N° d'ordre : 20/2016-C/MT

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université des Sciences et de la Technologie Houari Boumediène

Faculté de Mathématiques

جامعة هواري بومدين
للعلوم والتكنولوجيا
U S T H B

THESE

Présentée pour l'obtention du **diplôme** de **DOCTORAT 3 eme Cycle (LMD)**

En : MATHEMATIQUES

Spécialité : Mathématiques et Applications

Par : Oualid BENAMARA

Sujet

Eléments Arithmétiques pour la Cryptographie et Techniques de Cryptanalyse Statistiques

Soutenue publiquement, le 15 /03/2016, devant le jury composé de :

M B. BENZAGHOU	Professeur	USTHB	Président
M K. BETINA	Professeur	USTHB	Directeur de thèse
M A. ADDA	Professeur	Université d'Oran	Examinateur
Mme K. GUENDA	Maitre de conférences/A	USTHB	Examinatrice
Mme A. LAOUDI	Maitre de conférences/A	USTHB	Examinatrice
M L. NOUI	Professeur	Université de Batna	Examinateur

Dédicaces

Je dédie ce travail à :

Mes très chers parents qui sont mes références dans la droiture.

Mes chers frères et sœurs

Toute ma famille.

A la mémoire de mes grands parents paternels et maternels.

Remerciements

Mes sincères remerciements et ma profonde gratitude s'adressent à Monsieur le Professeur Kamel BETINA, mon promoteur pour sa disponibilité et ses conseils durant toute cette formation doctorale. Qu'il trouve ici l'expression de ma gratitude.

Mes sincères remerciements s'adressent aussi à :

- Mr Benali BENZAGHOU, pour l'honneur qu'il nous fait de présider le jury et de juger ce travail. Qu'il trouve ici mon profond respect et toute ma gratitude.
- Mr Adda ALI-PACHA, Professeur à l'université d'Oran d'avoir accepté de prendre de son temps pour faire partie du jury et examiner ce travail. Qu'il trouve ici mon profond respect et toute ma gratitude.
- Mr Lemnouar NOUI, Professeur à l'université de Batna pour l'honneur qu'il nous fait d'intégrer le jury et de juger ce travail. Qu'il trouve ici mon profond respect et toute ma gratitude.
- Mme Aini LAOUDI, Maitre de Conférence à l'USTHB pour l'honneur qu'elle nous fait d'intégrer le jury et de juger ce travail. Qu'elle trouve ici mon profond respect et toute ma gratitude.
- Mme Kenza GUENDA, Maitre de Conférences à l'USTHB pour l'honneur qu'elle nous fait d'intégrer le jury et de juger ce travail. Qu'elle trouve ici mon profond respect et toute ma gratitude.

Mes chaleureux remerciements s'adressent à Mme le professeur Fatiha MERAZKA, Professeur à l'institut d'électronique de l'USTHB pour ses précieux conseils.

A vous tous, mes sincères remerciements

Contents

Chapter 1

Introduction

La thématique de cette thèse porte sur "les éléments arithmétiques pour la cryptographie et techniques de cryptanalyse statistiques"

Nous avons pu travailler particulièrement sur la distribution du cryptosysteme BGN (Boney Goh Nissim). Nous avons pu relever le défi de la distribution de ce cryptosystème dont les propriétés arithmétiques sont intéressantes car on y trouve l'usage des courbes elliptiques. Ces dernières sont largement prouvée plus intéressantes que les outils arithmétiques classiques savoir le calcul modulaire dans le RSA par exemple. La performance des courbes elliptique sur le calcul modulaire est du au faite que les tailles des clé utilisées dans le ECC est moins importante que celles du RSA, pour une sécurité équivalente. Nous renvoyons la littérature dans ce domaine et particulièrement ceux de Guyeux et al pour les tests expérimentaux. Cette propriété a attiré notre attention et par conséquent nous avons essayé de le distribuer (généraliser).

La distribution des cryptosystèmes a une longue histoire et date des années 90. Dans un cryptosysteme classique, l'émetteur envoie un texte

chiffré sur un canal non sécurisé. Le receveur va par la suite décrypté le texte chiffré pour obtenir un texte claire en utilisant une clé secrète. Dans un cryptosysteme distribué, plusieurs parties interviennent, plutot que deux seulement, comme montré précédemment. Nous parlons alors d'un cryptosysteme (n, t) pour signifier que n personnes interviennent et que il faut au moins t personnes, parmi les n, pour déchiffré le teste chiffré. Cet algorithme est également divisé en deux catégories: sois le texte clair est distribué aux n parties ensuite chaque partie chiffre sont texte clair, ou bien chaque partie reoit ce qu'on appelle un share, mais qui est cette fois chiffré. Nous avons utilisé les deux procédés pour distribué BGN. La distribution a beaucoup d'application notamment dans le domaine du vote électronique ou celui de la sécurité des senseurs. Nous avons réussi le défi arithmétique de distribuer le cryptosystème BGN, qui, pour le meilleur de nos connaissances, n'a jamais été fait auparavant. Nous avons également prouvé sa sécurité dans un cadre bien précis savoir la sécurité sémantique. La sécurité repose sur l'intractabilité du calcul du logarithme discret sur courbe elliptique pour des tailles des clés suffisamment importantes. Egalement, nous avons montré, grce la simulation sur le logiciel Sage, que BGN distribué est plus performant quEl Gamal d'un point de vue rapidité. Sur le plan efficacité, des questions encore se posent et nous estimons que c'est une bonne direction pour des recherches futures.

Nous avons également réalisé une simulation en utilisant le logiciel open source Omnet++. Pour trouver les fonctions cryptographiques nécessaires, la librairie Crypto++, qui est également open source, a été utilisée. Cette simulation a pour but d'illustration plutt que de prouver la performance de

l'algorithme. Un autre calcul a eu lieux sur le logiciel Sage. Les opérations arithmétiques s'y trouvent implémentés et une grande puissance de calcul est possible sur ce genre de logiciel. Les résultats obtenus sur différentes courbes sont noté but de comparaison avec El Gamal.

Le second plan de cette thèse porte sur la cryptanalyse statistique. Nous avons amélioré un travail portant sur la cryptanalyse d'un cryptosystème classique l'aide des chaines de Markov. Les Chaines de Markov Monte Carlo (MCMC) ont beaucoup d'applications notamment en économie ou pour modéliser des phénomènes naturels. L'étude présente s'intéresse son application dans le domaine de la cryptanalyse.

Une chaine de Markov est un processus aléatoire dit sans mémoire. Cela veut dire que la probabilité d'un événement future dépend seulement de l'état précédent, et non de l'ensemble des états depuis l'état initial. Le but de l'algorithme MCMC est de générer une chaine de Markov qui converge vers la clé sécrète partir du texte chiffré. Nous nous trouvons donc avec une chaine de Markov dont les états sont l'ensemble des clés possibles qui ont pu tre utilisé.

Le cryprosystème en question est une combinaison d'un chiffrement par substitution et d'un autre par transposition. La substitution est l'application d'une permutation au texte claire et la transposition est le déplacement des lettres selon un motif régulier qui est par ailleurs la clé secrète. Combiner ces deux opérations génère une plus grande entropie.

Notre approche originale consiste changer le générateur de nombre aléatoire pour chaque simulation. Nous avons opté pour le xorshift, chaotique itération et le générateur classique fourni sur Linux pour effectuer les tests. Les

résultats obtenus prouvent la performance du xorshift sur les autres générateurs. Afin de mesure la performance d'une simulation, chaque simulation consiste en l'exécution du processus de cryptanalyse 100 fois. Une fois la simulation terminée, le nombre de décryptage réussi est enregistré. Le processus se comporte comme suit: Une clé aléatoire est générée. Cette dernière est utilisée pour chiffrer un texte de référence. Une fois le texte chiffré obtenu, le processus de décryptage Markov Chain Monte Carlo (MCMC) commence. Si la fin de cette dernière étape le texte clair est généré, nous estimons que le décryptage a réussi. Si toutefois le texte clair n'est pas retrouvé, nous estimons que le decryptage a échoué. Nous obtenons la fin une note sur 100 qui est la note attribuée la simulation.

Les lecteurs interessé par une introduction au domaine peuvent consulter les référence suivante: [23] [27] [35] .

Chapter 2

The Arithmetic over the group of points of an EC

In this chapter we will deal with the arithmetic properties of the elliptic curves. We will define the algebraic meaning of these objects and then present the explicit formula. Our goal is to give an expression of the point addition and multiplication.

2.1 Definitions

A curve whose genus is one is called an elliptic curve, more precisely:

Definition 1 *An elliptic curve is a pair* (E, O) *where:*

- E *is an smooth irreducible variety of genus one,*
- $O \in E$

The solutions (the points) of the cubic equation (weirstrass form) can be found on an extension field of the field K. The following definition give a more concise notion:

Definition 2 *An elliptic curve is defined over a field* K *if :*

- *E is a curve on K (i.e. given by the the zeros of a polynomial of $K[X,Y]$),*
- *O is a point of the curve whose coordinates are in K.*

We note the set of the points of E where the coordinates are in K as $E(K)$.

2.2 The Weierstrass Form

Theorem 1 *If E is an elliptic curve defined over K, then there exists $\Phi : E(K) \to P^2(K)$ which is an isomorphism of $E(K)$ on a curve $C(K)$ such that $\Phi(O) = (0:1:0)$ given by the Weierstrass equation:*

$$C : F(X,Y,Z) = Y^2Z + a_1XYZ + a_3YZ^2 - X^3 - a_2X^2Z - a_4XZ^2 - a_6Z^3 = 0$$

wherein, $(a_1, ..., a_6) \in K^5$.

So, the set of points of an elliptic curve E defined over K is given by :

$$E(K) = \{(X:Y:Z) \in P^2(K), F(X,Y,Z) = 0\}$$

- Only one class of this set verifies $Z = 0$: it is the class $(0:1:0)$ named point at infinity and is noted O. For the other classes, there exists a unique representation of the form $(X:Y:1)$. So, we consider in the following $E(K)$ as the union of O and the set of pairs (x,y), where $x = X/Z$ and $y = Y/Z$ verifying the equation

  $$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6,$$

  i.e. :

  $$E(K) = O \cup \{(x,y) \in P^2(K) / y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6\}$$

- O is the only point at infinity at it is not singularity, since $\partial F/\partial Z(0,1,0) = 1 \neq 0$.

