The security of ciphers and cryptographic protocols is based on hard mathematical problems for which algorithms that solve these problems with sufficiently low computational complexities are not known. Examples of such problems are factoring large integers, discrete logarithm problem, solving system of polynomial equations over a finite field, and others. Solving such problems implies breaking the security of the corresponding ciphers or protocols. The aim of this bachelor's thesis is to describe at least three such mathematical problems used in cryptography. For each problem, the student will:
- describe the problem in detail,
- provide a list of ciphers and protocols briefly described, the security of which is based on the problem,
- install available programs that solve the problem, and compare them (e.g., scalability, speed) with the tools implemented in Magma.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Hard Mathematical Problems in Cryptography

Marek Holík

Department of Information Security
Supervisor: Mgr. Martin Jureček, Ph.D.

May 11, 2022

Acknowledgements

I would like to thank Mgr. Martin Jureček, Ph.D. for his both friendly and professional attitude and I am grateful for all his remarks and recommendations for this thesis. I also thank my family for supporting me.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 11, 2022 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2022 Marek Holík. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Holík, Marek. *Hard Mathematical Problems in Cryptography.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstract

This thesis describes protocols and ciphers which base their security on integer factorization problem, quadratic residuosity problem and discrete logarithm problem. Random instances of mathematical problems are generated based on the described protocols and ciphers for comparison of Magma, SageMath and MATLAB implementations. Efficiency is evaluated in terms of average time needed and success rate for solving instances of a fixed length. The time for solving one instance is limited to one hour.

Keywords integer factorization problem, quadratic residuosity problem, discrete logarithm problem, Magma, SageMath, MATLAB

Abstrakt

Práce představuje šifry a protokoly založené na problému faktorizace čísel, problému kvadratických residuí a problému disktrétního logarithmu. Náhodné instance matematických problémů jsou generovány na základě popsaných protokolů a šifer pro porovnání Magma, SageMath a MATLAB implementací. Efektivita je vyhodnocena základě průměrného času potřebného k vyřešení a úspěšnost algoritmu vydat správný výsledek pro instance dané délky. Čas na vyřešení jedné instance je omezen na jednu hodinu.

Klíčová slova problém faktorizace čísel, problém kvadratických residuí, problém diskrétního logaritmu, Magma, SageMath, MATLAB

Contents

List of Algorithms

List of Tables

Introduction

Most of us rely on the fact that our communication over the internet is secure whenever we connect to a website, use social media applications, play games online or use online bank services. Such features would not be available if it were not for cryptographic protocols mainly encryption protocols ensuring that only the sender and the intended receivers can read the transmitted data and signature protocols enabling the intended receivers to verify that the transmitted data had not been modified by a third party.

Encryption and signatures can be done with symmetric cryptography. Symmetric cryptography uses the same key for operations such as encryption and decryption. The same key for such operations means it cannot be transmitted over an unprotected network and it is not always the case that the sender and the receiver can exchange the key in private. Such problems can be solved by using asymmetric cryptography.

Asymmetric cryptography offers protocols enabling to transfer some "parts" of the key over an unprotected network such as a key for encryption or verification while other key "parts" are kept in private such as a key for decryption or signature. In addition, asymmetric cryptography offers some new concepts such as zero-knowledge proofs enabling a party to prove knowledge of some information without disclosing the information and blind signatures enabling a party to sign data without knowing the data.

Asymmetric cryptography protocols and ciphers are based on "hard" mathematical problems enabling to make some of the information public, usually, a public key, while keeping some other information private, usually a private key. Although the public and private key pair is computed based on the same information a third party not having this information is unable to derive the private key from the public key in a feasible time.

The inability of such computations is uncertain as no efficient algorithms exist however it has never been proven that such algorithms cannot exist. Therefore it is of great interest to study algorithms solving these problems as well as protocols and ciphers based on these problems to ensure that the

protocols are still secure.

This thesis aims to define at least three mathematical problems and introduce protocols and ciphers basing their security on the particular mathematical problem. Furthermore, Magma and other available software implementing algorithms solving these problems are compared in terms of time efficiency and success rate.

Integer factorization problem, quadratic residuosity problem and discrete logarithm problem are chosen as hard mathematical problems. Each of the first three chapters is dedicated to one of the chosen problems. Protocols and ciphers are presented as pseudocodes and rarely additional requirements for a protocol are specified.

The fourth chapter and the fifth chapter contain the practical part of the thesis. Magma, SageMath and MATLAB implementations are tested on randomly generated instances of the mathematical problems used in described protocols. In the real world, the instances of the problems are chosen so that they cannot be solved even with enormous time and computational power resources. In this thesis both time and computational power resources are limited and the instances are chosen such that solving a single instance does not take more than one hour.

Notation

Mathematical notation

This section introduces the notation used in the thesis.

Carmichael's function λ

Let $\lambda(n)$ be the smallest integer such that $k^{\lambda(n)} \equiv 1 \pmod{n}$ for all $k < n$ and relatively prime to n [1].

Euler totient function ϕ

Let $n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k}$ then $\phi(n) = n\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\cdots\left(1 - \frac{1}{p_k}\right)$ where $p_1, p_2, \ldots, p_k$ are pairwise distinct primes and $\alpha_1, \alpha_2, \ldots, \alpha_k \in \mathbb{N}$ [2].

Jacobi symbol

Let m be and odd integer and a any integer Jacobi symbol is defined by $\left(\frac{a}{m}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1}\left(\frac{a}{p_2}\right)^{\alpha_2}\cdots\left(\frac{a}{p_k}\right)^{\alpha_k}$ where $m = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k}$ [3].

Legendre symbol

Let q be an odd prime and a be any integer.
If $q|a$ then $\left(\frac{a}{q}\right) = 0$.
If $q \nmid a$ then $\left(\frac{a}{q}\right) = 1$ if there is N such that $q|N^2 - a$
else $\left(\frac{a}{q}\right) = -1$ [2].

Operation $\oplus$

Symbol $\oplus$ is notation for bitwise xor.

Digital signature schemes

$\mathcal{M}$ is the set of elements to which a signer can affix a digital signature.
$\mathcal{M}_{\mathcal{S}}$ is the set of elements to which the signature transformations are applied.
$\mathcal{M}_{\mathcal{R}}$ is the image of a redundancy function $\mathcal{R}$ where $\mathcal{R}$ is one-to-one mapping from $\mathcal{M}$ to $\mathcal{M}_{\mathcal{S}}$.
$\mathcal{Q}_n$ be the set of all quadratic residues in $\mathbb{Z}_n$ [4].

Appendix scheme

Appendix scheme can only verify message m. If a scheme uses a redundancy function $\mathcal{R}$, $\mathcal{R}$ is typically selected as collision-free hash function [4].

Message recovery scheme

Message recovery scheme recovers the original message m from the signature s. Redundancy function $\mathcal{R}$ and $\mathcal{R}^{-1}$ are public knowledge. Suitable choice of R is critical to security of the scheme and should be chosen such that $\mathcal{M}_{\mathcal{R}}$ should be much smaller than $\mathcal{M}_{\mathcal{S}}$. If $\mathcal{M}_{\mathcal{R}} \approx \mathcal{M}_{\mathcal{S}}$ then it is probable that a modified message m' can be transformed by $\mathcal{R}^{-1}(m')$ and the signature seems valid [4].

Chapter 1

Integer factorization

1.1 Description

Definition 1.1.1 (Integer factorization problem)**.** Given a positive integer n, find its prime factorization; that is, write $n = p_1^{e_1} \cdot p_2^{e_2} \cdots p_k^{e_k}$ where the p_i are pairwise distinct primes and each $e_i \geq 1$ [4].

1.2 Ciphers and protocols

In this section protocols and ciphers basing their security on integer factorization problem are described.

1.2.1 RSA encryption scheme

RSA encryption scheme is named after its authors – Ron Rivest, Adi Shamir and Leonard Adleman [4]. This scheme was however first invented at Government Communications Headquarters by Clifford Cocks in 1973, four years before Rivest, Shamir and Adleman came up with this idea. This was kept secret and after 24 years, in the year 1997, the true history of RSA was announced by Cocks [5].

Algorithm 1.1 [4] generates a pair of a private key and a public key.

Algorithm 1.1 RSA key generation

Output: private key K_s, public key K_p

1: Choose two distinct primes p and q
2: $n \leftarrow pq$, $\phi \leftarrow (p-1)(q-1)$
3: Choose e, $1 < e < \phi$ such that $\gcd(e, \phi) = 1$
4: Compute d such that $ed \equiv 1 \pmod{\phi}$
5: Private key $K_s \leftarrow (n, d)$
6: Public key $K_p \leftarrow (n, e)$

Algorithm 1.2 [4] encrypts a chosen message using the public key generated by Algorithm 1.1 and outputs a ciphertext.

Algorithm 1.2 RSA encryption

Input: public key K_p
Output: ciphertext c
1: Choose a message m, $0 \leq m < n$
2: Ciphertext $c \leftarrow m^e \pmod{n}$

Choosing a message $m = 0$ or $m = 1$ will result in ciphertext $c = m$.

Algorithm 1.3 [4] decrypts the ciphertext generated by Algorithm 1.2 using the private key generated by Algorithm 1.1 and outputs the original message.

Algorithm 1.3 RSA decryption

Input: ciphertext c, private key K_s
Output: message m
1: Message $m \leftarrow c^d \pmod{n}$

1.2.2 RSA signature scheme

RSA signature scheme is a message recovery signature scheme. This scheme is based on the RSA encryption scheme, the difference is that the order of encryption and decryption is reversed and a redundancy function $\mathcal{R} : \mathcal{M} \rightarrow \mathbb{Z}_n$ is used [4].

RSA signature scheme key generation is the same as in Algorithm 1.1. Algorithm 1.4 [4] signs a chosen message by using the private key generated by Algorithm 1.1 and outputs a signature.

Algorithm 1.4 RSA signature

Input: private key K_s
Output: signature s
1: Choose a message m
2: $\tilde{m} \leftarrow \mathcal{R}(m)$
3: Signature $s \leftarrow \tilde{m}^d \pmod{n}$

Algorithm 1.5 [4] verifies the signature generated by Algorithm 1.4 using the private key generated in Algorithm 1.1 and outputs the original message if the signature is valid.

Algorithm 1.5 RSA verification

Input: public key K_p, signature s
Output: message m

1: $\tilde{m} \leftarrow s^e \pmod{n}$
2: Verify that $\tilde{m} \in \mathcal{M}_{\mathcal{R}}$
3: Message $m \leftarrow \mathcal{R}^{-1}(\tilde{m})$

1.2.3 RSA pseudorandom bit generator

RSA pseudorandom bit generator is a cryptographically secure pseudorandom bit generator [4].

Algorithm 1.6 [4] generates a pseudorandom sequence of bits.

Algorithm 1.6 RSA pseudorandom bit generation

Output: pseudorandom sequence $(z_1, z_2, \ldots, z_l)$

1: Choose two distinct primes p and q
2: $n \leftarrow pq$, $\phi \leftarrow (p-1)(q-1)$
3: Choose e, $1 < e < \phi$ such that $\gcd(e, \phi) = 1$
4: Choose a random integer x_0, $1 \leq x_0 \leq n-1$
5: **for** $i \leftarrow 1, 2, \ldots, l$ **do**
6: $\quad x_i \leftarrow {x_{i-1}}^e \pmod{n}$
7: $\quad z_i \leftarrow$ the least significant bit of x_i
8: **end for**

Choosing $x_0 = 1$ will result in generating a constant sequence of 1's and choosing $x_0 = n - 1$ will result in a constant sequence of -1's.

1.2.4 Dependent-RSA encryption scheme

David Pointcheval proposed three encryption schemes based on RSA in 1999 [6]. The author formulates a new algebraic problem called the Dependent RSA problem. Dependent RSA problem is defined as: given an element $\alpha \in \mathbb{Z}_n^*$, a composite modulus n and an exponent e relatively prime to $\phi(n)$, find $(a+1)^e \pmod{n}$ where $\alpha = a^e \pmod{n}$. All of the three schemes are semantically secure meaning that a ciphertext does not leak any information about the plaintext except the length.

Dependent-RSA key generation is same as in Algorithm 1.1. Algorithm 1.7 [6] encrypts a chosen message using the public key generated by Algorithm 1.1 and outputs a ciphertext.

Algorithm 1.7 Dependent-RSA encryption

Input: public key K_p
Output: ciphertext c

1: Choose a message m, $0 \leq m < n$
2: Choose a random integer $k \in \mathbb{Z}_n^*$
3: $A \leftarrow k^e \pmod{n}$
4: $B \leftarrow m(k+1)^e \pmod{n}$
5: Ciphertext $c \leftarrow (A, B)$

Choosing a message $m = 0$ will result in $B = 0$ and if $B = 0$ then either $m = 0$ or $\gcd(k + 1, n) > 1$. Choosing k such that $\gcd(k, n) > 1$ will result in obtaining a different message instead of the original message m in the decryption phase.

Algorithm 1.8 [6] decrypts the ciphertext generated by Algorithm 1.7 using the private key generated by Algorithm 1.1 and outputs the original message.

Algorithm 1.8 Dependent-RSA decryption

Input: private key K_p, ciphertext c
Output: message m

1: $k \leftarrow A^d \pmod{n}$
2: Message $m \leftarrow B/(k+1)^e \pmod{n}$

1.2.5 DRSA-1 encryption scheme

DRSA-1 encryption scheme was proposed by David Pointcheval in 1999. This scheme is based on the Dependent RSA problem and is semantically secure [6].

DRSA-1 key generation is the same as in Algorithm 1.1. Algorithm 1.9 [6] encrypts a chosen message using the public key generated by Algorithm 1.1 and outputs a ciphertext. Algorithm 1.9 uses a hash function $\mathcal{H} : \mathbb{Z}_n \times \mathbb{Z}_n \rightarrow \{0, 1\}^l$ where l is a security parameter.

Algorithm 1.9 DRSA-1 encryption

Input: public key K_p
Output: ciphertext c

1: Choose a message m, $0 \leq m < n$
2: Choose a random integer $k \in \mathbb{Z}_n^*$
3: $A \leftarrow k^e \pmod{n}$
4: $B \leftarrow m(k+1)^e \pmod{n}$
5: $H \leftarrow \mathcal{H}(m, k)$
6: Ciphertext $c \leftarrow (A, B, H)$.

Algorithm 1.10 [6] verifies and decrypts the ciphertext generated by Algorithm 1.9 using the private key and the public key generated by Algorithm 1.1 and outputs the original message on successful verification.

Algorithm 1.10 DRSA-1 decryption

Input: private key K_s, public key K_p, ciphertext c
Output: message m

1: $k \leftarrow A^d \pmod{n}$
2: Message $m \leftarrow B/(k+1)^e \pmod{n}$
3: Verify that $H = \mathcal{H}(m, k)$.

1.2.6 DRSA-2 encryption scheme

DRSA-2 encryption scheme was proposed by David Pointcheval in 1999. This scheme is based on the Dependent RSA problem and is semantically secure [6].

DRSA-2 key generation is same as in Algorithm 1.1. Algorithm 1.11 [6] encrypts a chosen message using the public key generated by Algorithm 1.1. Algorithm 1.11 uses a hash function $\mathcal{H}_1 : \mathbb{Z}_n \rightarrow \{0,1\}^{k_1}$, where k_1 is the size of the message and a hash function $\mathcal{H}_2 :\rightarrow \{0,1\}^{k_2}$ where k_2 is a security parameter.

