

# **Computer Methods for Integer Factorization and Discrete Logarithm: A Cryptographic Perspective**

**Shi Bai**

A subthesis submitted in partial fulfillment of the degree of  
Master of Information Technology (eScience) at  
The Department of Computer Science  
Australian National University

November 2006

© Shi Bai

Typeset in Palatino by  $\text{\TeX}$  and  $\text{\LaTeX}$  2e.

Except where otherwise indicated, this thesis is my own original work.

Shi Bai  
15 November 2006



Dedicated with loving appreciation to my parents for their love and inspiration.



---

# Acknowledgements

---

I am greatly indebted to my supervisor Professor Richard Brent for his insightful advice, extensive guidance, endless patience and encouragement throughout my study and research. Without those I can't complete the project smoothly. I would like to express my sincerest gratitude to Dr Alistair Rendell for his constant guidance, generous suggestion and commendable support, which kept me on track. I feel very proud to have worked with them. I would also like to take the opportunity to thank everyone who helps me during my project. I am indebted to you for both technical and moral support. Last but not least, I would like to express my heartfelt thanks and deepest appreciation to my parents. I am greatly indebted to them for their great and continuous teaching, love and caring.



---

# Abstract

---

Computer cryptology and related techniques underlie the security features of personal communications, commercial transactions and military systems nowadays. It has been revolutionized in the past thirty years by the development of public-key cryptography. Furthermore, many public-key cryptosystems base their security on the difficulty of solving some mathematical problems, such as integer factorization and discrete logarithms problems. Integer factorization is concerned with the problem of decomposing an integer into its prime factors and discrete logarithm problems are analogues of ordinary logarithms but modulo prime  $p$ . Such mathematical issues receive much research interest because their difficulties in solutions are exploited in many cryptosystems and they build the foundations of public-key cryptosystems from a theoretical perspective.

This study reviews contemporary literature concerning algorithms for solving integer factorization and discrete logarithm problems from a cryptographic perspective. We look into different types of algorithms for understanding and computing the problems; classify various algorithms by their characteristics and compare them in complexity; discuss the key papers in the field. A survey of related algorithms and their cryptographic applications is presented here as well as the understandings and analysis in the hope that this comprises a clear overview of the current state of the art of above issues.

The first chapter, which serves as an introductory section, briefly discusses about the background information including history, motivation and prerequisites. Integer factorization problems are introduced in the following chapter and detailed analyses of algorithms such as Pollard's rho algorithm and Brent's algorithm are discussed. The third chapter involves the theory of discrete logarithms problems with their analyses and applications. Then this thesis concludes with a summary of the analyses of algorithms. In the end, a detailed plan for future research is developed according to the proposal. Further research is suggested to focus on the empirical investigation of calculating integer factorization and discrete logarithm problems using Pollard' rho algorithms and its variants.

x

---

---

# Contents

---

<b>Acknowledgements</b>	<b>vii</b>
<b>Abstract</b>	<b>ix</b>
<b>1 Background</b>	<b>1</b>
1.1 History . . . . .	1
1.1.1 Cryptography . . . . .	1
1.1.2 Integer Factorization and Discrete Logarithm . . . . .	2
1.2 Mathematical Preliminaries . . . . .	3
1.2.1 Theory of Numbers . . . . .	3
1.2.2 Probability Theory . . . . .	5
1.2.3 Complexity Theory . . . . .	6
1.2.4 Abstract Algebra . . . . .	8
1.3 Concepts and Terminology . . . . .	8
1.4 Motivation and Objective . . . . .	11
1.5 Literature Review Plan . . . . .	12
1.6 Notations . . . . .	12
<b>2 The Large Integer Factorization Problem</b>	<b>15</b>
2.1 Introduction and Problem Formulation . . . . .	15
Definition . . . . .	15
Difficulty and Complexity . . . . .	16
Classification . . . . .	16
Importance . . . . .	16
2.2 Trial Division . . . . .	16
Principle . . . . .	16
Example . . . . .	17
Performance . . . . .	17
Parallel Trial Division . . . . .	17
Improvements . . . . .	17
Implementation of Algorithm . . . . .	18
2.3 Pollard's rho Method for Factorization . . . . .	18
Introduction . . . . .	18
"Birthday" Paradox . . . . .	18
Pseudorandom Generator . . . . .	19
Floyd's Cycle-finding Algorithm . . . . .	20
Principle of Pollard rho Method . . . . .	22

---

	Example . . . . .	23
	Notes . . . . .	23
	Performance . . . . .	24
	Parallel Pollard rho Method . . . . .	24
	Implementation of Algorithm . . . . .	24
2.4	Brent Algorithm . . . . .	25
	Introduction . . . . .	25
	Principle . . . . .	25
	Example . . . . .	25
	Performance . . . . .	26
	Implementation of Algorithm . . . . .	26
2.5	Conclusion and Comparison . . . . .	27
2.6	Application in Cryptography: RSA . . . . .	28
	Principle . . . . .	28
	Example . . . . .	28
	Security . . . . .	29
	Performance . . . . .	29
<b>3</b>	<b>The Discrete Logarithm Problem</b>	<b>31</b>
3.1	Introduction and Problem Formulation . . . . .	31
	Definition . . . . .	31
	Complexity and Importance . . . . .	32
3.2	Exhaustive Search . . . . .	32
	Principle . . . . .	32
	Performance . . . . .	32
3.3	Baby-step Giant-step Algorithm . . . . .	32
	Principle . . . . .	32
	Example . . . . .	33
	Implementation of Algorithm . . . . .	33
	Performance . . . . .	34
3.4	Pollard's Rho Method for DL problem . . . . .	34
	Introduction . . . . .	34
	Principle . . . . .	34
	Example . . . . .	35
	Implementation of Algorithm . . . . .	36
	Performance and Improvements . . . . .	36
3.5	Application in Cryptography: Diffie-Hellman . . . . .	37
	Principle . . . . .	37
	Example . . . . .	37
	Security . . . . .	37
<b>4</b>	<b>Conclusion</b>	<b>39</b>
4.1	Research Plan . . . . .	39

---

<b>A Proposal for Future Research</b>	<b>41</b>
<b>B Key Abstracts</b>	<b>43</b>
<b>C Floyd's cycle-finding algorithm and results</b>	<b>45</b>
<b>Bibliography</b>	<b>49</b>



# Background

---

Cryptography is used in many applications encountered in everyday life such as the security of ATM cards, computer passwords, E-commerce and is even more important in military applications. As a classical definition in [Rivest 1990], cryptography is all about communication in the presence of adversaries or eavesdroppers. The ultimate goal of cryptography is privacy: two parties wish to communicate with each other privately, so that an adversary knows nothing about what was communicated. Therefore, a broad research interest in cryptography arises from the requirements of more secure and reliable algorithms. Many modern cryptographic algorithms, such as RSA [Rivest et al. 1978] and Elliptic Curve Cryptography (ECC) [Cohen and Frey 2005], are based on trapdoor functions, which are widely used in cryptography. Generally speaking, a trapdoor function is a function that is easy to compute in one direction, yet believed to be difficult and time-consuming to find its inverse without extra special information. Although the existence of trapdoor function functions is still an open conjecture [Mollin 2003], many famous cryptographic algorithms nowadays rest on the existence of it. Such mathematical primitives as integer factorization and discrete logarithm problems are best known candidates which may be used to construct the trapdoor functions [Goldreich 2001]. Therefore a review on them would help better understand the theoretical foundation of cryptography.

In this introductory chapter, brief information about the history is discussed first; some mathematical preliminaries are explained for a better understanding of the topic; then basic concepts and definitions related with the topic are interpreted; finally, the literature review plan and schedule are briefly talked as a conclusion. This chapter serves as a framework on which the following chapters based.

## 1.1 History

### 1.1.1 Cryptography

Cryptography has a long and fascinating history. Before the modern era, cryptography was mainly concerned with messages' confidentiality [Kahn 1967]. Encryption-decryption was used to ensure the goal, which was the process to convert of messages from a comprehensible form into an incomprehensible one, and back again. In the

process, the messages must be rendered in the way that the interceptors or eavesdroppers were unable to find the information about original message. The earliest types of cryptography are transposition ciphers, which rearrange the order of alphabets in a message, and substitution ciphers, which replace alphabets with other alphabets using the some substitution rule. The earliest record about the application of cryptography in military traced back to the around 20 B.C. by Romans soldiers. They used a classical substitution cipher to write the secret letter, which was named as Caesar cipher because it was said to have been used by Julius Caesar to communicate with his army. The Caesar cipher needed little more than pen and paper, in which each letter in the message was replaced by another letter some fixed number of positions further down the alphabet. For example, the word "war" may become "xbs" in the secret message. However, they offered little confidentiality even in ancient times. Although cryptography has a long and complex history, there were less advances and the developments were not so exciting compared to its importance until the beginning of last century. By World War II, mechanical and electromechanical cipher machines were in wide use. Cryptography became more important as a consequence of political competition and religious revolution. It also played a crucial role in the outcome of World War II, especially in the Pacific war ground. The famous Battle of Midway could serve as an classical example to demonstrate the important of cryptography.

In recent decades, cryptography contributes much more in our daily life, business and government fields. The scope of cryptography has also expanded beyond confidentiality concerns to include techniques for authentication of message integrity or sender-receiver identity, digital signatures, interactive proofs, and secure computation [Kahn 1967]. More recently, the proliferation of computers and large communication systems help advance the modern cryptography techniques. Significantly, Diffie and Hellman introduced the revolutionary concept of public-key cryptography and also provided a key exchange method [Diffie and Hellman 1976] based on the discrete logarithm problems. Generally speaking, public-key cryptography is based on the trapdoor functions, such as discrete logarithm problems and large integer factorization problems. Following that in 1978, Rivest, Shamir and Adleman discovered the first practical public-key encryption algorithm, named as RSA [Rivest et al. 1978]. It's based on the problems of factoring large integers. From a modern cryptography perspective, the adversary is assumed to have limited computational resources, which is often analogous to an algorithm which runs in polynomial time. So we speak of the infeasibility of breaking a crypto-system if there is no algorithm to solve it in polynomial time. Similarly, a problem is considered to be difficult, hard or infeasible if no known algorithm exists which could solve it in polynomial time.

### 1.1.2 Integer Factorization and Discrete Logarithm

The concept of factorization had been introduced around 300 B. C. by Euclid's famous of fundamental theorem of arithmetic <sup>1</sup>. However, there were not much advances

---

<sup>1</sup>In number theory, the fundamental theorem of arithmetic states that every natural number either is itself a prime number, or can be written as a unique product of prime numbers.

---

until the time of Fermat. Fermat introduced a new factoring method other than the naive trial division method<sup>2</sup> in 16th century. Some other distinguished mathematicians contributed much to the problems as well, such as Euler, Gauss and Legendre. However, it was still very hard and labour some to factor integers which were large. Then the various sieving machines were built to do parts of the tedious computations and they remained to be the fastest factoring method until the late middle of last century because of the advent of computers.

It has advanced considerably during recent decades by the advent of high performance computers. Not only the existing classical methods have been advanced, some exciting new ideas have also been proposed, such as Pollard rho method [Pollard 1975]. On the other side, even that there existed some rather ingenious algorithms, the problem is still considered to be hard now and there is not any known efficient algorithm.

Discrete logarithm has a long history in number theory. Initially they were used primarily in computations in finite fields. The status of discrete logs started to grow in the beginning of last century, as more computations were done. As described above, research in discrete logarithm and integer factorization problems are being motivated and promoted by the need for developing better cryptographic techniques in recent decades. The main impetus for the intensive current interest in discrete logs came from the invention of the Diffie-Hellman cryptographic protocol [Diffie and Hellman 1976]. However, the discrete logarithm problems were rather obscure and no known efficient algorithms exist until now, just like integer factorization problems [Odlyzko 2000].

The history is referred from [Williams and Shallit 1994] [Riesel 1994] [Odlyzko 2000] and it reflects that both the integer factorization and discrete logarithm problems require great computational powers.

## 1.2 Mathematical Preliminaries

This chapter gives the mathematical background which is necessary for the thesis as well as basic definitions and notations. They are organized in several areas such as number theory, probability theory, complexity theory and abstract algebra. The theorems are presented as facts with proofs omitted.

### 1.2.1 Theory of Numbers

**The Sets of Integers** We denote the set of positive integers, as known as natural numbers, by symbol  $\mathbb{N}$ . The integer, rational, real and complex numbers are denoted by  $\mathbb{Z}$ ,  $\mathbb{Q}$ ,  $\mathbb{R}$  and  $\mathbb{C}$  respectively.