2.3 The Group Law

In this section we introduce the group law on an elliptic curve. This additive law, is the main operation in elliptic curve based cryptosystems.

We begin by giving the *Poincaré theorem*

Theorem 2 *Let K be a field and (E,O) an elliptic curve over K. Define the law $+ : E(K) \times E(K) \to E(K)$ as $(P,Q) \to O * (P * Q)$. Then (E,O) together with this law is an abelian group. ($P * Q$ is the point of intersection of the line (PQ), or the tangent at the point P, with the curve)*

2.4 Points Addition on EC

Elliptic curves used in cryptography are typically defined over two types of finite fields: prime fields $\mathbb{F}_p$, where p is a large prime number and binary extension fields $\mathbb{F}_{2^m}$. We focus on elliptic curves over $\mathbb{F}_p$. For simplicity, let $p > 3$, and consider the special case where an elliptic curve over $\mathbb{F}_p$ is defined by cubic equation $y^2 = x^3 + ax + b$ as the set

$$\Sigma = \{(x,y) \in \mathbb{F}_p \times \mathbb{F}_p \mid y^2 \equiv x^3 + ax + b \ (mod\ p)\}$$

where $a, b \in \mathbb{F}_p$ are constants such that $4a^3 + 27b^2 \neq 0 \ (mod\ p)$. The point addition and its special case, point doubling over Σ, is defined as follows (the arithmetic operations are defined in $\mathbb{F}_p$):

Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be two points of Σ. Then:

$$P + Q = \begin{cases} O & \text{if } x_2 = x_1 \text{ and } y_2 = -y_1, \\ (x_3, y_3) & \text{otherwise.} \end{cases}$$

where:

- $x_3 = \lambda^2 - x_1 - x_2$,
- $y_3 = \lambda \times (x_1 - x_3) - y_1$,
- $\lambda = \begin{cases} (y_2 - y_1) \times (x_2 - x_1)^{-1} & \text{if } P \neq Q, \\ (3x_1^2 + a) \times (2y_1^{-1} & \text{if } P = Q. \end{cases}$

Finally, we define $P + Q = O + P = P, \forall P \in \Sigma$, which leads to an abelian group $(\sigma, +)$. The multiplication $n \times P$ means $P + P + ... + P$ n times, and $-P$ is the symmetric of P for the group law $+$ defined above, for all $P \in \Sigma$.

2.5 The group structure

In the general case we have the following theorem of *Kronecker*

Theorem 3 *Let G be an abelian group of finite order, then there exists a sequence $(a_n)_{n\in\mathbb{N}}$ such that :*

- $\forall i \in 1, ..., s-1, n_{i+1} \mid n_i$,
- $G \cong \mathbb{Z}/n_1\mathbb{Z} \oplus \cdots \oplus \mathbb{Z}/n_s\mathbb{Z}$, *where* $s \geq 2$.

In this case, the group G is then called of type $(n_1, ..., n_s)$ and of rank s.

In the case of elliptic curve, we have the following corollary :

Corollary 2.5.1 *$E(F_q)$ is an abelian group of rank 1 or 2, called cyclic or bi-cyclic. The type of this group is (n_1, n_2), i.e. :*

$$E(F_q) = \mathbb{Z}/n_1\mathbb{Z} \oplus \mathbb{Z}/n_1\mathbb{Z}$$

with $n_2 \mid n_1$ *and* $n_2 \mid q-1$.

Chapter 3

Elliptic Curve Discrete Logarithm Problem

3.1 Introduction

In public-key cryptography, each participant possesses two keys: a public key and a private key. These are linked in a unique manner by a one way function. We will define this notion in the following section. These are functions that are easy to calculate but difficult to inverse. The security of a cryptographic protocol is related to the difficulty of inverting. So, we will estimate this difficulty in term of the time needed for computers to inverse this function. Complexity theory deals with this notion of complexity of algorithms. Searching for such function is an important task in cryptography, this function being used as a cryptographic primitive in protocols. RSA is one of the well known protocol that rely on the difficulty of factoring large numbers. There exists sub exponential time algorithm that can factor such numbers. For a high-level security, the RSA is not convenient and we will search for an other one way function for which inverting is match harder.

Discrete logarithm systems over finite fields are another cryptographic primitive used in security protocols. But they present certain weakness and a special attention is consequently required in designing such system. On elliptic curves, we know construct discrete logarithm on the group of the point of the curve. In this chapter we explain how to construct such system leading to a suitable one way function and how to avoid attacks and weakness. This work is an abstract of the results obtained in [5].

3.2 Definitions

3.2.1 One Way Function

Let S be the set of binary strings and f be a function from S to S. We say that f is a one-way function if

- the function f is one-on-one and for all $x \in S$, $f(x)$ is at most polynomially longer or shorter than x
- for all $x \in S$, $f(x)$ can be computed in polynomial time
- there is no polynomial time algorithm which for all $y \in S$ returns either no if $y \notin S$ or $x \in S$ such that $y = f(x)$.

Given a one way function one can choose as the private key an $a \in S$ and obtains the public key $f(a)$. This value can be published since it is computationally unfeasible to compute a from $f(a)$.Complexity theory gives a mathematical measure to define what is meant by (computationally unfeasible). We treat this notion in a next section. For some applications it will be necessary to have a special class of computational one way functions

that can be inverted if one possesses additional information. These functions are called trapdoor one way functions. We will explain in this chapter how to construct such functions in a manner that no algorithm can invert it in a raisonnable computation time.

3.2.2 Generic Discrete Logarithm Systems

Let $(G, \oplus)$ be a cyclic group of prime order l and let P be a generator of G. Let the map

$$\phi : \mathbb{Z} \to G$$

$$n \to [n]P = \underbrace{P \oplus P \oplus ... \oplus P}_{ńtimes}$$

The problem of computing the inverse of this map is called the discrete logarithm problem (DLP) to the base of P. If $[n]P = Q$ so we note $log_P Q = n$.

In this section we consider the group of point of elliptic curve over finite field and the related DLP.

3.3 Elliptic Curves

An elliptic curve over a field K is a non singular protective algebraic curve over K with genus 1 together with a given point, I. Elliptic curve is defined as all the points on the curve given by the Weirstrass equation:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

with $a_1, a_2, a_3, a_4, a_5, a_6 \in K$ together with I, the point at infinity. We also require that the curve is non singular. Note that if the field we are working

over has a characteristic not equal to two or three, then we can transform this equation to:

$$y^2 = x^3 + ax + a_4x + b$$

We refer to section 2 for the point addition definition and further details.

3.4 Choice of Parameters

One has to be careful with the choice of the pair (C, F_q) if one wants to have instances in which the complexity of algorithms computing the discrete logarithm is $O(l^{1/2})$.

- l does not divide q to avoid transferring the DLP to a group in witch there is efficient attacks [34] [32] [33].
- The extension field of F_q is of a sufficiently large degree k. For small k, fast algorithms for DLP can be found in [18] [7].
- Let $q = p^d$, to avoid a weaker DLP [19], one has to avoid the case where $\frac{d}{d_0}$ is small (d_0 is a divisor of d) and the case where d is prime with the existence of a small t for which $2^t \equiv 1 \mod (d)$. Also, d must not be a Mersenne or a Fermat prime number.

3.5 Point Counting

We are searching for groups of prime order or of order of large prime divisor. This is why the speed of point counting is important in designing elliptic curve DLP. The computation of the order of a group for curves C of genus

g defined over fields F_q can be performed efficiently by not too complicated algorithms if:

- The curve C is already defined over a small subfield F_{q_0} of F_q, [37] [29] [36] or
- The genus g is equal to 1 [3] [4] [13] or
- The characteristic of F_q is small or
- The genus of C is 1 or 2, the field F_q is a prime field, and the curve C is the reduction modulo q of a curve C' with complex multiplication over a given order End_C in a CM-field. Here is the algorithm [38]

3.6 Primality

In the applications, we would have to make sure that a given integer p is prime. The most obvious way is to try for all integers $n \leq \sqrt{N}$ whenever n devise N. This is not realizable in practice. In fact, it is too time-consuming to prove primality. The best primality test algorithms will only prove p to be prime. Most algorithms will be probabilistic in nature in the sense that one output is always true while the other is only true with a certain probability. Iterating this algorithm allows us to enlarge the probability that the answer that was given only with a certain probability actually holds true. These techniques offer quite good performance. To prove primality using probabilistic algorithms one usually starts with some iterations of an algorithm whose output non prime is always correct. While the output prime is true only with a certain probability (Trial division, Fermat tests, Rabin Miller

test, Lucas pseudoprime tests, BPSW tests). After passing some rounds one uses an algorithm that is correct when it outputs prime (Atkin Morain ECPP test [17], APRCL Jacobi sum test [12]). The reason for this order is that usually the algorithms of the first type have a shorter running time and allow us to detect composite integers very efficiently. On the other hand, factoring algorithms (Pollard s rho method, Pollard s p-1 method, Factoring with elliptic curves, Fermat Morrison Brillhart approach [30] [31]) are usually much slower.

3.7 Conclusion

In this chapter, we have presented a general method to use when designing elliptic curve discret logarithm problem. We have reviewed the important results obtained in this field and presented them in a step by step approach. The goal is to give the general image and introduce a basic understanding of the field. Even if the security of such system remains on the fact that there is no polynomial time algorithm witch solve DLP, there still no proof of the non existence of such algorithm.

Chapter 4

A Note on a Cryptanalysis Algorithm

4.1 Introduction

Cryptography refers to the science of encrypting secret data. By definition, without certain information which is the key, a third person other than the sender and the receiver could not recover the secret data. At the same time, cryptanalyst tries to break cryptosystems in order to prove that there is a security flow. According to Kerckhoffs's principal, the algorithm should be known by all the parties and even the adversary ho want to break the code. In this chapter, we deal with classical cryptosystems and we try to improve a cryptanalysis algorithm by pseudo random number generators. Classical cryptosystems operate at the byte level of data, whereas the modern one operate at the bit level. Pseudo random number generators have been deeply studied for their important application in the computer science. This note is organized as follows. The next section present MCMC algorithm and its related theory foundation. After that, results of our computation are given.

Finally, we analyze those results in the explanation section.

4.2 Markov Chain Monte Carlo

Theorem 4.2.1 *If a Markov chain X_n on a finite or countable state space X is irreducible and aperiodic, with stationary distribution π, then for every subset $A \subseteq X$,*

$$\lim_{n\to\infty} P(X \in A) = \int_A \pi(x)dx$$

[22]

4.2.1 MCMC algorithm

We refer to [22] for an extended explanation.