Algorithm 1.11 DRSA-2 encryption

Input: public key K_p
Output: ciphertext c

1: Choose a message m, $0 \leq m < n$
2: Choose a random integer $k \in \mathbb{Z}_n^*$
3: $A \leftarrow k^e \pmod{n}$
4: $B \leftarrow m \oplus \mathcal{H}_1((k+1)^e \pmod{n})$
5: $H \leftarrow \mathcal{H}_2(m, k)$
6: Ciphertext $c \leftarrow (A, B, H)$

Algorithm 1.12 [6] verifies and decrypts the ciphertext generated by Algorithm 1.11 using the private key and the public key generated by Algorithm 1.1 and outputs the original message on successful verification.

Algorithm 1.12 DRSA-2 decryption

Input: private key K_s, public key K_p, ciphertext c
Output: message m

1: $k \leftarrow A^d \pmod{n}$
2: Message $m \leftarrow B \oplus \mathcal{H}_1((k+1)^e \pmod{n})$
3: Verify that $H = \mathcal{H}_2(m, k)$

1.2.7 Chaum blind signature scheme

Chaum blind signature scheme was proposed by David Chaum who invented the notion of blind signatures and blind unanticipated signatures. This scheme [7] is based on the RSA algorithm.

Chaum key generation is the same as in Algorithm 1.1. Algorithm 1.13 [7] enables B to sign the message of A without revealing the message to B and outputs a signature. B uses a private key, A uses a public key, and both keys are generated by Algorithm 1.1.

Algorithm 1.13 Chaum signature

Input: private key K_s, public key K_p
Output: signature s

1: A chooses a message m and a random integer k, $1 < k < n$
2: A computes $t \leftarrow mk^e \pmod{n}$
3: B signs the blind message by $s_b \leftarrow t^d \pmod{n}$
4: A unblinds s_b and obtains signature s by $s \leftarrow s_b/k \pmod{n}$

Chaum blind signature is based on RSA therefore the message m should be $0 \leq m < n$. Choice of $m = 0$ will result in $t = 0$ and if $t = 0$ then either $m = 0$ or $\gcd(k, n) > 1$. Choice of k such that $\gcd(k, n) > 1$ will result in obtaining a different signature for the message m and consequently obtaining a different message in the verification phase.

The author in [7] does not specify any verification algorithm. Algorithm 1.14 verifies the signature generated by Algorithm 1.13 using the public key generated by Algorithm 1.1. The verification algorithm requires the original message.

Algorithm 1.14 Chaum verification

Input: public key K_p, message m, signature s

1: $m' \leftarrow s^e \pmod{n}$
2: Verify that $m' = m$

1.2.8 Rabin encryption scheme

Rabin encryption scheme was proposed by Michael C. Rabin. This scheme is based on the intractability of finding square roots modulo a composite number which is equivalent to factoring [7].

Algorithm 1.15 [7] generates a pair of a private key and a public key.

Algorithm 1.15 Rabin key generation

Output: private key K_s, public key K_p

1: Choose two distinct primes p and q such that $p \equiv q \equiv 3 \pmod 4$
2: $n \leftarrow pq$
3: Private key $K_s \leftarrow (p, q)$
4: Public key $K_p \leftarrow n$

Algorithm 1.16 [7] encrypts a chosen message using the public key generated by Algorithm 1.15 and outputs a ciphertext.

Algorithm 1.16 Rabin encryption

Input: public key K_p
Output: ciphertext c

1: Choose a message m, $m < n$
2: Ciphertext $c \leftarrow m^2 \pmod n$

Choosing a message $m = 0$ or $m = 1$ will result in ciphertext $c = m$.

Algorithm 1.17 [7] decrypts the ciphertext generated by Algorithm 1.16 using the private key generated by Algorithm 1.15 and outputs the original message.

Algorithm 1.17 Rabin decryption

Input: private key K_s, ciphertext c
Output: message m

1: $m_1' \leftarrow c^{(p+1)/4} \pmod p$
2: $m_2' \leftarrow p - c^{(p+1)/4} \pmod p$
3: $m_3' \leftarrow c^{(q+1)/4} \pmod q$
4: $m_4' \leftarrow q - c^{(q+1)/4} \pmod q$
5: $a \leftarrow q(q^{-1} \pmod p)$
6: $b \leftarrow p(p^{-1} \pmod q)$
7: $m_1 \leftarrow am_1' + bm_3' \pmod n$
8: $m_2 \leftarrow am_1' + bm_4' \pmod n$
9: $m_3 \leftarrow am_2' + bm_3' \pmod n$
10: $m_4 \leftarrow am_2' + bm_4' \pmod n$
11: One of the four results m_1, m_2, m_3, m_4 is the original message m

Determining which m_i is the original message is easy if the original message is in human language however if the original message is a random bit string there is no way to determine the original message [7].

1.2.9 Rabin signature scheme

Rabin signature scheme [4] is a message recovery scheme proposed by Michael C. Rabin. As in the RSA signature scheme, the redundancy function is critical

to the security of the scheme. In practice choosing a redundancy function for this scheme is a hard task as the image of every possible message must be a quadratic residue modulo n.

Algorithm 1.18 [4] generates a pair of a private key and a public key.

Algorithm 1.18 Rabin key generation

Output: private key K_s, public key K_p

1: Choose two distinct primes p and q
2: $n \leftarrow pq$
3: Private key $K_s \leftarrow (p, q)$
4: Public key $K_p \leftarrow n$

Algorithm 1.19 [4] signs a chosen message using the private key generated by Algorithm 1.18 and outputs a signature.

Algorithm 1.19 Rabin signature

Input: private key K_s
Output: signature s

1: Choose a message m
2: $\tilde{m} \leftarrow \mathcal{R}(m)$ where $\mathcal{R} : \mathcal{M} \rightarrow \mathcal{Q}_n$
3: Signature $s \leftarrow \text{sqrt}(\tilde{m}) \pmod{n}$

Algorithm 1.20 [4] verifies the signature generated by Algorithm 1.19 using the public key generated by Algorithm 1.18 and outputs the original message on successful verification.

Algorithm 1.20 Rabin verification

Input: public key K_p, signature s
Output: message m

1: $\tilde{m} \leftarrow s^2 \pmod{n}$
2: Verify that $\tilde{m} \in \mathcal{M}_{\mathcal{R}}$.
3: Message $m \leftarrow \mathcal{R}^{-1}(\tilde{m})$

1.2.10 Modified Rabin signature scheme

Modified Rabin signature scheme [4] is a message recovery scheme and is more useful in practice than the original Rabin signature scheme as this scheme does not require a redundancy function.

Algorithm 1.21 [4] generates a pair of a private key and a public key.

Algorithm 1.21 Modified Rabin key generation

Output: private key K_s, public key K_p

1: Select two primes $p \equiv 3 \pmod 8$ and $q \equiv 7 \pmod 8$
2: $n \leftarrow pq$
3: $d \leftarrow (n - p - q + 5)/8$.
4: Private key $K_s \leftarrow d$
5: Public key $K_p \leftarrow n$

Algorithm 1.22 [4] signs a chosen message using the private key generated by Algorithm 1.21 and outputs a signature.

Algorithm 1.22 Modified Rabin signature

Input: private key K_s
Output: signature s

1: Choose a message $m \in \mathbb{Z}_n$ such that $m \leq \lfloor (n-6)/16 \rfloor$
2: $\tilde{m} \leftarrow 16m + 6$
3: $J \leftarrow \left(\frac{\tilde{m}}{n}\right)$
4: **if** $J = 1$ **then**
5: Signature $s \leftarrow \tilde{m}^d \pmod n$
6: **else if** $J = -1$ **then**
7: Signature $s \leftarrow (\tilde{m}/2)^d \pmod n$
8: **end if**

Algorithm 1.23 [4] verifies the signature generated by Algorithm 1.22 using the public key generated by Algorithm 1.21 and outputs the original message on successful verification.

Algorithm 1.23 Modified Rabin verification

Input: public key K_p, signature s
Output: message m

1: $m' \leftarrow s^2 \pmod n$
2: **if** $m' \equiv 6 \pmod 8$ **then**
3: $\tilde{m} \leftarrow m'$
4: **else if** $m' \equiv 3 \pmod 8$ **then**
5: $\tilde{m} \leftarrow 2m'$
6: **else if** $m' \equiv 7 \pmod 8$ **then**
7: $\tilde{m} \leftarrow n - m'$
8: **else if** $m' \equiv 2 \pmod 8$ **then**
9: $\tilde{m} \leftarrow 2(n - m')$
10: **end if**
11: Verify that $\tilde{m} \equiv 6 \pmod{16}$
12: Message $m \leftarrow (\tilde{m} - 6)/16$.

1.2.11 Rabin oblivious transfer scheme

Rabin oblivious transfer scheme [7] was proposed by Michael C. Rabin. This scheme enables A to send a message to B with 50% chance of success and A does not know whether the message has been sent successfully. This scheme was not proven to be secure and there may be a way for B to gain some information about the message even if the transfer was not successful [8].

Rabin key oblivious transfer key generation is same as in Algorithm 1.18. Algorithm 1.24 [7] enables A to send two primes p and q to B with the probability of 50%.

Algorithm 1.24 Rabin oblivious transfer

Input: private key K_s, public key K_p

1: B chooses a random integer x, $x < n$ such that $\gcd(x, n) = 1$
2: B sends $a \leftarrow x^2 \pmod{n}$ to A
3: A then computes roots of a: x, $n - x$, y and $n - y$, randomly chooses one of them and sends it to B
4: If B receives y or $n-y$, B obtains either p or q by computing $\gcd(x+y, n)$

This scheme has a weakness – there is a possibility that B can compute an a such that it is possible to compute factor of n in all cases [7].

1.2.12 Williams encryption scheme

Hugh Williams redefined Rabin encryption scheme that the decryption is no longer ambiguous [7]. This scheme is proven to be as secure as factoring [9].

Algorithm 1.25 [7] generates a pair of a private key and a public key.

Algorithm 1.25 Williams key generation

Output: private key K_s, public key K_p

1: Choose two primes $p \equiv 3 \pmod 8$ and $q \equiv 7 \pmod 8$
2: $n \leftarrow pq$.
3: Choose s such that $\left(\frac{s}{n}\right) = -1$
4: $k \leftarrow 1/2 * (1/4 + (p-1)(q-1) + 1)$
5: Private key $K_s \leftarrow k$
6: Public key $K_p \leftarrow (n, s)$

Algorithm 1.26 [7] encrypts a chosen message using the private key generated by Algorithm 1.25 and outputs a ciphertext.

Algorithm 1.26 Williams encryption

Input: public key K_p
Output: ciphertext c

1: Choose a message m, $m < n$
2: Compute c_1 such that $\left(\frac{m}{n}\right) = (-1)^{c_1}$
3: $m' \leftarrow s^{c_1} m \pmod{n}$
4: $c_m \leftarrow m'^2 \pmod{n}$
5: $c_2 \leftarrow m' \pmod{2}$.
6: Ciphertext $c \leftarrow (c_m, c_1, c_2)$.

Message m should be chosen such that $\gcd(m, n) = 1$, otherwise $\left(\frac{m}{n}\right) = 0$ and c_1 such that $\left(\frac{m}{n}\right) = (-1)^{c_1}$ does not exist.

Algorithm 1.27 [7] decrypts the ciphertext generated by Algorithm 1.26 using the private key generated by Algorithm 1.25.

Algorithm 1.27 Williams decryption

Input: private key K_s, ciphertext c
Output: message m

1: Compute m'' such that ${c_m}^k \equiv \pm m'' \pmod{n}$
2: Message $m \leftarrow s^{c_1}(-1)^{c_1} m'' \pmod{n}$

1.2.13 ESIGN

ESIGN was proposed by Atsushi Fujioka, Tatsuaki Okamoto, Shoji Miyaguchi. This scheme was patented in the United States, Canada, England, France, Germany and Italy. Due to the efficiency of the scheme, it is suitable for smart card implementation [7, 10].

Algorithm 1.28 [10] generates a pair of a private key and a public key.

Algorithm 1.28 ESIGN key generation

Output: private key K_s, public key K_p

1: Choose two distinct primes p and q
2: $n \leftarrow p^2 q$
3: Private key $K_s \leftarrow (p, q)$
4: Public key $K_p \leftarrow n$

Algorithm 1.29 [10] signs a chosen message using the private key generated by Algorithm 1.28 and outputs a signature.

Algorithm 1.29 ESIGN signature

Input: private key K_s
Output: signature s

1: Choose a message m
2: Choose a random integer x, $x < pq$
3: $w \leftarrow \lceil (\mathcal{H}(m) - x^k \pmod n)/pq \rceil$ where $\mathcal{H}$ is a hash function such that $\mathcal{H}(m) \in \mathbb{Z}_n$ for any positive integer m and $k \geq 4$
4: $y \leftarrow w/(kx^{k-1}) \pmod p$
5: Signature $s \leftarrow x + ypq$

The recommended values for security parameter k are 8, 16, 32, 64, 128, 256, 512, 1024 [7].

Algorithm 1.30 [10] verifies the signature generated by Algorithm 1.29 using the public key generated by Algorithm 1.28. The verification algorithm requires the original message.

Algorithm 1.30 ESIGN verification

Input: public key K_p, signature s, message m

1: Verify that $\mathcal{H}(m) \leq s^k \pmod n < \mathcal{H}(m) + 2^{2/3(\lfloor \log_2(n) \rfloor + 1)}$

1.2.14 Schmidt-Samoa encryption scheme

Schmidt-Samoa encryption scheme [11] was proposed by Katja Schmidt-Samoa. Author presents a new trapdoor permutation with integers of type p^2q.

Algorithm 1.31 [12] generates a pair of a private key and a public key.

Algorithm 1.31 Schmidt-Samoa key generation

Output: private key K_s, public key K_p

1: Choose two distinct primes p and q
2: $n \leftarrow p^2q$
3: $d \leftarrow n^{-1} \pmod{\text{lcm}(p-1, q-1)}$
4: Private key $K_s \leftarrow (p, q, d)$
5: Public key $K_p \leftarrow n$

Algorithm 1.32 [12] encrypts a chosen message using the public key generated by Algorithm 1.31 and outputs a ciphertext.

Algorithm 1.32 Schmidt-Samoa encryption

Input: public key K_p
Output: ciphertext c

1: Choose a message m
2: Ciphertext $c \leftarrow m^n \pmod n$

The authors in [12] dot no specify any requirements for choosing the message m however $m = 0$ or $m = 1$ will result in ciphertext $c = m$ and should m be chosen as $m \geq n$, the decryption is ambiguous.

Algorithm 1.33 [12] decrypts the ciphertext generated by Algorithm 1.32 using the private key generated by Algorithm 1.31 and outputs the original message.

Algorithm 1.33 Schmidt-Samoa decryption

Input: private key K_s, ciphertext c
Output: message m
1: Message $m \leftarrow c^d \pmod{pq}$.

1.2.15 Schmidt-Samoa signature scheme

Schmidt-Samoa signature scheme [12] is based on Schmidt-Samoa encryption scheme as the only difference is that the order of encryption and decryption is reversed.