**Divisibility** Let  $a, b$  be integers, then  $a$  divides  $b$  if there exists an integer  $c$  such that  $b = ac$ . If  $a$  divides  $b$ , then it is denoted by  $a|b$ . In this case,  $a$  is a divisor of  $b$ .

---

<sup>2</sup>Briefly, trial dividing the candidate number by every prime number less than it. Details are discussed in the following chapters.

**Prime and Composite Numbers** A prime number is a natural number with only two distinct natural number divisors, which are exactly one and itself. Therefore, composite numbers are the positive integers other than prime numbers. By Euclid's Theorem, there are infinitely many primes.

**Greatest Common Divisor (GCD)** Greatest common divisor  $d$  of two integers  $a$  and  $b$  is the largest positive common divisor of  $a$  and  $b$ , denoted  $d = \gcd(a, b)$ . For example,  $3 = \gcd(6, 27)$ . Furthermore, two integers  $a, b$  are said to be relatively prime or coprime if  $\gcd(a, b) = 1$ .

**Division Algorithm** For integers  $a$  and  $d$ , where  $d > 0$ , there exist unique integers  $q$  and  $r$  such that  $a = qd + r$  and  $0 \leq r < |d|$ , where  $|d|$  denotes the absolute value of  $d$ .

- $q$  is called the quotient.
- $r$  is called the remainder.
- $d$  is called the divisor.
- $a$  is called the dividend.

For example, let  $a = 17$  and  $d = 3$ , since  $17 = 5 \cdot 3 + 2$ , then  $q = 5$  and  $r = 2$ .

**Modular Arithmetic** Modular Arithmetic is a system of arithmetic for integers, where the results are always related to the remainders in division operations. For example, the equation below is a modulo operation.

$$19\%5 = 4 \tag{1.1}$$

**Congruence** Two integers  $a$  and  $b$  are said to be congruent modulo  $n$ , if  $a$  and  $b$  have the same remainder when divided by  $n$ . Hence their difference  $(a - b)$  is a multiple of  $n$ . It is denoted as

$$a \equiv b \pmod{n} \tag{1.2}$$

For instance,

$$26 \equiv 12 \pmod{7} \tag{1.3}$$

**Euclidean Algorithm** Euclidean Algorithm is a method to find the greatest common divisor of two integers without factoring the two integers. It repeatedly divides the number with the remainder in turns until the remainder is zero and the previous remainder is the gcd of two integers. For  $a = 55$  and  $b = 10$ ,  $55 = 10 \cdot 5 + 5$  and then  $10 = 2 \cdot 5 + 0$ . Since remainder is zero now, then  $\gcd(55, 10) = 5$ . A simple implementation of Euclidean Algorithm is described below,

**Listing 1.1: Calculating  $\gcd(a, b)$  by Euclidean algorithm**

```

INPUT: Integers  $a, b$ .
OUTPUT: Greatest common divisor of  $a, b$ .

int gcd (int a , int b) {
    if (b == 0) return a;
    else return gcd (b, a % b);
}

```

The above algorithm has a running time of  $O((\log n)^2)$ .

**Fundamental Theorem of Arithmetic** Every positive integer  $n > 1$  can be expressed as a product of primes and this representation is unique as  $n = p_1^{k_1} \cdot p_2^{k_2} \cdots p_r^{k_r}$ . As an example,  $4725 = 3^3 \cdot 5^2 \cdot 7$ .

**The Euler  $\phi$  Function** Given a positive integer  $n$ ,  $\phi(n)$  denotes the number of positive integers not exceeding  $n$  that are coprime to  $n$ . Let  $n = 8$ , then  $\phi(n) = 4$  since there are four numbers (1, 3, 5, 7) which are coprime with 8.

- If  $p$  is a positive prime number, then  $\phi(p) = p - 1$ .
- Given  $n$  is a positive integer,  $\phi(n) = (p_1^{k_1} - p_1^{k_1-1}) \cdot (p_2^{k_2} - p_2^{k_2-1}) \cdots (p_r^{k_r} - p_r^{k_r-1})$ .

**Smooth Number** A positive integer  $n$  is called B-smooth if all of its prime factors  $p_i$  are bounded by integer B such that

$$p_i \leq B \quad (1.4)$$

For example, 320000 is 5-smooth since  $320000 = 2^9 \cdot 5^4$  and hence its biggest prime factor is 5.

**Prime Number Theorem** Let  $\pi(x)$  denotes the number of prime numbers which are not larger than  $x$ . Then

$$\lim_{x \rightarrow \infty} \frac{\pi(x)}{x / \ln x} = 1 \quad (1.5)$$

## 1.2.2 Probability Theory

**Definition** The probability that one particular event  $E$  occurs, denoted  $P(E)$ .

**Conditional Probability** Let  $E_1$  and  $E_2$  be two events. The conditional probability of  $E_1$  given  $E_2$  measures the probability of event  $E_1$  occurring, given that  $E_2$  has happened. It is denoted as

$$P(E_1|E_2) = \frac{P(E_1 \cap E_2)}{P(E_2)} \quad (1.6)$$

**Probability of No Repetition** If an experiment with  $n$  equally likely and independent events occurring  $k$  times, where  $1 \leq k \leq n$ , then the probability that all  $k$  events differ from each other is

$$\frac{n}{n} \cdot \frac{n-1}{n} \cdots \frac{n-k+1}{n} = \frac{n(n-1) \cdots (n-k+1)}{n^k} \quad (1.7)$$

### 1.2.3 Complexity Theory

**Time Complexity** An algorithm is a finite set of well-defined steps which will terminate when it reaches a defined condition by given a variable input. The computational complexity is important in computing as it is relevant to the efficiency of the algorithm. The running time of a problem is the number of steps that it takes to solve the problem as a function of the size of the input. The worst-case running time of an algorithm is an upper bound on the running time for any input as a function of the input size. The average-case running time is the average running time over all inputs of a fixed size [Menezes et al. 1997]. Because the exact running time of an algorithm is difficult to compute and it is decided by the input size, the asymptotic running time is often used to get an approximation [Hopcroft et al. 2001].

- **Big  $O$  notation:** The symbol  $O$  is used to describe an asymptotic upper bound for the magnitude of a function  $f(n) \in O(g(n))$  if there exists a positive constant  $c$  and a positive integer  $n_0$  such that for every integer  $n > n_0$ ,  $f(n) \leq cg(n)$ . Hence Big  $O$  notation gives a way to say that one function is asymptotically no more than another.

For example,

$$f(n) = 10 + 9(\log n)^3 + 7n + 3n^2 + 4n^3 \in O(n^3) \quad (1.8)$$

- **Small  $o$  notation:** The Big  $O$  notation has a companion called Small- $o$  notation. It means that one function is asymptotically less than another [Sipser 2006]. Say that  $f(n) \in o(g(n))$  if  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ . For example,

$$\sqrt{n} \in o(n) \quad (1.9)$$

- **Comparative growth rates of several functions:** Let  $c$  be an arbitrary constant,

where  $c > 1$ . The asymptotic growth rates are in increasing order in the following functions [Menezes et al. 1997]:

$$1 < \ln \ln n < \ln n < n^c < n^{\ln n} < c^n \quad (1.10)$$

**Polynomial-time Algorithm** A polynomial-time algorithm is an algorithm that takes at most  $O(n^c)$  steps to solve, where  $n$  is the input size and  $c$  is a constant. If a problem could be solved by a polynomial-time algorithm, the problem is often considered to be tractable or feasible and the algorithm is believed to be good or efficient. For example, calculating the greatest common divisor by Euclidean's Algorithm is an algorithm in polynomial-time.

**Exponential-time Algorithm** Any algorithm whose running time cannot be bounded in polynomial is called an exponential-time algorithm.

**Subexponential-time Algorithm** A subexponential-time algorithm is an algorithm whose worst-case running time function is in the form  $e^{o(n)}$ , where  $n$  is the input size and  $e$  is the base of the natural logarithm ( $e \approx 2.71828$ ).

**Running time estimate function  $L_n$**  The function  $L_n$  below is often used in run time estimates, where  $c$  is a positive constant integer, and  $a$  is a constant satisfying  $0 \leq a \leq 1$ .

$$L_n[a, c] = e^{[c+o(1)] \cdot (\ln n)^a \cdot (\ln \ln n)^{1-a}} \quad (1.11)$$

$$f = \begin{cases} L_n[a, c] \text{ is polynomial time in } \ln n, & \text{if } a = 0, \\ L_n[a, c] \text{ is exponential time in } \ln n, & \text{if } a = 1, \\ L_n[a, c] \text{ is a subexponential-time algorithm,} & \text{if } 0 < a < 1 \end{cases} \quad (1.12)$$

**Deterministic Algorithm** A deterministic algorithm will always produce the same correct outputs when given the same inputs. Moreover, the underlying calculation steps are the same given the same inputs. The concept of "mathematical function" serves as an example of deterministic algorithm because it always produces the same result given the same input. The main advantage of deterministic algorithm is that the process and result are always correct and predicted. However, for some problems deterministic algorithms are hard to find and the known deterministic algorithms remain considerably slower in practice. Furthermore, sometimes the predictable results are not expected and people want random results. In such situation, a randomized algorithm is needed.

**Randomized algorithm** A randomized algorithm or probabilistic algorithm makes some random or pseudorandom choices as part of its logic [Cormen et al. 1990]. It could be categorized into two types. If the randomized algorithm always outputs the correct answer with less consideration in running time, it is called Las Vegas algorithms. Another one is Monte Carlo algorithm, which may produce the error in a small probability but often works fast.

### 1.2.4 Abstract Algebra

**Binary Operation** A binary operation is a calculation involving two input quantities within a single set  $S$ . Typical examples of binary operations are the addition (+) and multiplication (\*) of numbers.

**Group** A group  $(G, *)$  is a set  $G$  with a defined binary operation  $*$  (or others) as  $G * G \rightarrow G$  that satisfies the following three axioms [Seress 1997]:

- Associativity: For all  $a, b$  and  $c$  in  $G$ ,  $(a * b) * c = a * (b * c)$ .
- Identity element: There is an element  $e$  in  $G$  such that for all  $a$  in  $G$ ,  $e * a = a * e = a$ .
- Inverse element: For each  $a$  in  $G$ , there is an element  $b$  in  $G$  such that  $a * b = b * a = e$ , where  $e$  is an identity element.

The order of a group  $G$  is the number of elements in the set  $G$ . If the order is not finite, then the group is an infinite group. A group is abelian (or commutative) if the operation is commutative, that is, for all  $a, b$  in  $G$ ,  $a * b = b * a$ . A group  $G$  is said to be cyclic if there exists an element  $g \in G$  such that, for  $a \in G$ , there is an integer  $i$  satisfying  $a = g^i$ . Such  $g$  is called a generator of the group  $G$ . For example, the set  $\mathbb{Z}_n$  is a group of order  $n$  with the operation of addition modulo  $n$ .

## 1.3 Concepts and Terminology

Key concepts in cryptography and related information are introduced in this section. The problem definition and formulations of integer factorization and discrete logarithm are left to discuss in the following chapters.

**Cryptography** Cryptography is the study of mathematical techniques related to aspects of information security. For example, it's related with information confidentiality, data integrity, entity authentication and data origin authentication [Menezes et al. 1997]. A cipher is a way or an algorithm for hiding ordinary text, called plaintext, by transforming it into ciphertext [Wagsaff 2003]. The process is called enciphering or encryption of the plaintext into ciphertext. The reverse process is said to be deciphering or decryption. The detailed operation of a cipher is controlled both by the encryption or decryption algorithm and keys. Keys are parameters for enciphering

---

or deciphering algorithm. They could be regarded as information that helps translate plain text to encoded text and vice versa. According to the characteristic of keys, we could classify them into two groups: symmetric-key (or private-key) and public-key cryptography.

**Symmetric-Key Cryptography** In symmetric-key cryptography scheme, both the sender and receiver share the same (or related in an easily computable way) key. It's equally to say that parties use the trivially related cryptographic keys for both decryption and encryption. This was the only kind of encryption publicly known until 1976. Cryptosystems in this category includes the famous DES, Blowfish and AES [Schneier 1994]. There are main two disadvantages in this kind cryptography. First, it's difficult to ensure the security of the keys, both encryption and decryption key, when a secure channel doesn't already exist between them. Exposure of any key would make the cryptosystem insecure. Another significant disadvantage of symmetric-key cryptography is in the management method of keys. Because each distinct pair of key must be established for one communication, the number of keys required may increase very quickly along with more communications, and hence require complex key management schemes.

