



TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies ■

Distributed application for cryptanalysis of public–key cryptosystems

Master thesis

Study programme: N2612 – Electrical Engineering and Informatics

Study branch: 1802T007 – Information Technology

Author: **Bc. David Salač**

Supervisor: doc. RNDr. Miroslav Koucký, CSc.





TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií ■

Distribuovaná aplikace pro kryptoanalýzu asymetrických kryptosystémů

Diplomová práce

Studijní program: N2612 – Elektrotechnika a informatika

Studijní obor: 1802T007 – Informační technologie

Autor práce: **Bc. David Salač**

Vedoucí práce: doc. RNDr. Miroslav Koucký, CSc.



ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Bc. David Salač**
Osobní číslo: **M15000251**
Studijní program: **N2612 Elektrotechnika a informatika**
Studijní obor: **Informační technologie**
Název tématu: **Distribuovaná aplikace pro kryptoanalýzu asymetrických kryptosystémů**
Zadávací katedra: **Ústav nových technologií a aplikované informatiky**

Z á s a d y p r o v y p r a c o v á n í :

Primárním cílem je analyzovat potenciál distribuovaných systémů v kryptografických systémech s veřejným klíčem. Speciálně pak problému faktorizace velkých čísel a diskretního logaritmu. Praktická část je zaměřena na vytvoření distribuované aplikace využívající vybrané algoritmy pro řešení těchto problémů.

Pokyny pro vypracování:

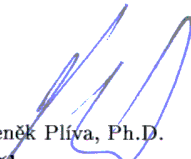
1. Popište vztah mezi diskretním logaritmem, faktorizací celých čísel a moderními kryptografickými algoritmy s veřejným klíčem
2. Popište efektivní algoritmy pro řešení těchto problémů. Navrhněte strategii implementace těchto algoritmů v distribuovaných systémech.
3. Vytvořte distribuovanou aplikaci pro dešifrování zpráv zašifrovaných pomocí některých popsaných algoritmů využívajících veřejný klíč.
4. Analyzujte potenciál této aplikace v reálných situacích.
5. Práce bude vypracována v anglickém jazyce.

Rozsah grafických prací: **dle potřeby**
Rozsah pracovní zprávy: **40 - 60 stran**
Forma zpracování diplomové práce: **tištěná/elektronická**
Seznam odborné literatury:

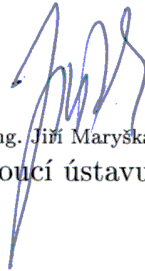
- [1] DELFS, Hans and Helmut KNEBL. Introduction to Cryptography: Principles and Applications. Third edition. Berlin: Springer, 2015.
[2] Y. YAN, Song, Moti YUNG and John RIEF. COMPUTATIONAL NUMBER THEORY AND MODERN CRYPTOGRAPHY. Higher Education Press: Singapore, 2013. ISBN 9781118188583.

Vedoucí diplomové práce: **doc. RNDr. Miroslav Koucký, CSc.**
Katedra aplikované matematiky

Datum zadání diplomové práce: **20. října 2016**
Termín odevzdání diplomové práce: **15. května 2017**


prof. Ing. Zdeněk Plíva, Ph.D.
děkan




prof. Dr. Ing. Jiří Maryška, CSc.
vedoucí ústavu

V Liberci dne 20. října 2016

Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

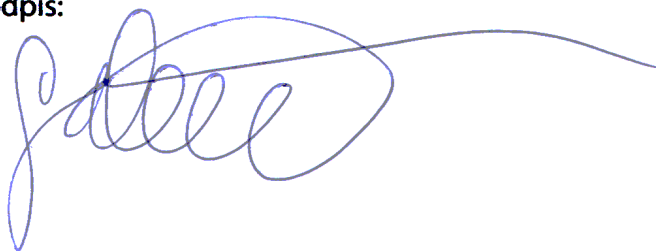
Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Současně čestně prohlašuji, že tištěná verze práce se shoduje s elektronickou verzí, vloženou do IS STAG.

Datum:

5. 5. 2017

Podpis:

A handwritten signature in blue ink, consisting of a large, stylized initial 'J' followed by several loops and a long horizontal stroke extending to the right.

Abstrakt

Práce zkoumá potenciál distribuované aplikace při kryptoanalýze kryptosystémů s veřejným klíčem. V práci je uvedeno vysvětlení vztahu mezi populárními kryptosystémy s veřejným klíčem, jako je šifra RSA, Diffie–Hellmanova výměna klíčů a šifra ElGamal, a řešení problému faktorizace celých čísel nebo diskrétního logaritmu. Existují numerické metody na řešení těchto problémů, neefektivnější z nich jsou popsány v této práci. V případě řešení problému diskrétního logaritmu, jsou zde popsány metody jako Shankův baby–step giant–step algoritmus nebo metoda index calculus. Pro účely řešení problému faktorizace celých čísel jsou zde popsány metody jako Pollardova Rho metoda, Dixonova metoda náhodných čtverců, kvadratické síto a obecné číselné síto. Téma práce bylo řešeno vytvořením distribuované aplikace. Jedná se o kompozici webových a desktopových aplikací. Webová aplikace představuje řídicí uzel distribuovaného systému. Pro uživatele je využitelná při správě úloh v systému. Poskytuje také základní funkcionalitu pro distribuci úloh podřízeným uzlům. Podřízené uzly jsou reprezentovány desktopovou aplikací. Jedná se o část, kde jsou implementovány popsané numerické metody pro řešení problému faktorizace čísel či diskrétního logaritmu. Nakonec je zde analýza použitelnosti distribuované aplikace pro reálné situace. Ta je složena z měření efektivity metod a jejich potenciálu v distribuované aplikaci. Ukázalo se, že distribuovaná aplikace představuje použitelný přístup pro řešení těchto typů problémů. Nicméně se také prokázalo, že pokud neudělá kryptograf žádnou chybu během implementace popsaných systémů, je téměř nemožné být úspěšný při kryptoanalýze těchto systémů. Práce analyzuje důležité téma související bezpečností dnes používaných kryptosystémů s veřejným klíčem. Toto téma je relevantní nejen pro vědecké účely, ale má také mnoho praktických konsekvencí.

Klíčová slova:

Kryptoanalýza kryptosystémů s veřejným klíčem, Distribuovaná aplikace, Problém faktorizace čísel, Problém diskrétního logaritmu, Numerické metody

Abstract

The thesis studies the potential of distributed application in cryptanalysis of public-key cryptosystems. There is an explanation of the relation among a popular public-key cryptosystems, such as RSA cypher, Diffie-Hellman key exchange and ElGamal cypher, and solving of integer factorization or discrete logarithm problem. There exists numerical methods for solving of these problems, the most effective ones are described in this thesis. In the case of solving discrete logarithm problems there are described method such as Shank's baby-step giant-step algorithm and Index calculus method. For the purpose of solving integer factorization problem there are described methods such as Pollard's rho method, Dixon's random square method, Quadratic Sieve and General number field sieve. The theme of the theses was solved by creating of distributed application. It is the composition of the web application and the desktop application. The web application represents master nod in the distributed system. It is usable for managing of task in the system for the users. It also provides basic functionality for distributing of the tasks to the slave nods. The slave nod is represented by the desktop application. It is the part where there are implemented described numerical methods for solving of integer factorization or discrete logarithm problem. Finally there is an analysis of usability of the distributed application for real situations. It consists of measurements of efficiency of methods and its potentials in distributed applications. It is shown that distributed application represents usable approach for solving of this kind of problems. However it is also shown that if cryptographers does not do any mistake during implementation of described cryptosystems, it is almost impossible to be successful with cryptanalysis of such system. The thesis analyzes important issue related with security of public-key cryptosystems of nowadays. This issue is relevant not only for scientific purposes but has also many practical consequences.

Key words:

Cryptoanalysis of public-key cryptosystems, Distributed application, Integer factorization problem, Discrete logarithm problem, Numerical methods

Acknowledgements

I would like to express my gratitude to my supervisor doc. Miroslav Koucký for the useful comments, remarks and engagement through the learning process of this master thesis.

Contents

List of abbreviations	11
Introduction	12
1 Public-key cryptography	14
1.1 RSA cryptosystem	14
1.2 Discrete logarithm	15
1.3 Integer factorization	15
1.4 Diffie-Hellman key exchange	15
1.5 ElGamal encryption	16
1.6 Summary	16
2 Integer factorization problem	17
2.1 Factoring by trial division	17
2.2 Pollard's rho	18
2.2.1 Realization in distributed application	18
2.3 Legendre's congruence	18
2.3.1 Realization in distributed application	19
2.4 Dixon's random squares method	19
2.4.1 Realization in distributed application	21
2.5 Quadratic Sieve	21
2.5.1 Tonelli-Shanks algorithm	23
2.5.2 Realization in distributed application	23
2.6 General number field sieve	24
2.6.1 Realization in distributed application	26
2.7 Summary	26
3 Discrete logarithm problem	28
3.1 Brute force algorithm	28
3.2 Baby-step giant-step algorithm	29
3.2.1 Realization in distributed application	30
3.3 Index calculus	30
3.4 Summary	31
4 Realization of distributed application	32
4.1 Web server	32
4.1.1 Realization of web application	33

4.1.2	Summary	37
4.2	Workstations	37
4.2.1	Receiving tasks and transmitting results	38
4.2.2	Processing of received tasks	38
4.2.3	Methods for integer factorization	40
4.2.4	Methods for solving of discrete logarithm	45
4.2.5	Summary	50
5	Using of application in real situation	51
5.1	Integer factorization problem	52
5.1.1	Real situations	53
5.2	Discrete logarithm problem	55
5.2.1	Real situations	56
5.3	Summary	58
	Conclusion	59
	Bibliography	62
	List of all appendixes	64

List of abbreviations

CPU	Central Processing Unit
DL	Discrete Logarithm
GCD	Greatest Common Divisor
GUI	Graphical User Interface
GNFS	General Number Field Sieve
MPQS	Multiple Polynomial Quadratic Sieve
PDO	PHP Data Objects
QS	Quadratic Sieve
RDBMS	Relational database management system
SSL	Secure Sockets Layer

Introduction

Public-key cryptography and its security is crucial for large scales of nowadays technologies, it is used almost everywhere. Not only banks during transactions but also each user connected to the internet network sooner or latter uses some kind of public-key cryptosystem. Not only because of this it is important to know how secure such cryptosystems are. This question is also the subject of many research studies of nowadays. This is also the motivation for writing of this thesis which tends to analyze potential of distributed system in this kind of problem.

In the beginning of thesis there is a description of relation among public-key cryptosystems and solving of discrete logarithm or integer factorization problem. There are only most popular cryptosystems described in this part, which include RSA cypher, Diffie–Hellman key exchange algorithm and ElGamal cypher. The security RSA cypher is related with solving of integer factorization problem, which could be described as finding prime factors p and q of number n such that $n = p \cdot q$. The rest of cryptosystems are based on solving of discrete logarithm problem, which could be described as finding integer x of congruence $g^x \equiv a \pmod{p}$ where g, a and p are known integers.

Solving of these kinds of problems is enormously time consuming process. By using of brute force method it is almost impossible to be successful in finding solution of relatively trivial task in some rational time – means for example less than one year. There exists some algorithms how to solve these kinds of problems that are much more effective than brute force – these are described in the following chapters. There is also the theoretical conception of its realization in distributed application. This part of theses is the research one. This is theoretical base of the following work and there is also a summary of relevant discoveries of past years. These chapters represents theoretical part of thesis, other chapters refer to the practical part.

The practical part of thesis consists chiefly of realization of distributed application for cryptanalysis of public-key cryptosystems and at last measuring of its efficiency and approximating of the time that is necessary for solving of real situations. The conception of distributed system is that there is one master nod (web application) and many nods for solving of inserted tasks managed by master nod. It means that the application consists of two parts, the first one is a web application and the other one is desktop application. The web application provides interface for users to standard operations with tasks, such as its inserting, modifying and removing. The task represents cryptographic problem that should be solved – this is represented by finding of encrypted message or shared key. The purpose of the desktop application is finding solution of task by using of implemented cryptanalytic

methods and sending found results back to the server where they are accessible via web application.

The last chapter tends to analyze a potential of distributed application in real situations. Primary aim of this chapter is the approximation of time that would be necessary to solve real cryptographic tasks. This is achieved by measuring of the set of values and using of regression analysis. The most important relation that is measured and analyzed is between the size of key and time that is necessary for cryptanalysis of this key. This relation is usable for computation of time which is necessary for solving of real tasks. The standard key size of such tasks is well known or could be easily found. There is also discussion about some progress in this area of past years.

1 Public–key cryptography

There are introduced the most popular public–key ciphers and protocols of nowadays in this chapter including RSA, Diffie–Hellman key exchange protocol and ElGamal. The integer factorization and discrete logarithm problems are also introduced as same as the relation among this issues.

Public–key cryptography is based on simple idea that there are two different keys without any trivial relation (mean in mathematical sense). One of this keys e is used for encrypting the message and the other one d is used for decrypting the message m .

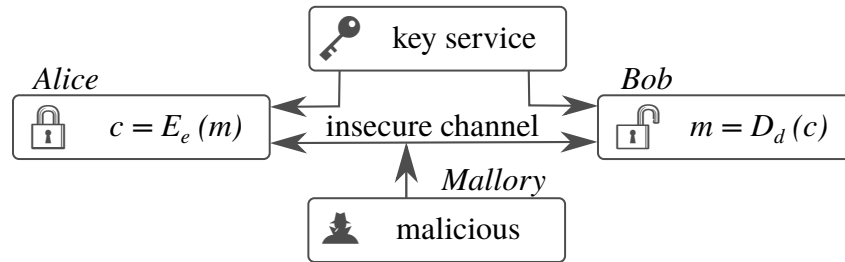


Figure 1.1: Standard encryption schema

1.1 RSA cryptosystem

Let $p, q \in \mathbb{P}$ (where \mathbb{P} denotes to set of all prime numbers) be large prime numbers (usually about 1000 bits lengths), $n = p \cdot q$, $\varphi(n) = (p - 1)(q - 1)$ is Euler totient function of n , and $e \in \mathbb{N}$, $2 \leq e < \varphi(n) \wedge \text{GCD}(e, \varphi(n)) = 1$, most often $e = 65537$ [1, p. 58]. Then compute d as the result of congruence $de \equiv 1 \pmod{\varphi(n)}$.

Plain–text m ($m \in \mathbb{N}, m < n$) is encrypted using public key set (e, n) :

$$c = m^e \pmod{n} \tag{1.1}$$

and ciphertext c is decrypted using private key set (d, n) :

$$m = c^d \pmod{n} \tag{1.2}$$

1.2 Discrete logarithm

Let g is a primitive root mod n , g is primitive root mod n if and only if:

$$g \in [0, n) \cap \mathbb{Z} \wedge \text{GCD}(g, n) = 1 \wedge \forall p \in \mathbb{P}, p \mid \varphi(n) \Rightarrow g^{\frac{\varphi(n)}{p}} \not\equiv 1 \pmod{n}$$

If $\text{gcd}(a, n) = 1$, then the smallest positive integer k such that $a \equiv g^k \pmod{n}$ is called index of a to the base g modulo n and is denoted by $\text{ind}_{g,n} a$ or simply by $\text{ind}_g a$ [2, p. 137].

The function $\text{ind}_g a$ is called discrete logarithm (or just index) and is sometimes denoted by $\log_g a$.

In the case of RSA n, e are public keys and c is ciphertext, then:

$$d = \text{ind}_e m$$

if m is chosen it is possible to get d :

$$d = \text{ind}_{(m^e \pmod{n})} m \quad (1.3)$$

It is obvious that it would be easy to break RSA if it exists effective algorithm to compute discrete logarithm but no such algorithm has been found yet [2, p. 137].

1.3 Integer factorization

Suppose following task, for given integer $n \in \mathbb{N}$ find all p_i, α_i where $p_i \in \mathbb{P}, \alpha_i \in \mathbb{N}, p_i < p_{i+1}, i = 1, 2, \dots, N$ where [2, p. 191]:

$$n = \prod_{i=1}^N p_i^{\alpha_i}$$

It is also evident that if it exists simple way for finding this factorization it would be easy to break RSA cipher because if someone is able to find p, q can easily find private key d by solving relation $de \equiv 1 \pmod{\varphi(n)}$ [1, p. 58].

