MATHEMATICAL CRYPTOLOGY

Keijo Ruohonen

(Translation by Jussi Kangas and Paul Coughlan)

2014

Contents

Foreword

These lecture notes were translated from the Finnish lecture notes for the TUT course "Matemaattinen kryptologia". The laborious bulk translation was taken care of by the students Jussi Kangas (visiting from the University of Tampere) and Paul Coughlan (visiting from the University of Dublin, Trinity College). I want to thank the translation team for their effort.

The notes form the base text for the course "MAT-52606 Mathematical Cryptology". They contain the central mathematical background needed for understanding modern data encryption methods, and introduce applications in cryptography and various protocols.

Though the union of mathematics and cryptology is old, it really came to the fore in connection with the powerful encrypting methods used during the Second World War and their subsequent breaking. Being generally interesting, the story is told in several (partly) fictive books meant for the general audience.[1]

The area got a whole new speed in the 1970's when the completely open, fast and strong computerized cryptosystem DES went live, and the revolutionary public-key paradigm was introduced.[2] After this, development of cryptology and also the mathematics needed by it

[1] An example is Neal Stephenson's splendid *Cryptonomicon.*

[2] Steven Levy's book *Crypto. Secrecy and Privacy in the New Code War* gives a bit romanticized description of the birth of public-key cryptography.

—mostly certain fields of number theory and algebra—has been remarkably fast. It is no exaggeration to say that the recent popularity of number theory and algebra is expressly because of cryptology. The theory of computational complexity, which belongs to the field of theoretical computer science, is often mentioned in this context, but in all fairness it must be said that it really has no such big importance in cryptology. Indeed, suitable mathematical problems for use in cryptography are those that have been studied by top mathematicians for so long that only results that are extremely hard to prove still remain open. Breaking the encryption then requires some huge theoretical breakthrough. Such problems can be found in abundance especially in number theory and discrete algebra.

Results of number theory and algebra, and the related algorithms, are presented in their own chapters, suitably divided into parts. Classifying problems of number theory and algebra into computationally ”easy” and ”hard” is essential here. The former are needed in encrypting and decrypting and also in setting up cryptosystems, the latter guarantee strength of encryption. The fledgling quantum cryptography is briefly introduced together with its backgrounds.

Only a few classical cryptosystems—in which also DES and the newer AES must be included according to their description—are introduced, much more information about these can be found e.g. in the references BAUER, MOLLIN and SALOMAA. The main concern here is in modern public-key methods. This really is not an indication of the old-type systems not being useful. Although the relevance of old classical methods vanished quite rapidly[3], newer methods of classical type are widely used and have a very important role in fast mass-encryption. Also stream encrypting, so important in many applications, is not treated here. The time available for a single course is limited. A whole different chapter would be correct implementation and use of cryptosystems, which in a mathematics course such as this cannot really be touched upon. Even very powerful cryptosystem can be made inefficient with bad implementation and careless use.[4]

Keijo Ruohonen

[3] As an example of this it may be mentioned that the US Army field manual FM 34-40-2: *Basic Cryptanalysis* is publicly available in the web. The book BAUER also contains material quite recently (and possibly still!) classified as secret.

[4] A great book on this topic is Bruce Schneier's *Secrets and Lies. Digital Security in a Networked World.*

"Cryptography involves one genius trying to
work out what another genius has done."

(MAI JIA: *Decoded*)

Chapter 1

Introduction

Encryption of a message means the information in it is hidden so that anyone who's reading (or listening to) the message, can't understand any of it unless he/she can *break* the encryption. An original plain message is called *plaintext* and an encrypted one *cryptotext.* When encrypting you need to have a so-called *key*, a usually quite complicated parameter that you can use to change the encryption. If the encrypting procedure remains unchanged for a long time, the probability of breaking the encryption will in practise increase substantially. Naturally different users need to have their own keys, too.

The receiver of the message *decrypts* it, for which he/she needs to have his/her own key. Both the encrypting key and decrypting key are very valuable for an eavesdropper, using the encrypting key he/she can send encrypted fake messages and using the decrypting key he/she can decrypt messages not meant to him/her. In symmetric cryptosystems both the encrypting key and the decrypting key are usually the same.

An encrypting procedure can encrypt a continuous stream of symbols (*stream encryption*) or divide it into blocks (*block encryption*). Sometimes in block encryption the sizes of blocks can vary, but a certain maximum size of block must not be exceeded. However, usually blocks are of the same size. In what follows we shall only examine block encryption, in which case it's sufficient to consider encrypting and decrypting of an arbitrary message block, and one arbitrary message block may be considered as the plaintext and its encrypted version as the cryptotext.