1. Choose an initial state $X_0 \in X$.

2. For n = 1, 2, 3, ...

   - Propose a new state $Y_n \in X$ from some symmetric proposal density $q(X_{n-1}, ...)$.
   - Let U_n $Uniform[0, 1]$, independently of $X_0, ..., X_{n-1}, Y_n$.
   - If $U_n < (\pi(Y_n)/\pi(X_{n-1}))$, then "accept" the proposal by setting $X_n = Y_n$, otherwise "reject" the proposal by setting $X_n = X_{n-1}$

4.2.2 An MCMC Algorithm to Break a Substitution-Transposition Cryptosystem

1. Choose an initial state (state here are all possible encryption keys), and a fixed scaling parameter $p > 0$.

2. Repeat the following steps for many iterations (e.g. 10 000 iterations).

- Given the current state x, propose a new state y from some symmetric density $q(x, y)$.
- Sample U_n $Uniform[0, 1]$, independently from all other variables.
- If $u < (\pi(y)/\pi(x))^p$, then "accept" the proposal by replacing x with y, otherwise reject y by leaving x unchanged.

$$\pi(x) = \prod_{\beta_1,\beta_2} r(\beta_1, \beta_2)^{f_x(\beta_1,\beta_2)}$$

where r is the frequencies of letters of the reference text and f are those of the decrypted text using the key x.

4.2.3 Testing Methodology

At each time, we tested the MCMC algorithm with a different pseudo random number generator. We recorded the results obtained in terms of accuracy and number of successful decryption. Our results are in the next paragraph.

4.3 The drand48 Pseudo Random Number Generator

This generator generate a sequence of numbers according to this linear congruence

$$X_{n+1} = (aX_n + c) \mod [m]$$

where a, c and m are constants.

Table 4.1: Score of the 100 Simulations of the drand48 PRNG

Scores
83-86-80-87-82-83-84-78-84-76-84-83-79-87
78-81-78-84-82-83-85-82-78-85-81-78-83-83
86-80-87-84-82-81-83-83-83-79-89-79-85-80
80-81-80-82-76-89-77-79-88-85-87-76-81-79
86-78-79-79-80-74-80-80-86-82-83-78-82-85
80-81-78-80-82-82-83-84-86-87-80-89-81-82-84
79-79-88-78-82-81-89-81-84-86-86-87-84-85-73

4.4 The xorshift Pseudo Random Number Generator

The properties of this generator is that it is a very fast algorithm with a great period $(2^{128} - 1)$ [26].

```
#include <stdint.h>
uint32_t xor128(void) {
  static uint32_t x = 123456789;
  static uint32_t y = 362436069;
  static uint32_t z = 521288629;
  static uint32_t w = 88675123;
  uint32_t t;

  t = x ^ (x << 11);
  x = y; y = z; z = w;
  return w = w ^ (w >> 19) ^ (t ^ (t >> 8));
}
```

Table 4.2: Scores of the 100 Simulations of the xorshift PRNG

Scores
85-78-86-87-85-83-75-80-80-80-82-85-90-90
79-79-87-85-83-81-82-80-80-84-82-86-91-78
83-78-93-76-85-76-87-80-86-86-81-88-83-78
93-76-85-76-87-87-78-88-88-79-79-80-82-83
85-79-85-86-84-82-78-80-87-80-91-81-88-80
82-82-83-84-82-82-87-84-78-88-87-84-74-81-80
81-84-83-87-84-85-80-84-82-82-75-86-79-78-85

4.5 The Chaotic Iteration Pseudo Random Number Generator

This generator is from [6]. As shown in this later, this generator bypass xorshift in some tests and a deep theoretic study proved that this generator has good randomness properties. Here x is a binary array of length N.

```
a := XORshift1();
m := a mod 2 + c
for i = 0, . . . ,m do
b := XORshift2();
S := b mod N;
x[S] := 1 - x[S] mod 2;
end for
r := x;
return r;
```

4.6 Theoretic Analysis

In our experiments, we have shown that with different prng, the output of the cryptanalysis algorithm is different. The better the statistical properties of

Table 4.3: Score des 100 Simulation du CI PRNG

Scores
87-82-88-84-87-85-81-82-81-87-78-88-85-83
76-80-85-82-70-85-81-85-84-80-79-78-81-82
86-83-84-87-81-86-84-84-79-90-85-82-86-86
80-86-83-85-76-81-83-82-87-86-88-75-84-83
83-84-85-83-82-86-77-86-86-83-79-83-80-83
81-78-82-81-81-80-85-86-81-88-88-80-82-89-79-76
86-83-84-86-84-79-86-77-83-80-77-84-86-85

the generator are, the greater are the accuracy and the number of successful decryption. This due to the fact that the resulting Markov Chain converge to the decryption key more accurately.

4.7 Conclusion

In this chapter we presented our experiments on decryption of classical cryptosystems. Results show that xorshift pseudo random number generator is more convenient to this kind of application since the highest scores were achieved. Previous studies show that CI prng have the best statistical proprieties, but surprisingly this does not imply that they are more suitable for all kind of applications like those we have discussed.

Chapter 5

The Boneh-Goh-Nissim EC Distributed Scheme

5.1 Introduction

Public key cryptosystems are widely used nowadays in electronic banking, lotteries, voting systems and so one. However, a well known drawback in public key cryptography, is that the knowledge of the secret key gives a huge power to the owner. Hence a new direction in research named distributed systems.

Several elliptic curve threshold cryptosystems had been proposed in the literature [15] [16] [25] [11] [42]. We aims to introduce a novel distributed scheme for an elliptic curve cryptosystem (ECC) version introduced in [9] following the paradigm of [15]. Fouque et al. [15] proposed a first distribution version of Paillier together with a proof of semantic security.Fournaris [16] proposed a distributed version of ECC. However, no proof of security was given by this author.

The definition of semantic security of threshold cryptosystem used follow

the one stated in [15].

The deciphering operation of the *Boneh Elliptic Curve Cryptosystem* (ECCB) is done by computing the discrete logarithm on elliptic curve. We use *Pollard Lambda* algorithm which has complexity $O(\sqrt{T})$ in time of computation, where T is the length of the interval to which the message m belongs.

The security of the Paillier, ElGamal ECC (ECCEG) and ECCB cryptosystems rely on the so called *Decisional Composite Residuosity* assumption (DCRA) [28], the *Elliptic Curve Discrete Logarithm Problem* (ECDLP) [8] and the *Subgroub Decision Problem* assumptions [9], respectively. So, building on the ECDLP and the SDP assumptions, we prove that our distributed scheme is semantically secure, in the meaning introduced by [15].

This chapter is organized as follows. In section 5.2, we recall the basic definition and the scenario when dealing with threshold cryptography. We present our novel threshold scheme together with a proof of security of this later in section 5.4, and finally we conclude this chapter in section 5.6.

5.1.1 RSA versus ECC

The efficiency of the ECC over RSA cryptosystem in terms of speed of computation and energy consumption is a key element that aroused our interest in this issue. Furthermore, we intend to apply our threshold cryptosystem in order to solve security issues in energy and computation power restricted networks. We recall a comparison done in [24] (see table 5.1).

Another comparison study is published in [14]. Taking into account different parameters such as decryption time and encrypted file size, the authors

Symmetric	ECC	RSA
80	163	1024
128	283	3072
192	409	7680
256	571	15360

Table 5.1: Key size of ECC and RSA algorithms for a given security level

compared three different cryptosystems: RSA, ElGamal and Paillier. From the point of view of the authors, the throughput is the most important parameter to take into account and RSA demonstrated better performance than the other cryptosystems.

Finally, to conclude this section, these previous studies demonstrate the advantage and the interest of ECC both in them of resources constraint and security enhancement.

5.1.2 Zero knowledge proof of equality

We will apply zero knowledge proof of equality algorithm given in [42] to produce proofs of validity in step 3 of our threshold ECC, see section 5.4, the algorithm runs as follows:

Suppose that party B need to authenticate that it has a valid secret share key s to A without divulging any information about s.

1. A selects two random integers a and b smaller then the order of the cyclic group of the point of the elliptic curve.

2. It computes $(a_1, a_2) = a(sg) \mod P$ and $(b_1, b_2) = bg \mod P$, where P is a prime number related to the finite field $GF(P)$ on which the elliptic curve E is constructed and g is a generator of the group G of

the points on E.

3. It also computes $t_1 = a_1 b_1 \mod P$ and $t_2 = a_2 b_2 \mod P$ and send (ag, t_1, t_2) to party B.

4. B computes $sag = (r_1, r_2) \mod P$.

5. B computes $z_1 = t_1 r_1^{-1} \mod P$ and $z_2 = t_2 r_2^{-1} \mod P$ and send (z_1, z_2) to A.

At the end, A verifies that $(b_1, b_2) = (z_1, z_2)$. If this is true, B has the secret s, otherwise, the proof $proof = (z_1, z_2)$ is not valid.

The correctness of this scheme can be verified from the Menezes- Vanstone elliptic curve cryptosystem [42]

5.1.3 The complexity of the Pollard Lambda logarithm on elliptic curves

An important criterion to the cryptosystem to be operational is the complexity of the logarithm function. We have mentioned above the version of the ECC we are using. This has theoretical complexity $O(\sqrt{T})$, where T is the length of the clear texts. This is due to the fast Pollard Lambda algorithm, which is a version of Pollard algorithm when the interval to which clear texts belong is known. We experienced this time complexity using Sage software [40]. Our approach is to construct an elliptic curve using the arithmetic parameters obtained from [10]. We vary the length T and record the time needed by the algorithm to output the discrete logarithm for a randomly chosen integer $m \in [0, T]$ figure 5.1. Our computation has been done on a

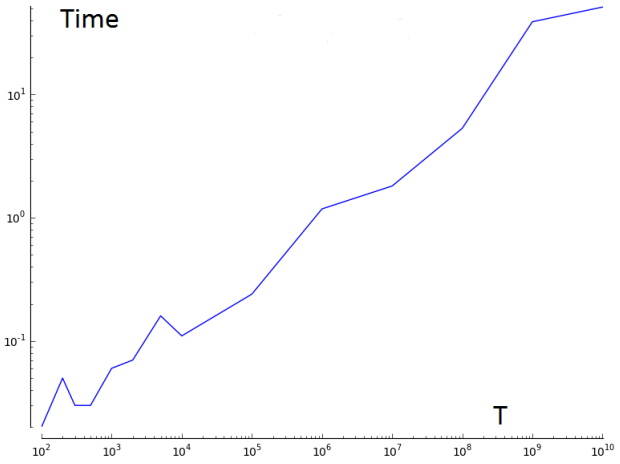

Figure 5.1: Computation time with T

virtual machine running with 512 Mo of ram, $i3$ processor, 2.10 GHz and 64-bit Operating System.