1.4 Diffie–Hellman key exchange

Diffie–Hellman is protocol for exchanging key value that is later used in symmetric–key algorithm. Let $p \in \mathbb{P}$ is a large prime and g is a primitive root mod p . Numbers p and g are publicly known. To establish secret share key Alice and Bob execute following protocol [1, p. 111]:

1. Alice choose randomly $a \in (1, p - 2] \cap \mathbb{N}$, then sets $c := g^a \pmod{p}$ and sends c to Bob.
2. Bob choose randomly $b \in (1, p - 2] \cap \mathbb{N}$ then sets $d := g^b$ and sends d to Alice.
3. Alice compute shared key $k = d^a \pmod{p} = (g^b)^a \pmod{p}$.
4. Bob compute shared key $k = c^b \pmod{p} = (g^a)^b \pmod{p}$.

Security level of Diffie–Hellman key exchange algorithm is based on difficulty of solving discrete logarithm problem. Private numbers a and b can be found as $a = \text{ind}_{g,p} c$ and $b = \text{ind}_{g,p} d$.

1.5 ElGamal encryption

ElGamal is based on discrete logarithm problem. ElGamal cryptosystem has no problem with integer factorization unlike RSA has.

The recipient of message Bob proceeds follows [1, p. 77]:

1. Bob chooses large prime $p \in \mathbb{P}$ such that $p - 1$ has a big prime factor and primitive root $g \bmod p$.
2. Bob randomly chooses an integer $x \in (1, p - 2] \cap \mathbb{Z}$. The triple (p, g, x) is Bob's secret key.
3. Bob compute $y \equiv g^x \pmod{p}$. Bob's public key is triple (p, g, y) . Only x is kept in secret. The y value is sometimes denoted to be h .

Generation of p such that $p - 1$ has large prime factor is done by algorithm: $q \in \mathbb{P}$ is large prime number and Bob is looking for primes of form $2kq + 1$ [1, p. 77].

Alice encrypts message to Bob by using public key triple (p, g, y) using follows [1, p. 78]:

1. Alice has message $m \in \mathbb{Z}_p$ to Bob.
2. Alice chooses an integer $k \in (1, p - 2] \cap \mathbb{N}$ at random.
3. Alice computes $(c_1, c_2) \equiv (g^k, y^k m) \pmod{p}$ and send vector (c_1, c_2) to Bob (vector c represents encrypted message).

Bob decrypts incoming message with private key triple (p, g, x) [1, p. 78]. Since of $y^k \equiv_p (g^x)^k \equiv_p (g^k)^x \equiv_p c_1^x$. To obtain plaintext m :

$$m \equiv_p (c_1^x)^{-1} c_2 \equiv_p (y^k)^{-1} y^k m \equiv_p m$$

Relation of discrete logarithm problem and ElGamal is evident:

$$x = \text{ind}_{g,p} y \tag{1.4}$$

If someone obtain private key x , decryption of messages is simple task.

1.6 Summary

There are introduced only the most popular public-key cryptosystems in this chapter. Various extensions of these systems exist (for example elliptic curve cryptography and so on).

It is evident that the security of public-key cryptosystems stays on the pillar of difficulty of solving discrete logarithm problem or integer factorization problem (especially RSA). It is recently common for every popular public-key cryptosystem of nowadays.

There are of course a lot of rules to generate parameters of introduced cryptosystems. If some developer ignores them, it could be easy to break concrete system without using complex techniques for solving discrete logarithm (or integer factorization) problem and this is also the most popular way for attacking these systems.

2 Integer factorization problem

Various methods for factoring integers exists. But apart from special cases none of them are effective enough to be expressed in polynomial level of complexity. Most effective modern integer factorization methods discussed below are Pollard's rho algorithm, Quadratic Sieve (QS) and currently fastest method (General / Special) Number Field Sieve (GNFS).

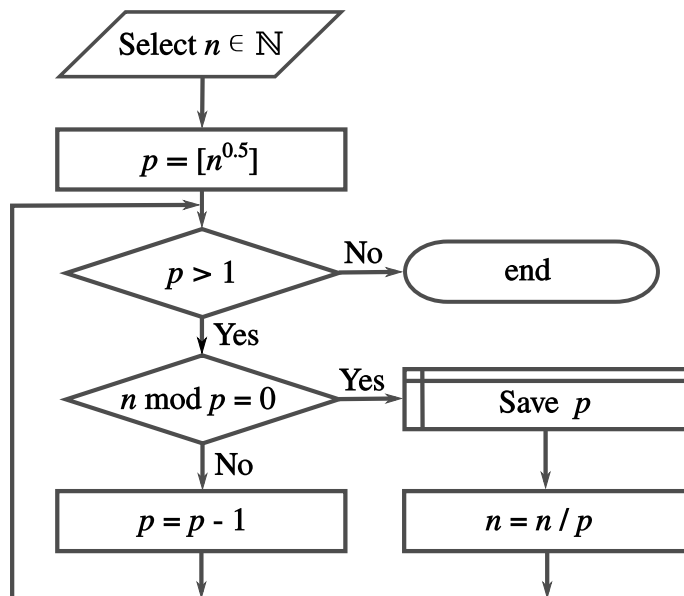


Figure 2.1: Trial division flowchart

2.1 Factoring by trial division

Trial division algorithm is the most straightforward algorithm of all as it is shown in figure 2.1. This algorithm has complexity $\mathcal{O}(2^{\lfloor \frac{m}{2} \rfloor})$ where m is a bit length of input (size of input, not specific number to be factoring).

Factoring by trial division is useful only for smaller integers (roughly smaller than 10^9) because of algorithm's simplicity (it is simple task to do in almost every programming language).

Algorithm of trial division has a few benefits (except of it's simplicity) in practice – it is not probabilistic algorithm (if it finds a solution it is surely nontrivial factor), and it is relatively fast algorithm for integers of size less than 20 bits

2.2 Pollard's rho

Pollard's rho (or ρ) method was proposed by John M. Pollard in 1975 as very efficient Monte Carlo method [1, p. 198].

Method uses an iteration of the form:

$$\begin{aligned} x_0 &= \text{random}(0, n-1) \\ x_i &\equiv f(x_{i-1}) \pmod{n}, \quad i = 1, 2, 3\dots \end{aligned} \tag{2.1}$$

where x_0 is random starting value, n is integer to be factored and $f \in \mathbb{Z}[x]$ is a polynomial with integer coefficient, usually $f(x) = x^2 \pm a$ with $a \neq 0, -2$ [1, p. 198].

Led d is a nontrivial divisor of n (d is small compared to n), since there are d congruent classes \pmod{d} (relatively few). There will probably exist integers x_j and x_i in the same congruent classes \pmod{d} but different classes \pmod{n} [1, p. 198]:

$$\begin{aligned} x_i &\equiv x_j \pmod{d} \\ x_i &\not\equiv x_j \pmod{n} \end{aligned} \tag{2.2}$$

since $d \mid (x_i - x_j)$ and $n \nmid (x_i - x_j)$, it follows that $\text{GCD}(x_i - x_j, n)$ is a nontrivial factor of n . The value of d is typically unknown but can be most likely found by counting $\text{GCD}(x_i - x_j, n)$ (where x_j is earlier $x_i \Rightarrow j < i$) until a nontrivial divider occurs.

Estimation of time complexity of Pollard's rho method:

$$\mathcal{O}(2^{\frac{m}{4}})$$

where m represents size of input (in bits).

2.2.1 Realization in distributed application

It exists many improvements of Pollard's rho algorithm, such as Brent–Pollard's ρ method or Pollard's $p-1$ method. But none of them is relevant in practical applications.

Pollard's rho method is useful for factoring numbers with less than 30 bits. Method is useless for larger numbers and so it is less significant in distributed application. But method could be useful for solving some subtasks of more complex methods such as searching of factor base in Dixon's method which is a part of all other effective methods.

2.3 Legendre's congruence

Every subsequent method for integer factorization is based on simple observation based on Legendre's congruence introduced by Adrien-Marie Legendre (1752 – 1833). If we want to factorize number n composed of factors $p, q \in \mathbb{P}$ there exists congruences of form [3, p. 234]

$$x^2 \equiv y^2 \pmod{n} \wedge x \not\equiv y \pmod{n} \tag{2.3}$$

where $x, y \in [2, n) \cap \mathbb{N}$ are some integers.

Congruence (2.3) could be written as:

$$x^2 - y^2 \equiv (x - y)(x + y) \equiv 0 \pmod{n} \Leftrightarrow pq \mid (x - y)(x + y)$$

which is the same as:

$$p \mid (x - y)(x + y) \wedge q \mid (x - y)(x + y)$$

because of condition $x \not\equiv y \pmod{n}$ there are only three options for each condition, consider situation if condition $p \mid (x - y)(x + y)$ is chosen:

1. $p \mid (x - y) \wedge p \nmid (x + y) \Rightarrow \text{GCD}(x - y, n) = p \wedge \text{GCD}(x + y, n) = q$
2. $p \nmid (x - y) \wedge p \mid (x + y) \Rightarrow \text{GCD}(x - y, n) = q \wedge \text{GCD}(x + y, n) = p$
3. $p \mid (x - y) \wedge p \mid (x + y)$ where $\text{GCD}(x \pm y, n)$ equals 1 or n

two options leads to nontrivial divisor of n , only third does not. It implies there is probability equals to $\frac{2}{3}$ to obtain nontrivial divisor of n for random x, y matches to congruence (2.3).

2.3.1 Realization in distributed application

Principle of factorization based on Legendre's is behind all modern method. Algorithms try to find integers x and y matches (2.3) in different ways. This effort is obvious in Dixon's method and Quadratic Sieve described bellow.

Application for integer factorization needs only effective algorithm for finding $\text{GCD}(x \pm y, n)$ values – Euclidean algorithm is usable for this purpose. The processing of Euclidean algorithm is not a task for parallel computing.

2.4 Dixon's random squares method

Dixon's factorization method was proposed by John D. Dixon in 1981 [5]. It was the first usable algorithm based on Legendre's congruence (2.3).

Algorithm consists of following steps ([5], modified):

1. Find Factor Base F which consists of prime numbers that occurs most frequently in prime factorization of $(x^2 \pmod{n})$ for random numbers $x \in (\sqrt{n}, n) \cap \mathbb{N}$.
2. Find at least $|F| + 1$ ($|F|$ is cardinality of set F) numbers $x_i \in (\sqrt{n}, n) \cap \mathbb{N}$ such that $(x_i^2 \pmod{n})$ is smooth over a set F .
3. Construct matrix \tilde{E} that represents an exponents of each prime number of F in prime factorization of $(x_i^2 \pmod{n})$ – find exponent α_k such that $(x_i \pmod{n}) = \prod_{p \in F} p^{\alpha_k}, k \in [1, |F|] \cap \mathbb{N}$. In this situation it is obvious that null space of matrix \tilde{E} is not empty set (matrix has more columns than rows).

Aim of these algorithm is to find numbers that fits the conditions defined in congruences (2.3). Only the parity of exponents is relevant to follow this purpose. Major task of algorithm is computation of null space of matrix E defined as $E = \widetilde{E} \pmod{2}$.

$$\widetilde{E} = \begin{bmatrix} x_1 & x_2 & \cdots & x_{|F|} & x_{|F|+1} \\ 7 & 2 & \cdots & 0 & 1 \\ 0 & 3 & \cdots & 0 & 2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & \cdots & 2 & 1 \\ 1 & 0 & \cdots & 0 & 3 \end{bmatrix} \begin{matrix} p_1 \\ p_2 \\ \vdots \\ p_{|F|-1} \\ p_{|F|} \end{matrix}$$

4. In case Dixon's algorithm succeeds in finding null space of E (over \mathbb{Z}_2 field) Legendre's congruence is constructed in following way):

$$n = (\beta_1, \beta_2, \dots, \beta_{|F|+1})^T \in \text{nullspace}(E), \quad \beta_i \in \{0, 1\}, i \in [1, |F| + 1] \cap \mathbb{N}$$

$$x = \prod_{i=1}^{|F|+1} x_i^{\beta_i}$$

$$y = \prod_{i=1}^{|F|} p_i^{(\sum_{j=1}^{|F|+1} \alpha_j \cdot \beta_j) \cdot 0.5}$$

In this situation we have congruence that fits (2.3).

5. Compute $\text{GCD}(x \pm y, n)$ and if it equals to nontrivial factor of n , algorithm is over (probability of this situation equals to $\frac{2}{3}$). Otherwise ($\text{GCD}(x \pm y, n)$ equals 1 or n) go back to step one (probability equals to $\frac{1}{3}$).

Time complexity (number of operations which will be required) of Dixon's method is estimated to [5]

$$\mathcal{O}\left(\exp\left(\alpha \cdot (\ln n \cdot \ln(\ln n))^{\frac{1}{2}}\right)\right), \alpha \geq 2\sqrt{2}$$

The idea of Dixon's algorithm is introduced in step number 3 of algorithm. If there is an aim to find the number with even exponents, it is possible to transform this problem via matrix of exponents to standard linear algebra task. First two steps of algorithm are called Data collection, third and fourth steps are both called Matrix processing.

There exist many improvements in each step of Dixon's algorithm such as Quadratic Sieve method or General Number Field Sieve (both described bellow).

2.4.1 Realization in distributed application

The easiest way for realization of the first step of algorithm (finding of the factor base) is to choose some random $x \in (\sqrt{n}, n) \cap \mathbb{N}$ and try to factorize result of $y(x) = (x^2 \bmod n)$. In practice the factorization itself could be done by brute force method or Pollard's rho method – if the $y(x)$ has larger prime factor (and selected method is not effective) it should be just put away. Number of random selections and ideal factor base size depends on many factors (especially on input size). In the second step algorithm only computes exponents of each primes in Factor Base F for some random x and at the same time checks if $y(x)$ is smooth over a set F . Data collection part of algorithm (step one and two) could easily become an object of parallelization (hypothetical thread of algorithm just takes random x and computes it's prime factorization with less complex algorithm and saves the result).

Data processing part of algorithm (consists of step three and four) takes much less time to process than Data collecting part. Algorithm almost could not become an object of parallelization. Naive method to obtain null space of matrix is Gaussian elimination which is also less effective one – it could be useful just for demonstration of algorithm logic. The complexity of Gaussian elimination is $\mathcal{O}(|F|^3)$ [6]. There are some optimization of this algorithm over finite field (for example [6]).

Matrix E is significantly sparse (it means that almost all of its element equals zero). Null space of matrix E could be most effectively found by block Lanczos algorithm which is not parallel or by partially parallel algorithm called block Wiedemann algorithm with time complexity $\mathcal{O}(|F|(w + |F| \ln(|F|) \ln \ln(|F|)))$ where w is approximately the number of operations required to multiply the matrix to a vector [7, p. 8]. Both algorithms are useful for sparse linear systems over finite field. But in practice both of them are relatively difficult to implement.

2.5 Quadratic Sieve

Quadratic Sieve is the improvement of Dixon's algorithm. Originally Quadratic Sieve algorithm was proposed by Carl Pomerance in 1982 [2, p. 214]. Quadratic Sieve is the fastest algorithm for factoring numbers up to 110 digit [7]. Before the method will be introduced, few definitions are necessary.

Quadratic residue modulo p : Let a is any integer and n a natural number, and suppose that $\text{GCD}(a, n) = 1$. Then a is called a quadratic residue modulo n if the congruence:

$$x^2 \equiv a \pmod{n}$$

is soluble. Otherwise, it is called a quadratic non-residue modulo n [2, p. 114].