**Public-Key Cryptography** The second type of modern cryptosystem allows users to communicate securely without having prior access to a shared secret key. This is done by using a pair of cryptographic keys, designated as public key and private key, which are related mathematically. Many of them are based on certain time-consuming problems in computational complexity theory. In public-key cryptosystems, the public key is typically used for encryption, while the private or secret key is used for decryption. The public key may be freely distributed, while its paired private key must remain secret. The two keys are related mathematically. Cryptosystems in this category includes RSA [Rivest et al. 1978], ElGamal [Gamal 1985], and Diffie-Hellman [Diffie and Hellman 1976]. The public-key cryptography will be considered as the applications in the thesis as it is relevant to the integer factorization and discrete logarithm problems.

Public-key cryptography is a series of protocols and there are many applications of public key cryptography such as public-key encryption, public-key digital signature and key agreement. We concentrate on the encryption in the thesis.

- encryption: encrypt the plaintext and keep it secret from the eavesdroppers.
- digital signature: verify the message and ensure that it is created by a specific private key by someone.
- key agreement: allows two parties that may not initially share a secret key to agree on one.

**Hard, Difficult, Infeasible, or Intractable Problems** A problem is considered to be hard, infeasible or intractable if there is no known deterministic algorithm which could solve it in polynomial time. Public-key cryptographic algorithms are often based on the computational complexity of hard problems, many of which are from number theory. For example, the hardness of RSA is related to the integer factorization problem, while Diffie-Hellman is relevant to the discrete logarithm problem. In these cryptosystems, one could only use the secret key to decrypt the ciphertext which is encrypted by the public key (except some pitfalls in implementation) and it's computational infeasible to deduce the secret key from the public key.

**One-way Function** A one-way function is a mapping that is easy to compute from one direction to another but computationally infeasible to invert. For a one-way function  $f$ , it is easy to compute  $y = f(x_1, x_2, \dots, x_i)$ , but difficult to compute  $f^{-1}(x_1, x_2, \dots, x_i)$ . We say that  $f$  is not computationally invertible when computing  $f^{-1}(x_1, x_2, \dots, x_i)$  is intractable. The existence of one-way functions is an open conjecture. However, we have to tacitly assume the existence of one-way function [Goldwasser and Bellare 2001]. The most commonly used candidates for such functions are large integer factorization and discrete logarithm problems.

**One-way Trapdoor Function** A trapdoor function is a one-way function that is easy to compute in one direction and assumed to be difficult to invert without special information. However, given such extra information, it is computationally feasible to compute the inverse. They are widely used in cryptography. RSA and Rabin related functions are the examples of trapdoor functions.

**Hybrid Cryptosystems** Because many public-key cryptographic algorithms are based on hard problem from number theory, the calculation involves operations related with modular arithmetic. They are rather computationally expensive than the techniques used in symmetric-key cryptosystems. Therefore, a hybrid cryptosystem is often used in practical public-key cryptosystems: a fast symmetric-key encryption algorithm is often used for the text itself, while the relevant symmetric key is encrypted by a public-key algorithm.

**Principle of Public-key Cryptography** The figure 1.1 is used to explain the principle of public-key cryptography. Alice and Bob denote the two sides in communication. It illustrates that how the public-key cryptography works when Bob is sending a message to Alice. As a communicative model based on public-key cryptography, for each party, there are two keys: the public key for encryption and the private one for decryption. They are mathematically related. For example, in our case Alice generates her public key and private key, which are related with each other. The public key is open to other persons and private key is kept secretly.

In the first step, Alice needs to generate two mathematically related keys, public key  $p$  and private key  $t$ . Because Bob does not want other people to see the message, Bob

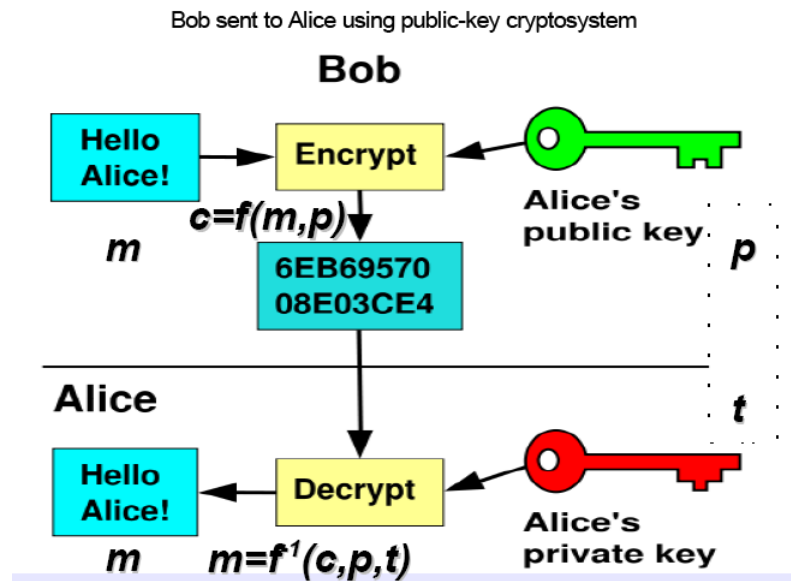


Figure 1.1: A Communicative Model Based On Public-key Cryptography

needs to encrypt his message  $m$ . Hence, Bob uses Alice's public key to encrypt the message as  $c = f(m, p)$ . The encrypted message is then sent to Alice. Assume that there are some eavesdroppers who are intercepting the communication line and they may possibly get the encrypted message. However, even they could get the encrypted message, the original plain message is still a secret. The reason is as described above, it is computationally infeasible to invert such encryption algorithm ( $m = f^{-1}(c, p)$ ) because it is a one-way function to the eavesdroppers. Then the encrypted message is received by Alice and she could decrypt the message using her private key as  $m = f^{-1}(c, p, t)$ . It is because the private key serves as the extra information here and it appears to be a trapdoor function to Alice. It is analogous to the mailbox which we are using everyday. People couldn't open others' mailboxes without the owners' keys.

Communication from another direction is similar. Alice use Bob's public key to encrypt the plain text, and Bob use her secret key to decrypt the encrypted text. There is no worry even if someone eavesdropped the communication because it's computationally infeasible to invert the coded message to plain one. We could sent an encrypted message to any person if we know his public key. And the secrecy depends on the security of his private key.

## 1.4 Motivation and Objective

Computer cryptography, in particular public-key cryptography, has become a vital component in our daily life, and it is extremely important in tasks such as military communication, e-Commerce. The significance of studying integer factorization and

discrete logarithm problems lies in that they build the foundation of modern public-key cryptography. They are used to ensure the security in many applications. On the other side, they are the underlying, presumably hard problem upon which several public-key cryptosystems are based, including RSA, ElGamal. As it has not been proven that trapdoor function must be difficult, and there remains a possibility that a quick and easy method might be discovered, although researchers consider this possibility remote. Solving either problem would allow people to figure out the private information. The result may also help us study and understand related problems in computational complexity field.

The literature review would look into the underlying mathematical algorithm of cryptography; especially concentrate on large integer factorization and discrete logarithm problems. The purpose is to review the most significant algorithms in the related areas. Then we would like to understand, classify and compare such algorithms and finally identify the opportunity for potential improvements.

## 1.5 Literature Review Plan

As discussed above, some public-key cryptography systems are based on big integer factorization problems, some are based on discrete logarithm problems, and some are based on elliptic curves which are related with discrete logarithm problems. So a survey with regard to both the factoring algorithms and methods for discrete logarithm problems would be necessary. The literature review composites mathematical foundations, algorithms and their applications in public-key cryptography.

The literature review would mainly emerge from below sources, but not restricted.

- Textbook chapters on Computational Number Theory and Cryptography.
- Recent journal articles and Bibliographies of same area.
- Conference proceedings and Transactions papers.
- Internet searches using Google, Citeseer and other general references.

## 1.6 Notations

The following standard notation will be used throughout, some of which has been introduced in preliminary part:

1.  $\mathbb{N}$ ,  $\mathbb{Z}$ ,  $\mathbb{Q}$ ,  $\mathbb{R}$  and  $\mathbb{C}$  denote set of natural, integer, rational, real and complex numbers respectively.
2.  $e$  denotes the base of natural logarithm, where  $e \approx 2.71828$ .
3.  $\gcd(a, b)$  denotes the greatest common divisor of  $a$  and  $b$ .
4.  $\phi(n)$  is the Euler  $\phi$  Function.

5.  $a \in A$  means that element  $a$  is a member of set  $A$ .
6.  $A \subseteq B$  means  $A$  is a subset of  $B$ .
7.  $A \subset B$  means  $A$  is a subset of  $B$ , yet  $A \neq B$ .
8.  $A \cap B = \{x | x \in A \text{ and } x \in B\}$ , denotes the intersection of sets  $A$  and  $B$ .
9.  $A \cup B = \{x | x \in A \text{ or } x \in B\}$ , denotes the union of sets  $A$  and  $B$ .
10.  $\ln x$  is the natural logarithm of  $x$ ;  $\lg x$  is the logarithm of  $x$  to base 2;  $\log x$  is the logarithm of  $x$  to an unspecified base.
11.  $\Pi$  is product symbol such that  $\Pi_{i=1}^n x_i = x_1 \cdot x_2 \cdots x_n$ .
12.  $\Sigma$  is summation symbol such that  $\Sigma_{i=1}^n x_i = x_1 + x_2 + \cdots + x_n$ .
13.  $\lfloor x \rfloor$  is the largest integer less than or equal to  $x$ ;  $\lceil x \rceil$  is the smallest integer larger than or equal to  $x$ ;



---

# The Large Integer Factorization Problem

---

In the previous Chapter we get a view of the history and background of integer factorization problems and their applications are mentioned. This chapter considers algorithms for computing large integer factorization and principle of RSA public-key cryptosystem.

In general, factoring a composite integer is believed to be a hard problem. As yet there is no rigorous mathematical proof on which the assumption could be based. We believe it to be a hard problem because so far we have no efficient and practical factoring method [Lenstra 2000]. While on the other side, newer and better algorithms emerge in these years along with the advance of computer and mathematics.

In this Chapter, we start with a general introduction and problem formulation, and then provide a more detailed discussion on selected algorithms. For each of the algorithms, principle of algorithms, examples, performance, improvements and variants are reviewed respectively. Finally, RSA public-key cryptography serves as an application related with integer factorization problems.

## 2.1 Introduction and Problem Formulation

**Definition** Integer factorization means finding positive non-trivial divisors of integers such that the product of them equals the original integer. A factorization of a composite number is not necessarily unique because the results may be composite; however, the prime factorization must be unique. Because according to the Fundamental Theorem of Arithmetic, every positive integer  $N$  could be expressed as a product of primes and the decomposition is unique without considering the order [Rosen 1993]. Prime factorization could be obtained by further factoring the factors that happened to be composite. Therefore the two concepts are closely related and can be used interchangeably without ambiguity. A formal definition of prime factorization is as below:

Any positive integer  $N$  has a unique prime power decomposition such that [Brent

1999],

$$N = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdots p_k^{\alpha_k} \quad (2.1)$$

$$(p_1 < p_2 < \cdots < p_k, \alpha_j > 0)$$

**Difficulty and Complexity** Although the above formula is true and easy to prove, it provides no efficient algorithm for computing the nontrivial factors. Not all the composites are difficult to factor (such as small integers), yet in general, factoring a composite integer is believed to be a hard problem. There is not any known algorithm to factor large composite numbers in polynomial time compared with the size of input [Crandall and Pomerance 2001]. On the other side, there is no rigorous proof on which the conjecture can be based. It is believed to be only because there is no such known efficient deterministic or randomized algorithm.

**Classification** Roughly speaking, we could classify the factoring algorithm into two types according to the performance characteristic [Lenstra 2000]. One could be named as general-purpose algorithm, which means the run time depends mainly on the size of  $N$ . The other type depends on not only the size of  $N$ , but also on the size of unknown factor  $p$  [Brent 1999]. The algorithms referred in this chapter are the second type.

**Importance** The relation between factoring and cryptography is the main reason why the integer factorization is studied and evaluated. Furthermore, the problem of computing integer factorization is a sort of sibling to the problem of discrete logarithm [Bach 1984], in that both problems are difficult and there are no efficient algorithms known now. Understanding the factorization problems would help study the discrete logarithm problems. As they have similar attributes, the algorithm from one problem may be adapted and modified to the other. Thus there is a potential possibility for one to refine the algorithms in factoring and put into discrete logarithm problems.

In the following sections, I would like to survey some of the most successful integer factorization algorithms. From now on, we assume that the candidate number for factoring is composite number, so that we can concentrate on the factoring problem but not primality test.

## 2.2 Trial Division

**Principle** Trial division is a straightforward method that derives naturally from the Fundamental Theorem of Arithmetic and Sieve of Erathostenes [Riesel 1994]. The smallest factor of a composite integer  $N$  could be found through trying division by all primes in succession, such as 2, 3, 5, 7, 11, 13,  $\cdots$ ,  $\sqrt{N}$ . Trial division is guaranteed to

find a factor of  $N$ , since it checks the smallest prime factor of  $N$  one by one. Thus, if the algorithm finds no factor, it is proof that  $n$  is prime.