5.2 Threshold cryptosystems

5.2.1 Formal definition

A threshold cryptosystem consists of the following four components:

- A key generation algorithm takes as input a security parameter k, the number l of decryption servers, the threshold parameter t and a random string ω. It outputs a public key PK, a list $PK_1, ..., PK_l$ of private keys and a list $VK, VK_1, ..., VK_l$ of verification keys.

- An encryption algorithm takes as input the public key PK, a random string ω and a clear text M. It outputs a ciphertext c.

- A share decryption algorithm takes as input the public key PK, an index $1 \leq i \leq l$, the private key SK_i and a ciphertext c. It outputs a

decryption share c_i and a proof of its validity $proof_i$.

- A combining algorithm takes as input the public PK, a ciphertext c, a list $c_1, ..., c_l$ of decryption shares, the list $VK, VK_1, ..., VK_l$ of verification keys and a list $proof_1, ..., proof_l$ of validity proofs. It outputs a cleartext M or fails.

5.2.2 The players and the scenario

The game includes the following players : a dealer, a combiner, a set of l servers P_i, an adversary and users. All are considered as probabilistic polynomial time Turing machines, playing in the following scenario :

- In an initialization phase, the dealer uses the key generation algorithm to create the public, private and the verification keys. The public key PK and all the verification keys VK, VK_i are publicized and each server receives its shares SK_i of the secret key SK.

- To encrypt a message, any user can run the encryption algorithm using the public key PK.

- To decrypt a ciphertext c, the combiner first forwards c to the servers. Using their secret key SK_i and their verification keys VK, VK_i, each server runs the decryption algorithm and outputs a partial decryption c_i with a proof of validity of the partial decryption $proof_i$. Finally, the combiner uses the combining algorithm to recover the cleartext if enough partial decryptions are valid.

5.2.3 Security requirements

We recall that an active non-adaptive adversary completely controls the behavior of the corrupted servers and who choose which servers he wants to corrupt before key generation. A threshold cryptosystem is said to be $t-robust$ if the combiner is able to correctly decrypt any ciphertext, even in the presence of an adversary who actively corrupts up to t servers. Let us consider an attacker who first issue two messages M_0 and M_1. We randomly choose one of these messages. We encrypt it and send this ciphertext to the attacker. Finally, he answers which message has been encrypted. We say that the encryption scheme is semantically secure if there exists no such polynomial time attacker able to guess which of the two messages has been encrypted with a non-negligible advantage. The semantic security for threshold cryptosystem is defined as follow:

Let be an attacker who actively and non-adaptively corrupts t servers learns the public parameters, the secret keys of the corrupted servers, the public verification keys, all the decryption shares and the proof of validity of those shares. We consider the following game A:

A1 The attacker chooses to corrupt t servers. She learns all their secret information and she actively controls their behavior.

A2 The key generation algorithm is run; the public keys are publicized, each server receives its secret keys and the attacker learns the secrets of the corrupted players.

A3 The attacker chooses a message M and a partial decryption oracle gives

her l valid decryption shares of the encryption of M, along with proofs of validity. This step is repeated as many times as the attacker wishes.

A4 The attacker issues two messages M_0 and M_1 and send them to an encryption oracle who randomly chooses a bit b and sends back an encryption c of M_b to the attacker.

A5 The attacker repeats step A3, asking for decryption shares of encryptions of chosen messages.

A6 The attacker outputs a bit b'.

A threshold encryption scheme is said to be semantically secure against active non-adaptive adversaries if for any polynomial time attacker, $b = b'$ with probability only negligibly greater than $1/2$.

5.3 The RSA cryptosystem

5.3.1 The standard version of RSA

Let us recall the RSA cryptosystem.

Key generation Choose two distinct prime numbers p and q. Compute $n = pq$. Choose an integer e prime with $\phi(n) = (p-1)(q-1)$. Compute the multiplicative inverse d of e modulo $\phi(n)$. e is the public key and d is the secret key.

Encryption Let the cleartext be an integer $0 \leq m \leq n$, to encrypt it we compute $m^e[n]$.

Decryption We have $m = c^d[n]$ and so we recover the message m.

5.3.2 Threshold version of RSA

Key generation algorithm. The dealer chooses two strong primes $p = 2p'+1$ and $q = 2q'+1$(p' and q' primes); the RSA modulus is $n = pq$ and the public exponent e is a prime number greater then l:$PK = (n, e)$. Let $m = p'q'$. The dealer then compute the secret key $d \in Z_m$ such that $de = 1 \mod m$. It is shared using Shamir's secret sharing scheme: let $f_0 = d$ and, for $i = 1, ..., l$ f_i is randomly chosen in Z_m. Let $f(X) = \Sigma_{i=0}^{t} f_i X^i$; the secret key SK_i is $d_i = f(i) \mod m$. Finally, the dealer randomly chooses v in the cyclic subgroup of squares in Z_m^* and computes the verification keys $VK = v$ and, for $i = 1, ..., l$, $VK_i = v^{d_i} \mod n$.

Encryption algorithm: In order to encrypt a message M, any user can compute

$$c = M^e \mod n$$

.

Share decryption algorithm: Let Δ be $l!$; in order to obtain his share of the decryption of a ciphertext c, each server computes $c_i = c^{2\delta d_i} \mod n$ and a proof of validity to convince the combiner, or anybody else, that the discrete logarithm of c_i^2 in the base $c^{4\delta}$ is the same as the discrete log of VK_i in the base VK, namely the secret key $SK_i = d_i$. If less then t decryption shares have valid proofs of correctness the algorithm fails. Otherwise, let S be a set of $t+1$ valid shares; for any $i \in \{0, ..., l\}$

and any $j \in S$, let us define a variant of Lagrange coefficients:

$$\mu_{i,j}^S = \delta \times \frac{\prod_{j' \in S \setminus j}(i - j')}{\prod_{j' \in S \setminus j}(j - j')} \in \mathbb{Z}$$

The Δ factor is used in order to obtain integers and to avoid the computation of inverses modulo the secret value m. Therefore, the Lagrange interpolation formula implies:

$$\Delta f(i) = \sum_{j \in S} \mu_{i,j}^S f(j)[m]$$

Using the fact that $f(0)$ is equal to the secret key d, we can partially decrypt c:

$$M^{4\Delta^2} = c^{4\Delta^2 d} = c^{4\Delta^2 f(0)} = \prod_{j \in S} c^{4\Delta \mu_{0,j}^S f(j)} = \prod_{j \in S} c_i^{2\mu_{0,j}^S}[n]$$

Finally, since the public exponent is a prime number greater then l, it is relatively prime with $4\delta^2$ so the extended euclidean algorithm provides integers a and b such that $a \times 4\delta^2 + b \times e = 1$. This allow to obtain the ciphertext

$$M = M^{a \times 4\Delta^2} \times M^{b \times e} = (M^{4\Delta^2})^a \times c^b[n].$$

5.4 Elliptic curve cryptosystem

We recall that the ECC we present in this section is semantically secure under the *subgroub decision problem* (SDP) assumption [9].

5.4.1 Standard version of ECC

It has been proved that the ECC we describe bellow is semantically secure under the SDP assumption.

Key generation The public key is (n, G, g, h), where G is a group of order

n, $n = q_1 q_2$, g and u are generators of G, $h = u.q_2$. The private key is q_1.

Encryption algorithm Take a random $r \in [0, n-1]$. Let $m \in [0, ..., T]$ be a message to encrypt. So the ciphertext is computed as

$$C = m \times g + r \times h$$

Decryption algorithm Compute

$$m = log_{q_1 g} q_1 C$$

5.4.2 Threshold version of the elliptic curve cryptosystem

Key generation The dealer chooses an elliptic curve E such that the order of the group of point of this curve is $p + 1 = n \times s$ and such that $n = q_1 q_2$. q_1 and q_2 are two sufficiently large prime. So there exist a subgroup G of order n. Let g and u be two generators of G, put $h = q_2 u$ and $g_0 = q_1 g$. The public key PK will be $PK = (n, G, g, h, g_0)$ and the secret key $SK = q_1$. It is shared using Shamir's secret sharing scheme: let $f_0 = SK$ and, for $i = 1, ..., l$, f_i is randomly chosen in $\mathbb{Z}_n$. Let $f(X) = \sum_{i=0}^{t} f_i X^i$; the secret key SK_i is $d_i = f(i) \mod n$. The process of generating the f_i's is repeated until all the d_i's are different from q_1 and q_2. This is to ensure that the decryption shares c_i may exist and that $c_i \in [0, T]$ for a suitable chosen T.

Encryption algorithm Compute $C = m \times g + r \times h$.

Share decryption algorithm: Let $D = l!$ Compute

$$c_i = log_{d_i g} d_i DC$$

and

$$g_i = d_i g,$$

for $i = 1, ..., l$ so the share of each server will be the couple (c_i, g_i). To convince anyone that the server i has a valid share d_i, it uses zero knowledge proof presented in section 5.1.2, resulting in a proof of validity.

Combining algorithm: Compute

$$m = log_B A,$$

where $B = \Sigma_{j \in S} \mu_{0,j} g_j$ and $A = \Sigma_{j \in S} \mu_{0,j} c_j g_j$ The coefficients $\mu_{i,j}$ are the Lagrange coefficients defined by

$$\mu_{i,j}^S = D \times \frac{\prod_{j' \in S \setminus j} (i - j')}{\prod_{j' \in S \setminus j} (j - j')} \in \mathbb{Z}$$

for any $i \in \{0, ..., l\}$ and any $j \in S$.

We will use this Lagrange interpolation formula:

$$Df(i) = \sum_{j \in S} \mu_{i,j}^S f(j) \mod n$$

for any $i \in \{0, ..., l\}$ and any $j \in S$.