Legendre symbol definition: Let p be an odd prime and a an integer. Suppose that $\text{GCD}(a, p) = 1$. Then the *Legendre symbol* $\left(\frac{a}{p}\right)$, is defined by [2, p. 117]:

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \text{if } a \text{ is quadratic residue modulo } p \\ -1 & \text{if } a \text{ is quadratic non-residue modulo } p \\ 0 & \text{if } p \mid a \end{cases} \quad (2.4)$$

Quadratic Sieve algorithm consists of following steps [7]:

1. Algorithm works with factor base F consists of small prime numbers. The size of the factor base depends on the size of input n . Each $f \in F$ has upper bound B that depends on the current task.
2. Unlike Dixon's algorithm that works with random x^2 , Quadratic Sieve method works with:

$$Q(x) = (x + \lfloor \sqrt{n} \rfloor)^2 - n \quad (2.5)$$

such that $Q(x) \equiv (x + \lfloor \sqrt{n} \rfloor)^2 \pmod{n}$. Algorithm tries to find congruence:

$$\prod_{i=1}^r (Q(x_{j_i}) + n) \equiv \prod_{i=1}^r \left((x_{j_i} + \lfloor \sqrt{n} \rfloor)^2 \pmod{n} \right) \pmod{n}$$

that has a form of Legendre's congruence (2.3).

3. Sieving is solved on interval $x \in [-M, M] \cap \mathbb{Z}$ (sieving interval). It is much more effective to compute value of x from factor base than using just a random generator for integers. Consider situation $p \in F \wedge p \mid Q(x)$ together with (2.5) it implies:

$$(x + \lfloor \sqrt{n} \rfloor)^2 \equiv n \pmod{p} \quad (2.6)$$

which means that n is quadratic residue mod p and also:

$$\forall p \in F : \left(\frac{n}{p} \right) = 1$$

4. Congruence (2.6) could be written in form (for some $s \in \mathbb{N}$ and $x \in \mathbb{Z}_p$):

$$Q(x) = s^2 - n \equiv 0 \pmod{p}$$

this congruence could be solved by Tonelli-Shanks algorithm which returns two solutions s_{1p} and $s_{2p} = p - s_{1p}$.

Value of $Q(x_i)$ is computed using s_{1p_i} or s_{2p_i} using $x_i = s_{1p_i} + kp$ or $x_i = s_{2p_i} + kp$ for $k \in \mathbb{Z}$ such that x_i is in sieving interval.

The rest of algorithm is the same as Dixon's method – especially constructing and processing matrix of exponent's parity.

Quadratic Sieve algorithm works with estimated time complexity (the number of steps which are needed to find the solution for given n) [2, p. 217]:

$$\mathcal{O} \left(\exp \left((1 + o(1)) \sqrt{\ln n \ln \ln n} \right) \right)$$

There are also many improvements of Quadratic sieve algorithm such as Multiple Polynomial Quadratic Sieve. Some of them has better time complexity in some specific situations.

2.5.1 Tonelli–Shanks algorithm

Algorithm is procedure to solve congruence of form

$$x^2 \equiv n \pmod{p}$$

where p is defined prime number greater than 2 and n is quadratic residue mod p , which is equivalent to condition $\left(\frac{n}{p}\right) = 1$, Legendre's symbol for prime p could be found by [10]:

$$\left(\frac{n}{p}\right) = n^{\frac{p-1}{2}} \pmod{p}$$

Algorithm consists of the following steps (from [8] and [9]):

1. Find integers Q and S such that $p - 1 = 2^S Q$ where Q is odd number. If $S = 1$ solution equals:

$$x \equiv \pm n^{\frac{p+1}{4}} \pmod{p}$$

2. Find quadratic non-residue W of p (it means that $\left(\frac{W}{p}\right) = -1$) and compute

$$V \equiv W^Q \pmod{p}$$

3. Find multiplicative inverse n' of $n \pmod{p}$
4. Compute

$$R \equiv n^{\frac{Q+1}{2}} \pmod{p}$$

and find the smallest integer $i \geq 0$ that satisfy:

$$(R^2 n')^{2^i} \equiv 1 \pmod{p}$$

5. If $i = 0$ algorithm stops and $x = R$, if it does not compute R' :

$$R' \equiv R V^{2^{S-i-1}} \pmod{p}$$

and go to step one with argument $R = R'$.

2.5.2 Realization in distributed application

Data collection part of quadratic sieve method could be divided to specific subtasks. The easier way how to do it is to split the sieving interval to subintervals and distribute it to each thread or process. Tonelli–Shanks algorithm itself is strictly sequential algorithm.

There is also a problem with the memory requirements of algorithm – practical size of factor base for integers of length above 400 bits is about hundreds of thousands primes. It means that algorithm has to save matrix of size at least $(10^5)^2$ bits and also the array of real exponents values of the same size. For example, factoring of 426 bits challenge integer (called RSA-129) in 1994 uses a factor base of 524 339 prime numbers [7, p. 9]. It could be useful to use some relatively low-level programming languages such as C where it is simple to work with each bits separately.

2.6 General number field sieve

GNFS was first proposed by John Pollard in 1988. It is the fastest known algorithm for factorization of large integers. There are also a few necessary definitions needed before method could be presented.

Algebraic number definition: A complex number $\alpha \in \mathbb{C}$ is an algebraic number if it is a root of some polynomial [2, p. 220]:

$$a_0x^k + a_1x^{k-1} + \dots + a_k = 0, \quad a_0, a_1, a_2, \dots, a_k \in \mathbb{Q} \quad (2.7)$$

Algebraic integer definition: A complex number $\beta \in \mathbb{C}$ is an algebraic integer if it is a root of some monic polynomial [2, p. 220]:

$$x^k + b_1x^{k-1} + \dots + b_k = 0, \quad b_0 = 1, b_1, b_2, \dots, b_k \in \mathbb{Z} \quad (2.8)$$

Theorem: The set of algebraic numbers forms a field, and the set of algebraic integers forms a ring [2, p. 221].

Let $\theta \in \mathbb{C}$ is the complex root of polynomial (2.8). Then the set $\mathbb{Z}[\theta]$:

$$\mathbb{Z}[\theta] = \left\{ \sum_{i=0}^k \theta^i b_i, \quad b_0, \dots, b_k \in \mathbb{Z} \right\} \quad (2.9)$$

forms a ring called polynomial ring.

Lemma: Let polynomial $f(x)$ has a form (2.8) and m is an integer such as $f(m) \equiv 0 \pmod{n}$ and α is a complex root of $f(x)$. There exists a unique (surjective) mapping $\Phi : \mathbb{Z}[\alpha] \rightarrow \mathbb{Z}_n$ satisfying (2.8) [2, p. 222]:

1. $\Phi(ab) = \Phi(a)\Phi(b), \forall a, b \in \mathbb{Z}[\alpha]$
2. $\Phi(a + b) = \Phi(a) + \Phi(b), \forall a, b \in \mathbb{Z}[\alpha]$
3. $\Phi(za) = z\Phi(a), \forall a \in \mathbb{Z}[\alpha], z \in \mathbb{Z}$
4. $\Phi(1) = 1$
5. $\Phi(\alpha) = m \pmod{n}$

Let $n \in \mathbb{N}$ is positive odd integer to be factorized. The GNFS algorithm consists of following steps [2, p. 223 – 224]:

1. The first step consists of selecting of two irreducible polynomials $f(x)$ and $g(x)$ with small integers coefficients for which exists integer m such that:

$$f(m) \equiv g(m) \equiv 0 \pmod{n} \quad (2.10)$$

And let α be a complex root of $f(x)$ and β of $g(x)$.

2. Algorithm searching for pairs (a, b) (where $\text{GCD}(a, b) = 1$) with smoothed integral norms over a chosen factor base F . Integral norm is defined:

$$N(a - b\alpha) = b^{\deg(f)} f(a/b) \quad N(a - b\beta) = b^{\deg(g)} g(a/b) \quad (2.11)$$

3. Find a set $U = \{a_i, b_i\}$ of indexes such that:

$$\prod_U (a_i - b_i\alpha) \quad \prod_U (a_i - b_i\beta) \quad (2.12)$$

both product are square of the product of prime ideals.

4. Let (2.12) defines set S . This will be used for finding of an algebraic numbers $\alpha' \in \mathbb{Q}(\alpha)$ and $\beta' \in \mathbb{Q}(\beta)$ such that:

$$(\alpha')^2 = \prod_U (a_i - b_i\alpha) \quad (\beta')^2 = \prod_U (a_i - b_i\beta) \quad (2.13)$$

Define $\Phi_\alpha : \mathbb{Q}(\alpha) \rightarrow \mathbb{Z}_n$ and $\Phi_\beta : \mathbb{Q}(\beta) \rightarrow \mathbb{Z}_n$ via $\Phi_\alpha(\alpha) = \Phi_\beta(\beta) = m$ where $m \in \mathbb{Z}$ is root of g and f . Then:

$$\begin{aligned} x^2 &\equiv \Phi_\alpha(\alpha')\Phi_\alpha(\alpha') \equiv \Phi_\alpha((\alpha')^2) \equiv \Phi_\alpha\left(\prod_U (a_i - b_i\alpha)\right) \equiv \prod_U \Phi_\alpha(a_i - b_i\alpha) \equiv \\ &\equiv \prod_U (a_i - b_i m) \equiv \Phi_\beta(\beta')^2 \equiv y^2 \pmod{n} \end{aligned}$$

This expression has a form of Legendre's congruence (2.3).

General number field sieve has time complexity (based on heuristic assumptions) for integer n [2, p. 229]:

$$\mathcal{O}\left(\exp\left((c + o(1))\sqrt[3]{\ln n \cdot (\ln \ln n)^2}\right)\right)$$

with $c \approx \sqrt[3]{\frac{64}{9}}$.

In practice there are two similar variants of number field sieve method. The first is GNFS and the second is called Special number field sieve which is usable just for one value of input integer n (it works with slightly better complexity).

GNFS algorithm is the best for factoring integers of size hundreds (or thousands) of bits – in this case it is the fastest known algorithm of all. The greatest disadvantage of GNFS is its complexity itself which causes many problems in practical realization.

2.6.1 Realization in distributed application

Almost everything is the same as it was in Dixon's algorithm or Quadratic Sieve method – it especially works with a large sparse matrix. The biggest difference between QS a GNFS algorithm consists in a difference of sieving process that decreases the complexity of algorithm. There is a possibility of distribution sieving process by the splitting of interval for b values between processors.

There exists a lot of academical papers about improving of each step of algorithm. For example, one of the latest academical works interested in possibilities of integrating parallel block Wiedemann algorithm to GNFS [11] for efficiency of work with sparse matrix. Another way of its achievement is presented in paper [12] where authors tends to use Montgomery variation of block Lanczos method which is implemented in Linbox math library.

Although there are lots of papers about improving complexity of each algorithm's step, the leading way how to increase efficiency of algorithm is still in distributing problem to as many independent nodes as possible. Many improvements of algorithm was motivated by RSA Factoring Challenge – where there was successfully factorized integers of size 768 bits at 2009.

2.7 Summary

This chapter tends to describe only the most popular methods for integer factorization. There are other effective methods (in some cases especially useful for special purposes) such as Lenstra's Elliptic Curve Method or Continued Fraction method. Each were superseded by Quadratic Sieve that is currently the fastest method for factoring of integers in range 20-110 bits. For integers of size less than 20 bits it is especially useful to use non probabilistic brute force algorithm. Pollard rho method is usable for factoring of integers with a lot of small factors. Fastest algorithm of year 2017 is still General Number Field Sieve that is about 30 years old.

There are some ways how to improve time complexity of each algorithm step. Parameters of each method such as optimal factor base length is set up heuristically. Finding of usable factor base and sieving process is potential task for parallel computing. Especially important are methods for working with sparse matrix over \mathbb{Z}_2 (for computing of matrix null space), such as Lanczos or block Wiedemann algorithm which is parallel. There exists also straightforward way how to find null space of matrix using Gaussian elimination. This process resulting in finding numbers of Legendre's congruence that could resulting in finding of nontrivial divisor of input with probability equals of $2/3$.

Memory requirements of each algorithm depends exponentially on size of input. It is possible to work on bit level in case of large input because of working on \mathbb{Z}_2 field in crucial part of algorithm. This is especially suitable task for relatively low level programming languages such as C / C++.

Despite significant research in this branch, most of successful attacks against RSA (and other cryptosystems based on integer factorization problem) are based on mistakes that developers have done during practical realization of system. It should

be mentioned that there is an algorithm with polynomial complexity solving integer factorization problem called Shor's algorithm, but it is designed just for quantum computers that currently does not exist.

3 Discrete logarithm problem

In the contrary to integer factorization problem there are no methods of solving discrete logarithm problem with comparable complexity. There are some methods that rely on errors in realization of special cryptographic application. The only practical usable method suitable for general purpose is called Baby-step giant-step discussed below.

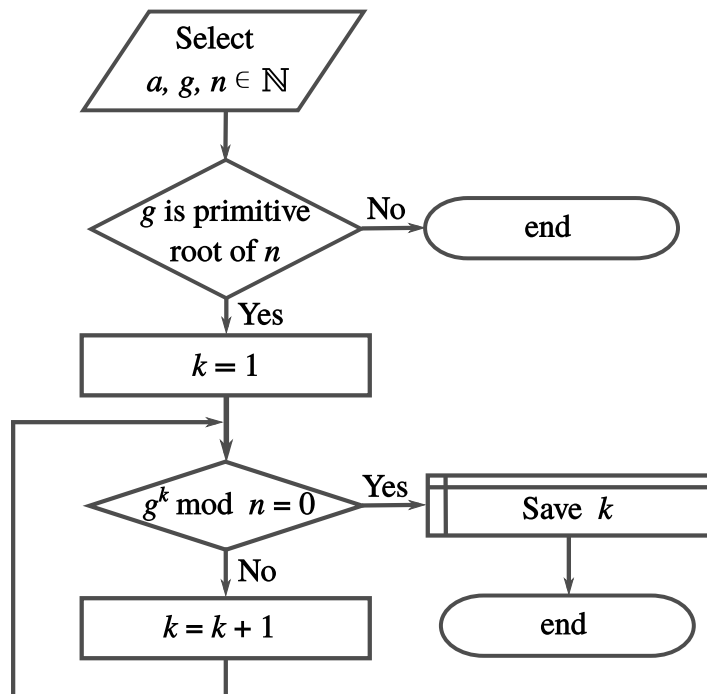


Figure 3.1: Discrete logarithm – brute force solver flowchart

3.1 Brute force algorithm

The most straightforward algorithm is solving discrete logarithm problem using brute force as it is shown in figure 1.1. The complexity of this method is:

$$\mathcal{O}(2^N)$$

where N represents length of n in bits.

The only advantage of this method is its simplicity and the fact that it could be relatively easily written in most of programming languages. In fact, first condition of algorithm could be skipped in some occasions. Because there could exist $k \in \mathbb{N}$ solving equation $g^k \equiv a \pmod{n}$ also in situation where g is not primitive root modulo n . For example $12^x \equiv 24 \pmod{30}$ has solution $x = 2$ and obviously 12 is not primitive root mod 30 (just because of $\text{GCD}(12, 30) \neq 1$).

3.2 Baby-step giant-step algorithm

The method is meet-in-the-middle algorithm described in 1968. Algorithm presuppose situation that equation

$$g^k \equiv a \pmod{n} \tag{3.1}$$

for $g, a, n, k \in \mathbb{N}$ has at least one solution.

Algorithm consists of following steps [2, p. 237–238]:

1. Compute $s = \lfloor n \rfloor$.
2. Compute pairs:

$$S = \{(ag^i, i), \quad i \in [0, s] \cap \mathbb{Z}\}$$

and save them in list. This step is called baby-step.

3. Compute the second sequence T of the following pairs:

$$T = \{(g^{is}, i), \quad i \in [1, s] \cap \mathbb{Z}\}$$

This step is called a giant step.

4. Search lists S and T for match $ag^r = g^{ts}$ where ag^r in S and g^{ts} in T . If algorithm find such numbers than $k = ts - r$ solving congruence (3.1).