Schmidt-Samoa key generation is same as in Algorithm 1.31. Algorithm 1.34 [12] signs a chosen message using the private key generated by Algorithm 1.31 and outputs a signature.

Algorithm 1.34 Schmidt-Samoa signature

Input: private key K_s
Output: signature s
1: Choose a message m
2: Signature $s \leftarrow m^d \pmod{n}$

Algorithm 1.35 [12] verifies the signature generated by Algorithm 1.34 using the public key generated by Algorithm 1.31. The verification algorithm requires the original message.

Algorithm 1.35 Schmidt-Samoa verification

Input: public key K_p, message m, signature s
1: Verify that $m = s^n \pmod{n}$

1.2.16 Benaloh encryption scheme

Benaloh encryption scheme [13] was proposed by Josh Benaloh in 1994. In the second chapter Goldwasser-Micali probabilistic encryption scheme will be introduced which is a special case of this encryption scheme with $r = 2$.

Algorithm 1.36 [14] generates a pair of a private key and a public key.

Algorithm 1.36 Benaloh key generation

Output: private key K_s, public key K_p

1: Choose two distinct primes p and q
2: $n \leftarrow pq$, $\phi \leftarrow (p-1)(q-1)$
3: Choose block size r such that:
4: $\quad r \mid p-1$
5: $\quad \gcd(r, (p-1)/r) = 1$
6: $\quad \gcd(r, q-1) = 1$
7: Choose y such that $x \leftarrow y^{\phi/r} \pmod{n} \neq 1$
8: Private key $K_s \leftarrow (p, q, x)$
9: Public key $K_p \leftarrow (y, r, n)$

Algorithm 1.37 [14] encrypts a chosen message using the public key generated by Algorithm 1.36 and outputs a ciphertext.

Algorithm 1.37 Benaloh encryption

Input: public key K_p
Output: ciphertext c

1: Choose a message m and an integer u such that $1 \leq u < n$
2: Ciphertext $c \leftarrow y^m u^r \pmod{n}$.

Choosing a message $m \geq \phi$ will result in ambiguous decryption. Choosing y or u such that $\gcd(y, n) \neq 1$ or $\gcd(u, m) \neq 1$ may result in ciphertext $c = 0$ and therefore unability to decrypt. Choosing $y = 0$ will result in ciphertext $c = 0$.

Algorithm 1.38 [14] decrypts the ciphertext generated by Algorithm 1.37 using the private key generated by Algorithm 1.36 and outputs the original message.

Algorithm 1.38 Benaloh decryption

Input: private key K_s, ciphertext c
Output: message m

1: $a \leftarrow c^{\phi/r} \pmod{n}$
2: $e \leftarrow 0$
3: **while** $x^e \pmod{n} \neq a$ **do**
4: $\quad e \leftarrow e + 1$
5: **end while**
6: Message $m \leftarrow e$.

1.2.17 Djebaili-Melkemi encryption scheme

Djebaili-Melkemi encryption scheme [15] was proposed by Karima Djebaili and Lamine Melkemi in 2020. This scheme is based on decisional generator

problem: Given $a, b, f, g \in \mathbb{Z}_n^*$ determine whether $f \in < a >$ and $g \in < b >$. Decisional generator problem is based on integer factorization problem.

Algorithm 1.39 [15] generates a pair of a private key and a public key.

Algorithm 1.39 Djebaili-Melkemi key generation

Output: private key K_s, public key K_p

1: Choose two distinct primes p and q
2: $n \leftarrow pq$
3: $\alpha \leftarrow \frac{p-1}{2}$ and $\beta \leftarrow \frac{q-1}{2}$
4: Compute γ, δ such that $\delta\alpha + \gamma\beta = 1$
5: Choose a, b where α and β are the least positive integers such that $a^\alpha \equiv 1 \pmod{n}$ and $b^\beta \equiv 1 \pmod{n}$
6: Private key $K_s = (p, q, \delta, \gamma)$
7: Public key $K_p = (n, a, b)$

Algorithm 1.40 [15] encrypts a chosen message using the public key generated by Algorithm 1.39 and outputs ciphertext.

Algorithm 1.40 Djebaili-Melkemi encryption

Input: public key K_p
Output: ciphertext c

1: Choose a message m where m is associated with the message space of K_p
2: Choose random integers $x, y \in \mathbb{Z}_n^*$
3: $c_1 \leftarrow a^x m \pmod{n}$
4: $c_2 \leftarrow b^y m \pmod{n}$
5: Ciphertext $c \leftarrow (c_1, c_2)$

Algorithm 1.41 [15] decrypts the ciphertext generated by Algorithm 1.40 using the private key generated by Algorithm 1.39 and outputs the original message.

Algorithm 1.41 Djebaili-Melkemi decryption

Input: private key K_s, ciphertext c
Output: message m

1: Message $m \leftarrow c_1^{\delta\alpha} c_2^{\gamma\beta}$.

1.2.18 Djebali-Melkemi signature scheme

Djebali-Melkemi signature scheme [15] was proposed by Karima Djebaili and Lamine Melkemi in 2020. This scheme is based on Djebaili-Melkemi encryption scheme.

Djebaili-Melkemi key generation is the same as in Algorithm 1.39. Algorithm 1.42 [15] signs a chosen message using the private key generated by Algorithm 1.39 and outputs a signature.

Algorithm 1.42 Djebaili-Melkemi signature

Input: private key K_s
Output: signature s

1: Choose a message m where m is associated with the message space of K_p
2: Choose random $\phi \in< a >$ and $\psi \in< b >$ where $< x >$ denotes the subgroup generated by x
3: $c_1 \leftarrow (\phi\mathcal{H}(m))^{\beta} \pmod{n}$ where $\mathcal{H}$ is a cryptographic hash function
4: $c_2 \leftarrow (\psi\mathcal{H}(m))^{\alpha} \pmod{n}$
5: $\omega \leftarrow (\phi\psi)^{-1} \pmod{n}$
6: Signature $s = (c_1, c_2, \omega)$

Algorithm 1.43 [15] verifies the signature generated by Algorithm 1.42 using the public key generated by Algorithm 1.39.

Algorithm 1.43 Djebaili-Melkemi verification

Input: public key K_p, signature s

1: Verify that $\mathcal{H}(m) = {c_1}^{\gamma}{c_2}^{\delta}\omega \pmod{n}$

1.2.19 Ariffin-Asbullah-Abu-Mahad encryption scheme

Ariffin-Asbullah-Abu-Mahad encryption scheme [16] was proposed by M.R.K. Ariffin, M.A. Asbullah, N.A. Abu and Z. Mahad in 2013. This scheme is quite efficient as it has lower complexity order for the encryption compared to RSA and ECC. The decryption is faster than RSA and marginally slower than ECC.

Algorithm 1.44 [16] generates a pair of a private key and a public key.

Algorithm 1.44 Ariffin-Asbullah-Abu-Mahad key generation

Output: private key K_s, public key K_p

1: Choose two distinct primes n-bit primes p and q such that $p, q \equiv 3 \pmod{4}$ where $2^n < p, q < 2^{n+1}$
2: Choose a random d such that $d > (p^2q)^{\frac{4}{9}}$
3: Compute e such that $ed \equiv 1 \pmod{pq}$
4: Add multiples of pq until $2^{3n+4} < e < 2^{3n+6}$
5: $A_1 \leftarrow p^2q$, $A_2 \leftarrow e$
6: Private key $K_s = (pq, d)$
7: Public key $K_p = (n, A_1, A_2)$

Algorithm 1.45 [16] encrypts a chosen message using the public key generated by Algorithm 1.44 and outputs a ciphertext.

Algorithm 1.45 Ariffin-Asbullah-Abu-Mahad encryption

Input: public key K_p
Output: ciphertext c

1: Choose a message m such that $m = 2^{4n}m_1 + m_2$ where $2^{4n} < m_1 < 2^{4n+1}$ and $2^{2n-2} < m_2 < 2^{2n-1}$
2: Ciphertext $c \leftarrow A_1 m_1 + A_2 {m_2}^2$

Algorithm 1.46 [16] decrypts the ciphertext generated by Algorithm 1.45 using the private key generated by Algorithm 1.44 and outputs the original message.

Algorithm 1.46 Ariffin-Asbullah-Abu-Mahad decryption

Input: private key K_s, ciphertext c
Output: message m

1: $W \leftarrow cd \pmod{pq}$
2: $x_p \leftarrow W^{\frac{p+1}{4}} \pmod{p}$
3: $x_q \leftarrow W^{\frac{q+1}{4}} \pmod{q}$
4: $M_1 = q^{-1} \pmod{p}$ and $M_2 = p^{-1} \pmod{q}$
5: $m_{21} = x_p M_1 q + x_q M_2 p \pmod{pq}$
6: $m_{22} = x_p M_1 q - x_q M_2 p \pmod{pq}$
7: $m_{23} = -x_p M_1 q + x_q M_2 p \pmod{pq}$
8: $m_{24} = -x_p M_1 q - x_q M_2 p \pmod{pq}$
9: **for** $i \leftarrow 1, 2, 3, 4$ **do**
10: $\quad m_{1i} \leftarrow \frac{c - {m_{2i}}^2 A_2}{A_1}$
11: **end for**
12: Choose the only value of m_{1j} which is equal to an integer
13: Sort the pair (m_{1j}, m_{2j})
14: Message $m = 2^{4n}m_1 + m_2$

1.2.20 Kurosawa-Itoh-Takeuchi encryption scheme

Kurosawa-Itoh-Takeuchi encryption scheme [17] was proposed by Kaoru Kurosawa, Toshiya Itoh and Masashi Takeuchi. This scheme is another modification of Rabin encryption scheme so that the decryption is no longer ambiguous and unlike Williams encryption scheme does not require primes p and q to be of a special form. It is proven to be as difficult as factoring a large number.

Algorithm 1.47 [17] generates a pair of a private key and a public key.

Algorithm 1.47 Kurosawa-Itoh-Takeuchi key generation

Output: private key K_s, public key K_p

1: Choose two distinct primes p and q
2: $n \leftarrow pq$
3: Choose c such that $\left(\frac{c}{p}\right) = \left(\frac{c}{q}\right) = -1$
4: Private key $K_s = (p, q)$
5: Public key $K_p = (n, c)$

Algorithm 1.48 [17] encrypts a chosen message using the public key generated by Algorithm 1.47 and outputs a ciphertext.

Algorithm 1.48 Kurosawa-Itoh-Takeuchi encryption

Input: public key K_p
Output: ciphertext c

1: Choose message m, $0 < m < n$ such that $\gcd(m, n) = 1$
2: $E \leftarrow m + cm^{-1} \pmod{n}$
3: **if** $(\frac{c}{m}) = 1$ **then**
4: $\quad s \leftarrow 0$
5: **else if** $(\frac{c}{m}) = -1$ **then**
6: $\quad s \leftarrow 1$
7: **end if**
8: **if** $c/m \pmod{n} > m$ **then**
9: $\quad t \leftarrow 0$
10: **else if** $c/m \pmod{n} < m$ **then**
11: $\quad t \leftarrow 1$
12: **end if**
13: Ciphertext $c \leftarrow (E, s, t)$

Algorithm 1.49 [17] decrypts the ciphertext generated by algorithm 1.48 using the private key generated by 1.47 and outputs the original message.

Algorithm 1.49 Kurosawa-Itoh-Takeuchi decryption

Input: private key K_s, ciphertext c
Output: message m

1: Let a_1, a_2 be the roots of $m^2 - Em + c = 0 \pmod{p}$
2: Let b_1, b_2 be the roots of $m^2 - Em + c = 0 \pmod{q}$
3: Then $m^2 - Em + c = 0 \pmod{n}$ has the following 4 roots:

$$m_1 \leftarrow [a_1, a_2], m_2 \leftarrow [a_2, b_2]$$
$$m_3 \leftarrow [a_1, b_2], m_4 \leftarrow [a_2, b_1]$$

4: **if** $s = 0$ **then**
5: **if** $t = 0$ **then**
6: Message $m \leftarrow \min(m_1, m_2)$
7: **else if** $t = 1$ **then**
8: Message $m \leftarrow \max(m_1, m_2)$
9: **end if**
10: **end if**
11: **if** $s = 1$ **then**
12: **if** $t = 0$ **then**
13: Message $m \leftarrow \min(m_3, m_4)$
14: **else if** $t = 1$ **then**
15: Message $m \leftarrow \max(m_3, m_4)$
16: **end if**
17: **end if**

1.2.21 Kurosawa-Itoh-Takeuchi signature scheme

Kurosawa-Itoh-Takeuchi signature scheme [17] was proposed by Kaoru Kurosawa, Toshiya Itoh and Masashi Takeuchi.

Kurosawa-Itoh-Takeuchi key generation is the same as in Algorithm 1.47. Algorithm 1.50 [17] signs a chosen message using the private key generated by Algorithm 1.47 and outputs a signature.

Algorithm 1.50 Kurosawa-Itoh-Takeuchi signature

Input: private key K_s
Output: signature s

1: Choose a message m, $0 < m < n$ such that $\gcd(m, n) = 1$
2: $E \leftarrow m + cm^{-1} \pmod{n}$
3: $E_j \leftarrow E + j$
4: **while** $\neg(((E_j{}^2 - 4c)/p) = ((E_j{}^2 - 4c)/q) = 1)$ **do**
5: $j \leftarrow j + 1$
6: **end while**
7: Signature $s \leftarrow j$

The authors in [17] do not specify any verification scheme for signature, therefore a proposal for a verification scheme is provided instead.

Algorithm 1.51 verifies the signature generated by Algorithm 1.50 using the public key generated by Algorithm 1.47. The verification requires the original message.

Algorithm 1.51 Kurosawa-Itoh-Takeuchi verification

Input: public key K_p, message m, signature s

1: $E \leftarrow m + cm^{-1} \pmod{n}$
2: $E_j \leftarrow E + j$
3: Verify that $((E_j{}^2 - 4c)/p) = ((E_j{}^2 - 4c)/q) = 1$

1.2.22 Galindo encryption scheme

Galindo encryption scheme [18] was proposed by David Galindo, Sebastià Martín, Paz Morillo, and Jorge L. Villar. This scheme is a modification of the RSA-Paillier scheme.

Algorithm 1.52 [18] generates a pair of a private key and a public key.

Algorithm 1.52 Galindo key generation

Output: private key K_s, public key K_p

1: Choose two distinct primes p and q such that $p \equiv q \equiv 3 \pmod{4}$
2: Choose $e > 2$ such that $\gcd(e, \lambda(n^2)) = 1$ and $\gcd(e, n) = 1$
3: $d \leftarrow e^{-1} \pmod{\lambda(n)}$
4: Private key $K_s \leftarrow (p, q, d)$
5: Public key$K_p \leftarrow (n, e)$

Algorithm 1.53 [18] encrypts a chosen message using the public key generated by Algorithm 1.52 and outputs a ciphertext.

Algorithm 1.53 Galindo encryption

Input: public key K_p

Output: ciphertext c

1: Choose a message $m \in \mathbb{Z}_n$
2: Choose a random integer $r \in \mathcal{Q}_n$
3: Ciphertext $c \leftarrow r^{2e} + mn \pmod{n^2}$

Algorithm 1.54 [18] decrypts the ciphertext generated by Algorithm 1.53 using the private key generated by Algorithm 1.52 and outputs the original message.