So for $i = 0$:

$$Df(0) = \sum_{j \in S} \mu_{0,j}^S f(j) \mod n$$

Thus

$$Dd_0 = \sum_{j \in S} \mu_{0,j}^S d_j$$

We have that

$$c_i = log_{d_i g} d_i DC \Leftrightarrow c_i g_i = d_i DC$$

and

$$m = log_{d_0 g} d_0 C \Leftrightarrow m d_0 g = d_0 C,$$

Multiplying by D we obtain:

$$m D d_0 g = d_0 DC,$$

so

$$m \Sigma_{j \in S} \mu_{0,j} d_j g = \Sigma_{j \in S} \mu_{0,j} d_j DC$$

$$\Leftrightarrow m \Sigma_{j \in S} \mu_{0,j} g_j = \Sigma_{j \in S} \mu_{0,j} c_j g_j$$

or

$$mB = A$$

So the proposed scheme is correct.

Theorem 5.4.1 *Under the SDA and the ECDLP assumptions, the threshold version of elliptic curve cryptosystem is semantically secure against active non-adaptive adversaries.*

Proof. The proof of this theorem is done with reduction; we show that if an adversary can break the semantic security of the threshold cryptosystem, then we can construct an attacker who can break the semantic security of the non-threshold one. Following this procedure, we have to simulate data received by the adversary in steps A2, A3 and A5 of game A. So the proof

consists of proving that the data simulated during the steps of the game A are indistinguishable from real one.

Let us assume the existence of an adversary $\mathcal{A}$ able to break the semantic security of the threshold scheme. We now describe an attacker which uses $\mathcal{A}$ in order to break the semantic security of the original ECC scheme. In a first phase, called the find phase, the attacker obtains the public key (n, G, g, h) and he chooses two messages M_0 and M_1 which are sent to an encryption oracle who randomly chooses a bit b and return an encryption c of M_b. In a second phase, called the guess phase, the attacker tries to guess which message has been encrypted.

We now describe how to feed an adversary $\mathcal{A}$ of the threshold scheme in order to make a semantic attacker. In step $A1$ of game A, the adversary chooses to corrupt t servers $P_1, ..., P_t$. In the find phase, the attacker first obtains the public key $PK = (n, G, g, h)$ of the regular ECC scheme. He randomly chooses t values $d_1, ..., d_t$ in the range $\{0, ..., n\}$. T is suitably chosen to efficiently compute the discrete logarithm on elliptic curve log. As q_2 is unknown and $T < q_2$, we have to simulate the value of T. However, as the computation efficiency of the function log is well known (section 5.1.3), a suitable choice of T may be obtained.

The attacker sends $(n, G, g, h, d_1, ..., d_t)$ to A in step 2 of game A.

During step A3, $\mathcal{A}$ chooses a message M and send it to the attacker. He computes $c = r \times h + M \times g$, where r is a random number, a valid encryption of M. The decryption shares of the corrupted players are correctly computed using the d_i's:$c_i = log_{d_i g} d_i Dc$, and $g_i = d_i g$ for $i = 1, ..., t$. The other shares

are computed from the following formula

$$c_i = log_a b$$

where $a = \sum_{j \in S} \mu_{ij} g_j$ and $b = D \sum_{j \in S} \mu_{ij} g_j c_j$. We have $c_i = log_{g_i} d_i Dc$ so $c_i g_i = d_i Dc$, and by multiplying by D: $Dc_i g_i = Dd_i Dc$. We have also $Dd_i = \sum_{j \in S} \mu_{ij} d_j$, this is why we have chosen the values of c_i as in the above formula. Furthermore, to compute the value of c_0, without the knowledge of d_0, we have $c_0 = log_{d_0 g} d_0 Dc$. $d_0 g = g_0$ is part of the public key and $d_0 c = d_0(r \times h + M \times g) = d_0 r \times h + d_0 M \times g = d_0 r \times h + M \times g_0 = M \times g_0$ (since h is of order d_0), so c_0 may be computed without any knowledge of the secret key d_0.

Finally, the adversary returns

$$(c, c_1, \ldots, c_l, proof_1, \ldots, proof_l).$$

In step A4, $\mathcal{A}$ chooses and outputs two messages M_0 and M_1. The attacker outputs those two messages as the results of the find phase.

The encryption oracle for the non-threshold ECC scheme chooses a random bit and sends an encryption c of M_b to the attacker. He forwards c to the adversary $\mathcal{A}$.

Step A5 is similar to step A3. Finally, in step A6, $\mathcal{A}$ answers a bit b' which is returned by the attacker in the guess phase.

We have $d_1, ..., d_t$ and the verification keys are randomly chosen on the interval $[0, T]$, so the distribution received by $\mathcal{A}$ during the key generation step is indistinguishable from a real one.

Also, the values received in step A3 and A5 are computed from the secret keys and the values of the non-corrupted servers are randomly chosen on the interval $[0, n]$.

Finally, all the data simulated by the attacker cannot be distinguished from real ones by $\mathcal{A}$. Consequently, if there exists a polynomial time adversary $\mathcal{A}$ able to break the semantic security of the threshold scheme, we have made an attacker able to break the semantic security of the original ECC scheme.

■

5.5 A second distribution of Boneh ECC (ECCB)

5.5.1 ECCB threshold algorithm

First we recall the distributed version of ElGamal elliptic curve cryptosystem (ECCEG) taken from [25]. Let the parameters of the cryptosystem be the finite field $GF(p)$, where p is a prime number, an elliptic curve $E_p(a, b)$ and a point G of order q on the group of point of E. Then we run the cryptosystem as follows :

Bob's private key is n_B with $0 < n_B < q$ and the public key is $K_B = n_B G$.

1. First we choose a prime number $p > max(M, n)$, and define $a_0 = M$, the message. Then we select $k - 1$ random, independent coefficients $a_1, ..., a_{k-1}, 0 \leq a_j \leq p - 1$, defining the random polynomial $f(x)$ over Z_p, a Galois prime field $GF(p)$.

2. We compute n shares, $M_i = f(x_i) \mod p, 1 \leq i \leq n$, where x_i can be just the public index i for simplicity, and convert it the to a point P_i

on the elliptic curve E.

3. Alice picks a random number r, and sends rG and $P_i + rK_B$ to Bob with index t.

4. Bob recovers each elliptic curve point by calculating $P_i + rK_B - n_B rG = P_i$.

5. Bob converts P_i to M_i, and deduces M by using Lagrange interpolation formula.

Our novel distributed crytosystem runs as follows : The public and private data are as in section 5.4.

1. First we define $a_0 = M$, the message. Then we select $k-1$ random, independent coefficients $a_1, ..., a_{k-1}, 0 \leq a_j \leq p-1$, defining the random polynomial $f(x)$ over Z_p, a Galois prime field $GF(p)$.

2. We compute n shares, $M_i = f(x_i) \mod p, 1 \leq i \leq n$, where x_i can be just the public index i for simplicity.

3. Alice picks a random number $r \in [0, n-1]$. So the ciphertext is computed as

$$C_i = M_i \times g + r \times h$$

.

4. Bob recovers each elliptic curve point by calculating

$$M_i = log_{q_1 g} q_1 C_i$$

.

5. Bob deduces M by using Lagrange interpolation formula.

5.5.2 Implementation details and computation efficiency

We compare in this section our threshold ECCB to ECCEG. We focus both on the ciphering plus the deciphering time of the two cryptosytems. We used Sage software [40] to compare the computational efficiency of the two algorithms. Rather then implementing the hole algorithm, we restricted to the encryption and the decryption functions, as the other parts of the two algorithms are the same. As we deal with the computation time, we have taken the average value of all the random values r. Also, when choosing the private key n_B, we have taken $q/2$.

For the ECCB cryptosystem, we used $T = 100$. So, for a given share $M_i < p$,M_i is written in base 100. Thus, we obtain $(M'_1, ..., M'_s)$, s depending on the value of p. Hence, for all $0 < i < s + 1$, $0 < M'_i < 100$.

Let compute the number of point addition during the ciphering and the deciphering process of the two cryptosystems. This is for this operation is a time consuming process. So, for the ECCEG, we will need $r+r+1+n_B+1 = 2r + 2 + n_B$ point addition to encryption and decryption one single point P_i. For the ECCB, we will need $r + M_i$ point addition and $s/2$ computation of the logarithm with the Pollard Lambda algorithm. As the value of s will have logarithm behavior, ECCEG will be more costly then ECCB for large keys.

We have taken 3 curves of different key lengths from [21] and computed the average time consumption of 100 runs of the two algorithms (See 5.2).

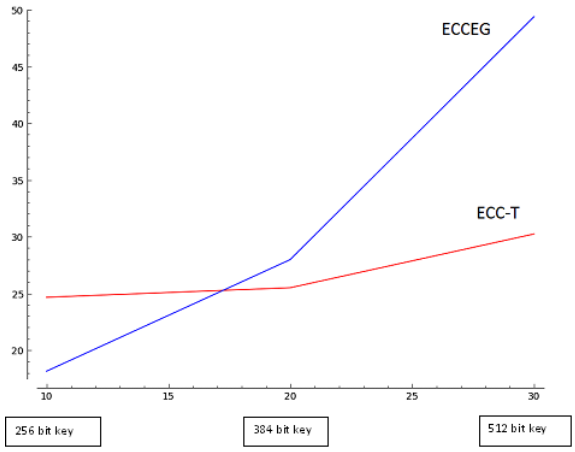

Figure 5.2: Computation time with T

5.6 Conclusion

We have introduced two novel threshold cryptosystems based on ECC along with a proof of security advantage for the first and the computational efficiently for the cryptosystem. Our scheme is accurate computationally due to the fast Pollard Lambda algorithm whose complexity is studied in section 5.1.3. And secure as stated in the theorem. The security of ECC rely on the difficulty of ECDLP on elliptic curves and the SDP. Furthermore, it is believed that ECDLP is more difficult then the problem of factoring large numbers. Another aspect of ECC is their important application in systems where the computational resources and energy are limited. So, in our future work, we will focus on the application of our cryptosystem in such environment like wireless sensor network and mobile social networks. To conclude, we recommend to use our first ECCTB for application where security is of great importance and to use our second ECCTB in application where speed is needed.

Chapter 6

Conclusion

In our thesis we have discussed arithmetic aspect of cryptography together with related cryptanalysis aspect. There is a duality between cryptology and cryptanalysis. When cryptology is concerned with encrypting sensitive data in order to achieve the confidentiality of the information, cryptanalysis try to break the codes. In this manner we are testing the robustness and the securities of the schemes.

We have restricted ourself to the classical cryptography case. This later is or no use in our days. However, modern cryptology relies on the same operating procedure. I mean the basic elementary operation over which modern cryptography relies, is a direct inspiration of the transposition and the substitution.