Algorithm above is also called Shanks' Baby-step giant step method. Time complexity of algorithm is:

$$\mathcal{O}(\exp(\sqrt{n} \log n))$$

Algorithm is a type of Square Root Method. There exist another similar algorithms such as ρ Method or λ Method (also called Kangaroo method) [2, p. 239]. Baby-step giant-step advantage is relatively straightforward way of realization in almost all programming language. Algorithm efficiency is comparable with other algorithms usable for solving of discrete logarithm problem. There also exists improvement of this method called Silver–Pohlig–Hellman algorithm which could find solution in $\sqrt{q_k}$ steps ($q_k = \max\{q \in \mathbb{P}, q \mid (p-1)\}$).

3.2.1 Realization in distributed application

Baby-step part of algorithm could be distributed to many processors, where each could operate with assigned interval of i values. The rest of algorithm could not use any advantages of parallel computing.

Another issue is memory requirements of an algorithm which fully depends on the length of input. If the algorithm should not be only probabilistic it is necessary to initialize array of \sqrt{n} values. That is possible only for relatively small values of n in the context of cryptography. For larger integers algorithm has to be probabilistic which means algorithm could fail.

Probabilistic version of algorithm generates only random baby-step pairs in set S to be compared with integers of set T , the rest of algorithm is the same.

3.3 Index calculus

Index calculus was proposed in 1979 by Adleman. Algorithm itself is a wide range of methods including Continued fraction method, QS, GNFS.

Algorithm consists of following steps [2, p. 255]:

1. Precomputation

- (a) For some $m \in \mathbb{N}$ create factor base F consisting of the first m prime numbers.
- (b) Choose randomly $e \in \mathbb{N}, e < p - 1$ and compute $g^e \bmod n$. If $g^e \bmod n$ is smooth over F then:

$$e \equiv \sum_{j=1}^m e_j \text{ind}_g p_j \pmod{p-1} \quad (3.2)$$

- (c) Repeat this process until algorithm has at least m congruences of form (3.2).

2. Compute $k \equiv \text{ind}_g a \pmod{n}$:

- (a) For each e in (3.2) determine the value of $\text{ind}_g p_j, j = 1, 2, \dots, m$ by solving m modular linear equations.
- (b) Choose randomly exponent $r \leq p - 2$ and compute $ag^r \bmod n$
- (c) Factor $ag^r \bmod p$ over F , if it is impossible go to step (2b) if not:

$$\text{ind}_g a \equiv -r \sum_{j=1}^m r_j \text{ind}_g p_j \pmod{p-1} \quad (3.3)$$

Index calculus algorithm has time complexity estimated:

$$\mathcal{O}\left(\exp\left(c\sqrt{\log n \log \log n}\right)\right)$$

Although index calculus has theoretically the best time complexity, it is not simple to realized it in practice. There are a few exceptions, such as [14] that has shown that this could be usable way of solving discrete logarithm problem but it is still a topic of academical discussion rather than practice. The problem of the algorithm is especially its complexity (for example working with matrices over \mathbb{Z}_n for some composed number n is difficult) and hardware requirements.

There also exist some improvements of index calculus algorithm such as Gordon's number field sieve and others [2, p. 258]. But despite of complexity decrease, any improvements nor index calculus itself is not widely used way for solving discrete logarithm problem.

3.4 Summary

Solving of discrete logarithm problem is done by much less effective algorithm than as it is in integer factoring problem. The most effective algorithm for DL problem is called index calculus which is the composite of many methods of number theory but is not widely used. The only algorithms that are usable in distributed application are Shrank's baby-step giant-step method and Silver-Pohlig-Hellman method.

In this chapter there is no mention about the problem of elliptic curve cryptography that is based on DL problem. There are some methods specialized for cryptoanalysing of this problem. One of the most effective algorithms in this branch is called Xedni calculus [2, p. 253].

Most of the reported successful attacks against DL based cryptosystems were based on mistakes of developers of such systems. There exists algorithm with polynomial complexity for quantum computers that was introduced by Peter Shor (together with algorithm for solving of integer factorization problem). The existence of algorithm with polynomial level of complexity for Turing machine has not been proven nor disproven yet (which is common fact for both discussed problems).

4 Realization of distributed application

The conception of the application is that there would be one web-server (master nod) where users (apps operators) would be submitting their tasks and finding corresponding results. There would also be a lot of work-stations (slave nods) for computing of inserted problems.

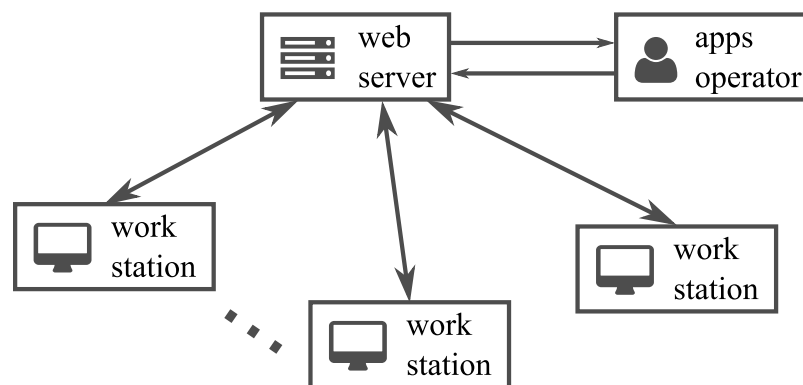


Figure 4.1: Conception of distributed application

Conception details of each part of the system (such as communication protocol) is discussed below including the details of realization.

4.1 Web server

The purpose of web-server is storing of task's list and providing interface for standard operations on this data set (inserting, updating and deleting of data). Server also shows results of solved tasks with other information about computing process and provides application interface for each station. Technically web server is standard database web application.

The list of all web application major features and fundamental parts of web application follows:

1. Inserting, modifying and removing users of the system. Each user has his own privilege levels. Admin of the system could create new users (and deleting or modifying existing users).
2. Inserting, modifying and removing tasks of the system that are later distributed for solving. Tasks of system are later converted to solving discrete logarithm or integer factorization problem. Each task has its priority level.

There are three kinds of task in the system:

- cryptanalysis of RSA cypher (finding message m and private key d using values c, e and n),
 - cryptanalysis of ElGamal cypher (finding message m and private key x using values c_1, c_2, p, g and h)
 - cryptanalysis of Diffie-Hellman key exchange protocol (finding shared key using values p, g, g^a and g^b).
3. Inserting, modifying and removing stations of system. Station is one node of the system that computes submitted tasks and returns results. Web application has to manage identification information of each station and provide functionality for assignment of station and task (this is done automatically by system with respect to task's priority level or by user).
 4. Providing detail information about each task and station and showing results of computation.

These details consist of answers for the following questions:

- how much time does the solving of task taken,
 - when the station was last active,
 - what is the solution of some task if it has been already found.
5. Web application also should provide manual pages (user guide). This should inform how to perform each step above.

4.1.1 Realization of web application

Some basic information about technical realization of each web server (and application running on it) part follows:

User interface: consists of control panel that is usable for inserting and modifying of tasks and also for fetching information about found results. Web application is available only for registered users (requires login and password for successful sign-in).

Navigation bar (menu) of sites is on the left side and contains reference for all major features of application.

The graphical user interface is designed as responsible web-site for large scale of resolutions. It is based on HTML5 and CSS3 technologies. Interface is designed only for relatively new browsers.

Application interface provides fundamental functionality for exchange of data between web server and workstations. All data are transferred via HTTP protocol and in JSON format (in the way from server to workstation) or using POST request method (in the way from workstation to server).

The task that goes from server to workstation contains definition of the task that follows this format (in case of RSA cypher):

```
{ "taskId": "(int)", "type": "RSA",
  "n": "(hex)", "c": "(hex)", "e": "(hex)" }
```

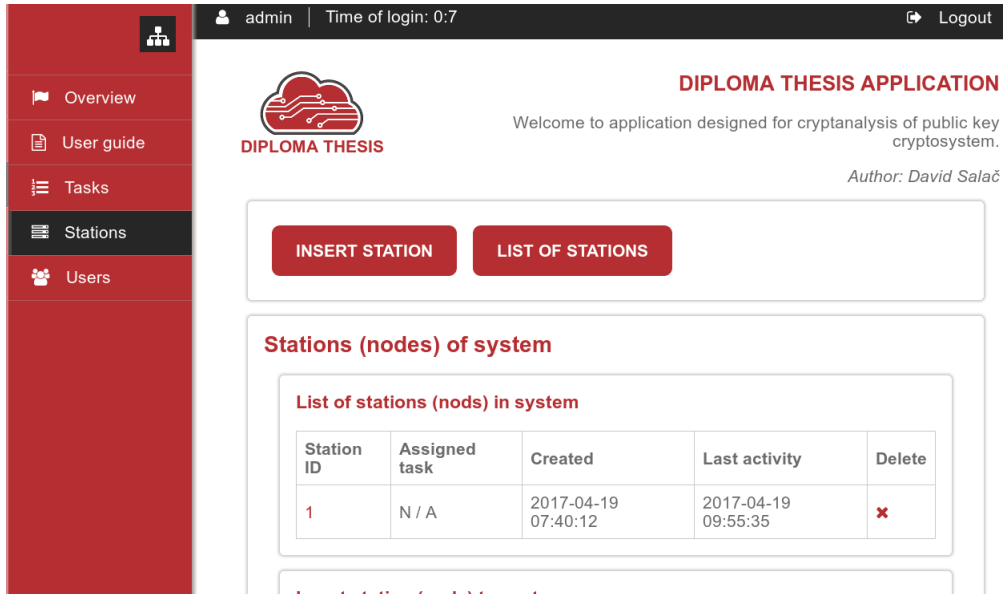


Figure 4.2: Screen of user's control panel

And the similar format is used in the case of other kind of problems. Difference is in the key value **"type"** and composition of task that fits to task selected task type. For ElGamal data has following format:

```
{ "taskId": "(int)", "type": "ElGamal", "p": "(hex)", "g": "(hex)",
  "h": "(hex)", "c1": "(hex)", "c2": "(hex)" }
```

For Diffie-Hellman key exchange problem task has following format:

```
{ "taskId": "(int)", "type": "DH", "p": "(hex)",
  "g": "(hex)", "gPowA": "(hex)", "gPowB": "(hex)" }
```

where **(hex)** represents the number in hexadecimal form and **(int)** represent integer (decimal system).

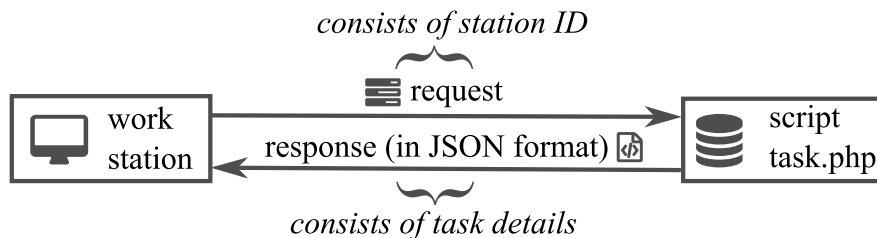


Figure 4.3: Processing of station requirement scheme

Workstations return results using POST request method to script **solution.php**. And also sends positive acknowledgement (also using POST

method) right after receiving data from the server. Each request send by POST method has the following format:

`type=(type)&stationId=(int)&taskId=(int)&par1=(hex)...`

where concrete form of `par` values depends on task type.

Request for data is sent to script `task.php` with identification of station (it is send using GET method).

Database solution Web application has to save at least the following information:

1. Users of system with login, privilege level, description, password (as HASH).
2. Logs that contain which user in which time was singed-up in the system.
3. Tasks inserted to system with the time of insertion, priority, type (RSA, ElGamal or Diffie-Hellman cryptosystem) and parameters of task such as n , g and p .
4. Solution of task with the time of computation, decrypted message or shared key.
5. The stations of system with the station identification, time of creation, last activity of station and optionally task to be solved.

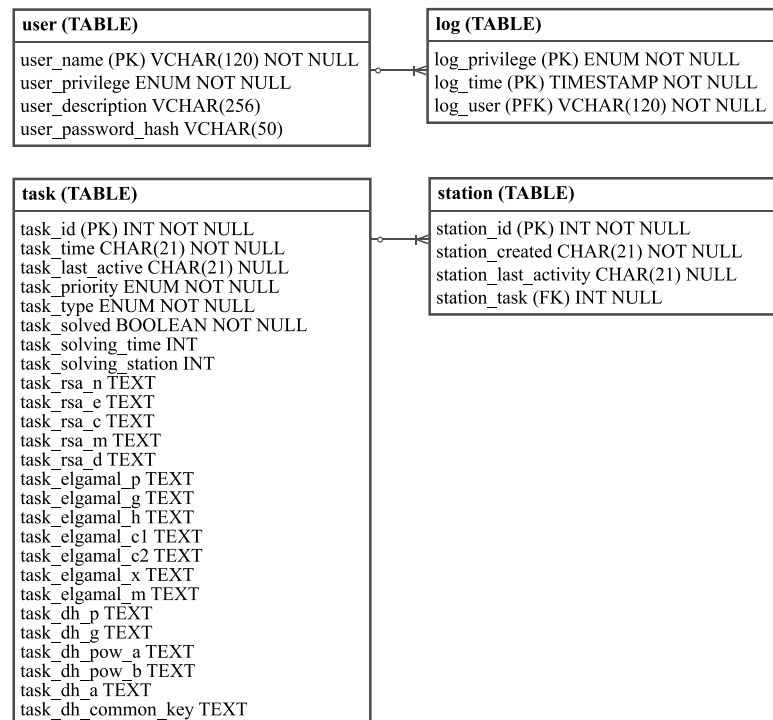


Figure 4.4: Entity-relationship model of database

Database of the system is created in MySQL RDBMS which is the low-coast solution with specific disadvantages (in compare to professional RDBMS, at

least PostgreSQL). It is, for example, impossible to create primary key of relation consisting larger data type – this is especially problematic in the case of this application. Installation file of database also contains insertion sequence for the first user of the system. Communication of PHP scripts with MySQL is managed by PDO.

Previously mentioned problem with the size of data type contained in primary key leads to bit more complex scheme of application database which is shown in figure 4.4 above.

Technical details Web application is written for PHP language of version 7.0 that provides some improvements of type checking which is relevant for security of application. The MySQL database is designed to version 5.5 and only InnoDB engine is used. Both technologies has significant level of portability and they were backward compatible historically.

Specific technical features are determined by popularity and license agreements of each technology. At this point of view both PHP and MySQL are selected in the top level (both are open-source, free, cross platform and widespread technologies). It practically means that web-application could run on almost every available web-hosting (in year 2017).

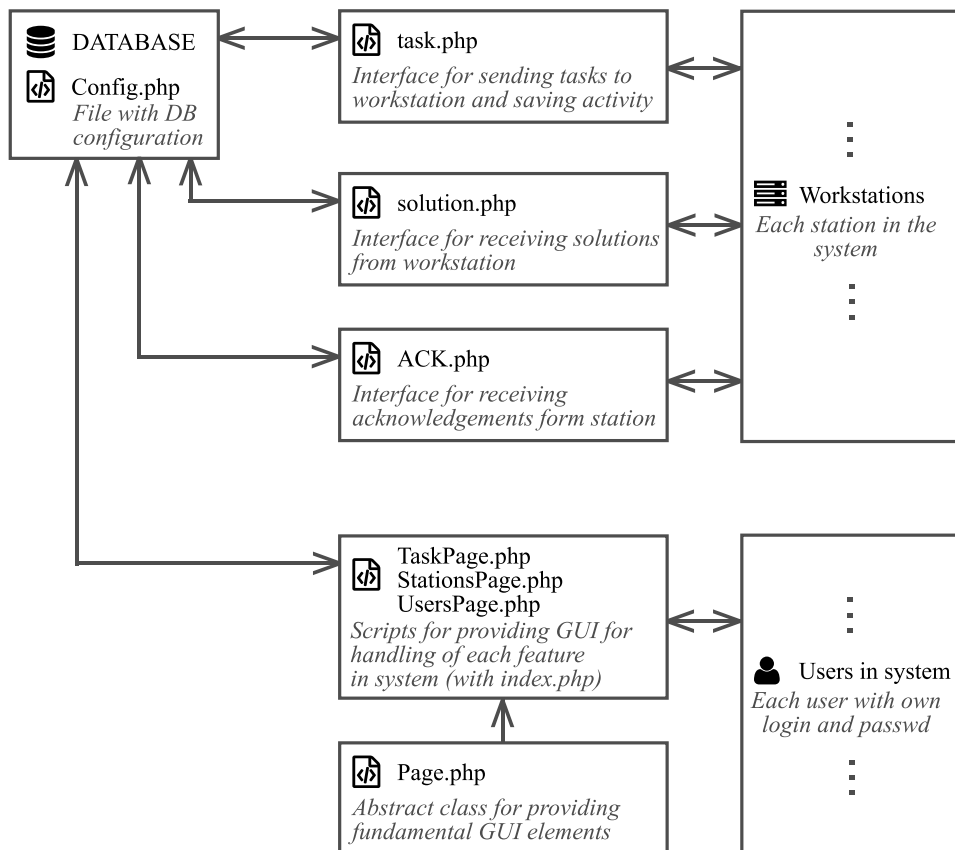


Figure 4.5: Block diagram of web application

Block diagram of application Web application was designed in the way that has been mentioned previously. For the purposes of making clear how application really works and the illustration of functionality – block diagram is included in figure 4.5.