**Example** Given a very simple example, say  $N = 221$ . Using trial division algorithm, we start from 2, 3, 5,  $\dots$ . Finally we could find 13 is a divisor of 221, we could also find another divisor which is 17 by  $\frac{221}{17}$ . Although it looks quite simple and fast here, it's inefficient to calculate large integers.

**Performance** If we don't take the overhead of obtain the prime numbers as candidate factors into account. In the worst case, trial division is a very laborious algorithm which will take  $\pi(\sqrt{N})$ <sup>1</sup> attempts. For a large factor, it's inefficient to use in practical.

On the other side, however, it is good for composite integers with small factors. Most numbers have a small factor and hence it is a fast method for such numbers. For a random number  $N$ , there is a 50% chance that 2 is a factor of  $n$ , and a 33% chance that 3 is a factor, and so on. It is also shown that 88% of all positive integers have a factor under 100, and that 92% have a factor below 1000.

**Parallel Trial Division** Speeding up the trial division by implementing several algorithms in parallel is a practical and effective way. In parallel computing, speedup is used to measure how much a parallel algorithm is faster than its sequential implementation. Informally, when the number of processors is doubled, there exists a linear speedup if the speedup is also doubled. It is straightforward to implement a parallel trial division algorithm. We could attempt the divisions by several parallel trials on different processors. There is a linear speedup provided the factor  $p$  is much bigger than number of parallel trials [Brent 1990].

**Improvements** In practice, the effort for obtaining the prime numbers as candidate factors will be taken into account. Either we could store a large table of primes up to some limit, or we could generate the number sequence including not only prime in run time. As we have to use the successive primes, there is a space-runtime trade in the trial division. Because it's quite time-consuming to run a sieve program in order to generate primes, it's suggested in [Riesel 1994] to generate a number sequence which includes every prime and some composites by a function such as,

$$2, 3, 6k \pm 1 \quad (k \in \mathbb{N}) \tag{2.2}$$

This sequence covers all the primes, but also introduces a few composites. The overhead will raise much if there are too many useless divisions. However, trial division by 2, 3, and all integers  $6k \pm 1$  is sufficiently efficient to be preferred [Riesel 1994].

<sup>1</sup> $\pi$  denotes the prime counting function, the number of primes less than itself.

**Implementation of Algorithm** A simple implementation of trial division using sequence  $(2, 3, 6k \pm 1)$  is described as below,

**Listing 2.1:** *Trial division by 2, 3,  $6k \pm 1$*  ( $k \in \mathbb{N}$ )

```

INPUT: Integer N.
OUTPUT: Smallest factor of N.

int trial (int N) {
    if (N % 2 == 0) return 2;
    if (N % 3 == 0) return 3;
    for (p = 1; p <= sqrt(N); p = 6p - 1) {
        if (N % p == 0) return p;
        if (N % (p + 2) == 0) return (p + 2);
    }
}

```

## 2.3 Pollard's rho Method for Factorization

**Introduction** In 1975 Pollard proposed a Monte Carlo method [Pollard 1975] for factoring integers named as rho method. It showed some very interesting ideas which are useful for future factoring algorithms. Pollard's rho method is based on three key ideas:

- Construct a sequence of integer  $x_i$  that is periodically recurrent modulo  $p$ , where  $p$  is the factor of an integer  $N$ .
- Find the periods of the sequence using Floyd's cycle-finding algorithm [Floyd 1967] [Knuth 1981].
- Identify the factor  $p$  based on the underlying theory of well-known "birthday" paradox in probability theory [Sayrafiezadeh 1994].

The three key ideas are discussed respectively in following paragraphs.

**"Birthday" Paradox** The probability of no repetition is discussed in the introduction Chapter. It states that the probability that all  $k$  events differ from each other is,

$$\bar{P}(k) = \frac{n}{n} \cdot \frac{n-1}{n} \cdots \frac{n-k+1}{n} = \frac{n(n-1) \cdots (n-k+1)}{n^k} \quad (2.3)$$

The assumption is that every event is fair and independent compared to others. The above formula is the key in "birthday" paradox problem. If a group of 23 or more people are chosen randomly, the probability that at least two of them have the same birthday exceeds 50%. That's the famous "birthday" paradox. The key assumption is that the group is chosen randomly so that it is considered to be a independent event. Hence we could use the above formula in probability theory. The probability that  $p$

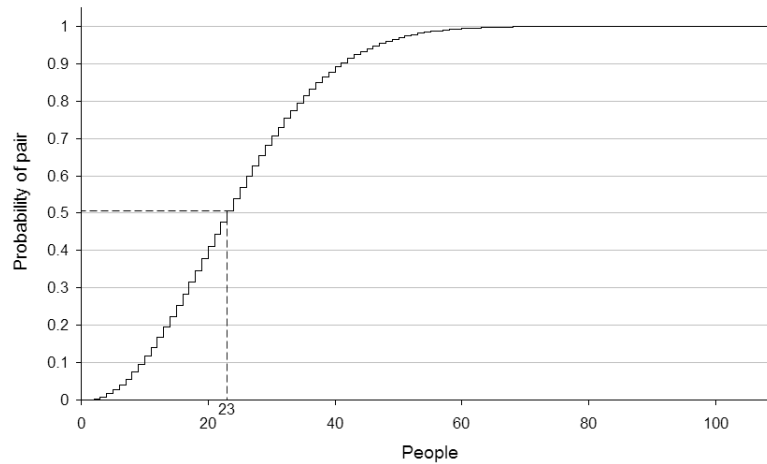
persons all have different birthdays is similar to events without repetition. In birthday problem the total number of events is 365 and  $p = 23$  people are considered first.

$$\bar{P}(23) = \frac{365}{365} \cdot \frac{365-1}{365} \cdot \frac{365-2}{365} \cdots \frac{365-23+1}{365} < 50\% \quad (2.4)$$

Therefore, the probability that people may have same birthday in a group of 23 persons is,

$$P(23) = 1 - \bar{P}(23) > 50\% \quad (2.5)$$

The key to understanding this problem is to think about the chances of no two people sharing the same birthday. As more people are chosen into a group, it becomes less and less likely that their birthday isn't already taken by someone else. Therefore, given a group of 23 randomly chosen people, the probability is more than 50% that at least two of them will have the same birthday. The probability will increase along with the increasing in number of people in a group. For 60 or more people, the probability is greater than 99%. The figure 2.1 shows the probability of at least two people sharing a birthday among a certain number of persons.



**Figure 2.1:** Probability at least two people share the same birthday in a group of people.

The "birthday" paradox can be applied in our case. Suppose that we have  $p$  as the total number of integers and if we choose  $k$  integers randomly out of  $p$ , the probability that at least two of them are congruent mod  $p$  is,

$$P(k) = 1 - \bar{P}(k) = 1 - \frac{n}{n} \cdot \frac{n-1}{n} \cdots \frac{n-k+1}{n} = \frac{n(n-1) \cdots (n-k+1)}{n^k} \quad (2.6)$$

The left side is approximately equal to,

$$\left(1 - \frac{k}{2p}\right)^{k-1} \approx e^{-k(k-1)/2p} \quad (2.7)$$

If we set the equation is 50%, then  $k \approx 1.177\sqrt{p}$  [Riesel 1994]. Thus the probability that two randomly chosen integers will be congruent mod  $p$  exceeds 50% after  $1.177\sqrt{p}$  numbers have been checked. Generally speaking, in a sequence of random integers mod  $p$ , a number is usually recurring after about  $c\sqrt{p}$  steps, with some small constant  $c$ . Such information could help us find the factor  $p$  of an integer  $N$ .

**Pseudorandom Generator** Since above numbers needed to be chosen randomly, a pseudorandom generator is considered to produce the group of numbers. Such sequence must contain information as  $p$ , the divisor of integer  $N$ . Pseudorandom numbers are often generated by an iteration of the form [Brent 1980],

$$x_{i+1} = f(x_i), \quad (2.8)$$

where  $f(x)$  is some easily computable function. Since the numbers of integers in a sequence is finite, there must be inputs which are same. Moreover the function is a deterministic mapping, the results of function are same by the same input. Therefore there exists  $n$  such that,

$$x_{\alpha+\lambda} = x_\lambda \quad (2.9)$$

The minimal such  $\alpha$  and  $\lambda$  are the length of aperiodic and periodic part of the sequence  $x_i$  respectively. For example,

$$x_{i+1} \equiv x_i^2 + 3 \pmod{11}, \quad x_0 = 2; \quad (2.10)$$

We successively obtain the following elements of the sequence,

$$x_0 = 2, \quad x_1 = 7, \quad x_2 = 8, \quad x_3 = 1, \quad x_4 = 2, \quad x_5 = 7, \quad x_6 = 8, \quad x_7 = 1, \quad (2.11)$$

After 4 elements the sequence is repeated. Note that this sequence has no aperiodic part since it repeats from the beginning. Because the randomness of the sequence is related with the concept of "Birthday" paradox. In general, the probability theory underlying it could only be applied on "good" numbers which are randomly enough. If we produce the sequence using a generator as  $x_{i+1} \equiv ax_i \pmod{p}$ , the generator does not produce numbers that are sufficiently random to give a short period of only  $c\sqrt{p}$  steps for  $x_i$  [Riesel 1994]; It's suggested a quadratic expression is appropriate for the generator [Brent 1980] [Pollard 1975]. Empirical results suggest that the generator is fairly random but this has not been proved so far.

$$x_{i+1} \equiv x_i^2 + a \pmod{p}, \quad a \neq 0, -2; \quad (2.12)$$

**Floyd's Cycle-finding Algorithm** We already have a periodic sequence  $x_i$  and we need to find the length  $\lambda$  of the period. For a periodic sequence, there exists an  $n > 0$  such that  $x_n = x_{2n}$  and the smallest  $n$  lies in the range  $a \leq n \leq a + \lambda$ . For instance we look into the periodic sequence  $1, 7, 4, 5, 6, 4, 5, 6, \dots$ . The aperiodic length is 2 and periodic part is 3. For  $n = 3$ , we have  $x_n = x_{2n}$ , and it satisfies  $a \leq n \leq a + \lambda$ . The

proof of this theorem is given below,

**Premise:** For a periodic sequence  $x_i$ , there exists an  $n > 0$  such that  $x_n = x_{2n}$  and the smallest  $n$  lies in the range  $a \leq n \leq a + \lambda$ .

**Proof:** For a periodic sequence  $x_i$ , assume that  $\alpha$  is the length of aperiodic part and  $\lambda$  is the length of periodic part. Then there exist  $n = k\lambda$ , where  $a \leq n \leq \alpha + \lambda$ . The least  $n$  could be  $\lambda$  and larger ones are multiple of  $\lambda$ . Hence we have  $x_n = x_{n+n}$  because  $n$  is a multiple of periodic part. Thus we have  $x_n = x_{2n}$ , and the smallest such  $n$  satisfies  $a \leq n \leq a + \lambda$ . We could also get the formula for compute such  $n$ ,

$$n = \lambda + \lfloor \frac{\alpha}{\lambda} \rfloor \tag{2.13}$$

The best way to understand the algorithm is to make a diagram of the sequence as figure 2.2. It looks like the Greek letter  $\rho$  as above. The sequence starts at the bottom of the tail using  $x_0$  and moves upward and clockwise around the loop. In the diagram, the length of aperiodic part is 3 and periodic part is 6. Hence the smallest  $n$  for  $x_n = x_{2n}$  happens at  $n = \lambda + \lfloor \frac{\alpha}{\lambda} \rfloor = 6$ .

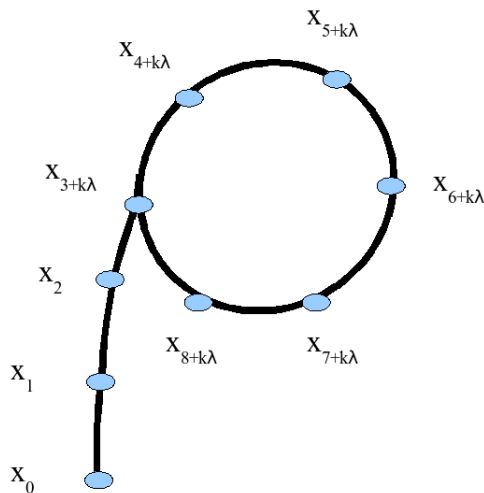


Figure 2.2:  $\rho$  Diagram

Armed with above, the period-finding algorithm may be described by the following code. Note the loop in below code is non-stop and hence not practical.