As a conclusion in the field of the cryptanalysis, we confirm the following points:

- Classical cryptography is of no use in our days.
- The choice of the PRNG used is a critical aspect in the cryptanalysis work.

- MCMC based algorithms are very accurate tools.

In the other part of this work, we have discussed the distribution of the BGN protocol. The main difficulty was technical and the security proof remains not solved. However, we have shown that the obtained scheme is interesting in the computational point of view.

We have discussed also the ECDLL. We have reviewed the main weaknesses and outlined the process of curve generation and the related point counting question.

Appendices

Appendix A

Application Section

A.1 Omnet++ Simulation of ECTC

Omnetpp software is an open source tool well known in the academic community. It is a powerful tool to simulate new algorithm before going to application in real scenario.

Here is Crypto Library, found in the web with open Licence that we have used for the simulation of the algorithm.

A.1.1 Code for Elliptic Curve Point Representation

Here we place the .h file.

```
#ifndef CRYPTOPP_ECP_H
#define CRYPTOPP_ECP_H

#include "modarith.h"
#include "eprecomp.h"
#include "smartptr.h"
#include "pubkey.h"


NAMESPACE_BEGIN(CryptoPP)
```

```
//! Elliptical Curve Point
struct CRYPTOPP_DLL ECPPoint
{
ECPPoint() : identity(true) {}
ECPPoint(const Integer &x, const Integer &y)
: identity(false), x(x), y(y) {}

bool operator==(const ECPPoint &t) const
{return (identity && t.identity) ||
(!identity && !t.identity && x==t.x && y==t.y);}
bool operator< (const ECPPoint &t) const
{return identity ? !t.identity : (!t.identity &&
(x<t.x || (x==t.x && y<t.y)));}

bool identity;
Integer x, y;
};

CRYPTOPP_DLL_TEMPLATE_CLASS AbstractGroup<ECPPoint>;

//! Elliptic Curve over GF(p), where p is prime
class CRYPTOPP_DLL ECP : public AbstractGroup<ECPPoint>
{
public:
typedef ModularArithmetic Field;
typedef Integer FieldElement;
typedef ECPPoint Point;

ECP() {}
ECP(const ECP &ecp, bool convertToMontgomeryRepresentation = false);
ECP(const Integer &modulus, const FieldElement &a, const FieldElement &b)
: m_fieldPtr(new Field(modulus)),
m_a(a.IsNegative() ? modulus+a : a), m_b(b) {}
// construct from BER encoded parameters
```

```
// this constructor will decode and extract the
//the fields fieldID and curve of the sequence ECParameters
ECP(BufferedTransformation &bt);

// encode the fields fieldID and curve of the sequence ECParameters
void DEREncode(BufferedTransformation &bt) const;

bool Equal(const Point &P, const Point &Q) const;
const Point& Identity() const;
const Point& Inverse(const Point &P) const;
bool InversionIsFast() const {return true;}
const Point& Add(const Point &P, const Point &Q) const;
const Point& Double(const Point &P) const;
Point ScalarMultiply(const Point &P, const Integer &k) const;
Point CascadeScalarMultiply(const Point &P, const Integer &k1,
const Point &Q, const Integer &k2) const;
void SimultaneousMultiply(Point *results, const Point &base,
const Integer *exponents, unsigned int exponentsCount) const;

Point Multiply(const Integer &k, const Point &P) const
{return ScalarMultiply(P, k);}
Point CascadeMultiply(const Integer &k1, const Point &P,
const Integer &k2, const Point &Q) const
{return CascadeScalarMultiply(P, k1, Q, k2);}

bool ValidateParameters(RandomNumberGenerator &rng, unsigned int level=3)
const;
bool VerifyPoint(const Point &P) const;

unsigned int EncodedPointSize(bool compressed = false) const
{return 1 + (compressed?1:2)*GetField().MaxElementByteLength();}
// returns false if point is compressed and not valid
//(doesn't check if uncompressed)
bool DecodePoint(Point &P, BufferedTransformation &bt,
size_t len) const;
```

```
bool DecodePoint(Point &P, const byte *encodedPoint,
size_t len) const;
void EncodePoint(byte *encodedPoint, const Point &P,
bool compressed) const;
void EncodePoint(BufferedTransformation &bt, const Point &P,
bool compressed) const;

Point BERDecodePoint(BufferedTransformation &bt) const;
void DEREncodePoint(BufferedTransformation &bt,
const Point &P, bool compressed) const;

Integer FieldSize() const {return GetField().GetModulus();}
const Field & GetField() const {return *m_fieldPtr;}
const FieldElement & GetA() const {return m_a;}
const FieldElement & GetB() const {return m_b;}

bool operator==(const ECP &rhs) const
{return GetField() == rhs.GetField()
&& m_a == rhs.m_a && m_b == rhs.m_b;}
int Dllambda(const Point &a, const Point &base,
const int &lb, const int &ub) const;

private:
clonable_ptr<Field> m_fieldPtr;
FieldElement m_a, m_b;
mutable Point m_R;
};

CRYPTOPP_DLL_TEMPLATE_CLASS DL_FixedBasePrecomputationImpl<ECP::Point>;
CRYPTOPP_DLL_TEMPLATE_CLASS DL_GroupPrecomputation<ECP::Point>;

template <class T> class EcPrecomputation;

//! ECP precomputation
template<> class EcPrecomputation<ECP> :
```

```
public DL_GroupPrecomputation<ECP::Point>
{
public:
typedef ECP EllipticCurve;

// DL_GroupPrecomputation
bool NeedConversions() const {return true;}
Element ConvertIn(const Element &P) const
{return P.identity ? P : ECP::Point(m_ec->GetField().ConvertIn(P.x),
m_ec->GetField().ConvertIn(P.y));};
Element ConvertOut(const Element &P) const
{return P.identity ? P : ECP::Point(m_ec->GetField().ConvertOut(P.x),
m_ec->GetField().ConvertOut(P.y));}
const AbstractGroup<Element> & GetGroup() const {return *m_ec;}
Element BERDecodeElement(BufferedTransformation &bt) const
{return m_ec->BERDecodePoint(bt);}
void DEREncodeElement(BufferedTransformation &bt, const Element &v)
const {m_ec->DEREncodePoint(bt, v, false);}

// non-inherited
void SetCurve(const ECP &ec)
{
m_ec.reset(new ECP(ec, true));
m_ecOriginal = ec;
}
const ECP & GetCurve() const {return *m_ecOriginal;}

private:
value_ptr<ECP> m_ec, m_ecOriginal;
};

NAMESPACE_END

#endif
```

A.1.2 Simulation Files

Here is the Client operating mode :

```
#include "DynaPacket_m.h"

#define STACKSIZE 16384

/**
 * Client computer; see NED file for more info
 */
class Client : public cSimpleModule
{
  public:
    Client() : cSimpleModule(STACKSIZE) {}
    virtual void activity();
};

Define_Module( Client );

void Client::activity()
{
    // query module parameters
    simtime_t timeout = par("timeout");
    cPar& connectionIaTime = par("connIaTime");
    cPar& queryIaTime = par("queryIaTime");
    cPar& numQuery = par("numQuery");

    DynaPacket *connReq, *connAck, *discReq, *discAck;
    DynaDataPacket *query, *answer;
    int actNumQuery=0, i=0;
    WATCH(actNumQuery); WATCH(i);

    // assign address: index of Switch's gate to which
```

```
//we are connected
int ownAddr = gate("port$o")->getNextGate()->getIndex();
int serverAddr = gate("port$o")->getNextGate()->size()-1;
int serverprocId = 0;
WATCH(ownAddr); WATCH(serverAddr); WATCH(serverprocId);

for(;;)
{
    if (ev.isGUI()) getDisplayString().setTagArg("i",1,"");

    // keep an interval between subsequent connections
    wait( (double)connectionIaTime );

    if (ev.isGUI()) getDisplayString().setTagArg("i",1,"green");

    // connection setup
    EV << "sending DYNA_CONN_REQ\n";
    connReq = new DynaPacket("DYNA_CONN_REQ", DYNA_CONN_REQ);
    connReq->setSrcAddress(ownAddr);
    connReq->setDestAddress(serverAddr);
    send( connReq, "port$o" );

    EV << "waiting for DYNA_CONN_ACK\n";
    connAck = (DynaPacket *) receive( timeout );
    if (connAck==NULL)
        goto broken;
    serverprocId = connAck->getServerProcId();
    EV << "got DYNA_CONN_ACK, my server process is ID="
       << serverprocId << endl;
    delete connAck;

    if (ev.isGUI())
    {
        getDisplayString().setTagArg("i",1,"gold");
        bubble("Connected!");
```

```
}

// communication
actNumQuery = (long)numQuery;
for (i=0; i<actNumQuery; i++)
{
    EV << "sending DATA(query)\n";
    query = new DynaDataPacket("DATA(query)", DYNA_DATA);
    query->setSrcAddress(ownAddr);
    query->setDestAddress(serverAddr);
    query->setServerProcId(serverprocId);
    query->setPayload("query");
    send(query, "port$o");

    EV << "waiting for DATA(result)\n";
    answer = (DynaDataPacket *) receive( timeout );
    if (answer==NULL)
         goto broken;
    EV << "got DATA(result)\n";
    delete answer;

    wait( (double)queryIaTime );
}

if (ev.isGUI()) getDisplayString().setTagArg("i",1,"blue");

// connection teardown
EV << "sending DYNA_DISC_REQ\n";
discReq = new DynaPacket("DYNA_DISC_REQ", DYNA_DISC_REQ);
discReq->setSrcAddress(ownAddr);
discReq->setDestAddress(serverAddr);
discReq->setServerProcId(serverprocId);
send(discReq, "port$o");

EV << "waiting for DYNA_DISC_ACK\n";
```

```
        discAck = (DynaPacket *) receive( timeout );
        if (discAck==NULL)
            goto broken;
        EV << "got DYNA_DISC_ACK\n";
        delete discAck;

        if (ev.isGUI()) bubble("Disconnected!");

        continue;

        // error handling
    broken:
        EV << "Timeout, connection broken!\n";
        if (ev.isGUI()) bubble("Connection broken!");
    }
}
```