Algorithm 1.54 Galindo decryption

Input: private key K_s, ciphertext c
Output: message m

1: $t \leftarrow c^d \pmod{n}$
2: $r \leftarrow \text{sqrt}(t) \pmod{n}$
3: Message $m \leftarrow \frac{(c-r^{2e}) \pmod{n^2}}{n}$

1.2.23 Okamoto–Uchiyama encryption scheme

Okamoto–Uchiyama encryption scheme [19] is a probabilistic encryption scheme proposed by Tatsuaki Okamoto and Shigenori Uchiyama. The authors present a new technique different from Rabin scheme and Diffie-Hellman.

Algorithm 1.55 [19] generates a pair of a private key and a public key.

Algorithm 1.55 Okamoto–Uchiyama key generation

Output: private key K_s, public key K_p

1: Choose two distinct primes p and q, such that $|p| = |q| = k$
2: Choose a random integer $g \in (\mathbb{Z}/n\mathbb{Z})^*$ such that the order of $g_p = g^{p-1} \pmod{p^2}$ is p
3: $h \leftarrow g^n \pmod{n}$
4: Private key $K_s \leftarrow (p, q)$
5: Public Key $K_p \leftarrow (n, g, h, k)$

Algorithm 1.56 [19] encrypts a chosen message using the public key generated by Algorithm 1.55 and outputs a ciphertext.

Algorithm 1.56 Okamoto–Uchiyama encryption

Input: public key K_p
Output: ciphertext c

1: Choose message m, $0 < m < 2^{k-1}$
2: Choose $r \in \mathbb{Z}/n\mathbb{Z}$
3: Ciphertext $c \leftarrow g^m h^r \pmod{n}$

Algorithm 1.57 [19] decrypts the ciphertext generated by Algorithm 1.56 using private key generated by Algorithm 1.55 and outputs the original message.

Algorithm 1.57 Okamoto–Uchiyama decryption

Input: private key K_s, ciphertext c
Output: message m

1: $c_p \leftarrow c^{p-1} \pmod{p^2}$
2: Message $m \leftarrow \frac{c_p - 1}{g_p - 1} \pmod{p}$

1.2.24 LUC

LUC [7] is a generalization of RSA using Lucas numbers. This scheme was proposed by Peter J. Smith and Michael J. J. Lenon and patented in 1993. The security of LUC is not better than RSA and several sources show how to break LUC at least in some implementations.

Algorithm 1.58 [7] generates a pair of a private key and a public key.

Algorithm 1.58 LUC key generation

Output: private key K_s, public key K_p

1: Choose two distinct primes p and q
2: $n \leftarrow pq$
3: Choose a random integer e such that $\gcd(e, p-1) = 1, \gcd(e, p+1) = 1, \gcd(e, q-1) = 1$ and $\gcd(e, q+1) = 1$
4: There are four possible decryption keys:

$$\begin{aligned} d &= e^{-1} \pmod{\operatorname{lcm}((p+1),(q+1))} \\ d &= e^{-1} \pmod{\operatorname{lcm}((p+1),(q-1))} \\ d &= e^{-1} \pmod{\operatorname{lcm}((p-1),(q+1))} \\ d &= e^{-1} \pmod{\operatorname{lcm}((p-1),(q-1))} \end{aligned}$$

5: Private key $K_s \leftarrow (d, n)$
6: Public key $K_p \leftarrow (e, n)$

Algorithm 1.59 [7] encrypts a chosen message using the public key generated by Algorithm 1.58 and outputs a ciphertext.

Algorithm 1.59 LUC encryption

Input: public key K_p
Output: ciphertext c

1: Choose a message m
2: Ciphertext $c \leftarrow V_e(m, 1) \pmod{n}$

Algorithm 1.60 [7] decrypts the ciphertext generated by Algorithm 1.59 using the private key generated by Algorithm 1.58 and outputs the original message.

Algorithm 1.60 LUC decryption

Input: private key K_s, ciphertext c
Output: message m

1: Message $m \leftarrow V_d(m, 1) \pmod{n}$ with the proper d

Chapter 2

Quadratic residuosity problem

2.1 Description

Definition 2.1.1 (Quadratic residuosity problem)**.** Given an odd composite integer n and an integer a having Jacobi symbol $\left(\frac{a}{n}\right) = 1$, decide whether or not a is a quadratic residue modulo n [4].

2.2 Ciphers and protocols

In this section protocols and ciphers basing their security on quadratic residuosity problem are described.

2.2.1 Blum-Blum-Shub pseudorandom bit generator

Blum-Blum-Shub pseudorandom bit generator was proposed by Lenore Blum, Manuel Blum and Michael Schub in 1982 [20]. This is a cryptographically secure pseudorandom bit generator meaning that no statistical polynomial time test can distinguish between random uniformly distributed sequences and sequences generated by the Blum-Blum-Shub generator.

Algorithm 2.1 [4] generates a pseudorandom bit sequence.

Algorithm 2.1 Blum-Blum-Shub pseudorandom bit generator

Output: pseudorandom bit sequence $(z_1, z_2, \ldots, z_l)$

1: Choose two primes p and q such that $p \equiv q \equiv 3 \pmod 4$
2: $n \leftarrow pq$
3: Choose a random integer s such that $1 \leq s \leq n-1$ and $\gcd(s, n) = 1$
4: $x_0 \leftarrow s^2 \pmod n$
5: **for** $i \leftarrow 1, 2, \ldots, l$ **do**
6: $\quad x_i \leftarrow {x_{i-1}}^2 \pmod n$
7: $\quad z_i \leftarrow$ the least significant bit of x_i
8: **end for**
9: Pseudorandom bit sequence is $(z_1, z_2, \ldots, z_l)$

2.2.2 Goldwasser-Micali encryption scheme

Goldwasser-Micali encryption scheme [21] was proposed by Shafi Goldwasser and Silvio Micali in 1984. This is a probabilistic encryption scheme meaning that the same message can be encrypted to different ciphertexts and is one of the first semantically secure schemes meaning that the ciphertext bears no information about the message except for the length.

Algorithm 2.2 [4] generates a pair of a private key and a public key.

Algorithm 2.2 Goldwasser-Micali key generation

Output: private key K_s, public key K_p

1: Choose two distinct primes p and q
2: $n \leftarrow pq$
3: Choose y such that y is quadratic non-residue modulo n and $\left(\frac{y}{n}\right) = 1$
4: Private key $K_s \leftarrow (p, q)$
5: Public key $K_p \leftarrow (n, y)$

Algorithm 2.3 [4] encrypts a chosen message using the public key generated by Algorithm 2.2 and outputs a ciphertext.

Algorithm 2.3 Goldwasser-Micali encryption

Input: public key K_p
Output: ciphertext c

Let the message m be represented as a t–bit vector
for $i \leftarrow 1, 2, .., t$ **do**
    Select random integer $x \in \mathbb{Z}_n$
    if $m_i = 1$ **then**
        $c_i \leftarrow yx^2 \pmod{n}$
    else
        $c_i \leftarrow x^2 \pmod{n}$
    end if
end for
Ciphertext $c \leftarrow (c_1, c_2, ..., c_t)$

Algorithm 2.4 [4] decrypts the ciphertext generated by Algorithm 2.3 using the private key generated by Algorithm 2.2 and outputs the original message.

Algorithm 2.4 Goldwasser-Micali decryption

Input: private key K_s, ciphertext c
Output: message m

for $i \leftarrow 1, 2, \ldots, t$ **do**
    if $\left(\frac{c_i}{p}\right) = 1$ **then**
        $m_i \leftarrow 0$
    else
        $m_i \leftarrow 1$
    end if
end for
Message $m \leftarrow (m_1, m_2, \ldots, m_t)$

2.2.3 Blum-Goldwasser probabilistic encryption scheme

Blum-Goldwasser probabilistic encryption scheme was proposed by Blum and Goldwasser in 1985. This scheme [21] is based on the Blum-Blum-Shub pseudorandom generator.

Algorithm 2.5 [4] generates a pair of a private key and a public key.

Algorithm 2.5 Blum-Goldwasser key generation

Output: private key K_s, public key K_p

Choose two distinct primes p and q such that $p \equiv q \equiv 3 \pmod{4}$
Compute a and b such that $ap + bq = 1$
Private key $K_p = (p, q, a, b)$
Public key $K_p = n$

Algorithm 2.6 [4] encrypts a chosen message using the public key generated by Algorithm 2.5 and outputs a ciphertext.

Algorithm 2.6 Blum-Goldwasser encryption

Input: public key K_p
Output: ciphertext c

1: $k \leftarrow \lfloor \log n \rfloor$, $h \leftarrow \lfloor \log k \rfloor$
2: Let m be a binary string $m = (m_1, m_2, \ldots, m_t)$ of length t where each m_i is a binary string of length h
3: Select x_0 such that x_0 is a quadratic residue modulo n
4: **for** $i \leftarrow 1, 2, \ldots, t$ **do**
5: $\quad x_i \leftarrow x_{i-1}^2 \pmod{n}$
6: $\quad$ Let p_i be the h least significant bits of x_i
7: $\quad c_i \leftarrow p_i \oplus m_i$
8: **end for**
9: $x_{t+1} \leftarrow x_t^2 \pmod{n}$
10: Ciphertext $c \leftarrow (c_1, c_2, \ldots, c_t, x_{t+1})$

Algorithm 2.7 [4] decrypts the ciphertext generated by Algorithm 2.6 using the private key generated by Algorithm 2.5 and outputs the original message.

Algorithm 2.7 Blum-Goldwasser decryption

Input: private key K_s, ciphertext c
Output: message m

1: $d_1 \leftarrow ((p+1)/4)^{t+1} \pmod{p-1}$
2: $d_2 \leftarrow ((q+1)/4)^{t+1} \pmod{q-1}$
3: $u \leftarrow x_{t+1}^{d_1} \pmod{p}$
4: $v \leftarrow x_{t+1}^{d_2} \pmod{q}$
5: $x_0 \leftarrow vap + ubq \pmod{n}$
6: **for** $i \leftarrow 1, 2, \ldots, t$ **do**
7: $\quad x_i \leftarrow x_{i-1}^2 \pmod{n}$
8: $\quad$ Let p_i be the h least significant bits of x_i
9: $\quad m_i \leftarrow p_i \oplus c_i$
10: **end for**
11: Message $m \leftarrow (m_1, m_2, \ldots, m_t)$

2.2.4 Feige-Fiat-Shamir identification scheme

Feige-Fiat-Shamir identification scheme was proposed by Uriel Feige, Amos Fiat and Adi Shamir. This scheme [22] uses zero-knowledge proofs and is suitable for smart card implementation.

Algorithm 2.8 [7] generates a pair of a private key and a public key.

Algorithm 2.8 Fiat-Feige-Shamir key generation

Output: private key K_s, public key K_p

1: Choose two distinct primes p and q
2: $n \leftarrow pq$.
3: Choose k different numbers $v_1, v_2, ..., v_k$ where each v_i is a quadratic residue mod n.
4: Compute $s_1, s_2, ..., s_k$ where each s_i is the smallest integer such that $s_i = \text{sqrt } (v_i^{-1}) \pmod{n}$.
5: Private key $K_s \leftarrow s_1, s_2, ..., s_k$
6: Public key $K_p \leftarrow v_1, v_2, ..., v_k$

Algorithm 2.9 [7] enables V to verify the identify of P. V uses the public key and P uses the private key, both keys are generated by Algorithm 2.9.

Algorithm 2.9 Feige-Fiat-Shamir identification

Input: private key K_p, public key K_p

P chooses a random integer r less than n
P computes $x \leftarrow r^2 \pmod{n}$
P sends x to V
V sends random binary k-bits long string $b_1, b_2, \ldots, b_k$ to P
P computes $y \leftarrow r s_1^{b_1} s_2^{b_2} \cdots s_k^{b_k} \pmod{n}$ and sends it to V
V verifies $x = y^2 (v_1^{b_1} v_2^{b_2} \cdots v_k^{b_k}) \pmod{n}$

In every round P has 2^{-k} chance of fooling V. If this scheme [7] is repeated t times, P has 2^{-kt} chance of fooling V.

2.2.5 Fiat-Shamir signature scheme

Fiat-Shamir signature scheme was proposed by Amos Fiat and Adi Shamir in 1986. This scheme [7] is a modification of Feige-Fiat-Shamir identification scheme, the main difference is turning V into a hash function.

The key generation is same as in Algorithm 2.8. Algorithm 2.10 [7] signs a chosen message using the private key generated by Algorithm Algorithm 2.8 and outputs a signature.

Algorithm 2.10 Fiat-Shamir signature

Input: private key K_s

Output: signature s

1: Choose t random integers $r_1, r_2, ..., r_t$ where $1 \leq r_i \leq n$
2: Compute $x_1, x_2, ..., x_t$ such that $x_i = {r_i}^2 \pmod{n}$.
3: Hash the concatenation of the message m and values of x_i, $\mathcal{H}(m, x_1, x_2, ..., x_t)$ and use first kt bits as values of b_{ij} where i goes from 1 to t and j goes from 1 to k.
4: Compute $y_1, y_2, ..., y_t$ where $y_i = r_i({s_1}^{b_{i1}} {s_2}^{b_{i2}} \cdots {s_k}^{b_{tk}}) \pmod{n}$
5: Signature $s \leftarrow ((b_{11}, b_{12}, \ldots, b_{tk}), (y_1, y_2, \ldots, y_t))$

Algorithm 2.11 [7] verifies the signature generated by Algorithm 2.10 using the public key generated by 2.8. Algorithm 2.11 requires the original message.

Algorithm 2.11 Fiat-Shamir verification

Input: public key K_p, message m, signature s

1: Compute $z_1, z_2, ..., z_t$ where $z_i = {y_i}^2({v_1}^{b_{i1}} {v_2}^{b_{i2}} \cdots {v_k}^{b_{ik}}) \pmod{n}$
2: Verify that the first kt bits of $\mathcal{H}(m, z_1, z_2, ..., z_t)$ are equal to b_{ij}.

Chapter 3

Discrete logarithm problem

3.1 Description

Definition 3.1.1. Given a prime p, a generator α of $\mathbb{Z}_p^*$, and an element $\beta \in \mathbb{Z}_p^*$, find the integer x, $0 \leq x \leq p-2$, such that $\alpha^x \equiv \beta \pmod{p}$ [4].

3.2 Ciphers and protocols

In this section protocols and ciphers basing their security on discrete logarithm problem are described.

3.2.1 Pohlig-Hellman encryption scheme

Pohlig-Hellman encryption scheme [7] was proposed by Stephen C. Pohlig and Martin E. Hellman and patented in the United States in 1984. This scheme is symmetric and unlike most of schemes based on discrete logarithm problem or integer factorization problem and is very similar to RSA.

Algorithm 3.1 [7] generates a key used for both encryption and decryption.

Algorithm 3.1 Pohlig-Hellman key generation

Output: Key K

1: Choose n
2: Choose e such that $\gcd(e, \phi(n)) = 1$
3: $d \leftarrow e^{-1} \pmod{n}$
4: Key $K \leftarrow (e, d)$

Encryption exponent e and decryption exponent d should be kept in secret due to no requirements on n therefore e may be easily computed from d and vice versa [7].

Algorithm 3.2 [7] encrypts a chosen message using the key generated by Algorithm 3.1 and outputs a ciphertext.

