square_free = (D%3!=0) ||
	((D%3==0)&&(mpz_perfect_square_p(factor)==0));

	//6 must be a square mod D (there exists a number x for which x^2=6 modD)
	minus_two_is_square_mod_D = 0;
	for(i=1;i<D;i++) {
		mpz_set_ui(i_mpz,(unsigned)i);
		mpz_powm_ui(square,i_mpz,(unsigned)2,D_mpz); //square=i^2 mod D
		if (mpz_cmp_ui(square,(unsigned)(D-2))==0) {
			//printf("-2 is %d^2 mod %d\n",i,D);
			minus_two_is_square_mod_D = 1;
		}
	}

	mpz_set_ui(factor,(unsigned)D);
	if (D_mod_24_is_9 && D_third_square_free
	 && minus_two_is_square_mod_D && (mpz_perfect_square_p(factor)==0)){
		//printf("Found a D:%d\n",D);

		//2
		mpz_set_si(factor,(signed)(-8));
		if (D > 64) {
			//3
			res = PellEquationSolver_1(&D_mpz, &factor, &x0, &y0);
			if (res = EC_FAILURE) {
				printf("\n D>64: Pell_1 failed\n");
				clear_mpz();
				return;
			}/* else {
				printf("\nPell_1 succeded\n");
				printf("x0 returned as number: ");
				mpz_out_str(stdout, 10, x0); printf("\n");
				printf("y0 returned as number: ");
				mpz_out_str(stdout, 10, y0); printf("\n");
			}*/
		} else {
			//5
			res = PellEquationSolver_2(&D_mpz, &factor, &x0, &y0);
			if (res = EC_FAILURE) {
				printf("\n D<=64: Pell_2 failed\n");
				clear_mpz();
				return;
			}/* else {
				printf("\nPell_2 succeded\n");
				printf("x0 returned as number: ");
				mpz_out_str(stdout, 10, x0); printf("\n");
				printf("y0 returned as number: ");
				mpz_out_str(stdout, 10, y0); printf("\n");
			}*/
		}

		//7
		mpz_set_ui(factor,(unsigned)1);
		res = PellEquationSolver_1(&D_mpz, &factor, &u, &v);
```

```
if (res = EC_FAILURE) {
	printf("\n u,v: Pell_1 failed\n");
	clear_mpz();
	return;
}/* else {
	printf("\nPell_1 succeded\n");
	printf("u returned as number: ");
	mpz_out_str(stdout, 10, u); printf("\n");
	printf("v returned as number: ");
	mpz_out_str(stdout, 10, v); printf("\n");
}*/

//8
mpz_mul(factor,x0,u);
mpz_mul(x1,y0,v);
mpz_mul(x1,x1,D_mpz);
mpz_add(x1,x1,factor);

mpz_mul(factor,x0,v);
mpz_mul(y1,y0,u);
mpz_add(y1,y1,factor);

//9
mpz_set(x,x0);
mpz_set(y,y0);
mpz_set(xx,x1);
mpz_set(yy,y1);

//10
mpz_mod_ui(factor,x0,(unsigned)6);
if(mpz_cmp_ui(factor,(unsigned)1)==0){
	mpz_abs(factor,x);
	while(mpz_cmp(limit,factor) >= 0) {
		//12
		//l1=(x-1)/6
		mpz_sub_ui(l1,x,(unsigned)1);
		mpz_tdiv_q_ui(l1,l1,(unsigned)6);

		//13
		comp_low=(N-3)/8; //-3 to round down
		comp_high=(N-1)/2; //-1 to round up
		//comp_low < log2 l1 is the same as saying
		2^comp_low is less than l1

mpz_ui_pow_ui(exp_low,(unsigned)2,(unsigned)comp_low);

mpz_ui_pow_ui(exp_high,(unsigned)2,(unsigned)comp_high);

		if((mpz_cmp(exp_low,l1) <= 0) &&
		(mpz_cmp(l1,exp_high) < 0)) {
			//14
			mpz_pow_ui(q1,l1,(unsigned)2);
			mpz_add(q1,q1,l1);
			mpz_add_ui(q1,q1,(unsigned)1);

			mpz_pow_ui(n1,l1,(unsigned)2);
			mpz_add_ui(n1,n1,(unsigned)1);

			if( MillerRabinPrimalityTesting (&q1,20)
			 && MillerRabinPrimalityTesting (&n1,20)) {
				printf("\nSuccess!\n");

				printf("k : 4\n");
				printf("D : %d\n",D);
				printf("n : ");
				mpz_out_str(stdout, 10, n1);
				printf("\n");
```

```
                                printf("q : ");
                                mpz_out_str(stdout, 10, q1);
                                printf("\n");
                                mpz_powm_ui(factor,q1,(unsigned)4,n1);
                                printf("q^4 mod n=");
                                mpz_out_str(stdout, 10, factor);
                                printf("\n");

                                clear_mpz();
                                return;
                        }
                }
                //19
                mpz_set(pre_x,x);
                //20-21
                mpz_pow_ui(factor,u,(unsigned)2);
                mpz_mul_ui(factor,factor,(unsigned)2);
                mpz_sub_ui(factor,factor,(unsigned)1);

                mpz_mul(x,x,factor);
                mpz_mul(y,y,factor);

                mpz_mul(factor,u,v);
                mpz_mul_ui(factor,factor,(unsigned)D);
                mpz_mul(factor,factor,y);
                mpz_mul_ui(factor,factor,(unsigned)2);
                mpz_add(x,x,factor);

                mpz_mul(factor,u,v);
                mpz_mul(factor,factor,pre_x);
                mpz_mul_ui(factor,factor,(unsigned)2);
                mpz_add(y,y,factor);
        }
}

//24
mpz_mod_ui(factor,x1,(unsigned)6);
if(mpz_cmp_ui(factor,(unsigned)1)==0){
        mpz_abs(factor,xx);
        while(mpz_cmp(limit,factor) >= 0) {
                //26
                //l1=(x'-1)/6
                mpz_sub_ui(l1,xx,(unsigned)1);
                mpz_tdiv_q_ui(l1,l1,(unsigned)6);

                //13
                comp_low=(N-3)/8; //-3 to round down
                comp_high=(N-1)/2; //-1 to round up
                //comp_low < log2 l1 is the same as saying 2
                ^comp_low is less than l1

        mpz_ui_pow_ui(exp_low,(unsigned)2,(unsigned)comp_low);

        mpz_ui_pow_ui(exp_high,(unsigned)2,(unsigned)comp_high);

                if((mpz_cmp(exp_low,l1) <= 0) &&
                (mpz_cmp(l1,exp_high) < 0)) {
                        //28
                        mpz_pow_ui(q1,l1,(unsigned)2);
                        mpz_add(q1,q1,l1);
                        mpz_add_ui(q1,q1,(unsigned)1);

                        mpz_pow_ui(n1,l1,(unsigned)2);
                        mpz_add_ui(n1,n1,(unsigned)1);

                        if( MillerRabinPrimalityTesting (&q1,20)
                         && MillerRabinPrimalityTesting (&n1,20)) {
```

```
                                        printf("\nSuccess!\n");