A.1.3 The Serveur Operating Mode

```
#include <string.h>
#include <omnetpp.h>
#include <ecp.h>
#include <integer.h>
#include <stdio.h>
#include <vector>
#include <time.h>
#include "algebra.cpp"

#include "sercli_m.h"

//NAMESPACE_BEGIN(CryptoPP)

//#include <simulations/Tst.h>
```

```
class Serveur : public cSimpleModule
{
  protected:
    // The following redefined virtual function holds the algorithm.
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};

// The module class needs to be registered with OMNeT++
Define_Module(Serveur);

void Serveur::initialize()
{
    EV << "Setting the curves and the parameters \n";
    CryptoPP::ECP Gr(62417,0,1);
    CryptoPP::ECPPoint G(52405,14877);
    CryptoPP::ECPPoint H(41628,47380);
    std::vector<int> par;
    par.push_back(3480);
    par.push_back(162229);
    par.push_back(2215238);
    par.push_back(15069165);
    par.push_back(68085916);
    par.push_back(235700045);
    par.push_back(676989834);
    par.push_back(1693825813);

 //   srand(time(NULL));
    //int r = rand() %  10 + 1;
    int r = 2;
    int mess = 10;

    //encryption

    CryptoPP::ECPPoint C = Gr.Add(Gr.Multiply(mess, G),
```

```
    Gr.Multiply(r,H));

    //EV << "abcisse du point C  :   "
    //<< C.x << " et l ordonne" << C.y <<"\n";
 //   std::vector<long long> gammai; std::vector<long long> betai;
 //   int i;
/*
    int p = 101;
    for (i = 0; i < 8; i++)
    {
        int h = int(par[i]/p);

        gammai.push_back(h*p);
        betai.push_back(par[i]-(h*p));
    }

*/
    std::vector<int> ci;



    EV << "computing the shares ... \n" ;

/*
    CryptoPP::ECPPoint Q = Gr.Multiply(p,G);
    for (i = 0; i < 8; i++)
    {
        CryptoPP::ECPPoint P = Gr.Multiply(gammai[i],C);

        ci.push_back(  Gr.Dllambda(P,Q,0,500000000));
        EV << Gr.Multiply(ci[i], Q).x
        << "**************" << P.x <<"\n";
    }
*/
/*
```

```
    ci.push_back(499853778);
    ci.push_back(499548756);
    ci.push_back(499687548);
    ci.push_back(499991184);
    ci.push_back(499953786);
    ci.push_back(499923810);
    ci.push_back(499854672);
    ci.push_back(499990524);

    for (i = 0; i < 8; i++)
    {

        EV << "ci[" << i << "]=" << ci[i] <<"\n";
    }
*/
    // the matrices

/*    std::vector<std::vector<long long> > muij;
    std::vector<long long> mui;
    int a = 1; int b = 1;
    int j;
    for(i = 0; i < 10; i++)
    {
        for(j = 1; j<9; j++)
        {
           for(int k=1; k<9; k++)
               if (k!=j)
               {
                   a = a*(i-k);
                   b = b*(j-k);
               }
           mui.push_back((3628800/b)*a);
           a = 1; b = 1;
        }
        muij.push_back(mui);
```

```
        mui.clear();
    }
*/
    // share decryption

    //CryptoPP::ECPPoint B;
    //for(i=0; i<8; i++)
    //    B = Gr.Add(B , Gr.Multiply(muij[0][i]*par[i],G));

    //CryptoPP::ECPPoint A;
    //for(i=0; i<8; i++)
    //    A = Gr.Add(A , Gr.Multiply(muij[0][i],
    Gr.Add(Gr.Multiply(ci[i]*p,G),    Gr.Multiply(betai[i],C))   ));

    ///int res = Gr.Dllambda(A,B,0,20);

    //B = Gr.Multiply(10,B);
    // EV << "*************** : " << Gr.Add(C, Gr.Inverse(C)).x
    << Gr.Add(C, Gr.Inverse(C)).y;
    //CryptoPP::ECPPoint vv = Gr.Add(H,G);
    //EV << "B="<< B.x << B.y << "     A=" <<  A.x <<A.y << "\n   ";


    //for(i = 0; i < 10; i++)
    //{
    /*
    EV << "*************** \n";
    for(j = 0; j<8; j++)
        {
        EV <<muij[0][j] << "   ";
        }

    EV<< "\n";

    for(j = 0; j<8; j++)
```

```
        {
        EV <<par[j] << " ";
        }
    EV<< "\n";
    for(j = 0; j<8; j++)
         {
             int k =  muij[0][j]*par[i];
             EV << k << " ";

         }
    EV<< "\n";

*/
    //EV << "\n";
    //}
     //par.push_back(2);
    // Q; Q = G.Multiply(5,P);
    // Initialize is called at the beginning
    //of the simulation.
    // To bootstrap the tic-toc-tic-toc process,
    //one of the modules needs
    // to send the first message. Let this be ‘tic’.

    // Am I Tic or Toc?
    //if (strcmp("t1", getName()) == 0)
    //{
        // create and send first message on gate "out". "tictocMsg" is an
        // arbitrary string which will be the name of the message object.
        SerCi *msg = new SerCi("tictocMsg0");
        msg->setSecretkey(par[0]);
        //EV << "the secret key is " << msg->getSecretkey() << "\n";
        send(msg, "port$o", 0);

        SerCi *msgg = new SerCi("tictocMsg1");
        msgg->setSecretkey(par[1]);
```

```
        send(msgg, "port$o", 1);

        SerCi *msggg = new SerCi("tictocMsg2");
        msggg->setSecretkey(par[2]);
        send(msggg, "port$o", 2);

        SerCi *msgggg = new SerCi("tictocMsg3");
        msgggg->setSecretkey(par[3]);
        send(msgggg, "port$o", 3);

        SerCi *mssg = new SerCi("tictocMsg4");
        mssg->setSecretkey(par[4]);
        send(mssg, "port$o", 4);

        SerCi *msssg = new SerCi("tictocMsg5");
        msssg->setSecretkey(par[5]);
        send(msssg, "port$o", 5);

        SerCi *mmsg = new SerCi("tictocMsg6");
        mmsg->setSecretkey(par[6]);
        send(mmsg, "port$o", 6);

        SerCi *mmmsg = new SerCi("tictocMsg7");
        mmmsg->setSecretkey(par[7]);
        send(mmmsg, "port$o", 7);


}

void Serveur::handleMessage(cMessage *msg)
{
    SerCi *ttmsg = check_and_cast<SerCi *>(msg);
    //EV <<  "share received : " << ttmsg->getShare() << "\n";

    // The handleMessage() method is called
```

```
    //whenever a message arrives
    // at the module. Here, we just send it to
    //the other module, through
    // gate ‘out’. Because both ‘tic’ and ‘toc’
    //does the same, the message
    // will bounce between the two.
  //  send(msg, "out");
    //send(ttmsg, "port$o", 0);
    //ECPPoint P(2,3) , Q;
    //signed long i(5);
    //signed long i (5);

    if (ttmsg->getState() == 1)
    {
    EV << "Computing the clear text ...";
    CryptoPP::ECPPoint B (43938,46878);
    CryptoPP::ECPPoint A (16790,19579);
    CryptoPP::ECP Gr(62417,0,1);
    EV <<"le message claire est :" << Gr.Dllambda(A,B,0,20);
    }
    //int j = 5 ; //CryptoPP::ECP G(11,1,1);
    //CryptoPP::ECPPoint P(2,3);   // Q; Q = G.Multiply(5,P);
    //EV << "All right !!!" << endl;
}
```

A.1.4 The Structure of the Messages

```
message SerCi
{
    int secretkey;
    int publickey;
```

```
    int state;
    int source;
    int destination;
    int share;
    //int hopCount = 0;
}
```

A.1.5 The .NED file

```
package tst3;

simple Serveur
{
    parameters:
        @display("p=210,50");
        @display("i=device/server");
    gates:
        inout port[];

}
simple Client
{
    parameters:
        @display("i=device/pc2");
    gates:
        inout port;

}
network ClientServer
{
    parameters:
        int numClients = default(8);
        @display("bgb=684,195");
    submodules:
```

```
        server: Serveur{
        //        switch: Switch {
        //            parameters:
        //                pkRate = 1.5*numClients; //
        //pkRate should be >= numClients,
        //otherwise switch will become the bottleneck
        //                queueMaxLen = 20; // buffer max 20 packets
        //                @display("p=210,170");
                    gates:
                        port[numClients];
          }
        client[numClients]: Client {
           // parameters:
           //     timeout = 5s;
           @display("p=70,150,m,10,80");
           //@display("p=441,64");
        }
    connections:
            for i=0..numClients-1 {
        //         client[i].port <--> {  delay = 10ms; } <--> server.port[i];
        client[i].port <--> {  delay = 10s; } <--> server.port[i];

        }

 //       server.port <--> {  delay = 10ms; } <--> switch.port[numClients];
}
```

A.1.6 The Omnet.ini File

This is the source code of the omnet.ini file (this is what we call a launcher):

```
[General]
#network =
```

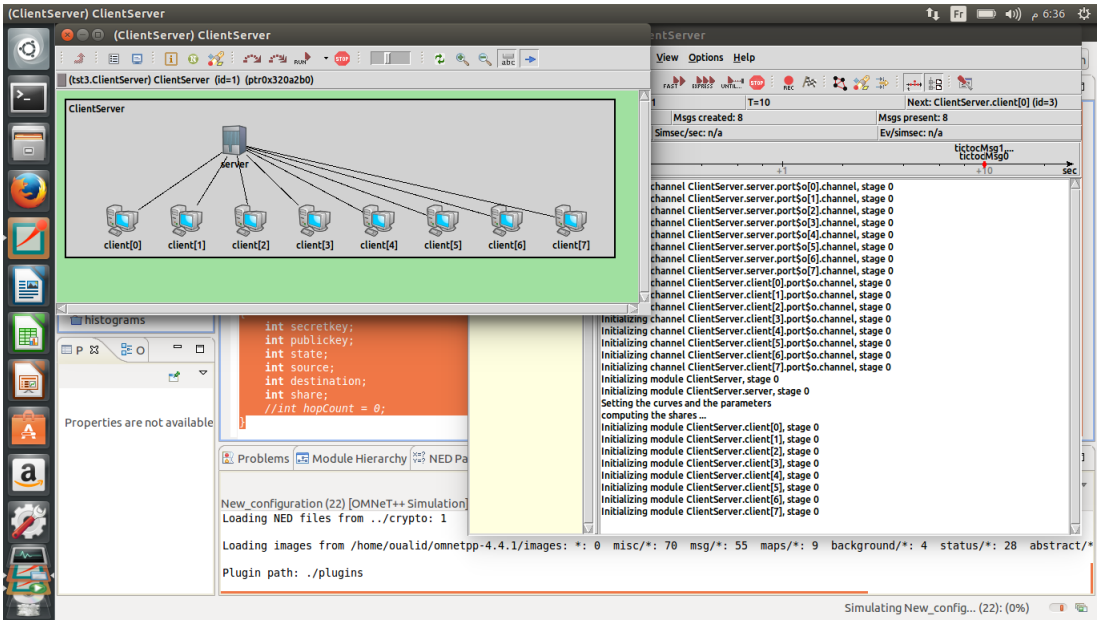

Figure A.1:

```
[Config ClientServer]

network = ClientServer
```

A.1.7 Screen shoot

This is the Screen shoot after raining the program. On the right we have the event log viewer which talk as which operation is being processed. On the left, we have a draw of the simulation devices. See A.1.