There are only most important functional blocks of web application included in figure 4.5. The rest of important application's features are mentioned above in the list of application major features.

4.1.2 Summary

There are two main purposes for existence of web application in the form as it is designed before. The first is to provide fundamental interface for users to editing of inserted problems and for inserting new ones. Other reason is to provide application interface for workstations that are designed to solve inserted problems and to distribute inserted problems to stations and manage of the synchronization.

Just for making the work with the system easier there is also implemented generator of random tasks in the system. This is done in class `RandomTaskGenerator`. Application access this file through its API using AJAX.

Chosen way of web application's realization is determined by popularity and openness of selected technologies. The PHP is the most popular language for programming of web application which is available for free and under open license. The same situation is in the case of chosen RDBMS which is MySQL (on InnoDB engine) that is the most popular database solution for web applications under GNU license.

4.2 Workstations


Workstation (or just station) represents one nod of distributed system. The function of station is straightforward: to obtain a task (and send acknowledgement), to compute it and to return the results back to the server.

Application is called SaFaDI (*motivated by Solve a Factorization and Discrete logarithm problem*) and it is composition of three main packages and one external application. It is written in Java SE language and external application called msieve [13, modified] is written in C++ language.

Technically workstation is standard console application. The biggest advantage of this approach is in portability of the output. Application could run almost on every machine where Java Virtual Machine does (mentioned external application written in C++ is also portable). Environment for running of application is not restricted only for desktop computers (meaning systems with operation system Windows or some distribution of Linux).

Application consists of four packages. The first package with main class is called `bid.mythesis`. There is some fundamental functionality of application contained in this package. This package contains the main class of application including infinity application loop. The second package is called `bid.mythesis.cryptanalysis`. This package is useful for transforming of input

to concrete cryptanalytic problem and provides basic functionality for final computation. At least there are two packages, first to solving of discrete logarithm problem called `bid.mythesis.logarithm` and other one to solving of integer factorization problem called `bid.mythesis.factorization`. These packages contains numerical methods for solving of each problem type.

A screenshot of a Java console window titled "dist:java — Konsole". The window has a menu bar with "File", "Edit", "View", "Bookmarks", "Settings", and "Help". The main area shows the following text:

```
addmin@addmina:~/NetBeansProjects/SaFaDl/dist$ java -jar SaFaDl.jar
Application started at: Apr 24, 2017, 9:31:08 PM
New data arrived at Apr 24, 2017, 9:31:08 PM
Solve: RSA, TaskID: 7, n: 3233, c: 2790, e: 17
System find solution! p: 47, q: b5, d: cd7, time: 0, type: RSA, m: 1569, ta
skId: 7, stationId: 1,
```

The window has a scrollbar on the right and a tab labeled "dist:java" at the bottom.

Figure 4.6: Screen of application

4.2.1 Receiving tasks and transmitting results

Tasks are received in the main package of application `bid.mythesis` in class `ReceiveData`. Data set is downloaded from selected URL defined in class `Configuration`. After downloading of data they are used for creating instance of class `CryptanalysisTask`. In the case that application succeed in creating of such instance, positive acknowledgement is sent back to server.

Sending of data set to server is done by using class `SendData`. Data are converted to string usable for POST request method and they are send to selected URL defined in configuration file. Whether transmitting of results were successful is checked using response code. Transmitting is done in independent thread and using infinite loop, data tries to be sent until it is not successful (with period equals to three seconds). The same method is used for sending of acknowledgement.

4.2.2 Processing of received tasks

After the receiving of task there is a package called `bid.mythesis.cryptanalysis` for handling of the problem. Major purpose of this package is converting of task to

discrete logarithm or integer factorization problem (depending on what kind of task is fetched).

The package contains abstract class `CryptanalysisTask` that encapsulate single system task. It also contains functionality such as simple JSON parser (task is received in JSON format). Static method `CryptanalysisTask` returns proper instance for the problem solving. Solution of the problem is found using abstract method `analyse` that returns map which is sent to server. Class implements `Runnable` interface because `run` method is called in independent thread. Method `run` called method `analyse` and send the found solution to the server asynchronously using class `SendData`. Data that are sent to server consist not only of found solution but also with time that finding of solution have taken, station ID, task ID and specification of task type.

There are three classes that extend `CryptanalysisTask`, each for one task type:

`DHCryptoanalysis` represents the class for cryptanalysis of Diffie-Hellman key exchange protocol. The purpose of this class is computation of shared key from known values p , g , $g^a \bmod p$ and $g^b \bmod p$. Computation began in finding private key a by solving of discrete logarithm:

$$a = \text{ind}_g(g^a \bmod p) \pmod{p}$$

using class `DiscreteLogarithm` in package `bid.mythesis.logarithm`. After finding of solution shared key is computed using as $(g^b)^a \bmod p$.

The following code shows how `analyse` function is implemented. Implementation of this function is similar in each situation.

```
@Override
public Map<String, String> analyse() {
    long startTime = System.currentTimeMillis() / 1000L;
    DiscreteLogarithm solver = DiscreteLogarithm.initInstance(g,
                                                                gPowA, p);
    this.a = solver.commitMethod();
    Map<String, String> res = new HashMap<>();
    if(a != null && g.modPow(a, p).compareTo(gPowA) == 0) {
        long totalTime = (System.currentTimeMillis() / 1000L
                          - startTime);

        this.sharedKey = gPowB.modPow(a, p);
        res.put("type", "DH");
        res.put("stationId", STATION_ID);
        res.put("taskId", this.getTaskId());
        res.put("a", this.a.toString(16));
        res.put("sharedKey", this.sharedKey.toString(16));
        res.put("time", Long.toString(totalTime));
        return res;
    }
}
```

```

    return null;
}

```

ElGamalCryptoanalysis represents the class for cryptanalysis of ElGamal cypher. It has a similar conception like there is in Diffie-Hellman case. Class tries to look for values of private key x and encrypted message m . Input of class consists of values c_1, c_2, p, g and h . Solution is also found by calculation of discrete logarithm value:

$$x = \text{ind}_g h \pmod{p}$$

There is also used class **DiscreteLogarithm** in package **bid.mythesis.logarithm** to do so. Finding of value m is simple:

$$m = (c_1^x)^{-1} c_2 \pmod{p}$$

RSACryptoanalysis represents the class for cryptanalysis of RSA cypher. The purpose of this class is converting the task to integer factorization problem. The input is represented by values of c, e and n . The output is represented by private key d and decrypted message m . Relation among this and factorization problem is well described in the first chapter.

Factorization of n is computed using class **Factorization** in package **bid.mythesis.factorization**. This is the common situation for n of size less than 70 bits. Larger numbers are factorized using msieve application. Msieve is open source application with implemented general number field sieve method. It is under public domain license since the end of year 2016. It is mainly written in C++ language and it is portable for large scale of platforms (especially Linux, BSD like and Windows).

The application uses slightly modified version of native msieve that differs in output format. Application is called using **ProcessBuilder** class (native class in java).

Each class prepares pairs in format **Map<String, String>** that will be sent to server using POST method. Such pairs are prepared in override method **analyse** in each case.

There could be a problem with a situation that incoming task does pass the lexical analysis but it is mathematically wrong. This could cause two situation. The first is that application does not do anything and stops its activity (no numerical method compute anything) and the second is that application falls to infinity loop of numerical method. In real situations it is not always simple to check whether the task has any solution (application could do only elementary tests in each task type).

4.2.3 Methods for integer factorization

Selected methods of integer factorization are implemented in package **bid.mythesis.factorization**. There is also functionality necessary for process of factorization. The most important class of this kind is **MatrixGF2** that is

usable for computing with matrix of parity (necessary for methods like the Dixon's one).

The effective work with parity matrix is important during the last step of algorithm (finding of null space of this matrix). The Gaussian elimination method is used in this class for this purpose. Technically parities in matrix are saved in `byte` array (two dimensional). Class also provides basic interface for working with matrix (such as transposing, inserting row or columns and so on).

Another important issue is working with relatively long numbers (meaning the number of decimal digit). Java has class `BigInteger` in standard library usable for this purposes. There are also a lot of other solutions usable for this purpose but none of them is not complex enough. Application use especially methods for arithmetic operations, comparing of integers and computing greatest common divider of two numbers (in this case using Euclidean algorithm). There also arise questions about real efficiency of methods in this class for numerical methods. The application also uses class `BigDecimal` in some cases for computing of square roots of number. Efficiency of methods in this class is not so relevant because it is only seldom used.

The conception of application is based on abstract class `Factorization` that encapsulate a lot of basic arithmetic functions necessary later. It also contains a method called `commitMethod` that returns desired factors of the number.

There are classes that extends class `Factorization` by implementation of each method:

`BruteForce` represents class for brute force factorization. It is most straightforward way how to achieve results. Method begins with with number two and tries to find factors of n computing modulus for each number which is less or equal than square root of n . Method is useful because of it's simplicity. It could be used for small numbers.

`PollardRho` is encapsulation of Pollard's Rho algorithm (for integer factorization). There are two methods for this purpose. The first represents one iteration of Pollard's Rho method. Another one pass all composite factors to the first method in cycle till it does not find relevant results. This method also sieves small factors by using brute force to some bound.

The method also checks number of iterations in each step, because there exists possibility that algorithm does not find any solution algorithm for given parameters.

Parameter x of the method is set up randomly in interval $[2, 35] \cap \mathbb{N}$. Example of source code for one iteration of Pollard's Rho algorithm is shown bellow this paragraph.

```
private BigInteger polardRhoEngine(BigInteger nr ) {
    BigInteger n = new BigInteger(nr.toString());
    Random randomGenerator = new SecureRandom();
    BigInteger x = new BigInteger(Integer.toString(
        randomGenerator.nextInt(33) + 2));
```

```

BigInteger y = new BigInteger("2");
BigInteger d = new BigInteger("1");

long freeze = 0;
while (d.compareTo(BigInteger.ONE) == 0 && freeze++ <
      maxIterations) {
    x = polyVal(x, new BigInteger[]
    { BigInteger.ONE, BigInteger.ZERO, BigInteger.ONE } ,n);
    y = polyVal(polyVal(y, new BigInteger[] { BigInteger.ONE,
    BigInteger.ZERO, BigInteger.ONE } ,n),
    new BigInteger[] { BigInteger.ONE, BigInteger.ZERO,
    BigInteger.ONE } , n);
    d = n.gcd(x.subtract(y).abs());
}
if(d.compareTo(n) == 0) {
    return null;
}
return d;
}

```

Pollard's Rho method is the first complex algorithm. It is useful for special purposes (such as factoring number with lot of small factors) and for numbers of size approximately less than 2^{30} .

Dixon represents Dixon's factorization method. It is the first complex method working with factor base. Dixon's method has more than one parameter (the number to be factorized and factor base). Composition of the factor base and size of the factor base is crucial for successful factorization of selected integer. In this case factor base is composed trivially by the first k prime numbers. Choosing of the right value of k is crucial, but there is no flexible manual how to choose it. It's value is chosen heuristically by polynomial:

$$|FB| = 10 + \frac{|n|^4}{192}$$

where $|FB|$ is size of factor base and $|n|$ is bit size of n (n is number to be factored).

Dixon's method is the first method that operates with parity matrix. Size of matrix is $|FB| \times |FB| + O$, where O is defined offset (equal at least one). The method is implemented in while cycle that is shown bellow this paragraph.

```

while (index < (factorBaseSize + parityMatrixOffset)) {
    if(++iteration > this.maxIterations) return null;
    BigInteger x = new BigInteger(this.getN().bitLength()*2,
    randomGenerator).mod(randomUpperBound);
}

```

```

x = x.add(sqrtN);
if(x.mod(Factorization.BIGINTEGER_TWO)
.compareTo(BigInteger.ZERO) == 0)
    x = x.add(BigInteger.ONE);
BigInteger x2 = x.modPow(Factorization.BIGINTEGER_TWO,
    this.getN());
exponentsOverFB[index] = Factorization.
    getFactorBaseCoefficients(x2, fb);
if(exponentsOverFB[index] == null) continue;
if(xOverview.contains(x)) {
    exponentsOverFB[index] = null;
    continue;
}
parityMatrix.insertRow(exponentsOverFB[index], index);
xList[index] = x;
xOverview.add(x); //Set of all x
index++;
}

```

After method succeed in finding enough vectors over factor base there is computation of null space of such matrix (transposed variant of matrix `parityMatrix` shown in the source code above). Vectors of null space are used for selecting proper vectors consist of exponents of integer over chosen factor base. The result is Legendre's congruence usable for finding of nontrivial factors.

There is a possibility of choosing factor base differently (by statistical analysis of situation) which could be useful in some special situations.

QuadraticSieve is the class of Quadratic Sieve algorithm. Although QS is one of the most effective algorithm, it is also method that is sensitive for proper value of the method's parameters. There are three parameters at all. The first is number n to be factorized, the second is factor base and the third is interval for value of x . In application there is only simple polynomial of form:

$$Q(x) = (x + \lceil n \rceil)^2 - n$$

used in application. The value of x is found using Tonelli-Shanks algorithm for found s_{i_1, i_2} values and then $x_{i_1, i_2} = s_{i_1, i_2} + kp_i$ for some integer k for such x is lower than some selected upper bound. There is also the biggest difference between Dixon's method and QS. Values of $Q(x)$ is factorized over factor base usable in Dixon's method but prime factors p_i used for computing of x value is element of QS factor base which is subset of the Dixon's one. It is because of equation $s_i^2 \equiv n \pmod{p_i}$ does not have solution for each prime number p_i . Whether p_i is suitable for Tonelli-Shanks algorithm could be easily checked by computing of Legendre symbol $\left(\frac{n \pmod{p_i}}{p_i}\right)$. If the symbol equals to value 1 then the solution of equation exists.

Selecting of method's parameters is also done heuristically. There exists a lot of strategies how to do it. Application uses parameters of following form:

```
int fbMinimalSize = 384;
BigInteger xIntervalDivisor = new BigInteger("16384");
if(n.bitLength() <= 55) {
    fbMinimalSize = 256;
    xIntervalDivisor = new BigInteger("256");
}
else if(n.bitLength() <= 65) {
    fbMinimalSize = 512;
    xIntervalDivisor = new BigInteger("256");
}
this.factorBaseSize = fbMinimalSize +
(n.toString().length() * n.toString().length() *
n.toString().length() * n.toString().length())
    / 1024;
this.parityMatrixOffset = 5;
this.xInterval = new BigInteger("512")
.add(Factorization.bigIntegerSqrt(
this.getN()).divide(xIntervalDivisor));
this.sqrtN = bigIntegerSqrt(this.getN());
```

parameters are modified in the situation when algorithm does not find any solution (it means return null value). In such case there is suitable to enlarge the size of factor base.

Rest of algorithm is similar to Dixon's method. Especially finding of null space of parity matrix and Legendre congruence. Implementation of QS method in application is usable for integers of size less than 2^{70} .