Listing 2.2: Floyd's cycle-finding algorithm

```

INPUT: Initial value for  $x_0$  and Pseudorandom function  $f(x)$ .
OUTPUT: Boolean value.

boolean cycling (int  $x_0$ ,  $f(x)$ ) {
    int  $x = x_0$ ;
    int  $y = f(x_0)$ ;
    for (;;) {
        if ( $x == y$ ) return TRUE;
        else {
             $x = f(x)$ ;
             $y = f(f(y))$ ;
        }
    }
}

```

The iteration result for each loop is tabulated below. There are two variables for loop and hence form two sequences. The subscript of sequence  $x$  increases by 1 for each loop. The subscript of sequence  $y$  increases by 2. Therefore, the second sequence loops faster than the first one.

Table 2.1: Iterations for Floyd-cycling Algorithm

Iteration	$x = f(x)$	$y = f(f(y))$	Comparison
0	$x_0$	$x_0$	Initialization
1	$x_1$	$x_2$	$x_1 \neq x_2$
2	$x_2$	$x_4$	$x_2 \neq x_4$
3	$x_3$	$x_6$	$x_3 \neq x_6$
4	$x_4$	$x_8$	$x_4 \neq x_8$
5	$x_5$	$x_{10}$	$x_5 \neq x_{10}$
...	...	...	...
n	$x_n$	$x_{2n}$	$x_n == x_{2n}$

There is no need to record the previous  $x_i$  values, and the Floyd-finding algorithm uses only a small constant amount of storage. The best running-time performance this algorithm is  $\alpha$  iterations and comparisons, since the slower sequence has at least to reach the beginning of periodic part. The worst-case performance is  $\alpha + \lambda$  comparisons.

**Principle of Pollard rho Method** The "birthday" paradox, pseudorandom generator and Floyd's cycle-finding are introduced in above sections. The Pollard rho method is based on Floyd's cycle-finding algorithm and on the birthday paradox that two numbers  $x$  and  $y$  are congruent modulo  $p$  with probability larger than 50% after  $1.177\sqrt{p}$  numbers have been randomly chosen. The suggested pseudorandom generator is as below,

$$x_{i+1} \equiv x_i^2 + a \pmod{p}, \quad a \neq 0, -2; \quad (2.14)$$

In practice, factor  $p$  is unknown. We use congruent modulo  $N$  instead of  $p$ .

$$x_{i+1} \equiv x_i^2 + a \pmod{N}, \quad a \neq 0, -2; \tag{2.15}$$

The information about  $p$  could be revealed from,

$$p = \gcd(x_{2i} - x_i, N) \tag{2.16}$$

The reason why using above formula is explained below. We assume that a integer  $N = pq$ . If two randomly chosen numbers are congruent modulo  $N$ , we have,  $x - y = kN = kpq$ , where  $k$  is an integer. Similarly, in our Floyd's cycle-finding algorithm, it becomes,

$$|x_{2i} - x_i| = kN = kpq \tag{2.17}$$

So  $|x_{2i} - x_i| = kpq$  is a multiple of  $p$ , although we don't know  $p$  yet. The value of  $p$  could be ultimately revealed if we compute  $\gcd(|x_{2i} - x_i|, N)$  since  $p$  divides both  $|x_{2i} - x_i|$  and  $N$ . When  $x_{2i} - x_i \neq mp$ , the result of  $\gcd$  is one; however the result turns out to be  $p$  or a multiple of  $p$  when  $x_{2i} \equiv x_i \pmod{p}$ .

**Example** Consider an example taken from [Riesel 1994], let  $N = 91643$  and  $a = 3$ . We set the generator as  $x_{i+1} = x_i^2 - 1$ ;

**Table 2.2:** Example of Pollard rho algorithm

Iteration	$x = f(x)$	$y = f(f(y))$	Comparison	$\gcd( x - y , N)$
0	$x_0 \equiv 3$	$x_0 \equiv 3$	Init	Init
1	$x_1 \equiv 8$	$x_2 \equiv 63$	$x_2 - x_1$	$\gcd(55, N) = 1$
2	$x_2 \equiv 63$	$x_4 \equiv 74070$	$x_4 - x_2$	$\gcd(74007, N) = 1$
3	$x_3 \equiv 3968$	$x_6 \equiv 35193$	$x_6 - x_3$	$\gcd(31225, N) = 1$
4	$x_4 \equiv 74070$	$x_8 \equiv 45368$	$x_8 - x_4$	$\gcd(62941, N) = 113$

**Notes** There are some notes in implementing the Pollard rho algorithm. First, the pseudorandom generator function needs to be easy to calculate. Since for each  $x_i$ , it will be calculated for twice, one in slower sequence and the other in faster sequence. Second, the number sequence should be constructed in the way that the period length is small. Furthermore, note that it's not appropriate to select the generator as a linear polynomial function such as  $x_{i+1} \equiv ax_i \pmod{N}$ , because the choice does not generate numbers that are random enough to produce a short period sequence which is easy to compute [Wagsaff 2003]. Also because it is based on a statistical idea, so we may have to change the choice for sequence generator for seeking the result. The  $\gcd$  operations are more expensive than modulo multiplication, it is better to reduce the computation cost by accumulating the product  $\prod_{k=1}^i (x_{2k} - x_k)$  and apply Euclidean algorithm occasionally. Furthermore, the original form of Euclidean algorithm is time consuming, some better algorithms such as Binary GCD, Sorenson's k-ary reduction

could be used in practice. [Brent and Zimmermann 2006] serves as a good reference in details about these algorithms. Finally, although no one has proved that any polynomials with degree bigger than 2 are random mapping, empirical results suggest that quadratic forms with  $a \neq 0, -2$  are good choices.

**Performance** As discussed above, if the numbers are genuine random, the expected complexity of Pollard's rho method is  $O(\sqrt{p})$ , where  $p$  is a factor of  $N$ . Hence the optimal bound of Pollard's algorithm is  $O(\sqrt{p})$ . Guy [?] conjectured that the worst case is  $O(\sqrt{p \log p})$ , which is slower by  $\sqrt{\log p}$  than the average case. Furthermore, because the numbers are produced by the pseudo-random generator, the running time depends on the randomness and hence different iteration function may take various steps of attempt. And they may be slower than that in truly random case. There is little rigorous proof on the performance issue because probability theory is appropriate only in random events, and therefore empirical investigations are good methods to explore the performance issue. Finally the Pollard rho method generally performs better in factoring composite integers with small factors.

**Parallel Pollard rho Method** In factoring problems, the parallel implementation of rho algorithm does not bring in a linear speedup. There are dependences between each of the element in the pseudorandom sequence. Moreover, even several different pseudorandom sequences could be used in parallel. There is still not a linear speedup. Assume we have  $P$  processors and conduct  $P$  sequences in parallel, the probability that the first  $k$  values in each sequence are distinct mod  $p$  is approximately  $e^{-k^2 P / (2p)}$ , hence the speed up is  $O(P^{1/2})$  [Brent 2000] [Brent 1990].

**Implementation of Algorithm** Below is an implementation of Pollard rho method for factorization problems,

Listing 2.3: Pollard rho method for factoring

```

INPUT: Integer  $N$ ; Pseudorandom function  $f(x) \pmod n$ ;
       Initial value  $x_0$  for iteration.
OUTPUT: Non-trivial factor  $p$  of  $N$  or 0 as failure.

int rho (int  $N$ ,  $f(x)$ , int  $x_0$ ) {
    int  $x = x_0$ ;
    int  $y = x_0$ ;
    int  $p = 1$ ;
    for (;;) {
        if ( $p == N$ ) return 0;
        if ( $p > 1 \ \&\& \ p < N$ ) return  $p$ ;
        else {
             $x = f(x)$ ;
             $y = f(f(y))$ ;
             $p = \text{gcd}(|x - y|, n)$ ;
        }
    }
}

```

## 2.4 Brent Algorithm

**Introduction** In [Brent 1980], a new cycle-finding algorithm has been proposed along with a faster refinement of the rho algorithm. The factoring is based on similar idea as Pollard's rho method, but uses a different method of cycle detection that is notably faster than Floyd's cycle-finding algorithm. It is this version that is commonly used in practice and about 25% faster than the original Pollard version,

**Principle** The Floyd's cycle-finding algorithm considers  $x_{2i} - x_i \pmod p$ . Instead, Brent's algorithm stop the computing for gcd after  $x_i = 2^k$  and subsequently compute  $x_j - x_{2^k} \pmod p$  with  $3 \cdot 2^{k-1} < j \leq 2^{k+1}$ . Then the iteration continues using the same rule. Thus in the calculation process, the values are computed as below,

Note that in the formula  $x_j - x_{2^k}$ , the subscripts of right side are always a power of 2. The subscripts of left side are decided by  $3 \cdot 2^{k-1} < j \leq 2^{k+1}$  and hence they are the intervals in the range  $(x_{i/2.3}, x_{2i}]$ , where  $i = 2^k$

**Example** Consider the same example as previous section, let  $N = 91643$  and  $0 = 3$ . We set the generator as  $x_{i+1} = x_i^2 - 1$ ;

The most remarkable success of the algorithm was the discovery of eighth Fermat number by Brent and Pollard. In [Brent and Pollard 1981], a modification of Brent's algorithm which is useful when the factors are known to lie in a certain congruence

**Table 2.3:** Brent's algorithm

elements	$x_j - x_{2^k}$
$x_1, x_2$	$x_2 - x_1$
$x_3, x_4$	$x_4 - x_2$
$x_5, x_6, x_7$	$x_7 - x_4$
$x_8$	$x_8 - x_4$
$x_9, x_{10}, x_{11}, x_{12}, x_{13}$	$x_{13} - x_8$
$x_{14}$	$x_{14} - x_8$
$x_{15}$	$x_{15} - x_8$
$x_{16}$	$x_{16} - x_8$
$x_{17}, \dots, x_{25}$	$x_{25} - x_{16}$
$x_{26}$	$x_{26} - x_{16}$
$x_{27}$	$x_{27} - x_{16}$
$\dots$	$\dots$

**Table 2.4:** Example of Brent's algorithm

Elements	Comparison	$\gcd( x - y , N)$
$x_1 \ x_2$	$x_2 - x_1$	$\gcd(55, N) = 1$
$x_2 \ x_4$	$x_4 - x_2$	$\gcd(74007, N) = 1$
$x_4 \ x_7$	$x_7 - x_4$	$\gcd(9672, N) = 1$
$x_4 \ x_8$	$x_8 - x_4$	$\gcd(62941, N) = 113$

class is used. The factorization of  $F_8$  is,

$$2^{2^8} + 1 = 1238926361552897 \times P_{62}^2 \quad (2.18)$$

**Performance** The algorithm is very fast for numbers with small factors. Compared to Pollard rho method, it is significantly 25% faster on average. The saving in Brent's algorithm comes from the aspect that there is no need to calculate the  $x_i$  for twice as in Floyd's finding algorithm.

**Implementation of Algorithm** Below is an implementation of Brent method for factoring,

---

<sup>2</sup> $P_{62}$  denotes a 62-digit prime number.

Listing 2.4: Brent method for factoring

```

INPUT: Integer  $N$ ; Pseudorandom function  $f(x) \pmod{n}$ ;
       Initial value  $x_0$  for iteration; Integer  $m > 0$ ;
OUTPUT: A non-trivial factor  $p$  of  $N$  or 0 as failure.

int rho (int  $N$ ,  $f(x)$ , int  $x_0$ ,  $m$ ) {
    int  $y = x_0$ ;
    int  $r = 1$ ;
    int  $p = 1$ ;
    int  $q = 1$ ;
    while ( $p == 1$ ) {
        int  $x = y$ ;
        for ( $i = 1$ ;  $i \leq r$ ;  $i++$ ) {
             $y = f(y)$ ;
        }
        int  $k = 0$ ;
        while ( $k < rp == 1$ ) {
            int  $ys = y$ ;
            for ( $i = 1$ ;  $\min(m, r - k); i++$ ) {
                 $y = f(y)$ ;
                 $q = q|x - y| \pmod{N}$ ;
            }
             $p = \text{gcd}(q, n)$ ;
             $k = k + m$ ;
        };
         $r = 2r$ ;
    }
    if ( $p == N$ ) {
        while ( $p == 1$ ) {
             $ys = f(ys)$ ;
             $p = \text{gcd}(x - ys, N)$ ;
        }
    }
    if ( $p == N$ ) return 0;
    else return  $p$ ;
}

```

## 2.5 Conclusion and Comparison

For trial division, in order to increase the performance, either we could store a large table of primes up to some limit, or we could generate the number sequence including not only prime in run time. Furthermore, speeding up the trial division by implementing several algorithms in parallel is a practical and effective way. Compared to trial division, the Pollard rho algorithm is a notable improvement with expected run time in  $O(\sqrt{p})$ . We could use rho method to find factors around twice in size comparing

trial division method. For Brent's algorithm [Brent 1980], it is significantly 25% faster than Pollard's original rho method on average. The speed-up is due to the reduction in computing the same element in the sequence from twice to once and hence need less run time.