                                        printf("k : 4\n");
                                        printf("D : %d\n",D);
                                        printf("n : ");
                                        mpz_out_str(stdout, 10, n1);
                                        printf("\n");
                                        printf("q : ");
                                        mpz_out_str(stdout, 10, q1);
                                        printf("\n");
                                        mpz_powm_ui(factor,q1,(unsigned)4,n1);
                                        printf("q^4 mod n=");
                                        mpz_out_str(stdout, 10, factor);
                                        printf("\n");

                                        clear_mpz();
                                        return;
                                }

                        }
                        //33
                        mpz_set(pre_xx,xx);
                        //34-35
                        mpz_pow_ui(factor,u,(unsigned)2);
                        mpz_mul_ui(factor,factor,(unsigned)2);
                        mpz_sub_ui(factor,factor,(unsigned)1);

                        mpz_mul(xx,xx,factor);
                        mpz_mul(yy,yy,factor);

                        mpz_mul(factor,u,v);
                        mpz_mul_ui(factor,factor,(unsigned)D);
                        mpz_mul(factor,factor,yy);
                        mpz_mul_ui(factor,factor,(unsigned)2);
                        mpz_add(xx,xx,factor);

                        mpz_mul(factor,u,v);
                        mpz_mul(factor,factor,pre_xx);
                        mpz_mul_ui(factor,factor,(unsigned)2);
                        mpz_add(yy,yy,factor);
                }
        }
    }

    printf("\nNo suitable EC with k=4 found, try increasing the value of z...\n");
    clear_mpz();
}
```

Example of the execution of the specific algorithm is illustrated below:

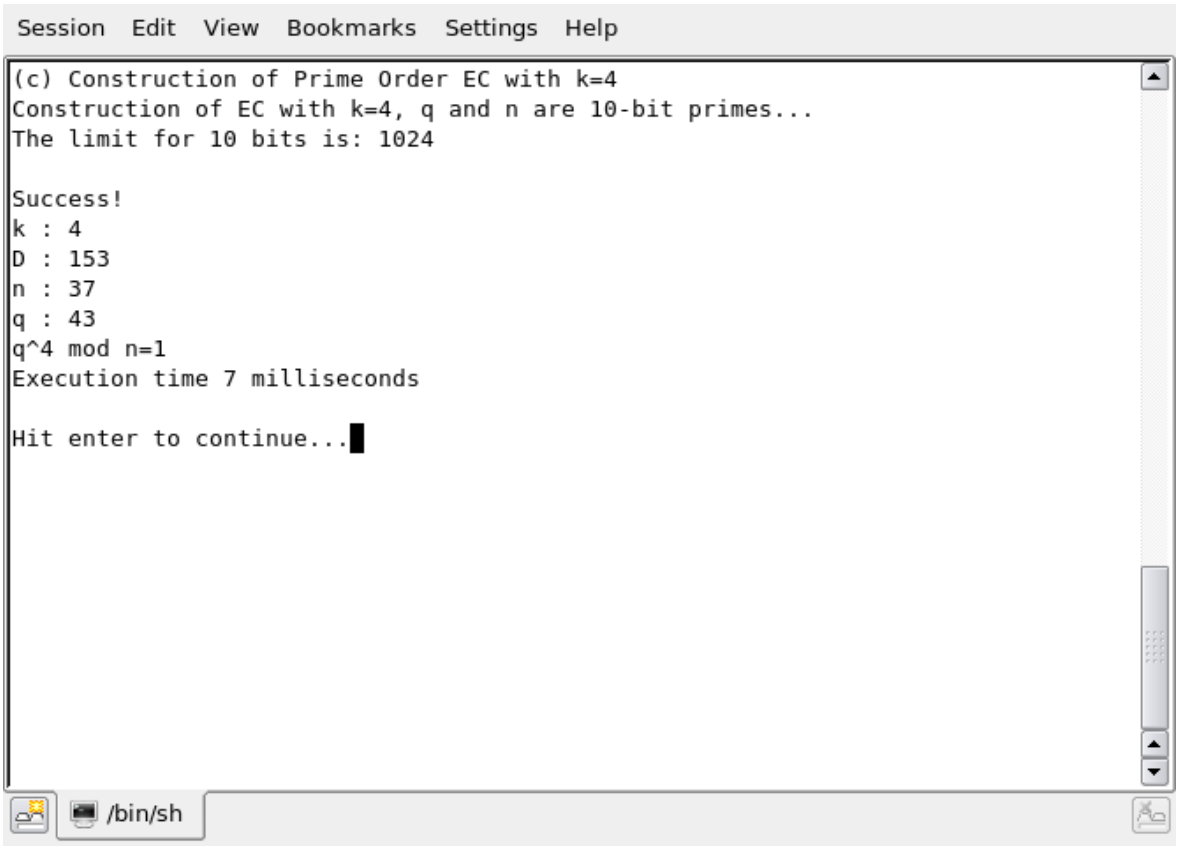


Figure 6 Construction of Elliptic Curve with Embedding Degree k=4

4.6 Constructing Elliptic Curves with Embedding Degree k=6

An algorithm for construction of Elliptic Curves with Embedding Degree k=6 has been implemented in the past in the PBC library[10]. This existing implementation has been utilized by the current thesis' implementation part for the taking of measurements that are used in the following chapter for performance analysis and comparison. The corresponding procedure from the PBC Library is called find_mnt_6_curve, and the corresponding source code that prepares its input and calls the specific procedure is shown below:

```
void Construct_k6_curve_Lynn () {
	darray_t L;
	d_param_t param;
	cm_info_ptr cm;
	int D = 9563;

	printf("Using procedure find_mnt6_curve by Ben Lynn\n");
	printf("for measurement and comparison purposes.\n");

	darray_init(L);

	printf("Using D = %d\n", D);

	find_mnt6_curve(L, D, 500);

	if (!L->count) {
		fprintf(stderr, "No suitable curves for this D\n");
		return;
	}

	cm = darray_at(L, 0);
	d_param_init(param);

	fprintf(stderr, "gendparam: computing Hilbert polynomial
				and finding roots...\n");
	d_param_from_cm(param, cm);
	fprintf(stderr, "gendparam: bits in q = %zu\n",
				mpz_sizeinbase(cm->q, 2));
	d_param_out_str(stdout, param);