There is also some functionality in package `bid.mythesis.factorization` that could be latter used for implementing of General Number Field Sieve method. Most of this functionality is available in class `NumberFieldSieve`. The class, for example, contains functionality for finding of sieving polynomial of GNFS method and other useful functionality. GNFS itself is quite difficult to implementation. It requires a lot of other mathematical functionality. Instead of this class, the application uses external program called `msieve`, which seems to be ideal for this purpose. This program also contains implementation of some advanced techniques such as MPQS (Multiple Polynomial Quadratic Sieve) that are usable for factoring of some larger integers. This method represents straightforward improvement of Quadratic Sieve method which differs only in selected polynomial.

The most difficult task for each method is to choose the right parameters that are suitable for selected problem. It is necessary to try maximal number of values and choose the right value. Parameters depends not only on the size of input number but also one to each other. This is in fact searching of minimal value (that represents the

running time of method with selected parameters) in some multidimensional space in which problem could not be easily described by some simple function.

There of course exists a potential for parallelism for each step of these algorithms and also there exists some concepts for increasing efficiency of these algorithms by using some specific hardware. But the leading issue is to analyze potential of distributed application.

4.2.4 Methods for solving of discrete logarithm

Numerical methods for solving of discrete logarithm problem are in a package `bid.mythesis.logarithm`. The package consists of abstract class `DiscreteLogarithm` that is extended by classes with numerical methods. Abstract class contains functionality necessary for computation (for example computing of square root of integer). Class also contains static method returning instance with chosen proper numerical method. The only abstract method is `commitMethod`.

In each method there is used Java native class `BigInteger` to working with large integers. Conception is similar to integer factorization problem. The biggest difference between interface discrete logarithm and integer factorization solver is that method return just one integer. This integer represents exponent x in equation:

$$g^x \equiv a \pmod{p}$$

for some given integer values g (group generator), a and prime p (group order).

There are some effective methods for solving of discrete logarithm problem in application. Classes with implemented method are mentioned in the following list:

`BruteForce` represents the class for solving of discrete logarithm problem using brute force. It is most straightforward method of all. It could be useful for some relatively small numbers. The biggest disadvantage of this method is its time complexity. Otherwise the biggest advantage of this method is its simplicity for programming that is shown in the code below this paragraph. Another advantage is in relatively small amount of required memory (no map for any values is needed). Application use brute force method for solving of discrete logarithm problem for groups of order less than 2^{20} (it means that prime number $p < 2^{20}$).

```
private BigInteger bruteForceMethod() {
    for(BigInteger x = BigInteger.ZERO; x.compareTo(n) <= 1 &&
        !Thread.currentThread().isInterrupted();
        x = x.add(BigInteger.ONE)) {
        if((g.modPow(x, n)).compareTo(a) == 0)
            return x;
    }
    return null;
}
```

The method also checks whether the current thread is not interrupted in each iteration. This is important feature for conception of this application (number of iterations of cycle could be great). The thread is usually interrupted in the situation when a new task is fetched to the system. It is not simple to stop the thread in Java (method `stop` is deprecated in current version of Java). Currently the only suitable way how to stop the thread is to use `interrupt` method in `Thread` class.

PollardsRhoDL represents method called Pollard's rho algorithm for logarithms. It has similar conception like method Pollard's rho for integer factorization. The realization of this method is straightforward. The core of method is in the following code:

```
do {
    pollardStepLittle();
    pollardStepLarge();
    pollardStepLarge();
    currentIteration++;
} while(x1.compareTo(X1) != 0 &&
    iterationLimit >= currentIteration);
```

Methods `pollardStepLittle` and `pollardStepLarge` operate with state variables of method (with variables x_1, a_1, b_1 in first case and with variables X_1, A_1, B_1 in other one). Methods later compute probable value of unknown integer value x .

This method is useful only in some special cases (it is not general purpose method). The biggest advantage of the method is its simplicity and run time (that is relatively short). Otherwise the biggest disadvantage of this method is that it does not return suitable results in each case. Another advantage of algorithm is its low memory requirement (which is common feature with brute force method).

Although Pollard's rho algorithm for logarithms could be used as the first time method, which could run before some other method (because of its fastness), application does not use it. It is because of presumption of fine implementation of inserted tasks which make using of this algorithm almost impossible.

BabyStepGiantStep represents Shank's baby-step giant-step algorithm for solving of discrete logarithm problem. This method is useful for a large scale of discrete logarithm problem. The biggest disadvantage of this method are memory requirements. In default version of algorithm it has to save \sqrt{p} (where p is order of group) values which makes it unusable for practical applications. It is possible to save only some subset instead of full set of \sqrt{p} integers and use algorithm as probabilistic one. The application use both versions (probabilistic for larger integers) of algorithm.

The algorithm consists of two steps. The first (called baby step) creates a map of values of following form:

```

for(BigInteger j = new BigInteger("0"); j.compareTo(m) < 0
    && !Thread.currentThread().isInterrupted();
    j = j.add(BigInteger.ONE)) {
    BigInteger tuple = g.modPow( j , n);
    gjA.put(tuple, j);
}

```

in this step the algorithm saves keys consists of values $g^j \bmod p$ for all $j < p$, where j is saved as value in map. Probabilistic version of algorithm saves only some values of j in interval $[0, \sqrt{p}) \cap \mathbb{Z}$.

This map is used in the second step of algorithm (called giant step):

```

for(BigInteger i = new BigInteger("0"); i.compareTo(m) <= 0 &&
!Thread.currentThread().isInterrupted();
i = i.add(BigInteger.ONE)) {
    BigInteger j = gjA.get(J);
    if(j != null) {
        BigInteger res = i.multiply(m);
        return new BigInteger(res.add(j).toString());
    }
    J = J.multiply(gInvPowerToM).mod(n);
}

```

variable `gjA` represents hash map used by algorithm (which is mentioned above) and `n` is modulus p also mentioned above. In this step algorithm tries to find matches of values J and the key in map which is found in baby step. There is also visible condition for checking of thread interruption `!Thread.currentThread().isInterrupted()`.

The only parameter of the method is maximal size of map structure which is defined by memory limits of application. This value is selected as 10 000 000, but could be easily changed. Probabilistic version of algorithm also uses infinity while loop in which it calls method in each iteration till it does not find relevant result (it means does not return null value).

The advantage of this algorithm is its time complexity which makes it usable especially in its original version. The disadvantage is that only relatively small values of x could be found which could be problematic in practical application. The method itself represents one of the meet-in-the-middle algorithm (this is special class of algorithms). The infinity loop is shown in following code:

```

BigInteger solution = babyStepGiantStep();
while(solution == null &&
!Thread.currentThread().isInterrupted()) {
    solution = this.commitMethod();
}
return solution;

```

in this code example, variable `solution` represents seeking exponent x . The application uses Shank's baby-step giant-step algorithm for solving of discrete logarithm problem for all integers $p \geq 2^{20}$ (where p represents order of group). It is crucial algorithm for solving of discrete logarithm problem not only in this application. Also some other algorithms use this one in some special part of its computation process.

SilverPohligHellman is the realization of Silver-Pohlig-Hellman method. It represents straightforward improvement of Shank's baby-step giant-step algorithm [2, p. 240]. This method could find solution at least $\sqrt{p_i}$ steps where p_i is maximal prime factor of $(p - 1)$ term factorization. It is one of the popular method that is especially useful for insecure implementation of some method based on discrete logarithm problem. In standard situation, this method has comparable complexity like the Shank's baby-step giant-step algorithm and it is also unusable for complex problems. The application uses this method for numbers that has modulus of size more than 35 bits.

Algorithm itself consists of two steps. The first is finding factorization of $(p - 1) = \prod_{\forall i} p_i^{\alpha_i}$ and setting up the congruences to be solved in the next step:

```
private BigInteger silverPohligHellmanMethod() {
    //----- Finding set of congruences -----
    BigInteger [][] congruences =
    new BigInteger[prime.length][2];
    for(int i = 0; i < this.prime.length; i++) {
        BigInteger x = new BigInteger("0");
        BigInteger pi = this.prime[i];
        int exponent = this.exp[i];
        BigInteger bi = this.b;
        BigInteger aInv = a.modInverse(q);
        for(int e = 1; e <= exponent; e++) {
            BigInteger val = bi.modPow(
                qSubtractOne.divide(pi.pow(e)), q);
            BigInteger j = findJ(pi, val);
            x = x.add(j.multiply(pi.pow(e-1)));
            bi = this.b.multiply(aInv.modPow(x, q)).mod(q);
        }
        congruences[i][0] = x;
        congruences[i][1] = pi.pow(exponent);
    }
    //-----

    //Use the congruence for computation
    return this.chineseRemainderTheorem(congruences);
}
```


and the last step is solving of set of congruence using Chinese remainder theorem that also represents the result of discrete logarithm problem. This part of algorithm is much faster one.

IndexCalculus is implementation of Index calculus method for discrete logarithm. Although it is one of the most effective method for this purpose (means in time complexity way), application does not use it. It is because of many problems that arise of working with matrix over \mathbb{Z}_n for some integer n which is not prime number. There exists a lot of algorithm for this purpose but none of them is effective enough and relative simple for implementation. This is also the reason why this method is not used in application. On the other hand, implementation of this method works correctly for most of inputs (and it could be especially useful for larger modulus).

The method consists of two blocks. The first is represented by class **MatrixGFn** that could work with matrix over \mathbb{Z}_n for some composed integer n . The rest of algorithm is written in method **indexCalculusIteration** in class **IndexCalculus**. This method consists of two steps. The first one is the sieving process over chosen factor base (similar principle to Dixon's algorithm) together with construction of matrix. The other one is finding solution of linear system that leads to final solution (shown in while cycle bellow).

```
while(iteration < maximalNumberOfIteration) {
    //Obtain random exponent in [0,p-2);
    BigInteger e = new BigInteger( p.bitLength(),
        randomGenerator).modPow(BIG_INTEGER_TWO,
        p.subtract(BIG_INTEGER_TWO));
    BigInteger gPowE = g.modPow(e, p);
    BigInteger[] factorsOverFB = factorOverFB((gPowE.multiply(a)).
        mod(p));
    if( factorsOverFB != null ) {
        BigInteger res = e.multiply(BIG_INTEGER_MINUS_ONE);
        for(int j = 0; j < factorBaseSize; j++) {
            res = res.add( factorsOverFB[j].multiply(solution[j]) );
        }
        BigInteger result = (res.mod(p.subtract(BigInteger.ONE)));
        if(g.modPow(result, p).equals(a)) {
            return result;
        }
    }
    iteration++;
}
```

The advantage of methods for solving of discrete logarithm problem is that they do not require so many parameters (instead of method for solving of integer factorization problem). Other advantage lies in relative simplicity of its programming

code. The disadvantage is that some effective method for discrete logarithm problem require relatively a lot of memory for successful running (in comparison to integer factorization methods).

4.2.5 Summary

The application conception follows a block diagram that is shown in figure 4.7. This block diagram is only conceptual and it is only for illustration of situation (and data flow) in distributed application.

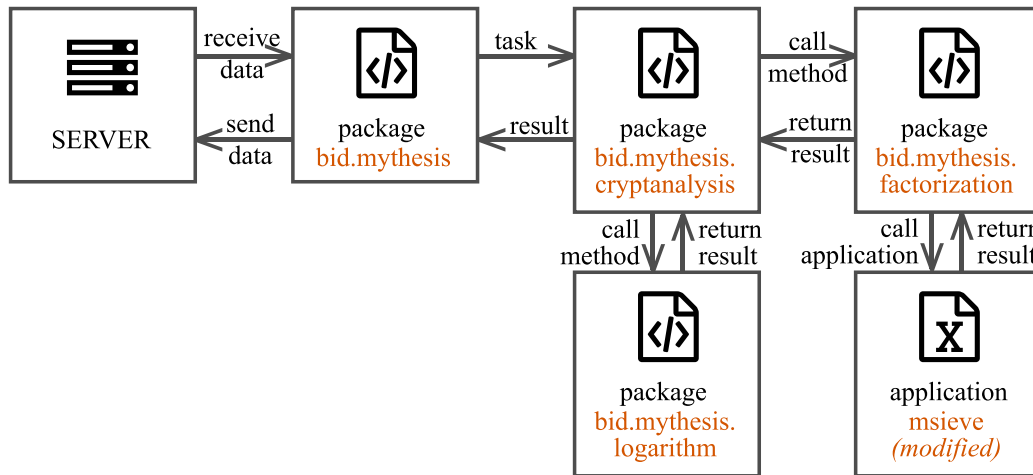


Figure 4.7: Block scheme of application

Web application is useful for providing of GUI, otherwise application that runs on workstation is primary target for computing of tasks. It could cryptanalyse most of popular cryptosystems, especially RSA and ElGamal cyphers or Diffie-Hellman key exchange problem. The application also contains methods for solving of integer factorization problem such as Pollard's rho, Dixon's method and Quadratic Sieve. It also uses external application that has implemented general number field sieve method. This methods represents most effective algorithms for solving of integer factorization problem of nowadays.

Tasks which are converted to discrete logarithm problem are solved with relatively less effective methods (in comparison to methods for integer factorization). But they also represents fundamental methods for this purposes and also most effective ones. These methods are Pollard's rho method for discrete logarithm and baby-step giant-step method (and also Silver-Pohlig-0.-Hellman method).

Application represents standard terminal program. It is especially useful because of portability of outputs and the possibility to be running as the background one. Application could run on all most popular operating systems.

Application is written in widely available Java language in way that allows its further extending. This could be especially useful for implementing new methods (or other functionality) in the future.

5 Using of application in real situation

Usability of application is determined by the running time of tasks. There are at last only two kinds of task. The first is to solve discrete logarithm problem and the other one is to solve integer factorization problem. Rest of algorithms, especially fetching and sending the tasks and decrypting messages from already found values, does not consume relevant amount of time (in situation that they are written in some standard usable way).

To measure the real effectiveness of application there are few presumptions. The first is that we suppose exponential characteristic of time t [s] (in seconds) to size of input N [b] (in bits) with number of nodes in distributed system S [—] (the number without unit). The result of this measurement should be some quantification of running time improvement for great amount of nodes S in system.

There are also practical limits of this approach. Only few nodes are available in real situation and no node is exactly the same one as the others (in this case there are three different computers with different CPU). This could have negative consequences for tolerance of measurement and its extrapolation. There are also problems with parameters of each method that are changing based on size of input N and also changing of methods itself. This could lead seemingly to paradox situation where there is smaller time of running for greater N .

Measurement was proceeded on at most three stations, three slightly different computers, last two differs only in hardware composition:

1. CPU information: Intel(R) Core(TM) i7-5557U CPU @ 3.10GHz
RAM capacity: 8 GB
Operating system: Linux 4.4.0-72-generic #93-Ubuntu x86_64 GNU/Linux
2. CPU information: Intel(R) Core(TM) i7-2620M CPU @ 2.70GHz
RAM capacity: 8 GB
Operating system: Linux 4.4.0-59-generic #80-Ubuntu x86_64 GNU/Linux
3. CPU information: Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz
RAM capacity: 8 GB
Operating system: Linux 4.4.0-72-generic #93-Ubuntu x86_64 GNU/Linux

The application were running only in console level with minimal number of other process, such as desktop environment. This is because each other application could negatively affected the running process.

This chapter refers on distribution of one task to multiple station. Not distribution set of task to multiple station. The efficiency increase of second approach is obvious but it has nothing to do with conception of this application.

5.1 Integer factorization problem

The conception of the measurement was to measure ten tasks for each task type. This type is determined by the number of connected nod in the system S , input size $N[bit]$ and generator of random tasks (the web application random task generator is used).