Algorithm 3.2 Pohlig-Hellman encryption

Input: Key k
Output: ciphertext c
1: Ciphertext $c \leftarrow m^e \pmod{n}$

Algorithm 3.3 [7] decrypts the ciphertext generated by Algorithm 3.2 using the key generated by Algorithm 3.1 and outputs the original message.

Algorithm 3.3 Pohlig-Hellman decryption

Input: Key k, ciphertext c
Output: message m
1: Message $m \leftarrow c^d \pmod{n}$

3.2.2 Diffie-Hellman key agreement

Diffie-Hellman key agreement [7] is the first public-key algorithm ever invented. It was proposed by Whitfield Diffie and Martin E. Hellman in 1976. This scheme was patented in the United States and Canada. Though the scheme is presented as key agreement between two parties, it can be used for key agreement between unlimited number of parties.

Algorithm 3.4 [4] generates a prime and a generator.

Algorithm 3.4 Diffie-Hellman precomputations

Output: prime p, generator α
1: Choose a prime p
2: Choose a generator α of group $\mathbb{Z}_p^*$

Algorithm 3.5 [4] enables two parties A and B to establish a shared key, both parties are using the prime and the generator generated by Algorithm 3.4.

Algorithm 3.5 Diffie-Hellman key exchange

Input: prime p, generator α
Output: shared key K
1: A chooses x such that $2 \leq x \leq p-2$
2: B chooses y such that $2 \leq y \leq p-2$
3: A sends $t_a \leftarrow \alpha^x \pmod{p}$ to B
4: B sends $t_b \leftarrow \alpha^y \pmod{p}$ to A
5: A receives t_b and computes $K \leftarrow {t_b}^x \pmod{p}$
6: B receives t_a and computes $K \leftarrow {t_a}^y \pmod{p}$

3.2.3 ElGamal encryption scheme

ElGamal encryption scheme [4] was proposed by Taher A. ElGamal. This scheme utilizes a random element in the encryption therefore the same message may be encrypted to different ciphertexts.

Algorithm 3.6 [4] generates a pair of a private key and a public key.

Algorithm 3.6 ElGamal key generation

Output: private key K_s, public key K_p

1: Choose a prime p
2: Choose generator α of the multiplicative group $\mathbb{Z}_p^*$
3: Choose random integer a such that $1 \leq a \leq p-2$
4: $y = \alpha^a \pmod{p}$
5: Private key $K_s \leftarrow a$
6: Public key $K_p \leftarrow (p, \alpha, y)$

Algorithm 3.7 [4] encrypts a chosen message using the public key generated by Algorithm 3.6 and outputs a ciphertext.

Algorithm 3.7 ElGamal encryption

Input: public key K_p
Output: ciphertext c

1: Choose a random integer k such that $1 \leq k \leq p-2$
2: $\gamma \leftarrow \alpha^k \pmod{p}$
3: $\delta \leftarrow m \cdot y^a \pmod{p}$
4: Ciphertext $c \leftarrow (\gamma, \delta)$

Algorithm 3.8 [4] decrypts the ciphertext generated by Algorithm 3.7 using the private key generated by Algorithm 3.6 and outputs the original message.

Algorithm 3.8 ElGamal decryption

Input: private key K_s, ciphertext c
Output: message m

1: $\gamma' \leftarrow \gamma^{-a} \pmod{p}$
2: Message $m \leftarrow \gamma'\delta \pmod{p}$

3.2.4 ElGamal signature scheme

ElGamal signature scheme key generation is the same as in Algorithm 3.6. Algorithm 3.9 [4] signs a chosen message using the private key generated by Algorithm 3.6 and outputs a signature.

Algorithm 3.9 ElGamal signature

Input: private key K_s
Output: signature s

1: Choose a random integer k such that $1 \leq k \leq p-2$ and $\gcd(k,(p-1)) = 1$
2: $r \leftarrow \alpha^k \pmod{p}$
3: $y \leftarrow k^{-1} \pmod{p}$
4: $t \leftarrow y(h(m) - ar) \pmod{p-1}$ where h is hash function such that $h : \{0,1\}^* \longrightarrow \mathbb{Z}_p$
5: Signature $s \leftarrow (r.t)$

Algorithm 3.10 [4] verifies the signature generated by Algorithm 3.9 using the public key generated by Algorithm 3.6. The verification requires the original message.

Algorithm 3.10 ElGamal signature verification

Input: public key K_p, message m, signature s

1: Check if $1 \leq r \leq p-1$
2: $v_1 \leftarrow y^r r^t$
3: $v_2 \leftarrow \alpha^{h(m)} \pmod{p}$
4: Check if $v_1 = v_2$

3.2.5 Guillou-Quisquater identification scheme

Guillou-Quisquater identification scheme was proposed by Louis Guillou and Jean-Jacques Quisquater. The scheme [7] is suited for smart card implementation.

Algorithm 3.11 [7] generates a pair of a private key and a public key.

Algorithm 3.11 Guillou-Quisquater key generation

Output: private key K_s, public key K_p

1: Let J be the string of information of Peggy's identity
2: If J is too long use hash of J instead
3: Let n be product of two primes
4: Choose an integer v
5: Compute B such that $JB^v \equiv 1 \pmod{n}$
6: Private key $K_s \leftarrow B$
7: Public key $K_p \leftarrow (J, n, v)$

Algorithm 3.12 [7] enables V to verify the identity of P. V uses the public key and P uses the private key, both keys are generated by Algorithm 3.11

Algorithm 3.12 Guillou-Quisquater identification

Input: private key K_s, public key K_p

1: P chooses a random integer r, $1 \leq r \leq n-1$
2: P computes $T \leftarrow r^v \pmod{n}$ and sends it to V
3: V chooses a random integer d, $0 \leq d \leq v-1$ and sends it to P
4: P computes $D \leftarrow rB^d$ and sends it to V
5: V verifies that $T \equiv D^v J^d \pmod{n}$

3.2.6 Guillou-Quisquater signature scheme

This scheme [7] is based on the Guillou-Quisquater identification scheme and is suited for smart card implementation. It can be also modified to a multiple signatures scheme.

Guillou-Quisquater key generation is the same as in Algorithm 3.11. Algorithm 3.13 [7] signs a chosen message using the private key generated by Algorithm 3.11 and outputs a signature.

Algorithm 3.13 Guillou-Quisquater signature

Input: private key K_s
Output: signature s

1: Let m be the message A wants to sign
2: Choose a random integer r, $1 \leq r \leq n-1$
3: $T \leftarrow r^v \pmod{n}$
4: $d \leftarrow \mathcal{H}(m, T) \pmod{v}$ where $\mathcal{H}$ is a one-way hash function
5: $D \leftarrow rB^d \pmod{n}$
6: Signature $s \leftarrow (d, D)$

Algorithm 3.14 [7] verifies the signature generated by Algorithm 3.13 using the public key generated by Algorithm 3.11. The verification algorithm requires the original message.

Algorithm 3.14 Guillou-Quisquater verification

Input: public key K_p, message m, signature s

1: $T' \leftarrow D^v J^d \pmod{n}$
2: $d' \leftarrow \mathcal{H}(m, T')$
3: Verify that $d = d'$

3.2.7 Schnorr authentication protocol

Schnorr authentication protocol was proposed by Claus Schnorr. This scheme [7] was patented in the United States and many other countries.

Algorithm 3.15 [7] generates a pair of a private key and a public key.

Algorithm 3.15 Schnorr key generation

Output: private key K_s, public key K_p

1: Choose two primes p and q such that $q|p-1$
2: Choose and integer $a \neq 1$ such that $a^q \equiv 1 \pmod{p}$
3: Choose a random integer s, $s < q$
4: $v \leftarrow s^{-s} \pmod{n}$
5: Private key $K_s \leftarrow s$
6: Public key $K_p \leftarrow v$

Algorithm 3.16 [7] enables V to authenticate P. V uses the public key, P uses the private key, and both keys are generated by Algorithm 3.15.

Algorithm 3.16 Schnorr authentication

Input: private key K_s, public key K_p

1: P chooses a random integer r, $r < q$
2: P computes $x \leftarrow a^r \pmod{p}$ and sends it to V
3: V chooses a random integer e, $0 \leq e \leq 2^t - 1$ where t is a security parameter and sends it to P
4: P computes $y \leftarrow (r + se) \pmod{q}$ and sends it to V
5: V verifies that $x \equiv a^v x^e \pmod{n}$

3.2.8 Schnorr signature protocol

Schnorr signature protocol was proposed by Claus Schnorr. This protocol [7] is a modification of the Schnorr authentication protocol.

Schnorr key generation is same as in Algorithm 3.15. Algorithm 3.17 [7] signs a chosen message using the private key generated by Algorithm 3.15 and outputs signature.

Algorithm 3.17 Schnorr signature

Input: private key K_s

Output: signature s

1: Choose a random integer $r < q$
2: $x \leftarrow a^r \pmod{p}$
3: $e \leftarrow \mathcal{H}(m, x)$ where $\mathcal{H}$ is a one-way hash function
4: $y \leftarrow (r + se) \pmod{q}$
5: Signature $s \leftarrow (e, y)$

Algorithm 3.18 [7] verifies the signature generated by Algorithm 3.17 using the public key generated by Algorithm 3.15. The verification requires the original message.

Algorithm 3.18 Schnorr verification

Input: public key K_p, message m, ciphertext c

1: $x' \leftarrow a^y v^e \pmod{p}$
2: Verify that $e = \mathcal{H}(m, x')$

3.2.9 Chaum undeniable signature scheme

Chaum undeniable signature scheme was proposed by David Chaum. Unlike signatures, undeniable signatures require the participation of both the signer and the verifier. It may be the case that the signer would try to give a false response to deny the signature however this can be detected by the verifier with exponentially high probability [23].

Algorithm 3.19 [7] generates a pair of a private key and a public key.

Algorithm 3.19 Chaum key generation

Output: private key K_s, public key K_p

1: Choose a prime p and a primitive element g
2: Choose an integer x
3: $e \leftarrow g^x \pmod{p}$
4: Private key $K_s \leftarrow x$
5: Public key $K_p \leftarrow (p, g, e)$

Algorithm 3.20 [7] signs a chosen message using the private key generated by Algorithm 3.19 and outputs a signature.

Algorithm 3.20 Chaum signature

Input: private key K_s
Output: signature s

1: B chooses two random integers $a < p$ and $b < p$ and sends them to A
2: A computes $t \leftarrow x^{-1} \pmod{p-1}$
3: A computes$c \leftarrow z^a e^b \pmod{p}$
4: Signature $s \leftarrow c^t \pmod{p}$

Algorithm 3.21 [7] verifies the signature generated by Algorithm 3.20 using the public key generated by Algorithm 3.19.

Algorithm 3.21 Chaum verificaton

Input: public key K_p, signature s

1: B verifies that $s = m^d g^b$

3.2.10 Cramer-Shoup encryption scheme

Cramer-Shoup encryption scheme was proposed by Ronald Cramer and Victor Shoup. The authors claim that this is the first scheme that is both practical and provably secure against adaptive chosen ciphertext attack under standard intractability assumptions [24].

Algorithm 3.22 [24] generates a pair of a private key and a public key.

Algorithm 3.22 Cramer-Shoup key generation

Output: private key K_s, public key K_p

1: Choose a group G of prime order q
2: Choose random $g_1, g_2 \in G$
3: Choose random $x_1, x_2, y_1, y_2, z \in \mathbb{Z}_q$
4: $c \leftarrow g_1^{x_1} g_2^{x_2}$, $d \leftarrow g_1^{y_1} g_2^{y_2}$, $h \leftarrow g_1^z$
5: Choose a hash function $\mathcal{H}$ from the universal family of one-way hash functions
6: Private key $K_s \leftarrow (x_1, x_2, y_1, y_2, z)$
7: Public key $K_p \leftarrow (g_1, g_2, c, d, h, \mathcal{H})$

Algorithm 3.23 [24] encrypts a chosen message using the public key generated by Algorithm 3.22 and outputs ciphertext.

Algorithm 3.23 Cramer-Shoup encryption

Input: public key K_p
Output: ciphertext c

1: Choose a message $m \in G$
2: Choose random integer $r \in \mathbb{Z}_n$
3: $u_1 \leftarrow g_1^r$, $u_2 \leftarrow g_2^r$
4: $e \leftarrow h^r m$
5: $\alpha \leftarrow \mathcal{H}(u_1, u_2, e)$
6: $v \leftarrow c^r d^{r\alpha}$
7: Ciphertext $c \leftarrow (u_1, u_2, e, v)$

Algorithm 3.24 [24] decrypts the ciphertext generated by Algorithm 3.23 using the private key generated by Algorithm 3.22 and output the original message.

Algorithm 3.24 Cramer-Shoup decryption

Input: private key K_s, ciphertext c
Output: message m

1: $\alpha \leftarrow \mathcal{H}(u_1, u_2, e)$
2: Verify that $v = u_1^{x_1 + y_1 \alpha} u_2^{x_2 + y_2 \alpha}$
3: Message $m \leftarrow e / u_1^z$

Chapter 4

Implementations setup

4.1 Tools

This section presents mathematical software chosen for implementation comparison.

4.1.1 Magma

Magma [25] is developed and distributed by the Computational Algebra Group at the University of Sydney. This software is designed for computations in algebra, number theory, algebraic geometry and algebraic combinatorics.

4.1.2 SageMath

SageMath [26] is a free open-source mathematics software, that offers python based language and builds on top of many mathematical packages such as NumPy, SciPy, R and more. The mission of SageMath is to be a viable free open-source alternative to Magma, Maple, Mathematica and MATLAB.

4.1.3 MATLAB

MATLAB [27] is a programming and numeric platform and offers many features such as signal processing, deep learning and machine learning. MATLAB is developed by MathWorks.

4.2 Integer factorization problem

Known algorithms for integer factorization are: Trial division, Pollard's Rho, Pollard $p-1$, Williams's $p+1$, Shanks square form factorization, Elliptic curve method, Quadratic sieve and Number field sieve and others [4].

4.2.1 Magma

This section specifies the choice of arguments for algorithms implemented in Magma. All Magma functions having an optional argument "Proof" were run with "Proof := false" meaning that no time was spent on proving that the number n being factored is a prime.

4.2.1.1 TrialDivision

Function TrialDivision requires 2 arguments: the number n to be factored and bound B as the upper bound [28]. B is chosen such that $B = \lfloor\sqrt{n}\rfloor$ in the case of $n = pq$ and $B = \lfloor n^{1/3}\rfloor$ in the case of $n = p^2q$.

4.2.1.2 Cunningham

Function Cunningham requires 3 arguments: b, k and c such that $n = b^k + c$, where $c \in \{-1, 1\}$ and n is the number to be factored [28]. Arguments k, c are determined for the lowest value of b where $b \in \{2, 3, 5, 6, 7, 10, 11, 12\}$ as in [29]. SageMath implementation [30] uses the same values.

4.2.1.3 PollardRho

Function PollardRho requires 4 arguments: the number n to be factored, c and s are parameters for $x_i = x_{i-1}^2 + c$ where $x_0 = s$ and the number of iterations k. Parameters c and s are set to 1 which are the default values and $k = 10^9$ which is the maximum value that can be chosen for number of iterations [28].