A.2 Sage Implementation of Elliptic Curve Discrete Log Lambda (ECLL)

ECLL was originally written in Sage environment and in Python programming language and I integrate it in c++ to omnet++ to the Crypto library.

Here is the code :

```
    from sage.rings.integer import Integer
    from operator import mul, add, pow

def discrete_log_lambda(a, base, bounds,
operation='*', hash_function=hash):

    if operation in addition_names:
        mult=add
        power=mul
    elif operation in multiplication_names:
        mult=mul
        power=pow
    else:
        raise ValueError("unknown operation")

    lb,ub = bounds
    if lb<0 or ub<lb:
        raise ValueError, "discrete_log_lambda()
        requires 0<=lb<=ub"

    # check for mutability
    mut = hasattr(base,'set_immutable')

    width = Integer(ub-lb)
    N = width.isqrt()+1

    M = dict()
    for s in xrange(10): #to avoid infinite loops
        #random walk function setup
        k = 0
        while (2**k<N):
            r = sage.misc.prandom.randrange(1,N)
            M[k] = (r , power(base,r))
            k += 1
```

```
    #first random walk
    H = power(base,ub)
    c = ub
    for i in xrange(N):
        if mut: H.set_immutable()
        r,e = M[hash_function(H)%k]
        H = mult(H,e)
        c += r
    if mut: H.set_immutable()
    mem=set([H])
    #second random walk
    H = a
    d=0
    while c-d >= lb:
        if mut: H.set_immutable()
        if ub > c-d and H in mem:
            return c-d
        r,e = M[hash_function(H)%k]
        H = mult(H,e)
        d += r

raise ValueError, "Pollard Lambda failed to find a log"
```

Appendix B

Some Definitions from the Complexity Theory

In this subsection we introduce some basic notion from complexity theory. The reader wishing further details can refer to [5]. The determination of the execution time of an algorithm consists of computing the number of basic operations needed for its execution. This last operation is called the time complexity of the algorithm. On the other hand, the space complexity measures the size of memory needed by an algorithm during its execution. Let f and g be two real functions. The function $g(N)$ is of order $f(N)$ denoted by $Of(N)$ if for a constant c one has:

$$|g(N)| \leq cf(N)$$

with $N \geqslant N$ for some constant N.

An algorithm has a polynomial time complexity when its complexity is of order $O(f(N))$, for some polynomial f and an input N. The complexity is exponential when the running time is of order $O(e^{f(N)})$, for some polynomial f. When the complexity is between the polynomial and the exponential, we

say that the complexity is sub-exponential.

Bibliography

[1] L.M. Adleman, J. Demarrais and M.D. Huang, A subexponential algorithm for discret logarithms over hyperelliptic curves of large genus over GF(q), *Theoret. Comput. Sci.*, 226(1999), 512-516.

[2] M.F.Atiyah and I.G.MacDonald. Introduction to commutative algebra. Addison-Wesley, London, 1969.

[3] A.O.L. Atkin, The number of points on an elliptic curve modulo a prime, *E-mail on the Number Theory Mailing List*, (1988)

[4] A.O.L. Atkin, The number of points on an elliptic curve modulo a prime, *E-mail on the Number Theory Mailing List*, (1991), 414-415.

[5] R. Avanzi, C. Doche, T. Lange, K. Nguyen and F. Vercauteren, *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, Chapman, (2006).

[6] J. Bahi, C. Guyeux, W. Qianxue, *Improving random number generators by chaotic iterations. Application in data hiding*,Int. Conf. on Computer Application and System Modeling,(2003).

[7] P.S.L.M. Barreto, H.Y. Kim, B. Lynn and M. Scott, Efficient algorithms for pairing-based cryptosystems, *Advenced in Cryptology*, 2442(354) (2002), 380-589.

[8] O. Benamara, *Elliptic Curve Discrete Logarithm Problem*, Genaral Mathematics Notes, vol. 15, pp. 84–91, March 2013.

[9] D. Boneh, E.J. Goh and N. Kobbi, *Evaluating 2-DNF formulas on ciphertexts*, Proceedings of the Second international conference on Theory of Cryptography, no. 17, pp. 325–341, 2005.

[10] R. Broker, P. Stevenhagen, *Constructing of elliptic curves of prime order*, Contemporary Mathematics, vol. 20, 2007.

[11] P. Changgen, and L. Xiang, *Threshold Signcryption Scheme Based on Elliptic Curve Cryptosystem and Verifiable Secret Sharing*, Proceedings of 2005 International Conference on Wireless Communications, pp. 1182–1185, 2005.

[12] H. Cohen and A.K. Lenstra, Implementation of a new primality test, *Math. Comp.*, 48(48) (1987), 103-121.

[13] N.D. Elkies, Explicite isogenies, *Draft*, (1991)

[14] S. Farah, M.Y. Javed, A. Shamim and T. Nawaz, *An experimental study on Performance Evaluation of Asymmetric Encryption Algorithms*, Recent Advences in Information Sciencce, vol. 62, no. 1, pp. 31–42, 2012.

[15] P.A. Fouque, G. Poupard and J. Stern, *Sharing decryption in the context of voting or lotteries*, Financial Cryptography, LNCS, pp. 90–104, 2000.

[16] A.P. Fournaris, *A Distributed Approach of a Threshold Certificate-Based Encryption Scheme with No Trusted Entities*, Information Security Journal: A Global Perspective, vol. 22, no. 3, pp. 126–139, 2013.

[17] J. Franke, T. Kleinjung, F. Morain and T. Wirth, Proving the primality of very large numbers with fast ECPP, *Algorithmic Number Theory Symposium*, 3076(2004), 194-207.

[18] S.D. Galbraith, K. Harrison and D. Soldera, Implementing the tate pairing, *Algorithmic Number Theory Symposium*, 2369(2002), 324-337.

[19] P. Gaudrey, F. Hess and N.P. Smart, Constructive and destructive facets of Weil descent on elliptic curves, *J. Cryptology*, 15(2002), 531-534.

[20] R.Hartshorne. *Algebraic Geometry.* Springer-Verlag. New York, 1977

[21] H. IVEY-LAW, R. ROLLAND, *Constructing a database of cryptographically strong elliptic curves*, Crypto'Puces, 9th-12th May 2011,http://galg.acrypta.com.

[22] C. Jian and J.S. Rosenthal, *Decrypting classical cipher text using Markov chain Monte Carlo*,(2012).

[23] Kahn, David (second edition, 1996), The Codebreakers: the story of secret writing, Scribners p.235

[24] K. Lauter, *The advantages of elliptic curve cryptography for wireless security*, Wireless Communications, IEEE, vol. 11, no. 1, pp. 62–67, 2004.

[25] E. Levent and L. Weimin, *ECC based threshold cryptography for secure data forwarding and secure key exchange in MANET (i)*, Proceedings of the 4th IFIP-TC6 international conference on Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; Mobile and Wireless Communication Systems, NETWORKING 05, no.12, pp. 102–113, 2005.

[26] G. Marsaglia, Xorshift RNGs,Journal of Statistical Software,(2003),1-6.

[27] Menezes, Alfred J. and Vanstone, Scott A. and Oorschot, Paul C. Van, Handbook of Applied Cryptography, isbn 0849385237, CRC Press, Inc., (1996)

[28] P. Paillier, *Public-key cryptosystems based on composite degree residuosity classes*, Proceedings of the 17th international conference on Theory and application of cryptographic techniques, Springer-Verlag, Berlin, Heidelberg, no. 16, pp. 223–238, 1999.

[29] S. Pohlig and M. Hellmann, An improvement algorithm for computing logarithms over GF(p) and its cryptographic signifiance, *IEEE Tran. Inform. Theory*, 24(1978), 106-110.

[30] C. Pomerance, Analysis and comparison of some integer factoring algorithms, *Math. Center Tracts*, 154(1983), 89-139.

[31] C. Pomerance, The quadratic sieve factoring algorithm, *Advances in Cryptology*, 209(1985), 169-182.

[32] I. Samaev, Evaluation of discrete logarithms in a group pf p-torsion points of an elliptic curve in charachteristic p, *Math. Compu.*, 67(1998), 353-356.

[33] N.P. Smart, The discrete logarithm problem on elliptic curve of trace one, *J. Cryptology*,12 (1999), 193-196.

[34] T. Satoh and K. Araki, Fermat quotients and the polynomail time discrete log algorithms for anomalous elliptic curves, *Comm. Math. Univ. Sancti Pauli*, 47 (1998), 92-91.

[35] Schneier, B.: Applied Cryptography, 2nd edn.Wiley, New York (1996)

[36] D. Shanks, Class number: A theory of factorization and generalization, *Proc. Symp. Math.*, 20(1971), 415-440.

[37] V. Shoup, Lower bounds for discrete logarithms and related problems, *Advences in Crryptology*, 1233(1997), 256-266.

[38] A.M. Spallek, Kurven vom Geschlecht 2 und ihre Anwendung in Public-key Kryptosystemen, *Universitat Gesamthchhschule Essen*PhD. Thesis, 1994

[39] S.Stepanov.*Codes on Algebraic Curves* , Plenum Publishers, New York, 1999.

[40] A. Stein and others, *Sage Mathematics Software (Version 5.11)*, The Sage Development Team, 2013.

[41] H.Stichtenot. Algebraic Function Field and Codes, Springer-Verlag, Berlin, 1993.

[42] X. Zhang, F. Zhang, Z. Qin, J. Liu, *ECC Based threshold decryption scheme and its application in web security*, Journal of Electronic Science and Technology of China, vol. 2, no. 4, 2004.