Table 5.1: Measurement of integer factorization, for $S=1$

N [bit]	n (hex)	e (hex)	c (hex)	time t [s]
64	692dfddb16527fb7	c5d7	5b34edfcac39d363	31
64	9dbff433e6f0911f	14ddd	9c08c7ada239d957	141
64	611a9238e637c67b	e705	0d7e3e1ee1ce899b	27
64	6f2b4db3493377ab	1d4b9	6b3bcdc63e505f02	56
64	a17873be393db4cf	293f	a1471952c0b12f21	101
64	c97c848ed56e2d01	187d	60708401df31349c	72
64	ae08933fd639ad8d	127c9	3fd73d0e06db6d12	193
64	88c2184154e4ec1f	1db07	1cf1f238308ee5df	143
64	825aca7cab1fc313	15df7	18b41a792dd43edf	110
64	7c1b2d9abb1e934f	eb63	306525c5f7bc6041	52

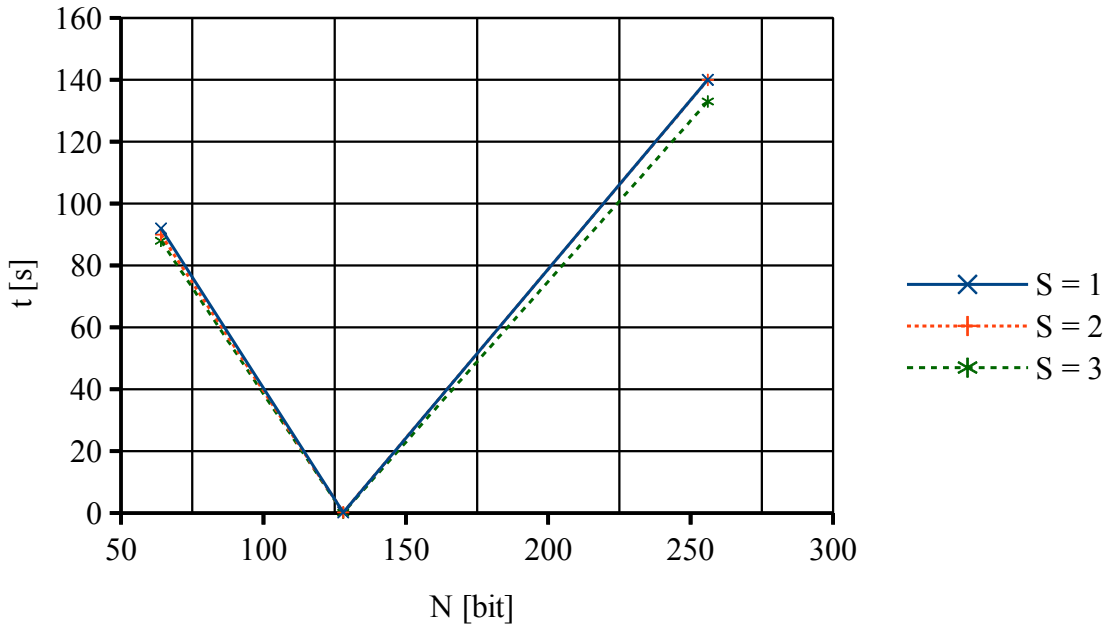
For illustration of the situation, there is a table 5.1 appended. It is obvious that variance of data set is wide in this case. There is also a measurement tolerance caused by time measuring. It is done by measuring the time at the start of the algorithm and ended after algorithm finishes its work. It could be done with accuracy equals to ± 2 s (worst case analysis, not considered in the following measurement processing). This is common feature for all following measurements (in each case).

Table 5.2: Measurement of integer factorization for multiple stations

S (nods)	N [bit]	$\mathcal{O}t$ [s]	Δt [s]	method
1	64	92	54	QS
2	64	90	48	QS
3	64	88	51	QS
1	128	0,2	0,4	MPQS (msieve)
2	128	0,2	0,4	MPQS (msieve)
3	128	0,1	0,3	MPQS (msieve)
1	256	140	21	GNFS (msieve)
2	256	137	17	GNFS (msieve)
3	256	133	19	GNFS (msieve)

Table 5.2 shows the result of measurement in complex way. The first column represents number of stations (nods) connected to server. The second represents complexity of integer factorization problem N by bit size of input word (n). The third column represents average time $\mathcal{O}t$ for each task type. The fourth column is standard deviation of measured time for each task type. Last column represents method (and external application) that was used to solving of problem.

Figure 5.1: Graph of relation between time t and input size N for S stations (RSA)



For illustration of situation graph 5.1 of average measured values (without error bars) is included. The graph shows described problem where relatively easier task takes much more time to be solved (because of selected numerical methods and its parameters).

5.1.1 Real situations

The problem of integer factorization is crucial to security of RSA cryptosystem. Most of nowadays certificates that use RSA are based on 1024 bit version (or sometimes 2048 bit one). For example Google's SSL certificate has 2048 bit modulus since 2013, before this enlargements it has 1024 bits [15]. This is also a common feature of almost all X.509 certificates (used for digital signature and SSL), that do not define exactly, whether RSA has to be used, but it still is a frequent variant.

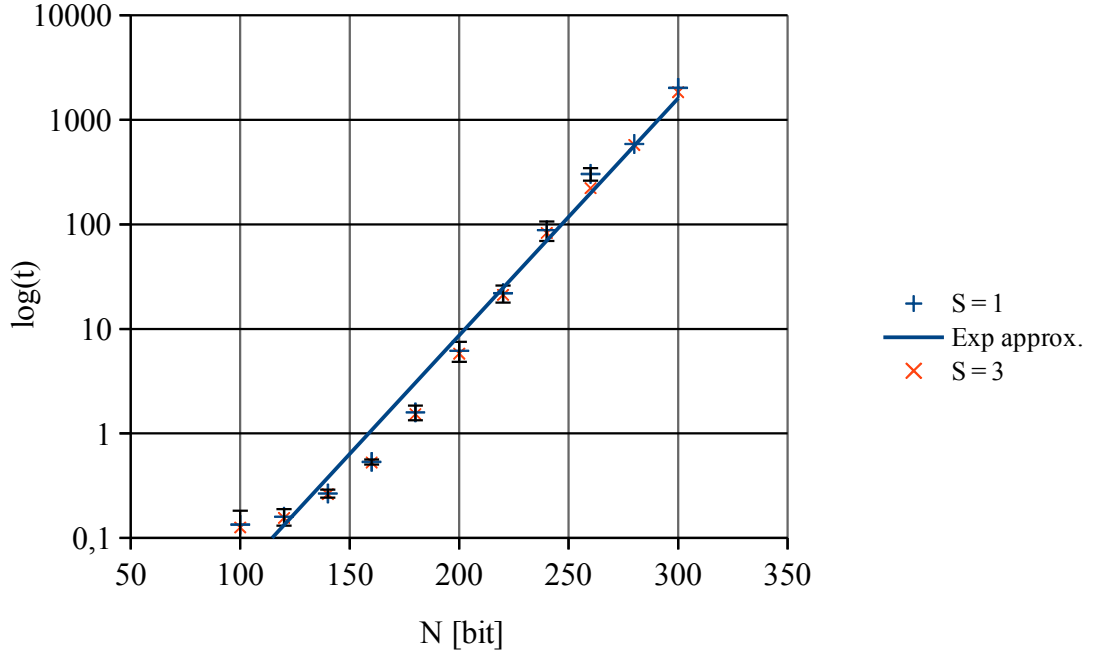
Estimation of the time that is necessary to solve some practical problem could be done by measuring of large set values and finding some relation. There are some problems with measurement because they are enormously time consuming. We consider to have exponential relation. That is also the way how to fit measured data (and find relation).

The data set that was measured is shown on a figure 5.2. The figure has logarithmic scale on vertical axis ($\log(t)$, t [ms]) and there a is size of input on horizontal axis (N). Data set for three station are fitted with the following function:

$$t(N) = \frac{0.3091422}{1000} \cdot \exp\left(\frac{0.1018556}{2}N\right) \quad (5.1)$$

that is also the function used for the following time approximations.

Figure 5.2: Graph of relation between time t and input size N (integer factorization)



If we consider equation (5.1) together with most popular sizes of input N we obtain approximation shown in table 5.3.

Table 5.3: Approximation of time for solving N bits RSA tasks

N [bit]	t [s]	t [year]	S
1024	$1.37 \cdot 10^{19}$	$4.36 \cdot 10^{11}$	3
2048	$6.12 \cdot 10^{41}$	$1.94 \cdot 10^{34}$	3
4096	$1.21 \cdot 10^{87}$	$3.84 \cdot 10^{79}$	3
8192	$4.76 \cdot 10^{177}$	$1.51 \cdot 10^{170}$	3

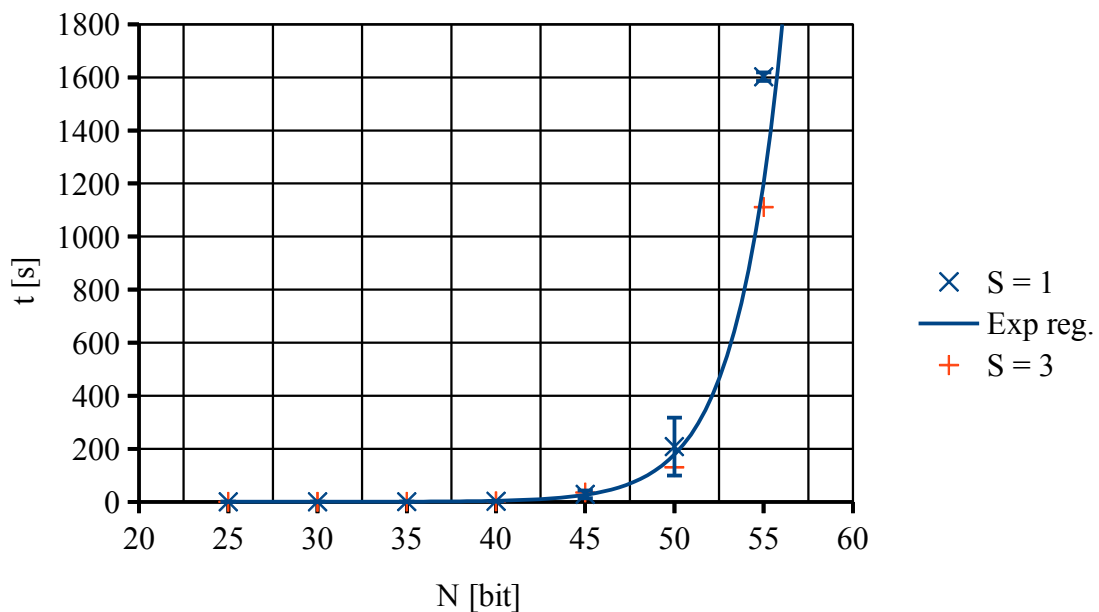
Table 5.3 represents only rough approximation of this problem. The number of station could be much height than $S = 3$ in real situation. The best known

results of integer factorization process is $N = 768$ [16] in 2010. Authors of this paper however presume (based on research) that in 10 years it could be possible to factorize $N = 1024$ problem.

5.2 Discrete logarithm problem

There is the same approach to measure the discrete logarithm problem tasks as it was in integer factorization. The measurement shows the relation among input size N (in bits) which represents bit size of modulus p , number of stations (nods) in system S and time t .

Figure 5.3: Graph of relation between time t and input size N (discrete logarithm)

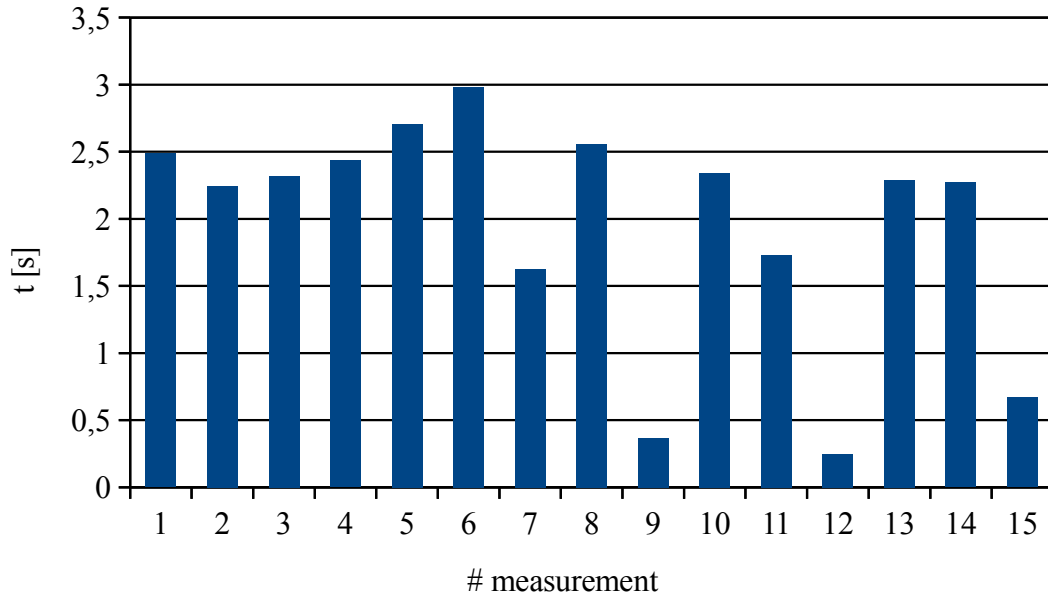


For illustration of the situation, there is 5.3 included (only error bar of $S = 1$ are included in graph). It is obvious that relation between time t and input size N is exponential. Relation is emphasized by using exponential regression (that would be lately used for analysis).

Discrete logarithm problem has much smoother characteristic than integer factorization one. But there are special situations that increase level of variance. This happen in case that number $\varphi(p) = p - 1$ is B -smooth for some small value of B , because Silver-Pohlig-Hellman algorithm is much more effective in this case. It means that modulus for which $p - 1$ could be factored to large number of relatively small factors is the best one to this method. This is shown in Figure 5.4 bellow. In this graph, there are measurements of time for one station and random data set of size $N = 40$ bits. There are obviously some values, such as 9, 12 and 15 that were solved in much shorter time. The Silver-Pohlig-Hellman algorithm requires $\sqrt{q_k}$ steps where $q_k = \max\{q \in \mathbb{P}, q \mid \varphi(p)\}$ (and could be faster in some situations). This is the

reason why there are so strict conditions to generating key pairs of Diffie-Hellman key exchange algorithm (or ElGamal cypher) where there is required to have $\varphi(p)$ composed of large prime numbers. The implementation of Silver-Pohlig-Hellman that is used in this application is the combination of this method with Baby-step giant-step method. This combination decrease time complexity of the method.

Figure 5.4: Graph of time for $N = 40$ and $S = 1$ (discrete logarithm)



There is a problem with memory requirements of Baby-step giant-step method that is crucial to time complexity. In native version of algorithm it requires \sqrt{p} values to be saved in memory. It is possible to relatively small values of p . Algorithm solve this problem by mapping only relatively tiny set of random values in interval $(0, \sqrt{p}) \cap \mathbb{Z}$. This otherwise increases time complexity of algorithm and variance of time necessary to solve of the task.