4.2.1.4 pMinus1

Function pMinus1 requires 2 arguments: the number n to be factored and B_1 such that all primes should be less or equal to B_1 except one which is less or equal to the optional parameter boound B_2 [28].

B_1 is chosen as $B_1 = \min(\frac{1}{2}\lfloor\sqrt{n}\rfloor, 10^9)$ in the case of $n = pq$ and $B_1 = \min(\frac{1}{2}\lfloor n^{1/3}\rfloor, 10^9)$ in the case of the $n = p^2q$ where 10^9 is the maximum value that can be chosen for bound B_1.

4.2.1.5 pPlus1

Function pPlus1 requires 2 arguments: the number n to be factored and B_1 such that $p + 1$ has all primes less or equal to B_1 except for one which may be less than or equal to B_2 where p is a prime factor of n. The algorithm succeeds only if $\left(\frac{{x_0}^2-4}{p}\right) = -1$ where x_0 is a randomly chosen seed but may be specified with paramater x_0 instead [28].

Therefore B_1 is chosen as $B_1 = \min(\frac{1}{2}\lceil\sqrt{n}\rceil, 10^9)$ in the case of $n = pq$ and $B_1 = \min(\frac{1}{2}\lceil n^{1/3}\rceil, 10^9)$ in the case of $n = p^2q$ where 10^9 is limit for bound

$B1$. An optional parameter x_0 is chosen as the least positive integer such that $\left(\frac{{x_0}^2-4}{p}\right) = -1$. This choice is however not possible without knowing the factorization of n beforehand.

4.2.1.6 ECM

Function ECM requires 2 arguments: the number n to be factored and bound B_1 [28]. As no additional information about bound B_1 is provided in [28] we assume that bound B_1 is equivalent to B_1 in pMinus1 based on description in [31].

Therefore B_1 should be chosen such that $p-1$ has all prime factors less than B_1 except one which is less than B_2. Therefore we choose $B_1 = \min(\frac{1}{2}\lfloor\sqrt{n}\rfloor, 10^9)$ in the case of form pq and $B_1 = \min(\frac{1}{2}\lfloor n^{1/3}\rfloor, 10^9)$ in the case of form p^2q. This will however result in worse performance [31].

4.2.1.7 MPQS

Function MPQS requires 1 argument: the number n to be factored, the function is specified in [28].

4.2.1.8 Factorization

Function Factorization requires 1 argument: the number n to be factored, the function is specified in [28].

4.2.2 SageMath

SageMath's functions factor_cunningham, factor_trial_division, factor, qsieve, ecm_factor, pollardrho_brent, pollard_pm1, williams_pp1 require 1 argument: the number n to be factored [30, 33].

4.2.3 MATLAB

MATLAB's function factor requires 1 argument: the number n to be factored, the function is specified in [34].

4.3 Quadratic residuosity problem

There are no known efficient algorithms for solving the quadratic residuosity problem if the factorization of the modulus n is not known [4].

4.3.1 Magma

Magma does not offer any algorithms for determining whether or not a given number a is a quadratic residue in $\mathbb{Z}_n$ where n is a product of two distinct

primes and $(\frac{a}{n}) = 1$. Therefore, exhaustive search was implemented for comparison with SageMath.

4.3.2 SageMath

This section specifies SageMath's implementations for solving quadratic residuosity problem. Also, exhaustive search was implemented as computing $i^2 \pmod{n}$ for $i \in \{0, 1, \ldots, n-1\}$ until $x = i^2 \pmod{n}$ where x is the element of $\mathbb{Z}_n$ for which the algorithm determines quadratic residuosity.

4.3.2.1 quadratic_residues

SageMath's only implementation for solving quadratic residuosity problem is the quadratic_residues function. This function requires one argument: the number n of $\mathbb{Z}_n$ and return value is a list of all quadratic residues in $\mathbb{Z}_n$, this function is specified in [35].

4.4 Discrete logarithm problem

Known algorithms for discrete logarithm problem are exhaustive search, Baby-step Giant-step, Pollard's rho, Pohlig-Hellman, Index calculus and others [4].

4.4.1 Magma

In this section Magma's implementations for solving discrete logarithm problem are specified.

4.4.1.1 Log

Function Log requires 2 parameters: a primitive element b of $\mathbb{Z}_p$ and the element x to compute discrete logarithm for, the function is specified in [36].

4.4.2 SageMath

In this section, SageMath's implementations for solving discrete logarithm problem are specified.

4.4.2.1 bsgs

Function bsgs requires 3 arguments: a primitive element a of $\mathbb{Z}_p$, the element b to compute discrete logarithm for and pair of bounds for exponent n, $l \leq n \leq u$ where $a^n \equiv b \pmod{p}$, the function is specified in [37]. The bounds were chosen as $l = 0$ and $u = p - 2$.

4.4.2.2 discrete_log

Function discrete_log requires 2 parameters: the element a to compute discrete logarithm for and a primitive element b of $\mathbb{Z}_p$, the function is specified in [37].

Chapter 5

Implementations comparison

5.1 Integer factorization problem

Implementations were tested on composite numbers of form pq and p^2q where p and q were primes of length of $10k$ bits, starting with $k = 2$ and k was incremented by 1 until the implementation gave correct results within 3600 seconds. For each bit length, 40 random primes were generated to form 20 composite numbers. All implementations were tested on the same set of composite numbers. Random primes were generated with OpenSSL [38] version 1.1.1k.

Magma's implementation of Number field sieve was not tested as neither SageMath nor MATLAB provide implementations of this algorithm and therefore could not be compared. Furthermore, choosing parameters for Number field sieve is out of the scope of this thesis.

5.1.1 Form pq

This section presents the results of integer factorization algorithms implementations tested on composite numbers of form pq.

5.1.1.1 Trial division

SageMath's implementation of trial division is faster as it could factor all numbers with a prime factor of 40 bits with an average time of 2074.35 seconds as shown in Table 5.2 whereas Magma's implementation could not factor any of those numbers within 3600 seconds as shown in Table 5.1.

5.1.1.2 Cunningham

Magma's implementation of Cunningham was not tested as none of the 80 composite numbers with a prime factor between 20 bits and 50 bits were in the required form.

5.1.1.3 Pollard's $p-1$

In Pollard's $p-1$, both speed and success of the algorithm depend on chosen parameters. Magma's implementation is concluded to be both faster and more reliable as it could factor 19 out of 20 numbers with a prime factor of 50 bits within 60 seconds on average as shown in Table 5.3. Sage's implementation could not factor 8 of these numbers within 3600 seconds and the remaining 12 took 125 seconds on average which is more than twice than Magma as shown in Table 5.4.

5.1.1.4 Pollard's Rho

The efficiency of the Magma and Sage implementations is the same for 50 bits factor however, Magma's implementation fails to factor 11 out of 20 numbers with 60 bits factor as shown in Table 5.5 whereas Sage's implementation factored all numbers with 60 bits factor although need almost 1480 seconds on average as shown in Table 5.6.

5.1.1.5 Williams's $p+1$

The results are similar to Pollard's $p-1$ implementations, Magma's implementation could factor 18 out of 20 numbers with a prime factor of 60 bits with 210 seconds on average as shown in Table 5.7. Sage's implementation factored 14 of these numbers with 633 seconds on average, the remaining 6 numbers were not factored within 3600 seconds as shown in Table 5.8.

5.1.1.6 Elliptic curve method

Sage's implementation outperformed Magma's implementation as it could factor all 20 numbers with a 120-bit prime factor in 968 seconds on average compared to Magma which could factor only 7 out of 20 numbers with a prime factor of 100 bits and needed almost twice as much time as shown in Table 5.9 and Table 5.10.

5.1.1.7 Quadratic sieve

Sage's implementation is faster than Magma as it factored all numbers with a factor of the length of 150 bits within 17 minutes and factored half of the 160-bit factor numbers within 1 hour as shown in Table 5.11. Whereas Magma's implementation needed more than 30 minutes on average for 150-bit factor numbers and could not factor any of 160-bit factor numbers as shown in Table 5.12.

5.1.1.8 Generic factor implementation

Both Magma's Factorization and Sage's factor factored all 20 numbers with a prime factor of 140 bits, Magma needed 2135 seconds on average seconds and Sage needed 2542 seconds on average as shown in Table 5.13 and Table 5.14. MATLAB's implementation was less efficient as it could factor 16 out of 20 numbers with a prime factor of 100 bits and needed 1947 seconds on average as shown in Table 5.15.

5.1.2 Form p^2q

This section presents the results of integer factorization algorithms implementations tested on composite numbers of form p^2q where p and q are distinct primes of the same size.

5.1.2.1 Trial division

As in the case of form pq SageMath's implementation is faster than Magma's implementation as it factored all 20 instances with a prime factor of the length of 40 bits whereas Magma's implementations did not factor any within one hour as shown in Table 5.16 and Table 5.17.

5.1.2.2 Cunningham

Magma's implementation of Cunningham was not tested as none of the 80 composite numbers with a prime factor between 20 bits and 50 bits were in the required form.

5.1.2.3 Pollard's $p-1$

As in case of form pq Magma's implementation was more efficient than SageMath's implementation as shown in Table 5.18 and Table 5.19.

5.1.2.4 Pollard's rho

SageMath's implementation of Pollard's Rho is concluded to be more efficient as it factored all 20 composite numbers with 60-bit prime factor as shown in Table 5.21. Magma's implementation factored only 9 out of 20 composite numbers with 60-bit prime factor as shown in Table 5.20.

5.1.2.5 William's $p+1$

Magma's implementation of William's $p+1$ is concluded to be more efficient. It has lower failure rates than SageMath's implementation for composite numbers with prime factor between 60 and 90 bits as shown in Table 5.22 and Table 5.23.

5.1.2.6 Elliptic curve method

SageMath's implementation was more efficient as it factored all 20 generated instances with 120 bit prime factor compared to Magma's implementation factored only 1 out of 20 instances with 110-bit prime factor as shown in Table 5.24 and Table 5.25.

5.1.2.7 Quadratic sieve

SageMath's implementation factored all 90-bit factor instances two times faster than Magma's implementation and factored 15 out of 20 instances with a 100-bit prime factor. Magma's implementation could not factor any of the 100-bit prime factor instances. SageMath's implementation is concluded to be more efficient as shown in Table 5.27 and Table 5.26.

5.1.2.8 Generic factor implementation

Magma's generic factor implementation is more efficient than both SageMath's and MATLAB's implementations. Magma's implementation needed less time and had higher success rate than SageMath's and MATLAB's for composite numbers with 120-bit factor as shown in Table 5.28, Table 5.29 and Table 5.30. SageMath and MATLAB performed similarly.

5.2 Quadratic residuosity problem

Implementations were tested in $\mathbb{Z}_n$ where $n = pq$ and p and q are primes of the same size. For each of 10-bit, 15-bit and 20-bit lengths of primes p and q, 40 random primes were generated to form 20 composite numbers. Primes were generated with OpenSSL [38] version 1.1.1k and random elements a in $\mathbb{Z}_n$ were generated with OpenSSL and then udjusted with Magma [25] so that $\left(\frac{a}{n}\right) = 1$. As Magma does not offer any implementation for solving quadratic residuosity problem, exhastive search was implemented in Magma. Also, exhaustive search was implemented in SageMath as SageMath implementation quadratic_residues computes all quadratic residues in $\mathbb{Z}_n$ which is memory demanding and therefore comparing exhaustive search in Magma to this implementation would not be impartial.

5.2.1 Type $\mathbb{Z}_n$

5.2.1.1 Exhaustive search implementation

Implementation of exhaustive search in Magma was more efficient than exhaustive search implementation in SageMath. For a 15-bit prime factor, Magma needed 194.80 seconds on average compared to 239.62 seconds needed by SageMath. For a 20-bit prime factor, both Magma and SageMath gave

results only for one instance within 3600 seconds, Magma needed 1112.26 seconds and SageMath needed 1270.00 as shown in Table 5.31 and Table 5.32. Also, the exhaustive search implemented in Magma was more efficient than SageMath's quadratic_residues as for 15-bit prime factor, Magma needed 194.80 seconds on average compared to 722.75 seconds on average needed by SageMath as shown in Table 5.31 and Table 5.33.

5.3 Discrete logarithm problem

Implementations were tested in $\mathbb{Z}_p$ with prime p, generator g and random element x of $\mathbb{Z}_p$. Prime p is of length of $10k$ bits starting with $k = 2$ and k was incremented until the implementation gave correct results within 3600 seconds. For each bit length of p, 20 random primes p, 20 generators g and 20 elements x were generated. Every implementation was tested on the same set of primes, generators and random elements of $\mathbb{Z}_p$. All primes and random elements of $\mathbb{Z}_p$ were generated with OpenSSL [38] version 1.1.1k, generators were computed with Magma [25].

Magma's implementation of Index calculus was not tested as neither SageMath nor MATLAB provide implementations of this algorithm and therefore could not be compared. Furthermore, choosing parameters for Index calculus is out of the scope of this thesis.

5.3.1 Type $\mathbb{Z}_p^*$

5.3.1.1 Generic discrete logarithm implementation

Magma's generic function log is supreme to SageMath's functions log and bsgs. Magma's log found all solutions in $\mathbb{Z}_p$ with p being a 210 bit prime whereas SageMath's log and bsgs could solve all instances only up to a 60 bit prime as shown in Tables 5.34, 5.35, 5.36. SageMath's implementation of the Baby-step Giant-step algorithm was part of the generic discrete logarithm function comparison as Magma's Log uses the Baby-step Giant-step algorithm for small instances of modulus p [36].

Conclusion

This thesis described almost 40 protocols and ciphers based on integer factorization problem, quadratic residuosity problem and discrete logarithm problem. Some of the described schemes are widely recognized such as the RSA encryption scheme whereas some other schemes are quite new such as the Djebaili-Melkemi encryption scheme.

SageMath and MATLAB were chosen as an alternative to Magma. Both Magma and SageMath offer a large number of algorithms for solving integer factorization problem. Integer factorization problem was tested on instances of n where $n = pq$ or $n = p^2q$ where p and q are primes of the same size. For both forms of n, SageMath's implementations of trial division, Pollard's Rho, Elliptic curve method and Quadratic sieve algorithms were more efficient than Magma's implementations.

For discrete logarithm problem, Magma's generic discrete logarithm implementation outperformed SageMath's generic discrete logarithm and Baby-step Giant-step implementations.

As this thesis has a wide range of interests many works can continue from this point. Choosing optimal parameters for Pollard's $p - 1$, Williams's $p + 1$ and Elliptic curve method, choosing parameters for NFS or index calculus algorithms. Also, different hard mathematical problems can be chosen.

Measurements

This chapter contains results presented in tables. 20 instances were generated for each bit length of p. Each table has 6 columns: avg – average time in seconds for instances that did not take longer than 3600 seconds, std – standard deviation in seconds, min – minimum time in seconds needed for solving one instance, max – maximum time in seconds needed for solving one instance, $>$ 3600s – number of instances that did not finish within 3600 seconds and fails – number of instances for which tested implementation did not give any result.

Integer factorization problem

This section contains tables of results of Magma, SageMath and MATLAB implementations of algorithms solving integer factorization problem.

Form pq

This section contains measurements of implementations tested on composite numbers of form pq where p and q are primes of the same size.