## 2.6 Application in Cryptography: RSA

**Principle** The RSA algorithm, devised by [Rivest et al. 1978], is the most well-known public-key cryptographic technique. Its security is related with the intractability of the integer factorization problem as discussed previously. As RSA is a public-key cryptography, it involves a public and private key. The public key is distributed to people and used for encryption. The encrypted message can only be decrypted by the private key. In order to illustrate the principle, we assume that Alice is sending a secret message to Bob, just like what is described in our introduction Chapter. Therefore, Alice needs Bob's public key to encrypt and Bob need his private key to decrypt. The keys for Bob the RSA algorithm are generated the following way,

### Keys Generation

- Bob picks two large primes  $p$  and  $q$  and compute  $n = pq$ ;
- Bob compute the Euler's (totient) function  $\phi(n) = (p - 1)(q - 1)$ ;
- Bob choose a random integer  $e$  (not base of natural logarithm here) with  $1 < e < \phi(n)$  and  $\gcd(e, \phi(n)) = 1$ ;  $e$  is released as the the public key exponent.
- Bob find a  $d$  such that  $de \equiv 1 \pmod{\phi(n)}$ ;  $d$  is kept as the private key exponent.
- Finally, Bob's public key is  $(n, e)$  and his decryption key is  $d$ .

### Encryption and Decryption

- The function for Alice to encrypt the message is  $c = m^e \pmod{n}$ ;
- The decryption function for Bob to recover the message is  $m = c^d \pmod{n}$ ;

**Example** Here is an example of RSA encryption and decryption. We use small primes  $p$  and  $q$  for convenience purpose and encrypt the number "24" for an example.

### Keys Generation Example

- Bob picks two primes 17 and 19 and compute  $n = pq = 323$ ;
- Bob compute the Euler's (totient) function  $\phi(n) = (p - 1)(q - 1) = 288$ ;
- Bob choose a random integer  $e$  with  $1 < e < 288$  and  $\gcd(e, 288) = 1$ ; The integer  $e = 95$  is chosen and it is released as the public key exponent for encryption purpose.

- 
- Bob compute a  $d = 191$  such that  $191 \cdot 95 \equiv 1 \pmod{288}$ ; The integer  $d = 191$  is computed and it is kept secret as the private key exponent for decryption purpose.
  - Finally, Bob's public key is  $(323, 95)$  and his decryption key is 191.

### Encryption and Decryption example

- The function for Alice to encrypt the message is  $c = m^e \pmod{n} = m^{95} \pmod{323}$ ; In our example, the number "24" is our message, thus  $c = 24^{95} \pmod{323} = 294$ .
- The decryption function for Bob to recover the message is  $m = 294^{191} \pmod{323} = 24$ ; Finally, the origin message is recovered through the decryption function.

**Security** Suppose Bob's public key is  $(n, e)$  and his private key is  $d$ , we have  $ed \equiv 1 \pmod{\phi(n)}$ . If the large integer factorization is not a hard problem, that is, suppose one could factor  $n = pq$  efficiently. Then we could compute  $\phi(n) = (p-1)(q-1)$  and hence we could ultimately calculate the private key  $d$  through  $ed \equiv 1 \pmod{\phi(n)}$ . Therefore, integer factorization problem is related with the security of RSA cryptography. When the two prime numbers  $p$  and  $q$  are chosen, we must ensure they are large primes and not close to each other, because it's relatively easy to factor it using the Fermat factorization method. However, on the other hand, it's better to choose them in similar size since Pollard's method is good at factoring the numbers with small factors.

**Performance** As discussed above, the modular arithmetics are expensive operations and public-key cryptography is generally slower than symmetric one. RSA is much slower than DES, AES and other symmetric cryptosystems. In practice, the secret content of message is often encrypted by a symmetric algorithm, while the symmetric key is encrypted with RSA algorithm. In the communication process, both the RSA-encrypted symmetric key and the symmetrically-encrypted message are sent.



---

# The Discrete Logarithm Problem

---

Discrete logarithm problem is another problem believed to be hard without any known efficient algorithm [Odlyzko 1987]. While the inverse problem - discrete exponentiation could be computed efficiently. This asymmetry is analogous to the one between integer factorization and integer multiplication. So there is certain similarity between discrete logarithm and integer factorization. Algorithms for discrete logarithm problem are possibly inspired by similar algorithms for integer factorization. For example, the Pollard rho method in factoring has an analogue in discrete logarithm problems. Both the integer factorization and discrete logarithm problems are used in practice in the construction of trapdoor functions in public-key cryptosystems.

This chapter will cover some basic topics in discrete logarithm problems and concentrate on the Pollard rho method. We start with a definition and problem formulation, then discuss on several selected algorithms. As before, for each kind of algorithm, the introduction, principle, example, improvements and performance are analyzed respectively. In the end, the Diffie-Hellman key exchange cryptosystem serves as an application.

## 3.1 Introduction and Problem Formulation

**Definition** In general, let  $G$  be a finite multiplicative cyclic group with order  $n$ . Let  $g$  be a generator of  $G$ , then every element  $a$  in  $G$  can be written in the form  $a = g^x$  for some integer  $x$  [Odlyzko 2000]. The problem of the discrete logarithm for the group  $G$  may be stated as follows:

Given  $a \in G$  and  $g \in G$  known, find an integer  $x$  such that  $g^x = a$ .

For instance, given that  $3^x \equiv 13 \pmod{17}$ , the discrete logarithm problem is to solve the equation and find such  $x$ . Obviously there are infinitely many answers, we only seek for one typical answer here, which is  $x = 4$ .

By analogy to ordinary logarithms, we could write  $x = \log_g a$  without misunderstanding. The its properties are similar to ordinary logarithms. For example,  $\log_g ab = \log_g a + \log_g b$ .

**Complexity and Importance** No efficient algorithm for computing general discrete logarithms is known. Hence as integer factorization problem, computing discrete logarithm is believed to be difficult. Both asymmetries have been exploited to construct cryptosystems. In practice, many commonly used cryptography systems are based on the assumption that the discrete logarithm is extremely difficult to compute. Many cryptosystems may be broken if we could compute discrete logarithms quickly. One way to increase the difficulty of the discrete logarithm problem is to base the cryptosystem on a larger group.

### 3.2 Exhaustive Search

**Principle** The naive algorithm is to raise  $x$  to higher and higher powers until the desired  $h$  is found, that is, to compute and try  $g^1, g^2, g^3, \dots$  until the desired  $a$  is obtained. Similar as trial division, this is sometimes called trial multiplication.

**Performance** This method is not practical because it requires  $O(n)$  multiplications, where  $n$  is the size of the group  $G$ . It requires running time linear in the size of the group  $G$  and thus exponential in the number of digits in the size of the group. Thus it's inefficient if the order of group is large. Although no algorithm has been found in polynomial time, there are some better algorithms that we will describe in following sections.

### 3.3 Baby-step Giant-step Algorithm

**Principle** Shank's Baby-step Giant-step Algorithm [Shanks 1971] computes the discrete logarithm using a time-space trade-off idea. It could be considered as a modification of trial multiplication. Given the discrete logarithm problem,

$$h = g^x \pmod{p}; \quad (3.1)$$

where  $h, g$  and group  $G$  with order  $n$ . The baby-step giant-step algorithm is based on restructure by replacing  $x = im + j$ , where  $m = \lceil \sqrt{n} \rceil$  and  $j < m$ . So we have,

$$h = g^{im+j} \pmod{p} \quad (3.2)$$

Furthermore, the above formula is equal to

$$h \cdot ((g^{-m})^i) = g^j \pmod{p} \quad (3.3)$$

The algorithm first pre-computes  $g^j$  for several values of  $j$ , where  $0 \leq j < m$ . We build a table and sort the table by the their values,

Then we compute  $g^{-m}$  and let  $\gamma = h \cdot g^{-mi} \pmod{p}$  for  $i = 0, 1, 2, \dots$  until the result is equal to any value in above table, find the the superscript  $j$ . Finally we know the current  $i$  and  $j$  and hence we could compute  $x = im + j$ .

**Table 3.1:** Table contains  $g^j$ .

$j$	0	1	2	$\dots$	$m-1$
$g^j \pmod{p}$	$x_0$	$x_1$	$x_2$	$\dots$	$x_{m-1}$

**Example** As an example from [Menezes et al. 1997], assume that  $p = 113$  and generator is 3 for a multiplicative modulo group  $\mathbb{Z}_{113}^*$  of order  $n = 112$ . We want to compute  $\log_3 57$ . First,  $m = 11$ , we compute to build the table as below,

**Table 3.2:** Sorted table contains  $3^{113} \pmod{113}$ .

$j$	0	1	2	$\dots$	10
$3^j \pmod{113}$	1	3	8	$\dots$	63

The second step is to sort the table by  $3^{113}$ , we omit the result here. And the third table is built for  $\gamma = h \cdot g^{-mi} \pmod{p}$  for  $i = 0, 1, 2, \dots$ .

**Table 3.3:** Table contains  $57 \cdot 58^i \pmod{113}$ .

$i$	0	1	2	$\dots$	9
$57 \cdot 58^i \pmod{113}$	57	29	100	$\dots$	3

Finally, we find that  $\gamma = h \cdot g^{-mi} \pmod{p} = 3 = g^1$ , where  $j = 1$  and  $m = 9$ , and thus  $x = 9 \cdot 11 + 1 = 100$ .

**Implementation of Algorithm** Below is an implementation of Baby-step giant-step algorithm for discrete logarithm problems,

Listing 3.1: *Baby-step giant-step algorithm*

INPUT: Generator  $g$  of a cyclic group  $G$  and an element  $h$  in  $G$ , prime  $p$ .  
 OUTPUT: The discrete logarithm  $x$  satisfying  $g^x = h \pmod{p}$ .

```

int shanks (g, h, p) {
    int m = ⌈√(p)⌉;
    for (int j=0; j < m; j++) {
        int table[j][1] = j;
        int table[j][2] = gj;
    }
    sort(table [] []);
    for (int i=0; i < m; i++) {
        r = hg-mi;
        for (int t=0; t < j; t++) {
            if (r == table[t][2]) {
                return (im + j);
            }
        }
    }
}

```

**Performance** It runs in time  $O(\sqrt{n})$  and require space  $O(\sqrt{n})$ . As a comparison, trial multiplication takes  $O(n)$  steps with a space complexity of  $O(1)$ .

### 3.4 Pollard's Rho Method for DL problem

**Introduction** Pollard's rho algorithm [Pollard 1978] for the discrete logarithm problem is a randomized algorithm with the same idea as his rho algorithm for solving the integer factorization problem. The rho method works by first defining a pseudo-random sequence of elements from a group, then looking for the cycle to appear in the sequence. Finally the information about cycle will lead to the solution of discrete logarithm with a high probability.

**Principle** The ideas underlying Pollard's rho algorithm for factoring and logarithm are similar. We will not repeat such ideas and we will only introduce the differences. The main difference lies in the pseudorandom generator. Assume we are going to solve  $x = \log_g h$ . The group  $G$  is divided into three sets  $G_1, G_2, G_3$  and we define the functions to generate a sequence of elements as below,

$$x_{i+1} = f(x_i) = \begin{cases} h \times x_i \pmod{n} & : x_i \in G_1 \\ x_i^2 \pmod{n} & : x_i \in G_2 \\ g \times x_i \pmod{n} & : x_i \in G_3 \end{cases}$$

The above sequence of group elements in turn defines other two sequences of integers  $a_i$  and  $b_i$  who satisfy  $x_i = g^{a_i}h^{b_i}$ ,

$$a_{i+1} = \begin{cases} a_i & : x_i \in G_1 \\ 2a_i \pmod{n} & : x_i \in G_2 \\ a_i + 1 \pmod{n} & : x_i \in G_3 \end{cases}$$

$$b_{i+1} = \begin{cases} b_i + 1 \pmod{n} & : x_i \in G_1 \\ 2b_i \pmod{n} & : x_i \in G_2 \\ b_i & : x_i \in G_3 \end{cases}$$

Then we use the Floyd's cycle-finding algorithm to find two group elements such that  $x_i = x_{2i}$ , which means that,

$$g^{a_i}h^{b_i} \equiv g^{a_{2i}}h^{b_{2i}} \pmod{n} \quad (3.7)$$

As we can easily compute inverses modulo  $n$ , above formula will lead to,

$$g^{a_{2i}-a_i} \equiv h^{b_{2i}-b_i} \pmod{n} \quad (3.8)$$

Then we take logarithm to the base  $g$  of both sides, it becomes

$$(a_{2i} - a_i) \equiv \log_g h \cdot (b_{2i} - b_i) \pmod{n} \quad (3.9)$$

Let  $d = \gcd((b_{2i} - b_i), n - 1)$ . The above congruence equation must have a solution because  $g$  is a primitive root modulo  $n$  and  $n$  does not divide  $h$ . The above congruence has  $d$  solutions, one of which is our target.  $d$  is usually small so that we can try all the solutions to find such  $x = \log_g h$ .