	while (darray_count(L)) {
		cm = darray_at(L, 0);
		darray_remove_index(L, 0);
		cm_info_clear(cm);
	}
	darray_clear(L);
}
```

The following figure illustrates a typical execution of this procedure, in this case for D=9563:

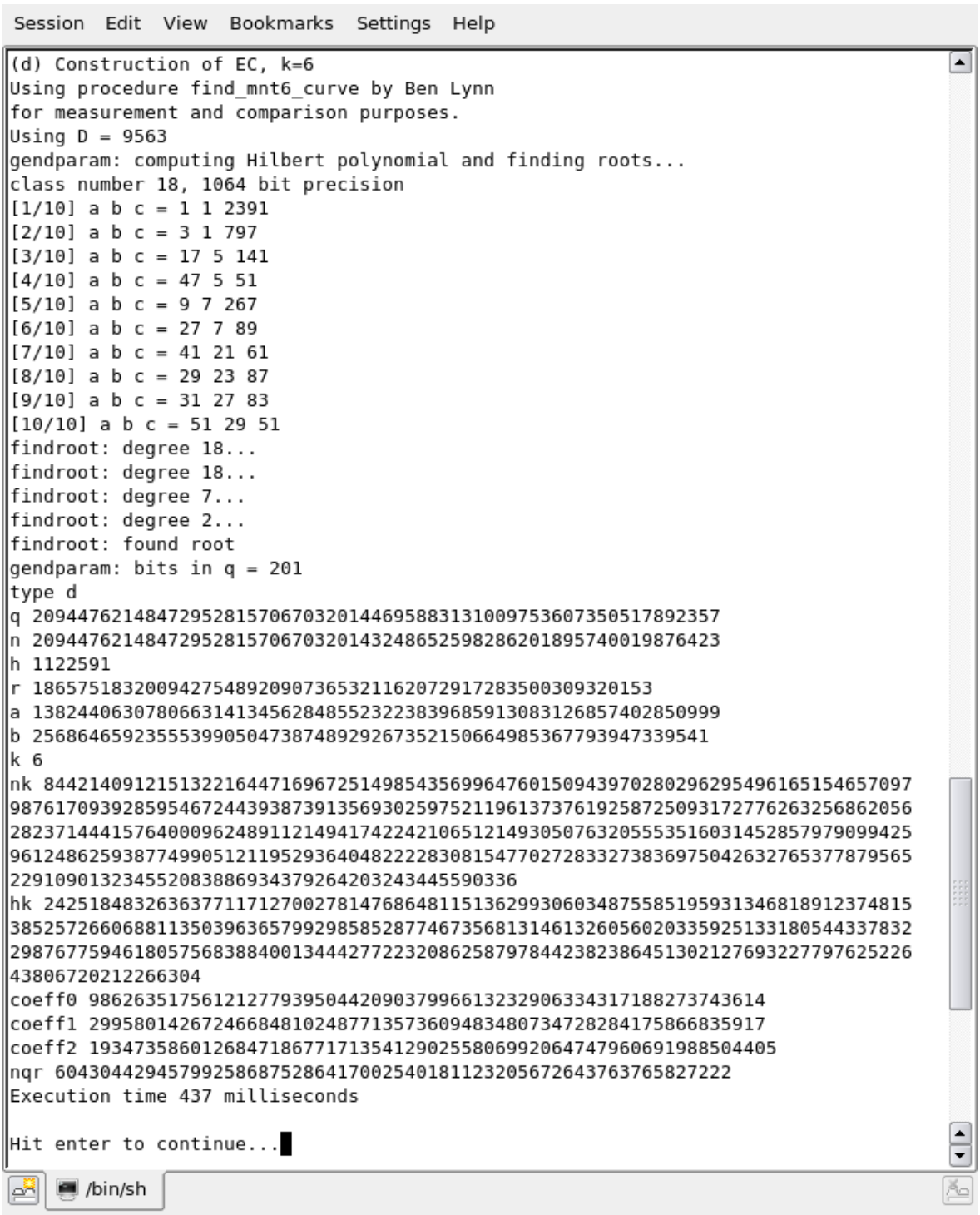


Figure 7 Construction of Elliptic Curve with embedding degree k=6 (PBC)

Further to Ben Lynn's algorithm, algorithm 3 of Chapter 3 has been implemented for k=6. The corresponding source code for this algorithm is:

```
void Construct_k6_curve (long N) {

	mpz_t D_mpz,i_mpz,factor,square,x0,y0,x1,y1,x,y,xx,yy,
	      pre_x,pre_xx,u,v,exp_high,exp_low;
	mpz_t l1,q1,n1,limit;
	long D,i,comp_low,comp_high;
	int D_mod_24_is_9,D_third_square_free, minus_two_is_square_mod_D;
	int res;

	void clear_mpz(){
	      mpz_clear(D_mpz);
	      mpz_clear(i_mpz);
	      mpz_clear(factor);
	      mpz_clear(square);
	      mpz_clear(x0);
	      mpz_clear(y0);
	      mpz_clear(x1);
	      mpz_clear(y1);
	      mpz_clear(x);
	      mpz_clear(y);
	      mpz_clear(xx);
	      mpz_clear(yy);
	      mpz_clear(pre_x);
	      mpz_clear(pre_xx);
	      mpz_clear(u);
	      mpz_clear(v);
	      mpz_clear(l1);
	      mpz_clear(q1);
	      mpz_clear(n1);
	      mpz_clear(exp_high);
	      mpz_clear(exp_low);
	      mpz_clear(limit);
	}

	//start of procedure
	mpz_init(D_mpz);
	mpz_init(i_mpz);
	mpz_init(factor);
	mpz_init(square);
	mpz_init(x0);
	mpz_init(y0);
	mpz_init(x1);
	mpz_init(y1);
	mpz_init(x);
	mpz_init(y);
	mpz_init(xx);
	mpz_init(yy);
	mpz_init(pre_x);
	mpz_init(pre_xx);
	mpz_init(u);
	mpz_init(v);
	mpz_init(l1);
	mpz_init(q1);
	mpz_init(n1);
	mpz_init(exp_high);
	mpz_init(exp_low);
	mpz_init(limit);

	printf("Construction of EC with k=6, q and n are %d-bit primes...\n",(N+1)/2);
	mpz_ui_pow_ui(limit,(unsigned)2,(unsigned)((N+1)/2)); //limit = 2^(N/2)
	printf("The limit for %d bytes is: ",(N+1)/2); mpz_out_str(stdout, 10, limit);
	printf("\n");
```

```
for(D=1;D<z;D++) {
	/*if (D % 1000==0) {
		printf(".");
	}*/

	mpz_set_ui(D_mpz,(unsigned)D);
	mpz_set_ui(factor,(unsigned)(D/3));

	D_mod_24_is_9 = (D % 24 == 9);
	//D is not good if it is divisible by 3
	AND the quotient is a perfect square
	D_third_square_free = (D%3!=0) ||
	((D%3==0)&&(mpz_perfect_square_p(factor)==0));

	//6 must be a square mod D (there exists a number x for which x^2=6 modD)
	minus_two_is_square_mod_D = 0;
	for(i=1;i<D;i++) {
		mpz_set_ui(i_mpz,(unsigned)i);
		mpz_powm_ui(square,i_mpz,(unsigned)2,D_mpz); //square=i^2 mod D
		if (mpz_cmp_ui(square,(unsigned)(D-2))==0) {
			//printf("-2 is %d^2 mod %d\n",i,D);
			minus_two_is_square_mod_D = 1;
		}
	}

	mpz_set_ui(factor,(unsigned)D);
	if (D_mod_24_is_9 && D_third_square_free
	 && minus_two_is_square_mod_D && (mpz_perfect_square_p(factor)==0)){
		//printf("Found a D:%d\n",D);

		//2
		mpz_set_si(factor,(signed)(-8));
		if (D > 64) {
			//3
			res = PellEquationSolver_1(&D_mpz, &factor, &x0, &y0);
			if (res = EC_FAILURE) {
				printf("\n D>64: Pell_1 failed\n");
				clear_mpz();
				return;
			}/* else {
				printf("\nPell_1 succeded\n");
				printf("x0 returned as number: ");
				mpz_out_str(stdout, 10, x0); printf("\n");
				printf("y0 returned as number: ");
				mpz_out_str(stdout, 10, y0); printf("\n");
			}*/
		} else {
			//5
			res = PellEquationSolver_2(&D_mpz, &factor, &x0, &y0);
			if (res = EC_FAILURE) {
				printf("\n D<=64: Pell_2 failed\n");
				clear_mpz();
				return;
			}/* else {
				printf("\nPell_2 succeded\n");
				printf("x0 returned as number: ");
				mpz_out_str(stdout, 10, x0); printf("\n");
				printf("y0 returned as number: ");
				mpz_out_str(stdout, 10, y0); printf("\n");
			}*/
		}