Trial division

$\lvert p\rvert$ / time	avg	std	min	max	$>$ 3600s	fails
20 bit	0.26	0.06	0.19	0.49	0	0
30 bit	9.42	0.71	8.63	11.45	0	0

Table 5.1: Magma TrialDivision

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	0.00	0.00	0.00	0.01	0	0
30 bit	1.90	0.14	1.69	2.19	0	0
40 bit	2074.35	320.81	1737.00	2827.00	0	0

Table 5.2: SageMath factor_trial_division

Pollard's $p-1$

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	1.39	0.30	0.65	1.69	0	0
30 bit	21.10	2.70	12.55	25.65	0	0
40 bit	65.42	23.66	37.17	136.88	0	1
50 bit	59.26	20.55	36.80	120.76	0	1
60 bit	60.15	25.51	36.99	145.21	0	2
70 bit	92.54	23.00	59.65	168.43	0	8
80 bit	86.58	22.37	59.17	145.42	0	6
90 bit	89.86	10.71	58.58	94.89	0	15
100 bit	119.10	8.21	84.79	123.40	0	17
110 bit	122.34	0.76	121.41	125.19	0	15
120 bit	119.39	8.68	82.54	121.96	0	18
130 bit	158.72	1.76	155.20	165.11	0	20

Table 5.3: Magma pMinus1

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	0.01	0.01	0.00	0.04	5	0
30 bit	1.73	6.87	0.00	28.40	3	0
40 bit	3.18	9.19	0.01	34.80	6	0
50 bit	135.42	191.81	0.51	461.00	8	0
60 bit	124.70	182.46	4.69	456.00	11	0
70 bit	535.25	18.39	520.00	562.00	16	0
80 bit	580.67	257.31	0.69	732.00	13	0
90 bit	66.00	0.00	66.00	66.00	18	0
100 bit	736.00	0	736.00	736.00	19	0

Table 5.4: SageMath pollard_pm1

Pollard's Rho

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	0.02	0.01	0.00	0.04	0	0
30 bit	0.46	0.35	0.11	1.38	0	0
40 bit	1.93	0.39	1.35	2.83	0	0
50 bit	30.82	16.78	6.16	60.00	0	0
60 bit	517.07	162.28	83.38	628.82	0	11
70 bit	918.31	6.65	905.86	926.85	0	20

Table 5.5: Magma PollardRho

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	0.00	0.00	0.00	0.00	0	0
30 bit	0.05	0.03	0.01	0.11	0	0
40 bit	0.88	0.49	0.27	2.12	0	0
50 bit	30.78	20.10	4.87	82.00	0	0
60 bit	1478.65	642.96	563.00	2518.00	0	0

Table 5.6: SageMath pollardrho_brent

Williams's $p + 1$

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	1.60	0.24	0.92	1.91	0	0
30 bit	119.06	9.53	97.19	136.24	0	0
40 bit	233.98	77.86	162.37	332.44	0	0
50 bit	238.40	87.48	156.44	422.14	0	0
60 bit	209.71	70.26	157.74	409.82	0	2
70 bit	269.49	89.76	199.54	593.29	0	6
80 bit	285.62	48.22	232.70	432.51	0	10
90 bit	372.73	61.35	353.38	629.03	0	15
100 bit	369.48	81.78	282.34	709.18	0	16

Table 5.7: Magma pPlus1

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	0.03	0.08	0.00	0.38	0	0
30 bit	0.61	2.38	0.00	10.70	0	0
40 bit	4.96	15.71	0.01	71.00	0	0
50 bit	182.48	419.75	0.05	1720.00	3	0
60 bit	632.68	914.97	6.60	3100.00	6	0
70 bit	1713.00	1241.64	67.00	3050.00	16	0
80 bit	1576.33	1231.79	227.00	3168.00	14	0
90 bit	101.00	0	101.00	101.00	19	0
100 bit	2922.00	0	2922.00	2922.00	19	0

Table 5.8: SageMath williams_pp1

Elliptic curve method

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	1.13	0.45	0.44	1.66	0	0
30 bit	23.76	15.72	3.99	56.74	0	0
40 bit	496.17	259.16	109.77	967.51	0	0
50 bit	699.64	355.50	199.64	1510.03	0	0
60 bit	886.90	276.30	482.48	1395.52	0	2
70 bit	1046.29	231.79	670.02	1695.95	0	5
80 bit	1132.34	179.59	847.30	1747.27	0	6
90 bit	1124.63	14.64	1102.81	1151.31	0	15
100 bit	1406.76	197.60	1075.73	2116.38	0	13
110 bit	1427.41	37.42	1372.09	1507.89	0	16
120 bit	1376.49	21.32	1337.03	1408.80	0	20

Table 5.9: Magma ECM

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	0.00	0.00	0.00	0.00	0	0
30 bit	0.02	0.01	0.01	0.03	0	0
40 bit	0.03	0.01	0.02	0.06	0	0
50 bit	0.07	0.03	0.03	0.13	0	0
60 bit	0.33	0.22	0.04	0.79	0	0
70 bit	1.81	1.35	0.32	5.44	0	0
80 bit	6.93	8.10	0.15	32.10	0	0
90 bit	22.44	17.76	2.63	64.00	0	0
100 bit	101.83	75.07	16.10	273.00	0	0
110 bit	286.38	215.50	15.90	887.00	0	0
120 bit	967.44	706.47	42.80	2596.00	0	0
130 bit	1950.45	1062.39	369.00	3411.00	9	0
140 bit	1947.75	832.14	909.00	2691.00	16	0

Table 5.10: SageMath ecm.factor

Quadratic sieve

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	0.45	0.22	0.32	1.15	0	0
30 bit	0.40	0.03	0.36	0.44	0	0
40 bit	0.66	0.05	0.56	0.73	0	0
50 bit	1.03	0.23	0.80	1.65	0	0
60 bit	1.60	0.14	1.11	1.71	0	0
70 bit	1.85	0.05	1.77	1.93	0	0
80 bit	2.19	0.31	1.63	2.67	0	0
90 bit	4.48	0.63	2.70	5.42	0	0
100 bit	16.99	2.59	10.98	22.34	0	0
110 bit	51.52	6.71	38.97	65.94	0	0
120 bit	170.91	20.74	135.48	199.58	0	0
130 bit	565.23	97.93	395.93	755.77	0	0
140 bit	1652.10	311.85	1146.69	2209.17	0	0

Table 5.11: Magma MPQS

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
70 bit	0.57	0.14	0.39	0.83	1	0
80 bit	0.87	0.15	0.69	1.29	0	0
90 bit	1.61	0.19	1.18	2.00	0	0
100 bit	8.45	1.68	5.34	10.20	0	0
110 bit	23.43	4.22	18.00	34.40	0	0
120 bit	68.95	8.55	55.10	84.00	0	0
130 bit	349.45	77.79	249.00	529.00	0	0
140 bit	836.00	128.58	550.00	1064.00	0	0
150 bit	3251.87	406.07	2595.00	3588.00	5	0

Table 5.12: SageMath qsieve

Generic factorization

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	0.01	0.01	0.00	0.02	0	0
30 bit	0.06	0.05	0.01	0.17	0	0
40 bit	0.88	0.56	0.15	1.74	0	0
50 bit	1.55	0.31	0.87	1.94	0	0
60 bit	1.95	0.24	1.60	2.32	0	0
70 bit	2.05	0.40	1.20	2.63	0	0
80 bit	2.13	0.52	0.45	3.03	0	0
90 bit	5.37	0.63	3.61	6.52	0	0
100 bit	20.19	2.75	14.09	26.02	0	0
110 bit	70.53	6.61	57.86	84.81	0	0
120 bit	223.07	21.38	183.12	254.53	0	0
130 bit	732.52	99.06	556.48	920.53	0	0
140 bit	2135.51	316.56	1606.66	2753.87	0	0

Table 5.13: Magma Factorization

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	0.00	0.00	0.00	0.00	0	0
30 bit	0.00	0.00	0.00	0.01	0	0
40 bit	0.04	0.06	0.01	0.25	0	0
50 bit	0.26	0.20	0.02	0.76	0	0
60 bit	0.06	0.02	0.04	0.12	0	0
70 bit	0.22	0.04	0.16	0.35	0	0
80 bit	0.61	0.08	0.45	0.73	0	0
90 bit	2.31	0.42	1.51	3.18	0	0
100 bit	9.81	1.32	7.15	11.90	0	0
110 bit	33.55	6.63	23.70	51.30	0	0
120 bit	122.05	26.59	86.00	188.00	0	0
130 bit	546.55	124.40	340.00	789.00	0	0
140 bit	2542.25	406.76	1886.00	3312.00	0	0

Table 5.14: SageMath factor

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	0.02	0.01	0.02	0.03	0	0
30 bit	0.19	0.04	0.18	0.36	0	0
40 bit	0.22	0.03	0.18	0.29	0	0
50 bit	0.57	0.37	0.19	1.34	0	0
60 bit	2.13	1.52	0.36	6.20	0	0
70 bit	15.80	17.48	3.65	76.41	0	0
80 bit	53.91	40.42	5.85	157.60	0	0
90 bit	456.25	359.97	5.40	1630.86	1	0
100 bit	1947.04	1048.55	33.98	3415.25	4	0
110 bit	1260.72	1702.06	596.19	2402.66	17	0

Table 5.15: MATLAB factor

Form p^2q

This section contains measurements of implementations tested on composite numbers of form p^2q where p and q are primes of the same size.

Trial division

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	0.23	0.05	0.18	0.41	0	0
30 bit	9.86	1.00	8.21	11.87	0	0

Table 5.16: Magma TrialDivision

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	0.00	0.00	0.00	0.01	0	0
30 bit	1.92	0.18	1.63	2.17	0	0
40 bit	2004.25	184.84	1737.00	2289.00	0	0

Table 5.17: SageMath factor_trial_division

Pollard's $p - 1$

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	0.89	0.24	0.59	1.35	0	0
30 bit	18.36	1.48	16.28	21.43	0	0
40 bit	41.86	19.68	37.16	125.46	0	0
50 bit	71.45	15.84	59.57	93.16	0	1
60 bit	80.35	29.09	59.20	182.23	0	2
70 bit	112.13	15.19	82.50	126.26	0	8
80 bit	112.21	26.02	82.56	186.35	0	6
90 bit	151.76	14.23	110.68	165.71	0	15
100 bit	155.90	10.67	110.96	162.81	0	17
110 bit	194.85	2.17	193.03	203.44	0	15
120 bit	193.07	13.20	138.08	204.29	0	18
130 bit	235.44	3.00	233.19	246.60	0	20

Table 5.18: Magma pMinus1

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	0.01	0.02	0.00	0.04	0	0
30 bit	1.94	8.28	0.00	37.10	0	0
40 bit	8.06	16.56	0.01	46.90	1	0
50 bit	152.73	228.82	0.53	556.00	7	0
60 bit	989.45	745.81	41.10	2930.00	0	0

Table 5.19: SageMath pollard_pm1

Pollard's Rho

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	0.02	0.01	0.00	0.05	0	0
30 bit	0.49	0.24	0.13	1.18	0	0
40 bit	1.89	0.39	1.21	2.77	0	0
50 bit	38.31	26.13	6.75	100.19	0	0
60 bit	800.50	276.80	93.89	1011.57	0	11
70 bit	1288.94	11.34	1275.03	1324.58	0	20

Table 5.20: Magma PollardRho

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	0.01	0.00	0.00	0.01	0	0
30 bit	0.04	0.02	0.01	0.08	0	0
40 bit	0.97	0.45	0.28	2.26	0	0
50 bit	39.70	21.08	9.95	85.00	0	0
60 bit	931.05	593.01	161.00	2929.00	0	0

Table 5.21: SageMath pollardrho_brent

William's $p+1$

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	1.55	0.19	0.89	1.77	0	0
30 bit	146.91	17.17	116.34	200.41	0	0
40 bit	230.23	80.92	157.67	341.99	0	0
50 bit	290.90	101.63	202.62	523.86	0	0
60 bit	245.02	68.66	195.97	482.73	0	2
70 bit	311.64	75.63	236.99	533.56	0	6
80 bit	280.48	36.46	231.91	352.04	0	10
90 bit	368.42	63.82	345.50	638.29	0	15
100 bit	369.79	82.34	284.64	712.84	0	16
110 bit	419.73	0.94	417.97	420.97	0	18
120 bit	423.21	1.79	421.43	427.26	0	20

Table 5.22: Magma pPlus1

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	0.04	0.10	0.00	0.47	0	0
30 bit	0.93	3.55	0.00	16.00	0	0
40 bit	9.63	31.44	0.02	142.00	0	0
50 bit	353.32	833.77	0.07	3407.00	3	0
60 bit	579.43	620.23	15.10	1625.00	8	0
70 bit	1758.50	2290.32	139.00	3378.00	18	0
80 bit	973.67	652.82	408.00	1688.00	17	0
90 bit	216.00	0	216.00	216.00	19	0

Table 5.23: SageMath williams_pp1

Elliptic curve method

$\|p\|$ / time	avg	std	min	max	>3600s	fails
20 bit	0.62	0.08	0.52	0.69	0	0
30 bit	33.90	23.19	4.82	88.85	0	0
40 bit	490.66	329.65	104.80	1224.08	0	0
50 bit	1062.83	511.38	189.21	2041.16	0	0
60 bit	1014.52	280.24	830.89	1912.37	0	2
70 bit	1382.99	368.38	555.62	2119.59	0	4
80 bit	1421.64	273.68	1052.07	2100.28	0	7
90 bit	1828.72	236.32	1329.75	2114.48	0	8
100 bit	1699.36	33.59	1648.05	1823.85	0	15
110 bit	1972.96	88.96	1601.47	2022.70	0	19

Table 5.24: Magma ECM

$\|p\|$ / time	avg	std	min	max	>3600s	fails
20 bit	0.02	0.01	0.01	0.04	0	0
30 bit	0.02	0.01	0.01	0.04	0	0
40 bit	0.03	0.01	0.02	0.07	0	0
50 bit	0.12	0.08	0.04	0.31	0	0
60 bit	0.56	0.25	0.06	1.07	0	0
70 bit	3.03	2.31	0.20	8.44	0	0
80 bit	6.80	5.44	0.71	20.20	0	0
90 bit	41.36	41.73	7.35	183.00	0	0
100 bit	121.72	120.53	22.00	530.00	0	0
110 bit	490.35	402.21	78.00	1549.00	0	0
120 bit	1287.60	791.37	279.00	3054.00	0	0
130 bit	1926.00	1073.63	752.00	3283.00	14	0
140 bit	712.00	0	712.00	712.00	19	0

Table 5.25: SageMath ecm.factor

Quadratic sieve

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	0.54	0.21	0.38	1.12	0	0
30 bit	0.95	0.26	0.66	1.46	0	0
40 bit	1.55	0.12	1.11	1.70	0	0
50 bit	2.05	0.10	1.93	2.33	0	0
60 bit	4.26	0.48	3.30	5.13	0	0
70 bit	27.39	3.91	19.35	35.19	0	0
80 bit	176.66	28.48	129.45	221.83	0	0
90 bit	907.44	110.60	762.75	1123.26	0	0

Table 5.26: Magma MPQS

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
50 bit	0.81	0.32	0.49	1.41	0	0
60 bit	2.17	0.35	1.68	3.02	0	0
70 bit	9.52	1.41	6.77	11.60	0	0
80 bit	67.10	12.34	51.20	94.00	0	0
90 bit	456.40	55.28	384.00	603.00	0	0
100 bit	3164.93	344.99	2821.00	3593.00	5	0