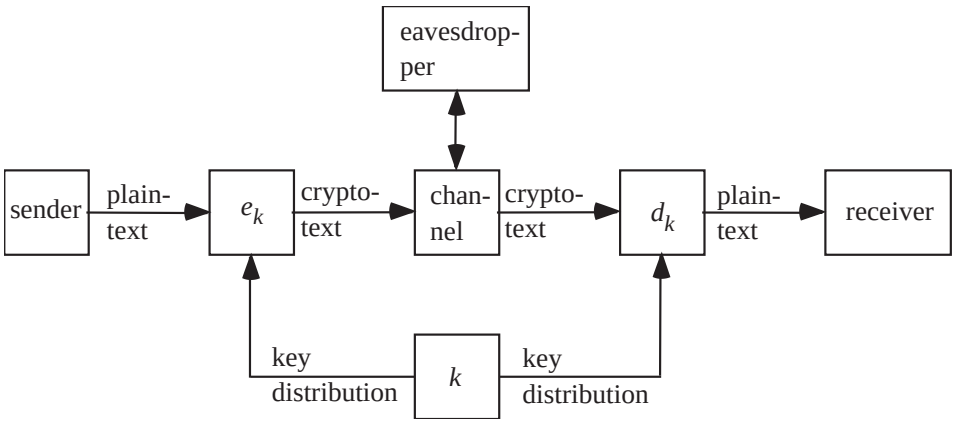
An encryption procedure is *symmetric,* if the encrypting and decrypting keys are the same or it's easy to derive one from the other. In *nonsymmetric* encryption the decrypting key can't be derived from the encrypting key with any small amount of work. In that case the encrypting key can be public while the decrypting key stays classified. This kind of encryption procedure is known as *public-key cryptography,* correspondingly symmetric encrypting is called *secret-key cryptography.* The problem with symmetric encrypting is the secret key distribution to all parties, as keys must also be updated every now and then.

Symmetric encryption can be characterized as a so called *cryptosystem* which is an ordered quintet (P, C, K, E, D), where

- P is the finite *message space* (plaintexts).
- C is the finite *cryptotext space* (cryptotexts).
- K is the finite *key space.*
- for every key $k \in K$ there is an *encrypting function* $e_k \in E$ and a *decrypting function* $d_k \in D$. E is called the *encrypting function space* which includes every possible encrypting function and D is called the *decrypting function space* which includes every possible decrypting function.

- $d_k(e_k(w)) = w$ holds for every message (block) w and key k.

It would seem that an encrypting function must be *injective,* so that it won't encrypt two different plaintexts to the same cryptotext. Encryption can still be random, and an encrypting function can encrypt the same plaintext to several different cryptotexts, so an encrypting function is not actually a mathematical function. On the other hand, encrypting functions don't always have to be injective functions, if there's a limited amount of plaintexts which correspond to the same cryptotext and it's easy to find the right one of them.



Almost all widely used encryption procedures are based on results in number theory or algebra (group theory, finite fields, commutative algebra). We shall introduce these theories as we need them.

"So in order to remove the contingent and subjective elements from cryptography there have been concerted efforts in recent years to transform the field into a branch of mathematics, or at least a branch of the exact sciences. In my view, this hope is misguided, because in its essence cryptography is as much an art as a science."

(NEAL KOBLITZ)

Chapter 2

NUMBER THEORY. PART 1

2.1 Divisibility. Factors. Primes

Certain concepts and results of number theory[1] come up often in cryptology, even though the procedure itself doesn't have anything to do with number theory. The set of all integers is denoted by $\mathbb{Z}$. The set of nonnegative integers $\{0, 1, 2, \dots\}$ is called the set of *natural numbers* and it's denoted by $\mathbb{N}$.

Addition and multiplication of integers are familiar commutative and associative operations, with identity elements 0 and 1 respectively. Also recall the distributive law $x(y+z) = xy + xz$ and the definitions of opposite number $-x = (-1)x$ and subtraction $x - y = x + (-1)y$. *Division* of integers means the following operation: When dividing an integer x (*dividend*) by an integer $y \neq 0$ (*divisor*), x is to be given in the form

$$x = qy + r$$

where the integer r is called *remainder* and fulfills the condition $0 \leq r < |y|$. The integer q is called *quotient.* Adding repeatedly $-y$ or y to x we see that it's possible to write x in the desired form. If it's possible to give x in the form

$$x = qy,$$

where q is an integer then it's said that x *is divisible by* y or that y *divides* x or that y *is a factor of* x or that x *is a multiple of* y, and this is denoted by $y \mid x$. The so-called *trivial factors* of an integer x are ± 1 and $\pm x$. Possible other factors are *nontrivial.*

The following properties of divisibility are quite obvious:

(1) 0 is divisible by any integer, but divides only itself.

(2) 1 and -1 divide all integers, but are divisible only by themselves and by one another.

(3) If $y \mid x$ and $x \neq 0$ then $|y| \leq |x|$.

(4) If $x \mid y$ and $y \mid z$ then also $x \mid z$ (in other words, divisibility is transitive).

[1] *Number theory* is basically just the theory of integers. There are however different extensions of number theory. For example, we can include algebraic numbers—roots of polynomials with integral coefficients—which leads us to *algebraic number theory,* very useful in cryptology, see e.g. KOBLITZ. On the other hand, number theory can be studied using other mathematical formalisms. For example, *analytic number theory* studies integers using procedures of mathematical analysis—integrals, series and so on—and this too is usable in cryptology, see SHPARLINSKI.