		//7
		mpz_set_ui(factor,(unsigned)1);
		res = PellEquationSolver_1(&D_mpz, &factor, &u, &v);
		if (res = EC_FAILURE) {
			printf("\n u,v: Pell_1 failed\n");
```

```
        clear_mpz();
        return;
}/* else {
        printf("\nPell_1 succeded\n");
        printf("u returned as number: ");
        mpz_out_str(stdout, 10, u); printf("\n");
        printf("v returned as number: ");
        mpz_out_str(stdout, 10, v); printf("\n");
}*/

//8
mpz_mul(factor,x0,u);
mpz_mul(x1,y0,v);
mpz_mul(x1,x1,D_mpz);
mpz_add(x1,x1,factor);

mpz_mul(factor,x0,v);
mpz_mul(y1,y0,u);
mpz_add(y1,y1,factor);

//9
mpz_set(x,x0);
mpz_set(y,y0);
mpz_set(xx,x1);
mpz_set(yy,y1);

//10
mpz_mod_ui(factor,x0,(unsigned)6);
if(mpz_cmp_ui(factor,(unsigned)1)==0){
        mpz_abs(factor,x);
        while(mpz_cmp(limit,factor) >= 0) {
                //12
                //l1=(x-1)/6
                mpz_sub_ui(l1,x,(unsigned)1);
                mpz_tdiv_q_ui(l1,l1,(unsigned)6);

                //13
                comp_low=(N-3)/8; //-3 to round down
                comp_high=(N-1)/2; //-1 to round up
                //comp_low < log2 l1 is the same as saying
                2^comp_low is less than l1

        mpz_ui_pow_ui(exp_low,(unsigned)2,(unsigned)comp_low);

        mpz_ui_pow_ui(exp_high,(unsigned)2,(unsigned)comp_high);

                if((mpz_cmp(exp_low,l1) <= 0) &&
                (mpz_cmp(l1,exp_high) < 0)) {

                        //14
                        mpz_pow_ui(q1,l1,(unsigned)2);
                        mpz_mul_ui(q1,q1,(unsigned)4);
                        mpz_add_ui(q1,q1,(unsigned)1);

                        mpz_mul_ui(factor,l1,(unsigned)2);
                        mpz_sub(n1,q1,factor);

                        if( MillerRabinPrimalityTesting (&q1,20)
                         && MillerRabinPrimalityTesting (&n1,20)) {
                                printf("\nSuccess!\n");

                                printf("k : 6\n");
                                printf("D : %d\n",D);
                                printf("n : ");
                                mpz_out_str(stdout, 10, n1);
                                printf("\n");
                                printf("q : ");
```

```
                                mpz_out_str(stdout, 10, q1);
                                printf("\n");
                                mpz_powm_ui(factor,q1,(unsigned)6,n1);
                                printf("q^6 mod n=");
                                mpz_out_str(stdout, 10, factor);
                                printf("\n");

                                clear_mpz();
                                return;
                        }
                }
                //19
                mpz_set(pre_x,x);
                //20-21
                mpz_pow_ui(factor,u,(unsigned)2);
                mpz_mul_ui(factor,factor,(unsigned)2);
                mpz_sub_ui(factor,factor,(unsigned)1);

                mpz_mul(x,x,factor);
                mpz_mul(y,y,factor);

                mpz_mul(factor,u,v);
                mpz_mul_ui(factor,factor,(unsigned)D);
                mpz_mul(factor,factor,y);
                mpz_mul_ui(factor,factor,(unsigned)2);
                mpz_add(x,x,factor);

                mpz_mul(factor,u,v);
                mpz_mul(factor,factor,pre_x);
                mpz_mul_ui(factor,factor,(unsigned)2);
                mpz_add(y,y,factor);
        }
}

//24
mpz_mod_ui(factor,x1,(unsigned)6);
if(mpz_cmp_ui(factor,(unsigned)1)==0){
        mpz_abs(factor,xx);
        while(mpz_cmp(limit,factor) >= 0) {
                //26
                //l1=(x'-1)/6
                mpz_sub_ui(l1,xx,(unsigned)1);
                mpz_tdiv_q_ui(l1,l1,(unsigned)6);

                //27
                comp_low=(N-3)/8; //-3 to round down
                comp_high=(N-1)/2; //-1 to round up
                //comp_low < log2 l1 is the same as saying
                2^comp_low is less than l1

        mpz_ui_pow_ui(exp_low,(unsigned)2,(unsigned)comp_low);

        mpz_ui_pow_ui(exp_high,(unsigned)2,(unsigned)comp_high);

                if((mpz_cmp(exp_low,l1) <= 0) &&
                (mpz_cmp(l1,exp_high) < 0)) {
                        //28
                        mpz_pow_ui(q1,l1,(unsigned)2);
                        mpz_mul_ui(q1,q1,(unsigned)4);
                        mpz_add_ui(q1,q1,(unsigned)1);

                        mpz_mul_ui(factor,l1,(unsigned)2);
                        mpz_sub(n1,q1,factor);

                        if( MillerRabinPrimalityTesting (&q1,20)
                         && MillerRabinPrimalityTesting (&n1,20)) {
                                printf("\nSuccess!\n");
```

```
                                        printf("k : 6\n");
                                        printf("D : %d\n",D);
                                        printf("n : ");
                                        mpz_out_str(stdout, 10, n1);
                                        printf("\n");
                                        printf("q : ");
                                        mpz_out_str(stdout, 10, q1);
                                        printf("\n");
                                        mpz_powm_ui(factor,q1,(unsigned)6,n1);
                                        printf("q^6 mod n=");
                                        mpz_out_str(stdout, 10, factor);
                                        printf("\n");

                                        clear_mpz();
                                        return;
                                }

                        }
                        //33
                        mpz_set(pre_xx,xx);
                        //34-35
                        mpz_pow_ui(factor,u,(unsigned)2);
                        mpz_mul_ui(factor,factor,(unsigned)2);
                        mpz_sub_ui(factor,factor,(unsigned)1);

                        mpz_mul(xx,xx,factor);
                        mpz_mul(yy,yy,factor);

                        mpz_mul(factor,u,v);
                        mpz_mul_ui(factor,factor,(unsigned)D);
                        mpz_mul(factor,factor,yy);
                        mpz_mul_ui(factor,factor,(unsigned)2);
                        mpz_add(xx,xx,factor);