5.2.1 Real situations

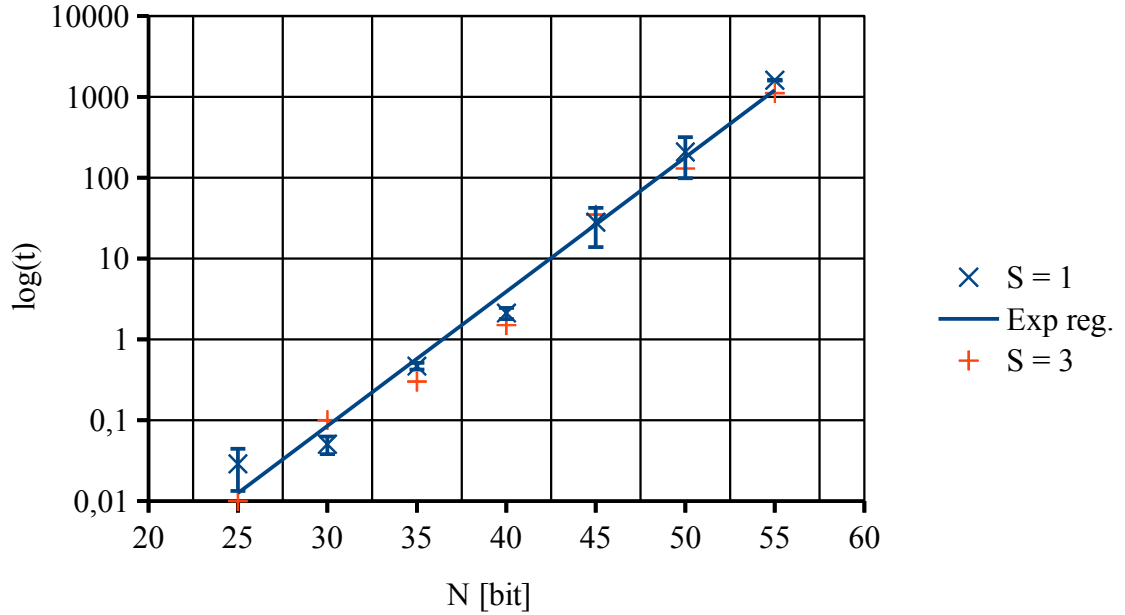
Hardness of solving discrete logarithm problem is fundamental thing in modern cryptography. Based on The Internet Engineering Task Force standard paper the minimal size of modulus p should be at least 512 bits (this is secure minimum till the year 1999), it is binding norm of The Internet Society [17]. Number p should be in format $p = jq + 1$ where $q \in \mathbb{P}$ is large prime number and $j \geq 2$. Recommended modulus size is however 1024 bits (since the year 1999). The reason why number p has format $p = jq + 1$ is that this makes Silver-Pohlig-Hellman method almost useless.

ElGamal is not so widely used cypher. But it is based on same idea like Diffie-Hellman key exchange method. So it is reasonable to presume that minimal modulus

size recommendation and also the algorithm for generating of modulus is the same as above.

The following approximations of time that is necessary to cryptanalysis of each cryptosystems are based on presumption that there is no mistake in implementation of methods (these are worst case analysis).

Figure 5.5: Graph of relation between time t and input size N (discrete logarithm)



The measured values are presented on graph 5.5, there is logarithmic scale of time t in seconds on vertical axis and there is size of input N (number of bits) on horizontal axes. Exponential regression that is used in graph 5.5 has equation:

$$t = 8.859638 \cdot 10^{-7} \exp(0.382359 \cdot N) \quad (5.2)$$

This equation is used in table 5.4 bellow.

Table 5.4: Approximation of time for solving N bits DL tasks

N [bit]	t [s]	t [year]	S
512	$9.29 \cdot 10^{78}$	$2.95 \cdot 10^{71}$	3
1024	$9.75 \cdot 10^{163}$	$3.09 \cdot 10^{156}$	3
2048	$1.07 \cdot 10^{334}$	$3.41 \cdot 10^{326}$	3
4096	$1.30 \cdot 10^{674}$	$4.13 \cdot 10^{666}$	3

Table 5.4 shows that time which is necessary to successful solving of discrete logarithm problem in real situation is enormous. There are no effective methods in 2017 that could reduce this time to some acceptable form. However, it is not impossible that there could be some discoveries in this area in the following years. For example, some usable improvements of cryptanalytic algorithm called Index calculus are shown in this [18] article. This could be the way how to achieve usable time complexity for algorithms for solving of discrete logarithm problem.

5.3 Summary

Although there is some improvement in numerical methods for cryptanalysis of public key cryptosystems, there is no method effective enough to be usable in real situation. It means that if someone uses public key cryptosystems in the recommended way (especially use of large keys generated by recommended pattern) it is still impossible to attack such cryptosystems nowadays. Using of complex distributed application does not change these facts at all. It is improvement in contrast to sequential approach but it is still limited in many ways.

The biggest improvement of past years is noticeable at methods for solving integer factorization problem (especially RSA). This is also the area where distributed application is fastest in cryptanalysis of given tasks. Although there are no such effective methods for the discrete logarithm problem, there is also some improvement over past years. The distributed application causes improvement also in this case.

Conclusion

The relation between public-key cryptography and solving of discrete logarithm or integer factorization problem is straightforward. The RSA cypher is based on integer factorization problem. On the other hand Diffie-Hellman key exchange or ElGamal cypher are based on discrete logarithm problem. The integer factorization problem could be defined in the following way: there is no effective algorithm for finding of all prime factors of given number in polynomial time. On the other hand discrete logarithm problem could be defined in the following way: there is no effective algorithm for solving congruence of form $g^x \equiv a \pmod{p}$ for given values g, a and p . If there would be any algorithm for solving these problems, each cryptosystem based on these problems would be useless.

There are no effective algorithm for solving of integer factorization in the way that solution could be found in polynomial time. However, there are algorithms that could find solution in subexponential time. The first algorithm that is described is called Pollard's rho, this algorithm does not works with usable level complexity but could be useful in some special situations (especially for numbers with great number of relatively small factors). Other described algorithms work with special pair of numbers such that $x^2 \equiv y^2 \pmod{n}$ and $x \not\equiv y \pmod{n}$ (so called Legendre's congruence). If such numbers are found there is probability which equals to $2/3$ that $\text{GCD}(x \pm y, n)$ is nontrivial divisor of n . First method that is based on Legendre's congruence is called Dixon's random squares method. It works with with fixed factor base composed of given prime numbers (this is the only parameter of method). The principle of this method is fundamental for all other methods. The straightforward improvement of Dixon's method is called Quadratic Sieve method. This is the fastest method for factorization of numbers upto approximately 100 digits. The most effective method for factoring of larger numbers is called General Number Field Sieve, this is rather complex method that uses many advanced issues of number theory.

There is similar situation in discrete logarithm problem. Methods for solving of this problem are less effective than the ones used for integer factorization. The most important method for solving of this problem is called Baby-step giant-step (or Shank's Baby-step giant-step) method. The biggest disadvantage of this method is that in native version it requires to save \sqrt{p} values (where p is modulus). Other method that partially improve Baby-step giant-step method is called Silver-Pohlig-Hellman algorithm. This algorithm is especially useful in situation where $(p-1)$ is composed of many relatively small factors. The last algorithm (and the most effective one) is called Index calculus. This algorithm theoretically has subexponential

level of complexity. But it has also many disadvantages that makes it almost useless in practical situations.

Distributed application that was created as practical part of thesis consists of two parts. The first is web application that represents master node of distributed system and the other part is a desktop (terminal) application that represents slave node of system. The web application is created in PHP in version 7.0 together with MySQL database management system. Both technologies represents the most popular ones for this purpose. The only purpose of web application is providing graphical interface for users of the system (especially for modifying of tasks in the system) and providing API for slave nodes, that have to obtain task to be solved and also send found solution. Both tasks and solutions are sent using HTTP protocol. Just for testing purposes there is also included random task generator in the web application.

The desktop application (slave node) is the part of application where there are implemented numerical methods for solving of integer factorization and discrete logarithm problem. It is written partially in Java SE programming language and it also used in some cases of integer factorization external application called msieve in slightly modified version that is written in C/C++ programming language. The application is designed to transform cryptographic task to solve of discrete logarithm or integer factorization problem. After the solving of this problem application decrypt the message or finds the shared secret key. The value of the key (or decrypted message) is afterward sent back to the server, where there is accessible via web application.

There is implementation of almost all relevant method for integer factorization in the application. There is also the functionality for working with matrices over \mathbb{Z}_2 in the application that includes operation such as finding reduce echelon form of matrix or the null space. This is necessary functionality for all complex methods. There is also implementation of Pollard's Rho method, Dixon's method and Quadratic Sieve method. Both Dixon's method and Quadratic Sieve method use the functionality for working with matrices over \mathbb{Z}_2 . Quadratic Sieve method needs also another functionality for working, such as Tonelli-Shanks algorithm that is also included in the application. Functionality for generating of method's parameters is also the part of the application. There is also some functionality that would be necessary for successful implementation of General Number Field Sieve method in the application.

The discrete logarithm problem is solved by all fundamental methods in the application. There is implementation of brute force method that could be useful for some simple tasks. The most important method is Shank's Baby-step giant-step method which is implemented in two variants. The first is native version that saves \sqrt{p} (where p is modulus of given congruence) values in memory during the running. This variant is not usable for real situations where the value of p is much bigger than the memory limits of the system. This problem is solved by using the probabilistic version of method that does not save all values, but only the random selection. Application also uses Silver-Pohlig-Hellman method for decrease the number of steps which are necessary to solve of discrete logarithm problem. There is also the implementation of Index calculus method in the application. This method requires

some advanced functionality such as working with matrix over \mathbb{Z}_n for some composed number n . This functionality for working with such matrices is also included in the application (together with methods for finding of reduce echelon form of given matrix and other functionality). There is also example of implementation of Pollard's rho method for discrete logarithm in the application (this method is useful during the pre-processing of task).

The analysis of using application in real situations shows that in case that cryptosystem is implemented without mistakes, it is almost impossible to be successful in cryptanalysis process. In the case of RSA there is requirement for the public key (modulus n) to be larger than 1024 bits and to be composed of two factor of size at least 512 bit. The only mistake in implementation of RSA that could happen is using predictable generator of pseudorandom prime numbers. Another potential mistake exists in the case of cryptosystems based on discrete logarithm problem. The modulus p using during the computation of discrete exponential has to be in the format in which $(p - 1)$ is composed of at least one large prime number. Otherwise there is a possibility of successful attack using Silver-Pohlig-Hellman method. The time which is necessary for computing of any real problem is still enormous.

The conception of application is that there is possibility of its expansion in the future. It is predicable that it is a question of time when the new effective methods will be discovered. Especially in the case of solving discrete logarithm problem there is visible progress during past few years. It could be also useful to analyze the potential of application on some embedded devices created for special purpose.

Bibliography

- [1] DELFS, Hans and Helmut KNEBL, 2015. *Introduction to Cryptography: Principles and Applications*. Third edition. Berlin: Springer.
- [2] Y. YAN, Song, Moti YUNG and John RIEF, 2013. *COMPUTATIONAL NUMBER THEORY AND MODERN CRYPTOGRAPHY*. Higher Education Press: Singapore. ISBN 9781118188583.
- [3] YAN, Song Y. *Number theory for computing*. 2nd ed. New York: Springer, 2002. ISBN 35-404-3072-5.
- [4] BRIGGS, Matthew E., 1998. *An Introduction to General Number Field Sieve*. Virginia Polytechnic Institute and State University. Online at: <https://vtechworks.lib.vt.edu/bitstream/handle/10919/36618/etd.pdf>
- [5] Dixon, J. D. (1981). DIXON, John D. Asymptotically fast factorization of integers. *Mathematics of Computation* [online]. 1981, **36**(153), 255-255 [cit. 2017-02-12]. DOI: 10.1090/S0025-5718-1981-0595059-1. ISSN 0025-5718.
- [6] ANDRÉN, Daniel, Lars HELLSTRÖM and Klas MARKSTRÖM. On the complexity of matrix reduction over finite fields. *Advances in Applied Mathematics* [online]. 2007, **39**(4), 428-452 [cit. 2017-02-12]. DOI: 10.1016/j.aam.2006.08.008. ISSN 01968858.
- [7] LANDQUIST, Eric. *The Quadratic Sieve Factoring Algorithm* [online]. 2001, 12 [cit. 2017-02-14]. Online at: <https://www.math.unl.edu/~mbrittenham2/classwk/445f08/dropbox/landquist.quadratic.sieve.pdf>
- [8] Shanks-Tonelli algorithm. *Planetmath.org* [online]. 2013 [cit. 2017-02-16]. Online at: <http://planetmath.org/sites/default/files/texpdf/30621.pdf>
- [9] SCHOOOF, René. *The Tonelli-Shanks algorithm* [online]. Roma: Università degli Studi di Roma Tor Vergata, 2008, , 1 [cit. 2017-02-18]. Online at: http://www.mat.uniroma2.it/~geo2/Shanks_Tonelli.pdf
- [10] Kvadratická rezidua. *Štěpán Holub* [online]. Prague: Department of Algebra, 2013 [cit. 2017-02-25]. Online at: <http://www.karlin.mff.cuni.cz/~holub/soubory/Rezidua.pdf>

- [11] YANG, Laurence T., Gaoyuan HUANG, Jun FENG and Li XU. Parallel GNFS algorithm integrated with parallel block Wiedemann algorithm for RSA security in cloud computing. *Information Sciences* [online]. 2017, **387**, 254-265 [cit. 2017-03-13]. DOI: 10.1016/j.ins.2016.10.017. ISSN 00200255.
- [12] YANG, Laurence T., Li XU, Sang-Soo YEO and Sajid HUSSAIN. An integrated parallel GNFS algorithm for integer factorization based on Linbox Montgomery block Lanczos method over $GF(2)$: note II. *Computers* [online]. 2010, **60**(2), 338-346 [cit. 2017-03-14]. DOI: 10.1016/j.camwa.2010.01.020. ISSN 08981221.
- [13] Msieve. *SourceForge.net* [online]. Cryptography, Mathematics: jasonp_sf, 2016 [cit. 2017-04-11]. Online at: <https://sourceforge.net/projects/msieve/>
- [14] VIVEK, Srinivas and C.E. VENI MADHAVAN. Cubic Sieve Congruence of the Discrete Logarithm Problem, and fractional part sequences. *Journal of Symbolic Computation* [online]. 2014, **64**, 22-34 [cit. 2017-03-15]. DOI: 10.1016/j.jsc.2013.12.004. ISSN 07477171.
- [15] Changes to our SSL Certificates. *Google Security Blog* [online]. Google, 2013 [cit. 2017-04-29]. Online at: <https://security.googleblog.com/2013/05/changes-to-our-ssl-certificates.html>
- [16] KLEINJUNG, Thorsten. *Factorization of a 768-bit RSA modulus: version 1.4, February 18, 2010* [online]. Netherlands, 2010 [cit. 2017-04-29]. Online at: <http://eprint.iacr.org/2010/006.pdf>
- [17] RESCORLA, Eric. *Diffie-Hellman Key Agreement Method* [online]. The Internet Engineering Task Force, 1999 [cit. 2017-04-30]. Online at: <https://www.ietf.org/rfc/rfc2631.txt>
- [18] PADMAVATHY, R. and Chakravarthy BHAGVATI. Discrete logarithm problem using index calculus method. *Mathematical and Computer Modelling* [online]. 2012, **55**(1-2), 161-169 [cit. 2017-04-30]. DOI: 10.1016/j.mcm.2011.02.022. ISSN 08957177.

List of all appendixes

There are following appendixes in the thesis:

1. Appendix A: Web application details
2. Appendix B: Desktop application details
3. Appendix C: List of files on appended CD

The CD is also appended to the thesis.

Appendix A: Web application details

The web application is available at the following URL:

```
http://www.mythesis.bid/
```

with the following sign-in information:

```
login:    admin
password: a1b456
```

User's guide

The User guide for web application is available after successful sign-in to system in left menu bar (item **User guide**).

Installation instruction

Instruction for installation of web application are available in file `installGuide.txt` that is located in folder with installation files.

Documentation of program source

Each used method and class in method is documented in source files.

Appendix B: Desktop application details

Desktop application files are available in directory `SaFaDl` of appended CD.

Installation instruction

The manual of how to install desktop application and how to run it is available in file `installSaFaDl.txt` which is in directory `SaFaDl/dist`.

The program `msieve` (which has modified outputs) is available in directory `SaFaDl/dist/modmsieve`.

Documentation of program source

The source files of application are available in directory `SaFaDl/src`.

The javadoc file (documentation of written source code) is available in directory `SaFaDl/dist/javadoc`.

Appendix C: List of files on appended CD

There are following relevant directories and files on CD:

- [SALACdipl.pdf](#) – diploma thesis
- [mythesis.bid](#) – directory of web application
 - [installGuide.txt](#) – installation guide of web application
- [SaFaDI](#) – directory of desktop application
 - [dist](#) – directory with executable file
 - * [modmsieve](#) – modified msieve application
 - * [javadoc](#) – documentation of application
 - * [installSaFaDI.txt](#) – installation guide of desktop application
 - [src](#) – source codes of application