**Example** Let  $n = 999959$ ,  $g = 7$ ,  $h = 3$ . Find  $x \equiv \log_g h \pmod{n}$ . By applying Floyd's cycle-finding algorithm, one could get that at  $i = 1174$ , there is

$$x_i \equiv x_{2i} \pmod{n} \quad (3.10)$$

Hence  $(b_{2i} - b_i) \equiv 310686 \pmod{n}$  and  $(a_{2i} - a_i) \equiv 764000 \pmod{n}$ , the congruence equation is,

$$310686 \cdot x \equiv 764000 \pmod{n} \quad (3.11)$$

By applying the extended Euclidean algorithm, we have,

$$2 = \gcd(310686, 999958) = 148845 \cdot 310686 - 46246 \cdot 999958 \quad (3.12)$$

We find that  $3 \equiv 7^{178162}$  and therefore  $x = \log_g h = 178162$ .

**Implementation of Algorithm** Below is an implementation of Pollard rho algorithm for discrete logarithm problems,

**Listing 3.2: Pollard rho algorithm for discrete logarithm**

INPUT: Generator  $g$  of a cyclic group  $G$  of prime order  $n$  and an element  $h$  in  $G$ ; Pseudo-random generator and initial values.  
 OUTPUT: Discrete logarithm  $x$  or 0 as failure.

```

int rho_dl (x0, a0, b0) {
    int x0 = x0;
    int a0 = a0;
    int b0 = b0;
    for (;;) {
        if (x_i == x_{2i}) {
            r = b_i - b_{2i} (mod n);
            if r = 0 return 0;
            else x = r^{(-1)}(a_{2i} - a_i) (mod n);
        }
        else {
            compute x_i, a_i, b_i and x_{2i}, a_{2i}, b_{2i}
                               from previous items;
        }
    }
}

```

**Performance and Improvements** The expected running time for Pollard's rho algorithm for logarithms problems is the same as Shank's Baby-step Giant-step algorithm. Under the presumption of randomness, the expected running time is  $O(\sqrt{n})$ . However, in contrast to Shank's algorithm, it's not necessary to store tables and search in the sequence to find a repeated group element. Because we could compute elements of the sequences  $x_i$  and  $x_{2i}$  until they equal to each other. Therefore, the space requirement is small.

In the analysis of its running time, the crucial assumption is made that a random walk in the underlying group is simulated. It shows that this assumption does not exactly hold for the walk originally suggested by Pollard [Teske 2001] [Teske 1998b]. Thus the performance is worse than expected in random case. Recently a rigorous estimate for the expected time has also been proposed by [Miller and Venkatesan 2006]. They show that the classical Pollard's rho algorithm for discrete logarithm produces a collision in expected time  $O(\sqrt{n}(\log n)^3)$ , which is close to the conjectured optimal bound  $O(\sqrt{n})$ . Furthermore, the iteration function or pseudo-random generator is used to define the random walk passing through the group members. Different iteration function would take various steps of attempts. It also has been found in [Teske 1998a] that the original Pollard method is slower than that in truly random case. Several better iteration functions have been proposed in [Teske 1998a] [Teske 2001]. Fi-

nally, as integer factorization problems, probability theory could be only applied on random events. Therefore empirical investigations are good methods to explore the performance issue.

### 3.5 Application in Cryptography: Diffie-Hellman

**Principle** Diffie-Hellman cryptography, which always refers to key agreement protocol, is a cryptographic algorithm that allows two users to exchange a secret key over an insecure medium without any prior secrets. It was developed by Diffie and Hellman in 1976 and published in the notable paper "New Directions in Cryptography" [Diffie and Hellman 1976]. Suppose that Alice and Bob want to agree on a shared secret key using the Diffie-Hellman key agreement protocol. They proceed as follows:

- First, Alice and Bob decide a prime  $p$  to use and choose a base such that  $1 < g < p$ .
- Then, Alice generates a random private value  $a$  and Bob generates a random private value  $b$ . Both of them are drawn from the set of integers .
- Alice computes  $g^a \pmod{p}$ , then sent it to Bob; On the other side, Bob computes  $g^b \pmod{p}$ , then sent it to Alice.
- Both of them could compute the shared secret key as  $k \equiv (g^a)^b \equiv (g^b)^a \equiv g^{ab} \pmod{p}$ .

**Example** According to above steps, an example is given below,

- Assume Alice and Bob have decided a number  $p = 97$  and base is 5.
- Alice chooses a random number 31 and Bob chooses 95.
- Then Alice computes  $5^{31} \equiv 7 \pmod{97}$ ; Bob has  $5^{95} \equiv 39 \pmod{97}$
- Finally, they have the shared key as  $k \equiv (g^a)^b \equiv (g^b)^a \equiv g^{ab} \equiv 14 \pmod{97}$ .

**Security** The protocol is considered secure against eavesdroppers if  $p$  and  $g$  are chosen properly. In order to obtain the secret key, the eavesdroppers need compute it from  $p, g, g^a, g^b$ . However, it is currently believed to be a hard problem, and this is often called the Diffie-Hellman assumption. Assume that one can compute  $a$  from  $g^a$ , the shared secret key will be revealed because we could compute  $k = (g^b)^a$  with  $a$  and  $g^b$  known. So an efficient algorithm to solve the discrete logarithm problem would make it easy to compute  $a$  or  $b$  and solve the Diffie-Hellman problem, making many types of public key cryptosystems insecure. Unlike our example above, much larger values of  $a, b$ , and  $p$  would be needed to make the system secure in practice.  $g$  need not be large at all, and in practice is usually either 2 or 5. Furthermore, if the random number generators for  $a, b$  are not random enough and can be predicted, then the cryptosystem is vulnerable.



---

# Conclusion

---

In this paper, we have presented and studied computer methods for integer factorization and discrete logarithm problems from a cryptographic point of view. In particular, we have seen how and why several popular public-key cryptosystems are related with such mathematical primitives. The interest in the subject was renewed and increased considerably during the last decades, because of the applications it found in public-key cryptography.

Furthermore this paper explains the mathematical primitives within the above field. The review looks into the principles of different algorithms; illustrates examples and instances; analyses and compares the performance; and applications of them in cryptosystems. Key algorithms such as Pollard's rho method for factoring and discrete logarithm, Brent's algorithm and Shank's algorithm are discussed and analysed in detail. The further work will also follow the same lines.

## 4.1 Research Plan

Using the techniques and algorithm discussed in the thesis, we hope to understand a number of things and build an foundation for further study and research. Generally speaking, the next stages research would concentrate on using the Pollards rho algorithm [Pollard 1975] [Pollard 1978] to solve integer factorisation, discrete logarithm problems.

First of all, for integer factorization, the conjectured optimal bound of Pollard's rho algorithms is  $O(\sqrt{n})$  under the genuine random assumption. However, the performance of original Pollard's rho algorithm is slower than that in the truly random case. In order to show how good the random walk assumption is, we could investigate the performance empirically. We could compute the the sum  $w$  of length of the period and non-periodic part of the walk experimentally for various primes  $p$ . Under the truly random assumption,  $w^2$  should have a certain exponential distribution. Thus we could show how good or how randomly the Pollard rho algorithm does by investigating the distribution. Some of work has already been done in the literature review phase. A cycle-finding algorithm along with simple analysis of performance is listed in Appendix C. More work will be done in next research phase.

Second, Guy [?] conjectured that the worst case differs from the average case only

by the slowly growing function  $\sqrt{\log p}$ , which is  $O(\sqrt{p \log p})$ . We can test this conjecture by computing  $\frac{w}{\sqrt{p \log p}}$  for all primes  $p$  up to some limit  $B$ . As a byproduct we can obtain a rigorous bound on the running time of the Pollard rho factoring algorithm on numbers whose least prime factor is at most  $B$ . This would be useful in applications where trial division is now used because it would allow trial division to be replaced by Pollard rho.

Furthermore, the running time of rho method depends on the randomness of the numbers in the sequence. Different iteration function may take various steps of attempts. Therefore, an investigation on difference pseudo-random generators will help us analyze the performance issue. There are some papers about this such as [Brent 1980] [Brent and Pollard 1981].

The discrete logarithm resembles integer factorization problem in some aspects and the key points of Pollard rho algorithm used in integer factorisation and discrete logarithm problems are similar. It is showed in [Teske 1998a] that the walk originally used by Pollard in discrete logarithm problems does not behave like a random walk. In addition, recently there is a rigorous proof [Miller and Venkatesan 2006] which shows that the classical Pollards rho algorithm is bounded by  $O(\sqrt{n}(\log n)^3)$ , which is near to the conjectured optimal bound. This could help us in the empirical analysis and we could also verify their results through experiments. The running time of rho method for discrete logarithm problems also depends on the randomness of the random walk. Several better iteration functions have been proposed in [Teske 1998a] [Teske 2001]. Thus we would like look into the performance of different iteration functions in discrete logarithm case too.

Finally, it would be attractive and exciting to develop some more efficient pseudo-random function. While due to the time limitation, this should be served as an optional target. The final outcome of this project is not only to study and research in the particular field, but also learn and develop a methodology in future research and work.

---

# Proposal for Future Research

---

## Empirical Investigation of the Pollard Rho Method

The Pollard rho algorithms for integer factorization and discrete log computation are of great interest because of their relevance to public key cryptography. The algorithms depend on making a pseudo-random walk in a finite abelian group. If the group is of order  $n$ , the algorithms have expected running time  $O(\sqrt{n})$ , using the "birthday paradox" idea, provided the pseudo-random walk behaves like a genuine random walk. Storage requirements are minimal and in some cases (e.g. the general discrete log problem for the group of an elliptic curve) the algorithms are the fastest known.

Very little has been proved about these algorithms because the walk performed is not truly random. The aim of this project is to investigate empirically how good the random walk assumption is. For example, in the case of integer factorisation, let  $w$  be the sum of the period and the length of the non-periodic part of the walk. Then  $w^2$  should have a certain exponential distribution and we can test this empirically by taking walks  $\pmod p$  for various primes  $p$ . If the exponential distribution is correct, then the expected value of  $w$  is  $E(w) = O(\sqrt{p})$ . Also, Guy conjectured that in the worst case  $w = O(\sqrt{p \log p})$ . In other words, he conjectured that the worst case differs from the average case only by the slowly growing function  $\sqrt{\log p}$ . We can test this conjecture by computing  $w$  for all primes  $p$  up to some limit  $B$  (say  $B = 10^6$ ). As a byproduct we can obtain a rigorous bound on the running time of the Pollard rho factoring algorithm on numbers whose least prime factor is at most  $B$ . This would be useful in applications where trial division is now used because it would allow trial division to be replaced by Pollard rho.



---

# Key Abstracts

---

The following articles are of key importance in both of current literature review process and future research plan.

- *Recent Progress and Prospects for Integer Factorisation Algorithms* [[Brent 2000](#)].

Abstract. The integer factorisation and discrete logarithm problems are of practical importance because of the widespread use of public key cryptosystems whose security depends on the presumed difficulty of solving these problems. This paper considers primarily the integer factorisation problem. In recent years the limits of the best integer factorisation algorithms have been extended greatly, due in part to Moore's law and in part to algorithmic improvements. It is now routine to factor 100-decimal digit numbers, and feasible to factor numbers of 155 decimal digits (512 bits). We outline several integer factorisation algorithms, consider their suitability for implementation on parallel machines, and give examples of their current capabilities. In particular, we consider the problem of parallel solution of the large, sparse linear systems which arise with the MPQS and NFS methods.

- *A Monte Carlo Method for Factorization* [[Pollard 1975](#)].

Abstract. We describe briefly a novel factorization method involving probabilistic ideas.

- *A Improved Monte Carlo Factorization Algorithm* [[Brent 1980](#)].

Abstract. Pollard's Monte Carlo factorization algorithm usually finds a factor of a composite integer  $N$  in  $O(N^{1/4})$  arithmetic operations. The algorithm is based on a cycle-finding algorithm of Floyd. We describe a cycle-finding algorithm which is about 36 percent faster than Floyd's (on the average), and apply it to give a Monte Carlo factorization algorithm which is similar to Pollard's but about 24 percentage faster.