                        mpz_mul(factor,u,v);
                        mpz_mul(factor,factor,pre_xx);
                        mpz_mul_ui(factor,factor,(unsigned)2);
                        mpz_add(yy,yy,factor);
                }
        }
    }
}

printf("\nNo suitable EC with k=6 found, try increasing the value of z...\n");
clear_mpz();
}
```

Example of the execution of the specific algorithm is illustrated below:

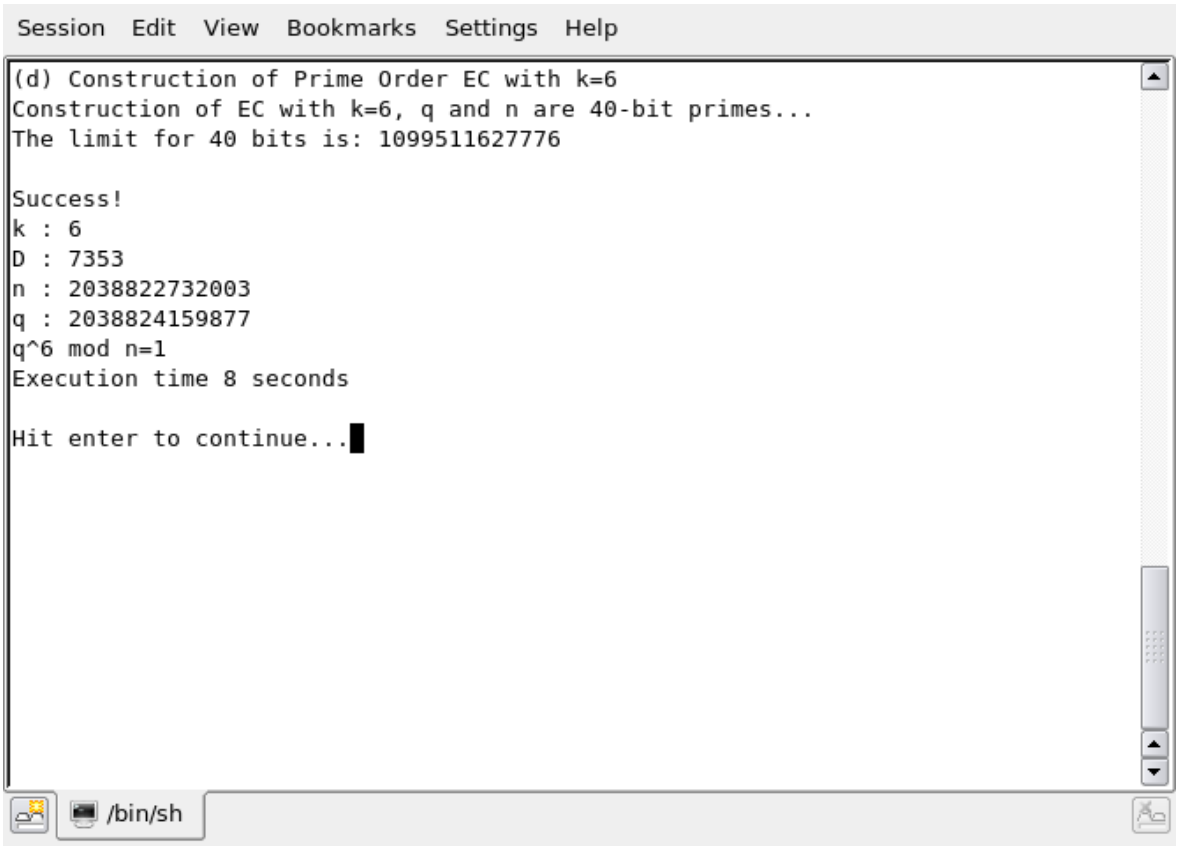


Figure 8 Construction of Elliptic Curve with Embedding Degree k=6

4.7 Experimental Results

The algorithms implemented were executed in order to evaluate their operation and performance. The execution platform for these algorithms was a PC based on the AMD Athlon-64 2800+ processor, having 512 MB RAM and running Ubuntu Linux 7.10, as seen by the System Information window displayed below.

Figure 9 Test-system configuration

For the execution of the algorithms, a test procedure was created. This procedure executes the algorithms described in the previous sections for generating prime order elliptic curves having embedding degrees of 3,4, and 6, for input values ranging from 10 to 120 (which results in q,n being approximately from 5 to 60 bits in length). The source code for this test procedure is given below:

```
static void ExperimentalProcedure(){
	struct timeb {
		time_t time;
		unsigned short millitm;
		short timezone;
		short dstflag;
	} t1,t2;

	long N;
	int start = 10;
	int limit = 120;
	int step = 10;

	for(N=10;N<=limit;N+=step) {
		(void) ftime(&t1);

		Construct_k3_curve(N);

		(void) ftime(&t2);

		if ( ((int)t2.time-t1.time) == 0 ) {
			printf("Execution time %d milliseconds\n",
				(int)t2.millitm-t1.millitm);
		} else {
			printf("Execution time %d seconds, %d milliseconds\n",
				(int)t2.time-t1.time, (int)t2.millitm-t1.millitm);
		}
	}

	for(N=10;N<=limit;N+=step) {
		(void) ftime(&t1);

		Construct_k4_curve(N);

		(void) ftime(&t2);

		if ( ((int)t2.time-t1.time) == 0 ) {
			printf("Execution time %d milliseconds\n",
				(int)t2.millitm-t1.millitm);
		} else {
			printf("Execution time %d seconds, %d milliseconds\n",
				(int)t2.time-t1.time, (int)t2.millitm-t1.millitm);
		}
	}

	for(N=10;N<=limit;N+=step) {
		(void) ftime(&t1);

		Construct_k6_curve(N);

		(void) ftime(&t2);