Table 5.27: SageMath qsieve

Generic factorization

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	0.06	0.04	0.01	0.18	0	0
30 bit	0.27	0.14	0.10	0.63	0	0
40 bit	0.84	0.52	0.17	1.60	0	0
50 bit	1.54	0.15	1.16	1.75	0	0
60 bit	2.07	1.13	1.15	5.62	0	0
70 bit	5.90	10.28	0.66	37.20	0	0
80 bit	9.93	9.31	0.51	35.28	0	0
90 bit	68.62	56.59	15.11	217.88	0	0
100 bit	213.66	139.72	23.15	542.01	0	0
110 bit	779.13	746.99	70.68	2550.94	0	0
120 bit	2256.47	1269.47	261.03	3427.71	10	0
130 bit	1700.09	2204.25	1666.92	1733.25	18	0
140 bit	2664.79	889.36	1638.45	3208.58	17	0

Table 5.28: Magma Factorization

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	0.00	0.01	0.00	0.01	0	0
30 bit	0.04	0.05	0.01	0.18	0	0
40 bit	0.05	0.02	0.03	0.10	0	0
50 bit	0.64	0.26	0.06	1.05	0	0
60 bit	2.09	0.48	0.35	2.75	0	0
70 bit	15.50	5.52	1.14	24.40	0	0
80 bit	116.20	27.88	80.00	198.00	0	0
90 bit	529.14	573.88	26.40	1629.00	0	0
100 bit	751.62	441.39	66.00	1423.00	4	0
110 bit	1380.00	1599.96	607.00	2229.00	16	0
120 bit	3324.00	0	3324.00	3324.00	19	0

Table 5.29: SageMath factor

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	0.18	0.04	0.16	0.34	0	0
30 bit	0.18	0.01	0.17	0.21	0	0
40 bit	0.21	0.03	0.17	0.27	0	0
50 bit	0.62	0.43	0.19	1.75	0	0
60 bit	3.35	2.42	0.58	10.10	0	0
70 bit	16.00	17.49	3.57	76.52	0	0
80 bit	53.95	40.45	5.84	157.64	0	0
90 bit	362.30	246.68	5.41	916.77	0	0
100 bit	1972.34	1052.21	34.04	3411.37	4	0
110 bit	1287.38	836.88	611.09	2466.68	17	0

Table 5.30: MATLAB factor

Quadratic residuosity problem

This section contains tables of results of Magma and SageMath implementations of algorithms solving quadratic residuosity problem.

Type $\mathbb{Z}_n$

This section contains measurements of implementations tested in $\mathbb{Z}_n$ where n is a product of two primes of the same size.

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
10 bit	1.47	0.63	0.00	2.04	0	0
15 bit	194.80	161.09	1.76	430.20	0	0
20 bit	1112.26	0	1112.26	1112.26	19	0

Table 5.31: Magma exhaustive search implementation

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
10 bit	0.31	0.21	0.00	0.56	0	0
15 bit	239.62	200.26	1.94	549.00	0	0
20 bit	1270.00	0	1270.00	1270.00	19	0

Table 5.32: SageMath exhaustive search implementation

$\lvert p\rvert$ / time	avg	std	min	max	$>$ 3600s	fails
10 bit	0.39	0.07	0.26	0.50	0	0
15 bit	722.75	147.54	391.00	1014.00	0	0

Table 5.33: SageMath quadratic_residues

Discrete logarithm problem

This section contains tables of results of Magma and SageMath implementations of algorithms solving discrete logarithm problem.

Type $\mathbb{Z}_p^*$

This section contains measurements of implementations tested in $\mathbb{Z}_p$ where p is a prime.

Generic discrete logarithm

Magma Log						
$\|p\|$ / time	avg	std	min	max	$>$ 3600s	fails
20 bit	0.00	0.01	0.00	0.01	0	0
30 bit	0.01	0.02	0.00	0.10	0	0
40 bit	0.07	0.14	0.01	0.62	0	0
50 bit	0.31	0.29	0.02	0.95	0	0
60 bit	0.55	0.45	0.04	1.66	0	0
70 bit	1.16	0.53	0.29	1.64	0	0
80 bit	1.61	0.41	0.04	2.16	0	0
90 bit	1.64	0.53	0.07	1.91	0	0
100 bit	1.74	0.49	0.81	3.25	0	0
110 bit	2.47	1.81	0.80	8.35	0	0
120 bit	3.21	3.02	0.29	15.73	0	0
130 bit	6.73	7.73	4.01	39.49	0	0
140 bit	9.36	0.85	8.06	11.73	0	0
150 bit	22.39	2.40	19.49	31.36	0	0
160 bit *	41.21	3.17	35.20	45.77	0	0
170 bit	81.18	4.55	71.85	89.48	0	0
180 bit **	161.82	11.40	135.32	185.62	0	0
190 bit	320.85	21.53	284.76	353.04	0	0
200 bit	658.28	89.94	534.73	917.19	1	0
210 bit	1539.21	138.74	1337.71	1806.49	0	0
220 bit	2987.64	373.13	2427.41	3453.76	5	0

Table 5.34: Magma Log

* Two instances caused segmentation fault, therefore the results are for 18 instances.
** One instance caused segmentation fault, therefore the results are for 19 instances.

$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	0.00	0.00	0.00	0.00	0	0
30 bit	0.00	0.01	0.00	0.02	0	0
40 bit	0.06	0.20	0.00	0.88	0	0
50 bit	2.67	6.64	0.00	28.70	0	0
60 bit	70.45	231.80	0.00	1038.00	0	0
70 bit	222.85	836.35	0.04	3358.00	4	0
80 bit	423.81	924.54	0.00	3387.00	6	0
90 bit	807.72	1016.07	0.01	2810.00	8	0
100 bit	248.45	629.31	0.17	2023.00	10	0
110 bit	272.62	468.31	0.10	1163.00	11	0
120 bit	424.28	1008.21	0.07	2482.00	14	0
130 bit	618.60	960.92	21.70	2036.00	16	0

Table 5.35: SageMath discrete_log

SageMath bsgs						
$\|p\|$ / time	avg	std	min	max	> 3600s	fails
20 bit	0.00	0.00	0.00	0.00	0	0
30 bit	0.02	0.01	0.02	0.03	0	0
40 bit	1.65	0.10	1.47	1.88	0	0
50 bit	52.90	4.29	44.20	59.10	0	0
60 bit	2156.80	268.62	1669.00	2558.00	0	0

Table 5.36: SageMath bsgs

Bibliography

[1] Carmichael Function. *Wolfram MathWorld* [online]. Weisstein, Eric W., ©1999–2022 [cit. 2022-04-26]. Dostupné z: `https://mathworld.wolfram.com/CarmichaelFunction.html`

[2] SHANKS, Daniel. *Solved and unsolved problems in number theory.* American Mathematical Soc., 2001.

[3] Jacobi Symbol. *Wolfram MathWorld* [online]. Weisstein, Eric W., ©1999–2022 [cit. 2022-04-26]. Dostupné z: `https://mathworld.wolfram.com/JacobiSymbol.html`

[4] MENEZES, A. J., Paul C. VAN OORSCHOT a Scott A. VANSTONE. *Handbook of applied cryptography.* Boca Raton: CRC Press, c1997. ISBN 0849385237.

[5] Dr Clifford Cocks CB. *University of Bristol* [online]. Bristol: University of Bristol, © 2002-2021 [cit. 2022-04-26]. Dostupné z: http://www.bristol.ac.uk/graduation/honorary-degrees/hondeg08/cocks.html

[6] POINTCHEVAL, David. New Public Key Cryptosystems Based on the Dependent-RSA Problems. In: *Advances in Cryptology — EUROCRYPT '99.* EUROCRYPT 1999. Lecture Notes in Computer Science, vol 1592. Springer, Berlin, Heidelberg. `https://doi.org/10.1007/3-540-48910-X_17`

[7] SCHNEIER, Bruce. *Applied cryptography: protocols, algorithms, and source code in C.* 2nd ed. New York: Wiley, c1996. ISBN 0-471-12845-7.

[8] FISCHER, Michael J.; MICALI, Silvio; RACKOFF, Charles. A secure protocol for the oblivious transfer. *Journal of Cryptology*, 1996, 9.3: 191-196.

[9] WILLIAMS, Henry. A modification of the RSA public-key encryption procedure (Corresp.). *IEEE Transactions on Information Theory*, 1980, 26.6: 726-729.

[10] FUJIOKA, Atsushi; OKAMOTO, Tatsuaki; MIYAGUCHI, Shoji. ESIGN: An efficient digital signature implementation for smart cards. In: *Workshop on the Theory and Application of Cryptographic Techniques*. Springer, Berlin, Heidelberg, 1991. p. 446-457.

[11] SCHMIDT-SAMOA, Katja. A new rabin-type trapdoor permutation equivalent to factoring. *Electronic Notes in Theoretical Computer Science*, 2006, 157.3: 79-94.

[12] AL-HAIJA, Qasem Abu; ASAD, Mohamad M.; MAROUF, Ibrahim. A systematic expository review of Schmidt-Samoa cryptosystem. *Int. J. Math. Sci. Comput.(IJMSC)*, 2018, 4.2: 12-21.

[13] BENALOH, Josh. Dense probabilistic encryption. In: *Proceedings of the workshop on selected areas of cryptography*. 1994. p. 120-128.

[14] BUDIMAN, M. A.; RACHMAWATI, D. A tutorial on using Benaloh public key cryptosystem to encrypt text. In: *Journal of Physics: Conference Series*. IOP Publishing, 2020. p. 012039.

[15] DJEBAILI, Karima; MELKEMI, Lamine. A Different Encryption System Based on the Integer Factorization Problem. *Malaysian Journal of Computing and Applied Mathematics*, 2020, 3.1: 47-51.

[16] ARIFFIN, Muhammad Rezal Kamel, et al. A New Efficient Asymmetric Cryptosystem Based on the Integer Factorization Problem of $N = p^2q$. *Malaysian Journal of Mathematical Sciences*, 2013, 7: 19-37.

[17] KUROSAWA, Kaoru; ITO, Toshiya; TAKEUCHI, Masashi. Public key cryptosystem using a reciprocal number with the same intractability as factoring a large number. Cryptologia, 1988, 12.4: 225-233.

[18] GALINDO, David, et al. A practical public key cryptosystem from Paillier and Rabin schemes. In: *International Workshop on Public Key Cryptography*. Springer, Berlin, Heidelberg, 2003. p. 279-291.

[19] OKAMOTO, Tatsuaki., UCHIYAMA, Shigenori. A new public-key cryptosystem as secure as factoring. In: *Advances in Cryptology — EUROCRYPT'98*. EUROCRYPT 1998. Lecture Notes in Computer Science, vol 1403. Springer, Berlin, Heidelberg. https://doi.org/10.1007/BFb0054135

[20] BLUM, Lenore; BLUM, Manuel; SHUB, Michael. *A simple secure pseudo-random number generator*. Electronics Research Laboratory, College of Engineering, University of California, 1982.

[21] RICHARDSON, Kert. Progress on probabilistic encryption schemes. 2006.

[22] FEIGE, Uriel; FIAT, Amos; SHAMIR, Adi. Zero-knowledge proofs of identity. *Journal of cryptology*, 1988, 1.2: 77-94. https://doi.org/10.1007/BF02351717

[23] CHAUM, David; ANTWERPEN, Hans Van. Undeniable signatures. In: *Conference on the Theory and Application of Cryptology*. Springer, New York, NY, 1989. p. 212-216.

[24] CRAMER, Ronald; SHOUP, Victor. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In: *Annual international cryptology conference*. Springer, Berlin, Heidelberg, 1998. p. 13-25.

[25] Magma Computational Algebra System. *Magma Computational Algebra System* [online]. Computational Algebra Group, © 2010-2022 [cit. 2022-05-08]. Dostupné z: http://magma.maths.usyd.edu.au/magma/

[26] SageMath - Open-Source Mathematical Software System. *SageMath - Open-Source Mathematical Software System* [online]. [cit. 2022-05-08]. Dostupné z: https://www.sagemath.org

[27] MATLAB & Simulink - MathWorks. *MATLAB & Simulink - MathWorks* [online]. The MathWorks, © 1994-2022 [cit. 2022-05-08]. Dostupné z: https://www.mathworks.com/products/matlab.html

[28] Factorization. *Magma Computational Algebra System* [online]. Computational Algebra Group, © 2010-2022 [cit. 2022-04-16]. Dostupné z: http://magma.maths.usyd.edu.au/magma/handbook/text/182#1445

[29] BRILLHART, John, et al. Factorizations of $b^n \pm 1, b = 2, 3, 5, 6, 7, 10, 11, 12$ Up to High Powers. 1988. Dostupné z https://cir.nii.ac.jp/crid/1571698599389824512

[30] Integer factorization functions. *SageMath Documentation* [online]. The Sage Development Team, © 2005-2022 [cit. 2022-04-16]. Dostupné z: https://doc.sagemath.org/html/en/reference/rings_standard/sage/rings/factorint.html

[31] SILVERMAN, Robert D.; WAGSTAFF, Samuel S. A practical analysis of the elliptic curve factoring algorithm. *Mathematics of computation*, 1993, 61.203: 445-462.

[32] Integer Factorization. *SageMath Documentation* [online]. The Sage Development Team, © 2005-2022 [cit. 2022-04-18]. Dostupné z:

https://doc.sagemath.org/html/en/thematic_tutorials/explicit_methods_in_number_theory/integer_factorization.html

[33] Primefac. *The Python Package Index* [online]. Lucas Brown, © 2022 [cit. 2022-04-18]. Dostupné z: https://pypi.org/project/primefac/

[34] Factorization - MATLAB factor. *MathWorks - Makers of MATLAB and Simulink - MATLAB & Simulink* [online]. The MathWorks, © 1994-2022 [cit. 2022-04-16]. Dostupné z: https://www.mathworks.com/help/symbolic/factor.html

[35] Elementary number theory. *SageMath Documentation* [online]. The Sage Development Team, © 2005-2022 [cit. 2022-04-18]. Dostupné z: https://doc.sagemath.org/html/en/constructions/number_theory.html

[36] Discrete Logarithms. *Magma Computational Algebra System* [online]. Sydney: Computational Algebra Group, © 2010-2022 [cit. 2022-04-16]. Dostupné z: http://magma.maths.usyd.edu.au/magma/handbook/text/211

[37] Miscellaneous generic functions. *SageMath Documentation* [online]. The Sage Development Team, © 2005-2022 [cit. 2022-04-16]. Dostupné z: https://doc.sagemath.org/html/en/reference/groups/sage/groups/generic.html

[38] OpenSSL. *OpenSSL* [online]. The OpenSSL Project Authors, © 1999-2021 [cit. 2022-05-08]. Dostupné z: https://www.openssl.org/

Appendix A

Acronyms

gcd Greatest common divisor

lcm Least common multiple

DLP Discrete logarithm problem

ECC Elliptic curve cryptography

ECM Elliptic curve method

MPQS Multiple polynomial quadratic sieve

QRP Quadratic residuosity problem

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `thesis` ..... text of the thesis
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format
  - `latex` ..... the directory of LaTeX source codes of the thesis
- `test_generator` ..... the directory of source codes for generating tests