- *Monte Carlo methods for index computation mod  $p$*  [[Pollard 1978](#)].

Abstract. We describe some novel methods to compute the index of any integer relative to a given primitive root of a prime  $p$ . Our first method avoids the use of stored tables and apparently requires  $O(N^{1/2})$  operations. Our second

algorithm, which may be regarded as a method of catching kangaroos, is applicable when the index is known to lie in a certain interval; it requires  $O(W^{1/4})$  operations for an interval of width  $W$ , but does not have complete certainty of success. It has several possible areas of application, including the factorization of integers.

- *Spectral Analysis of Pollard Rho Collisions* [[Miller and Venkatesan 2006](#)].

Abstract. We show that the classical Pollard rho algorithm for discrete logarithms produces a collision in expected time  $O(\sqrt{n}(\log n)^3)$ . This is the first non-trivial rigorous estimate for the collision probability for the unaltered Pollard rho graph, and is close to the conjectured optimal bound of  $O(\sqrt{n})$ . The result is derived by showing that the mixing time for the random walk on this graph is  $O((\log n)^3)$ ; without the squaring step in the Pollard rho algorithm, the mixing time would be exponential in  $\log n$ . The technique involves a spectral analysis of directed graphs, which captures the effect of the squaring step.

- *On random walks for Pollard's rho method* [[Teske 2001](#)].

Abstract. We consider Pollard's rho method for discrete logarithm computation. Usually, in the analysis of its running time the assumption is made that a random walk in the underlying group is simulated. We show that this assumption does not hold for the walk originally suggested by Pollard: its performance is worse than in the random case. We study alternative walks that can be efficiently applied to compute discrete logarithms. We introduce a class of walks that lead to the same performance as expected in the random case. We show that this holds for arbitrarily large prime group orders, thus making Pollard's rho method for prime group orders about 20% faster than before.

- *Speeding Up Pollard's Rho Method for Computing Discrete Logarithms* [[Teske 1998a](#)].

Abstract. In Pollard's rho method, an iterating function  $f$  is used to define a sequence  $(y_i)$  by  $y_{i+1} = f(y_i)$  for  $i = 0, 1, 2, \dots$ , with some starting value  $y_0$ . In this paper, we define and discuss new iterating functions for computing discrete logarithms with the rho method. We compare their performances in experiments with elliptic curve groups. Our experiments show that one of our newly defined functions is expected to reduce the number of steps by a factor of approximately 0.8, in comparison with Pollard's originally used function, and we show that this holds independently of the size of the group order. For group orders large enough such that the run time for precomputation can be neglected, this means a real-time speed-up of more than 1.2.

---

## Floyd's cycle-finding algorithm and results

---

As described in the conclusion section, some of work in future research plan has already been done for a better understanding of the plan. The Floyd's cycle-finding algorithm has been implemented along with a performance table,

**Listing C.1:** *Floyd's cycle-finding algorithm*

```

/* *****
*      rho . cc
*
*
*      Compile :
*          g++ -s -O4 -o rho rho . cc <OR>
*          g++ -o -rho rho . cc
*
*      Invoke :
*          ./rho
* ***** */

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define exit1 printf("\nNo such file, please try again.\n"); exit(1);
#define exit2 printf("\nCan't create file, please try again.\n"); exit(1);

main (int argc, char *argv[]) {

    // temporary data for each input in the loop for reading prime table.
    int input;

    // an array used for saving each prime. could only deal with 1000000 numbers of primes.
    int data[1000000];

```

```

// record length of array occupied by primes.
int length = 0;

// read filename.
char filename1[30];
printf("\\nPlease INPUT file including prime table :\\n");
scanf("%s", filename1);

// preparing for open data file - prime table.
FILE *fop;
fop = fopen(filename1, "rt");
if (fop == NULL ) {exit1;}

// read prime tables line by line.
while((fscanf(fop, "%d\\n", &input))!= EOF ) {
    // printf("data[length]=input;
    length = length + 1;
}

// close file
fclose(fop);

/* //display primes in the data array, for debugging purpose
for (int j = 0; j < length ; j ++ ) {
printf("%d\\n", data[j]);
}
*/

// record our result which is steps/sqrt(p) OR steps/sqrt(p*log(p));
float ratio[1000000];

// cycle finding
for (int i = 0; i < length; i++) {
    long long int x, y ;
    int p = data[i];
    int steps = 0;
    // initial value
    int x0 = 3;
    int a = 1;
    //float ratio[1000000];
    x = x0;
    y = x0;
    do {
        x = (x*x + a) % p;
        y = (y*y + a) % p;
        y = (y*y + a) % p;
        steps = steps + 1;
    }
    while (y != x);
}

```

---

```

        // ratio[i] = steps / (float) sqrt(p*log(p));
        ratio[i] = steps / sqrt(p);
        printf("%f\n", ratio[i]);
    }

    // preparing the file for saving result value
    char filename2[30];
    printf("\nOUTPUT file name. (DON'T overwrite exsiting file!) :\n");
    scanf("%s", filename2);
    fop = fopen(filename2, "wt");
    if (fop == NULL) {exit2;}

    for (int i = 0; i < length; i++ ) {
        fprintf(fop, "%f\n", ratio[i]);
    }
    fclose(fop);
}

```

An empirical investigation on various digits of number has been conducted and the result is as below, detailed analysis and further work will be done in next phase as described in research plan.

**Table C.1:** Performance results

Digits of prime numbers	$\frac{w}{\sqrt{p}}$	Number of examples computed
3	1.0846	143
4	1.0280	1061
5	1.0306	8363
6	1.0284	68906
7	1.0307	121503
8	1.0335	50000
Average	1.0393	249976



---

# Bibliography

---

- BACH, E. 1984. Discrete logarithms and factoring. Technical Report CSD-84-186, <http://digitalassets.lib.berkeley.edu/techreports/ucb/text/CSD-84-186.pdf>, Computer Science Division (EECS), University of California, Berkeley, California. (p.16)
- BRENT, R. P. 1980. An improved Monte Carlo factorization algorithm. *BIT Numerical Mathematics* 20, 2, 176–184. (pp.20, 25, 28, 40, 43)
- BRENT, R. P. 1990. Parallel algorithms for integer factorisation. In J. H. LOXTON Ed., *Number theory and cryptography, London Mathematical Society Lecture Note Series 154* (New York, NY, USA, 1990), pp. 26–37. Cambridge University Press. (pp.17, 24)
- BRENT, R. P. 1999. Some parallel algorithms for integer factorisation. In *EUROPAR: Parallel Processing, 5th International EURO-PAR Conference*, Volume 1685 (Berlin, 1999), pp. 1–22. Lecture Notes in Computer Science, Springer-Verlag. (pp.15, 16)
- BRENT, R. P. 2000. Recent progress and prospects for integer factorisation algorithms. In *COCOON: Annual International Conference on Computing and Combinatorics*, Volume 1858 (Berlin, 2000), pp. 3–22. Lecture Notes in Computer Science, Springer-Verlag. (pp.24, 43)
- BRENT, R. P. AND POLLARD, J. M. 1981. Factorization of the eighth Fermat number. *Mathematics of Computation* 36, 627–630. (pp.25, 40)
- BRENT, R. P. AND ZIMMERMANN, P. November 2006. *Modern Computer Arithmetic*. Version 0.1.1, <http://www.loria.fr/~zimmerma/mca/mca-0.1.1.pdf>. (p.24)
- COHEN, H. AND FREY, G. Eds. 2005. *Handbook of elliptic and hyperelliptic curve cryptography*. CRC Press. (p.1)
- CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms*. MIT Press/McGraw-Hill. (p.8)
- CRANDALL, R. AND POMERANCE, C. 2001. *Prime numbers: a computational perspective*. Springer-Verlag, New York. (p.16)
- DIFFIE, W. AND HELLMAN, M. E. 1976. New directions in cryptography. *IEEE Trans. Inform. Theory* IT-22, 644–654. (pp.2, 3, 9, 37)
- FLOYD, R. W. 1967. Non-deterministic algorithms. *Journal of the ACM* 14, 4, 636–644. (p.18)
- GAMAL, T. E. 1985. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inform. Theory* 31, 469–472. (p.9)

- GOLDREICH, O. 2001. *Foundations of Cryptography*, Volume Basic Tools. Cambridge University Press. (p.1)
- GOLDWASSER, S. AND BELLARE, M. 2001. Lecture notes on cryptography. *lecture notes in Massachusetts Institute of Technology*. (p.10)
- HOPCROFT, J. E., MOTWANI, R., AND ULLMAN, J. D. 2001. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley. (p.6)
- KAHN, D. 1967. *The Codebreakers*. Macmillan, New York. (pp.1, 2)
- KNUTH, D. E. 1981. *The Art of Computer Programming* (2nd ed.), Volume 2. Addison-Wesley, Reading, Mass. (p.18)
- LENSTRA, A. K. 2000. Integer factoring. *IJDCC: Designs, Codes and Cryptography* 19. (pp.15, 16)
- MENEZES, A. J., VAN OORSCHOT, P. C., AND VANSTONE, S. A. 1997. *Handbook of Applied Cryptography*. CRC Press. (pp. 6, 7, 8, 33)
- MILLER, S. D. AND VENKATESAN, R. 2006. Spectral analysis of pollard rho collisions. Algorithmic Number Theory Symposium (ANTS VII). (pp.36, 40, 44)
- MOLLIN, R. A. 2003. *RSA and Public-key Cryptography*. Chapman Hall CRC. (p.1)
- ODLYZKO, A. M. 1987. On the complexity of computing discrete logarithms and factoring integers. In T. M. COVER AND B. GOPINATH Eds., *Open Problems in Communication and Computation* (1987), pp. 113–116. Springer-Verlag. (p.31)
- ODLYZKO, A. M. 2000. Discrete logarithms: The past and the future. *Des. Codes Cryptography* 19, 2/3, 129–145. (pp.3, 31)
- POLLARD, J. M. 1975. A Monte Carlo method for factorization. *BIT Numerical Mathematics* 15, 331–334. (pp.3, 18, 20, 39, 43)
- POLLARD, J. M. 1978. Monte carlo methods for index computation mod p. *Mathematics of Computation* 32, 918–924. (pp.34, 39, 43)
- RIESEL, H. 1994. *Prime Numbers and Computer Methods for Factorization*. Progress in Mathematics. Birkhäuser. (pp.3, 16, 17, 19, 20, 23)
- RIVEST, R. L. 1990. Cryptography. In J. VAN LEEUWEN Ed., *Handbook of Theoretical Computer Science (Volume A: Algorithms and Complexity)*, Chapter 13, pp. 717–755. Elsevier and MIT Press. (p.1)
- RIVEST, R. L., SHAMIR, A., AND ADLEMAN, L. M. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM* 21, 2, 120–126. (pp.1, 2, 9, 28)
- ROSEN, K. H. 1993. *Elementary Number Theory and its Applications*, (Third Edition). Addison Wesley. (p.15)
- SAYRAFIEZADEH. 1994. The birthday problem revisited. *MATHMAG: Mathematics Magazine* 67. (p.18)
- SCHNEIER, B. 1994. *Applied Cryptography*. John Wiley & Sons. (p.9)

- 
- SERESS, Á. 1997. An introduction to computational group theory. *Notices of the AMS* 44, 6 (June/July), 671–679. (p. 8)
- SHANKS, D. 1971. Class number, a theory of factorization, and genera. In *Analytic Number Theory*, Volume 20 of *Proceedings of Symposia in Pure Mathematics*, pp. 415–440. American Mathematical Society. (p. 32)
- SIPSER, M. 2006. *Introduction to the Theory of Computation*. Second Edition. Pws Publishing. (p. 6)
- TESKE, E. 1998a. Speeding up pollard’s rho method for computing discrete logarithms. In *ANTS: 3rd International Algorithmic Number Theory Symposium (ANTS)* (1998). (pp. 36, 40, 44)
- TESKE, E. 1998b. Better random walks for pollard’s rho method. *Combinatorics and Optimization Research Report CORR 98-52*, University of Waterloo, Canada, [citeseer.ist.psu.edu/teske98better.htm](http://citeseer.ist.psu.edu/teske98better.htm). (p. 36)
- TESKE, E. 2001. On random walks for pollard’s rho method. *Mathematics of Computation* 70, 234, 809–825. (pp. 36, 40, 44)
- WAGSAFF, S. S. 2003. *Cryptanalysis of number theoretic ciphers*. CRC Press. (pp. 8, 23)
- WILLIAMS, H. C. AND SHALLIT, J. O. 1994. Factoring integers before computers (1994). pp. 481–531. In *Mathematics of Computation 1943-1993, Fifty Years of Computational Mathematics* (Ed. W. Gautschi). Providence, RI: American Math. Society. (p. 3)