		if ( ((int)t2.time-t1.time) == 0 ) {
			printf("Execution time %d milliseconds\n",
				(int)t2.millitm-t1.millitm);
		} else {
			printf("Execution time %d seconds, %d milliseconds\n",
				(int)t2.time-t1.time, (int)t2.millitm-t1.millitm);
		}
	}
}
```

The total execution time for this procedure was 8443 seconds (or 2:20:43), during which processor occupancy was constantly held at 99-100%, as seen from the System Monitor screenshot below:

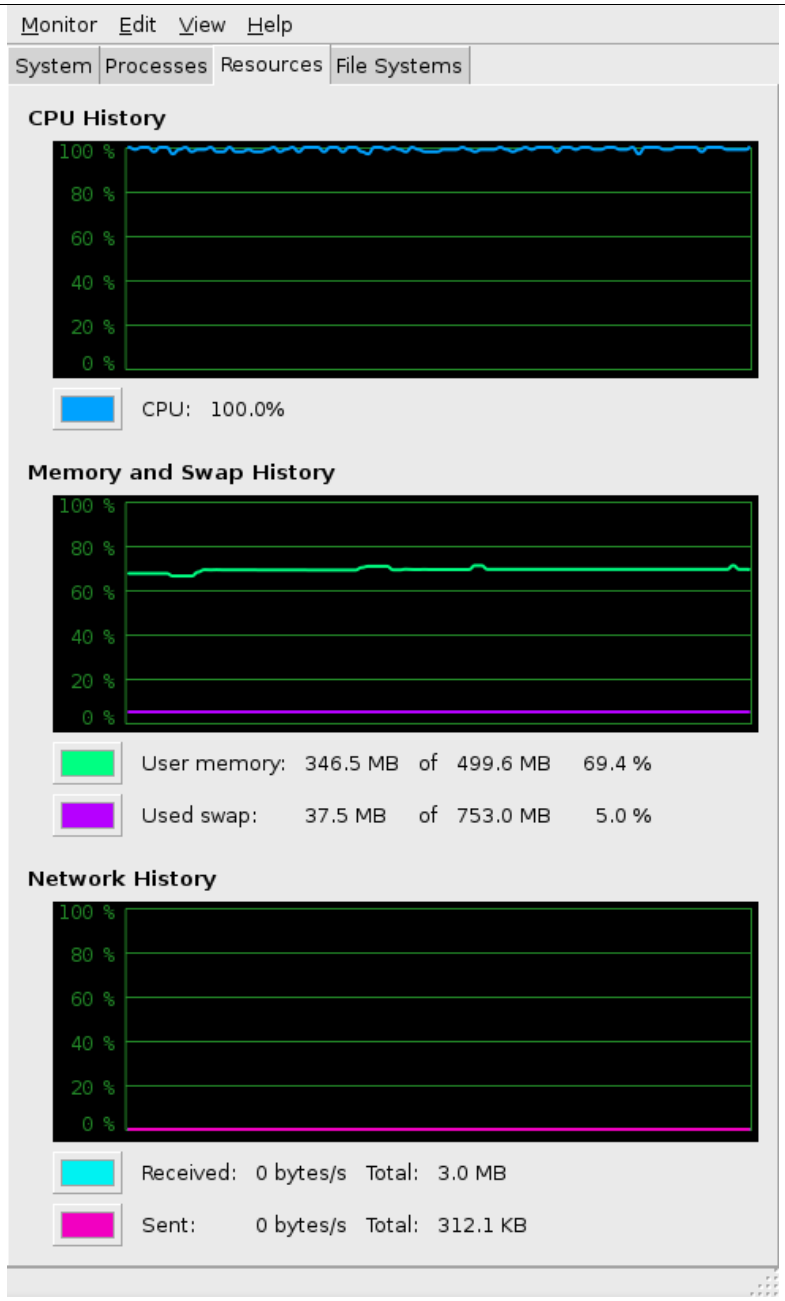


Figure 10 CPU and Memory usage during experimental procedure

During its execution, the execution time for each elliptic curve construction was measured, as well as the value of D reached before the procedure could successfully

generate a curve of the desired embedding degree and bit lengths. The value z, that defines the maximum value of D, was set to 10^6 or 100.000. The suggested value from [8] was 10^10, however this yields excessively large execution times and was not preferred.

4.7.1 Measurements

For each case k=3, k=4 and k=6, the execution time, value of D reached and the outcome of the algorithms (values q,n) are presented in the tables below. For all cases the criterion

$$q^k \bmod n = 1$$

was fulfilled by the produced results.

The results for k=3 are shown below:

N	Approx. Bits	Execution Time(sec)	D	n	q	q^3 mod n	(t/D)*1000
10	5	0,001	57	7	11	1	0,01754386
20	10	0,003	129	919	971	1	0,02325581
30	15	0,003	129	919	971	1	0,02325581
40	20	0,045	489	1527247	1529387	1	0,09202454
50	25	0,045	489	1527247	1529387	1	0,09202454
60	30	0,045	489	1527247	1529387	1	0,09202454
70	35	0,042	489	1527247	1529387	1	0,08588957
80	40	19,635	11217	78566419	78581771	1	1,75046804
90	45	19,896	11217	78566419	78581771	1	1,77373629
100	50	21,996	11217	78566419	78581771	1	1,96095213
110	55	56,753	19281	972702127	972648107	1	2,94346766
120	60	194,643	35913	142810066595527	142810045896971	1	5,41984797

Table 1 Summary of results for k=3

The results for k=4 are shown below:

N	Approx. Bits	Execution Time(sec)	D	N	q	q^4 mod n	(t/D)*1000
10	5	0,002	57	2	3	1	0,03508772
20	10	0,131	873	2917	2971	1	0,15005727
30	15	0,127	873	2917	2971	1	0,14547537
40	20	0,121	873	2917	2971	1	0,13860252
50	25	0,121	873	2917	2971	1	0,13860252
60	30	1106,744	85497	20153446237311557	20153446379274391	1	12,9448285
70	35	376,873	50241	251985827927610001	251985828429591901	1	7,50130372
80	40	1093,548	85569	20153446237311557	20153446379274391	1	12,7797216
90	45	1090,946	85497	20153446237311557	20153446379274391	1	12,7600501
100	50	1092,796	85497	20153446237311557	20153446379274391	1	12,7816882
110	55	376,827	50241	251985827927610001	251985828429591901	1	7,50038813
120	60	1091,798	85497	20153446237311557	20153446379274391	1	12,7700153

Table 2 Summary of results for k=4

The results for k=6 are shown below:

N	Approx. Bits	Execution Time(sec)	D	N	q	q^6 mod n	(t/D)*1000
10	5	1523,234	>100000	N/A	N/A	N/A	N/A
20	10	298,813	44241	43891	44101	1	6,7542099
30	15	6,398	6441	1321351	1322501	1	0,99332402
40	20	6,358	6441	1321351	1322501	1	0,9871138
50	25	8,283	7353	2038822732003	2038824159877	1	1,12647899
60	30	8,426	7353	2038822732003	2038824159877	1	1,14592683
70	35	8,357	7353	2038822732003	2038824159877	1	1,13654291
80	40	6,424	6441	1321351	1322501	1	0,99736066
90	45	8,401	7353	2038822732003	2038824159877	1	1,14252686
100	50	8,368	7353	2038822732003	2038824159877	1	1,1380389
110	55	8,285	7353	2038822732003	2038824159877	1	1,12675099
120	60	8,283	7353	2038822732003	2038824159877	1	1,12647899

Table 3 Summary of results for k=6

It is notable that, although the algorithm for constructing prime order elliptic curves having embedding degree k=4 and k=6 are almost identical (the only difference being the equations that determine the values of q and n in relation to l in every step), the results produced differ very much in terms of execution time and reached value of D in each case. It must be noted at this point that Ben Lynn's function for generation of elliptic curve having k=6 had execution time of 437 milliseconds (see figure 5, section 4.3) but accomplished this by starting from a predefined value of D that successfully generated such a curve. Additionally, Lynn's implementation does not impose a strict rule on having a prime order curve, meaning that the order could be also a composite made of a large prime and a much smaller coefficient, which of course results in a much slower construction of the elliptic curve in our implementation due to the fact that too many curves are rejected because q and n must both be prime.

4.7.2 Curve-generating value of D

The following table summarises the values of D that generated the curve in each round of the experimental procedure, for embedding degrees k=3,4 and 6 and input values N ranging from 10 to 120.

N	D (k=3)	D (k=4)	D (k=6)
10	57	57	N/A
20	129	873	44241
30	129	873	6441
40	489	873	6441
50	489	873	7353
60	489	85497	7353
70	489	50241	7353
80	11217	85569	6441
90	11217	85497	7353
100	11217	85497	7353
110	19281	50241	7353
120	35913	85497	7353

Table 4 Value of D reached by the algorithms

4.7.3 D and Time

By comparison of tables 1,2,3 it is obvious that as the value of D increases, the execution time of the algorithm will also increase.

		k=3		k=4		k=6	
N	Approx. Bits	Execution Time	D	Execution Time	D	Execution Time	D
10	5	0,001	57	0,002	57	1523,234	>100000
20	10	0,003	129	0,131	873	298,813	44241
30	15	0,003	129	0,127	873	6,398	6441
40	20	0,045	489	0,121	873	6,358	6441
50	25	0,045	489	0,121	873	8,283	7353
60	30	0,045	489	1106,744	85497	8,426	7353
70	35	0,042	489	376,873	50241	8,357	7353
80	40	19,635	11217	1093,548	85569	6,424	6441
90	45	19,896	11217	1090,946	85497	8,401	7353
100	50	21,996	11217	1092,796	85497	8,368	7353
110	55	56,753	19281	376,827	50241	8,285	7353
120	60	194,643	35913	1091,798	85497	8,283	7353

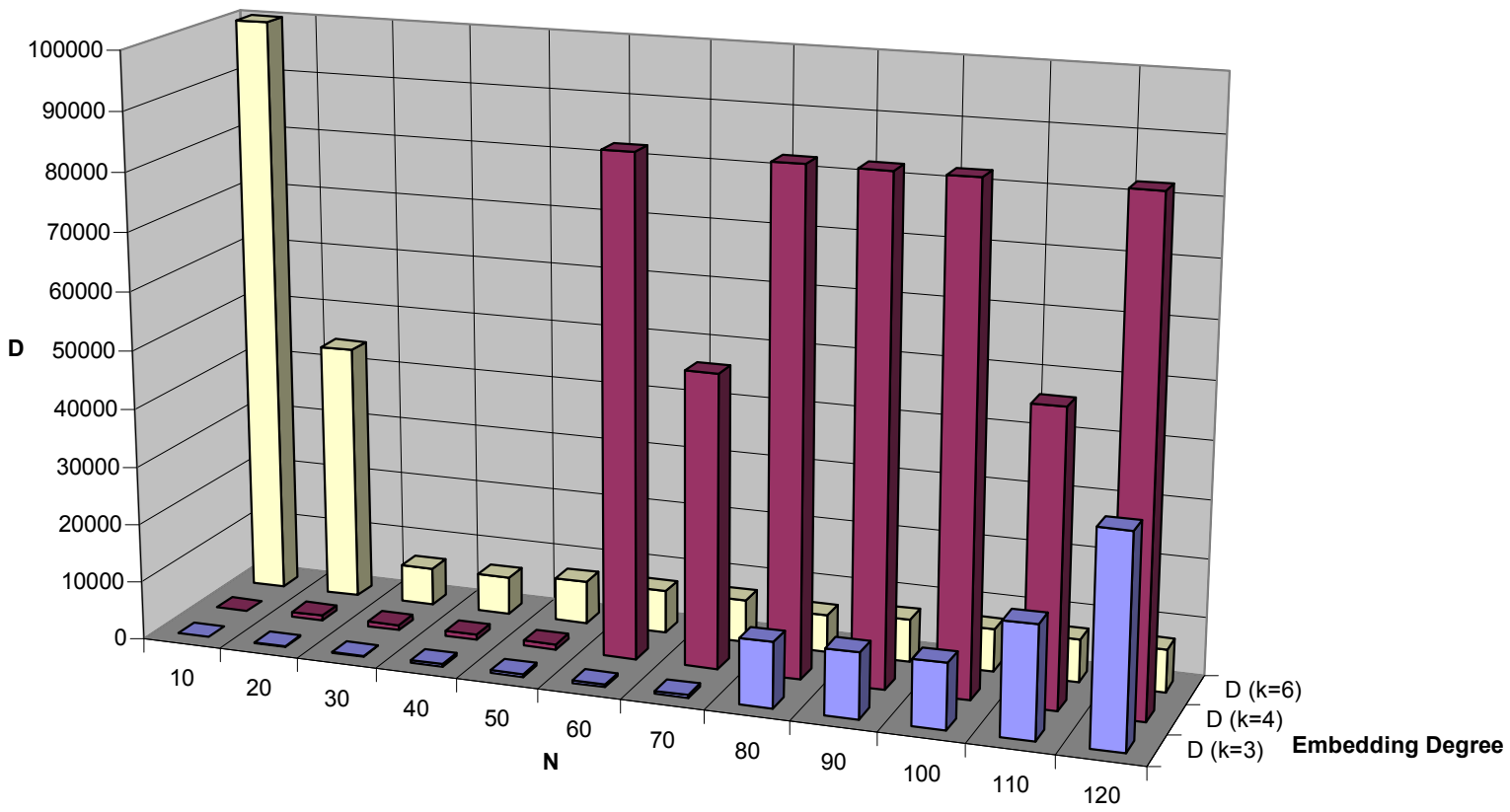
Table 5 Relation between D and Execution Time

It must be noted that the relationship between execution time and value of D reached is not linear; higher values of D are more time-consuming, leading to overall higher ration of t/D. The following table clearly illustrates this.

N	t/D k=3	t/D k=4	t/D k=6
10	0,01754386	0,03508772	N/A
20	0,02325581	0,15005727	6,754209896
30	0,02325581	0,14547537	0,993324018
40	0,09202454	0,13860252	0,987113802
50	0,09202454	0,13860252	1,126478988
60	0,09202454	12,9448285	1,145926833
70	0,08588957	7,50130372	1,136542908
80	1,75046804	12,7797216	0,997360658
90	1,77373629	12,7600501	1,142526860
100	1,96095213	12,7816882	1,138038896
110	2,94346766	7,50038813	1,126750986
120	5,41984797	12,7700153	1,126478988

Table 6 t over D for k=3,4 and 6

The corresponding t/D columns have all been multiplied by a factor of 1000, as most of the time t is less than D and the results were <1. The following figure is a graphical representation of Table 5, followed by the graph of figure 6.



	10	20	30	40	50	60	70	80	90	100	110	120
D (k=3)	57	129	129	489	489	489	489	11217	11217	11217	19281	35913
D (k=4)	57	873	873	873	873	85497	50241	85569	85497	85497	50241	85497
D (k=6)	100000	44241	6441	6441	7353	7353	7353	6441	7353	7353	7353	7353

Figure 11 Maximum value of D for each Embedding Degree

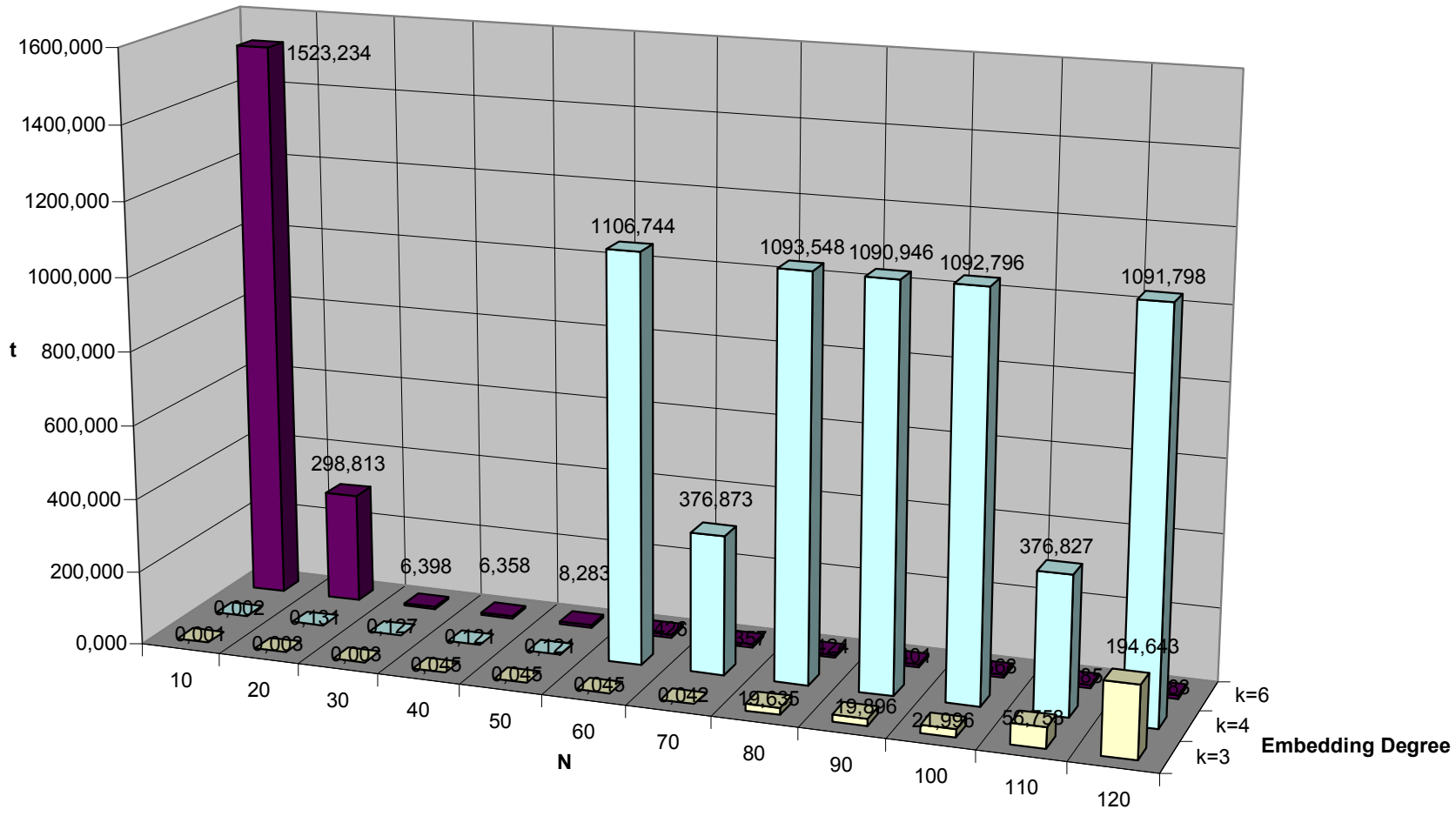


Figure 12 Execution time for each curve construction

5 CONCLUSIONS

This thesis dealt with the presentation, implementation and experimental evaluation of some well known methods for the generation of elliptic curves having arbitrary small embedding degree, and in particular embedding degree k=3, 4 or 6.

As a result, a primitive software library written in the C Programming language was produced. This software library contains procedures that solve Pell's Equation as well as procedures that produce prime-order elliptic curves of embedding degree k=3,4 or 6 with prescribed bit lengths for the parameters. These elliptic curves can then be used on the construction of Weil or Tate pairings in order to implement pairing-based cryptographic protocols.

The experimental evaluation of the library proved that it produces accurate results. Even without any performance optimizations in the implementation, most curves were constructed within a few seconds. For some cases where a high value of D was required, the construction requires a few minutes.

Concerning further work that could be undertaken in this area, it is possible to implement several optimizations in order to improve the performance of the algorithms of this library for larger values of D. For example, it is possible to implement all possible equations used for the calculation of l,q,n rather than only one of the equations available as was done here. This will inevitably result in more complicated source code, but will also increase the possibility of producing a prime order elliptic curve in each potential value of D calculated.

The library could be further enhanced by providing procedures for the creation of prime order elliptic curves having other embedding degrees, such as k=8 or k=12. In any case, since it is based on the GNU MP library that is distributed under the GNU licence, the limitations imposed by this licence must be taken into consideration concerning the library's use.

6 REFERENCES

[1] J. Lopez, R. Dahab, "An Overview of Elliptic Curve Cryptography", Technical Report IC-00-10, 2000.

[2] E. Konstantinou, Y. Stamatiou, C. Zaroliagis, "Efficient generation of secure elliptic curves ", Springer-Verlag, 2006.

[3] S. Galbraith, K. Paterson, N. Smart, "Pairings for Cryptographers", 2005.

[4] A. Menezes, "An Introduction to Pairing-Based Cryptography", Mathematics Subject Classification – Primary 94A60, 1991.

[5] B. Lynn, "On the Implementation of Pairing-Based Cryptosystems ", PhD Thesis, Stanford University, 2007.

[6] A. Miyaji, M. Nakabayashi, S. Takano, "New Explicit Conditions of Elliptic Curve Traces for FR-Reduction", *IEICE Transactions on Fundamentals* **E84-A(5)** (2001), pp. 1234--1243.

[7] P. S. L. M. Barreto, B. Lynn, M. Scott, "Constructing Elliptic Curves with Prescribed Embedding Degrees," *Security in Communication Networks -- SCN'2002*, LNCS **2576**, Springer-Verlag (2003), pp. 257--267.

[8] K. Karabina, "On Prime Order Elliptic Curves with Embedding Degrees 3,4 and 6", MSc Thesis, University of Waterloo-Canada, 2006.

[9] E. Konstantinou, Y. Stamatiou, C. Zaroliagis, "ECC-LIB, A Library for Elliptic Curve Cryptography", http://www.ceid.upatras.gr/faculty/zaro/software/ecc-lib/, accessed on September 2007.

[10] B. Lynn et al, "Pairing Based Cryptography Library", http://crypto.stanford.edu/pbc/, accessed on September 2007.

[11] T. Granlund et al, "GNU MP Bignum Library", http://gmplib.org/, accessed on September 2007.

[12] D. Boneh and X. Boyen, “Efficient selective-ID secure identity-based encryption without random oracles”, In Advances in Cryptology - Eurocrypt 2004, Springer-Verlag LNCS 3027, 223–238, 2004.

[13] D. Boneh, X. Boyen and H. Shacham, “Short group signatures”, In Advances in Cryptology – CRYPTO 2004, Springer-Verlag LNCS 3152, 41–55, 2004.

[14] D. Boneh and M. Franklin, “Identity-based encryption from the Weil pairing”, In Advances in Cryptology–CRYPTO 2001, Springer-Verlag LNCS 2139, 213–229, 2001.

[15] D. Boneh, B. Lynn, and H. Shacham, “Short signatures from the Weil pairing”, In Advances in Cryptology – ASIACRYPT 2001, Springer-Verlag LNCS 2248, 514–532, 2001.

[16] D. Boneh and H. Shacham, “Group signatures with verifier-local revocation”, In ACM CCS 2004, 168-177, ACM Press, 2004.

[17] L. Chen and Z. Cheng, “Security proof of Sakai-Kasahara’s identity-based encryption Scheme”, In Proceedings of Cryptography and Coding 2005, Springer-Verlag LNCS 3796, 442–459, 2005.

[18] A. Joux, “A one round protocol for tripartite Diffie–Hellman”, In Algorithmic Number Theory Symposium – ANTS IV, Springer-Verlag LNCS 1838, 385–394, 2000.

[19] R. Sakai, K. Ohgishi and M. Kasahara, “Cryptosystems based on pairing”, In The 2000 Symposium on Cryptography and Information Security, Okinawa, Japan, January 2000.

[20] R. Sakai and M. Kasahara, “ID based cryptosystems with pairing on elliptic curve”, Cryptology ePrint Archive, Report 2003/054. 2003.