(5) If $x \mid y$ and $x \mid z$ then also $x \mid y \pm z$.

(6) If $x \mid y$ and z is an integer then $x \mid yz$.

The result of division is unique since, if

$$x = q_1 y + r_1 = q_2 y + r_2,$$

where q_1, q_2, r_1, r_2 are integers and $0 \leq r_1, r_2 < |y|$, then y divides $r_1 - r_2$. From the fact that $|r_1 - r_2| < |y|$ it then follows that $r_1 = r_2$ and further that $q_1 = q_2$.

An integer that has only trivial factors is called *indivisible.* An indivisible integer is a *prime number* or just a *prime*[2], if it is ≥ 2. The first few primes are

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, \ldots$$

2 is the only even prime. One basic task is to test whether or not a natural number is a prime. An integer, which is ≥ 2 and is not a prime, is called *composite.*

Theorem 2.1. *If the absolute value of an integer is ≥ 2 then it has a* prime factor.

Proof. If $|x| \geq 2$ then a prime factor p of x can be found by the following algorithm:

1. Set $z \leftarrow x$.
2. If z is indivisible then $p = |z|$.
3. If z is divisible, we take its nontrivial factor u. Then set $z \leftarrow u$ and move back to #2.

The procedure stops because in the third step $|z|$ gets smaller and smaller, so ultimately z will be a prime. □

Corollary. *The number of primes is infinite.*

Proof. An infinite list of primes can be obtained by the following procedure, known already to ancient Greeks. (It is not believed to produce all primes, however, but this is an open problem.)

1. Set $\mathcal{P} \leftarrow 2$. Here $\mathcal{P}$ is a sequence variable.
2. If $\mathcal{P} = p_1, \ldots, p_n$ then compute $x = p_1 \cdots p_n + 1$. Notice that none of the primes in the sequence $\mathcal{P}$ divide x (remember uniqueness of division).
3. By Theorem 2.1, x has a prime factor p, which is not any of the primes in the sequence $\mathcal{P}$. Find some such p, and set $\mathcal{P} \leftarrow \mathcal{P}, p$ and return to #2.

The first few primes produced by the procedure are $3, 7, 43, 13, 53, 5, 6\,221\,671, \ldots$ □

Basic tasks concerning primes are for example the following:

(1) Compute the n^{th} prime in order of magnitude.

(2) Compute the n first primes in order of magnitude.

(3) Compute the largest (resp. smallest) prime, which is $\leq x$ (resp. $\geq x$).

(4) Compute primes, which are $\leq x$.

[2]The set of all primes is sometimes denoted by $\mathbb{P}$.

Theorem 2.2. *An integer $x \neq 0$ can be written as a product of primes (disregarding the sign), this is the so-called* factorization. *In particular, it is agreed that the number 1 is the so-called empty product, that is, a product which has no factors.*

Proof. The algorithm below produces a sequence of primes, whose product is $= \pm x$:

1. Set $\mathcal{T} \leftarrow \text{NULL}$ (the empty sequence).
2. If $x = \pm 1$ then we return $\mathcal{T}$, and stop. Remember that the empty product is $= 1$.
3. If $x \neq 1$ then we find some prime factor p of x (Theorem 2.1). Now $x = py$. Set $\mathcal{T} \leftarrow \mathcal{T}, p$ and $x \leftarrow y$ and go back to #2.

This procedure stops because in the third step $|x|$ gets smaller, and is eventually $= 1$ whereafter we halt at #2. In particular, the empty sequence is returned if $x = \pm 1$. □

Later we will show that this factorization is in fact unique when we disregard permutations of factors, see Section 2.3. Naturally, one basic task is to find the factorization of a given integer. This is computationally very hard, see Section 7.5.

2.2 Representations of Integers in Different Bases

The most common way to represent an integer is to use the familiar *decimal representation* or in other words base-10 representation. Base-2 representation, called the *binary representation,* is also often used and so is base-8 *octal representation* and base-16 *hexadecimal representation.* The general base representation is given by

Theorem 2.3. *If $k \geq 2$ then every positive integer x can be represented uniquely in the form*

$$x = a_n k^n + a_{n-1} k^{n-1} + \cdots + a_1 k + a_0$$

where $0 \leq a_0, a_1, \ldots, a_n \leq k - 1$ and $a_n > 0$. This is called base-k representation *of x, where k is the* base (number) *or* radix *and $n + 1$ is the* length *of the representation.*

Proof. The representation, i.e. the sequence $a_n, a_{n-1}, \ldots, a_0$, is obtained by the following algorithm:

1. Set $\mathcal{K} \leftarrow \text{NULL}$ (the empty sequence).
2. Divide x by the radix k:

   $$x = qk + r \quad \text{(quotient } q\text{, remainder } r\text{)}.$$

   Set $\mathcal{K} \leftarrow r, \mathcal{K}$ and $x \leftarrow q$.
3. If $x = 0$ then return $\mathcal{K}$ and quit. Else repeat #2.

x gets smaller and smaller in #2 with each iteration and so the procedure stops eventually in #3.

The base-k representation is unique because if

$$x = a_n k^n + a_{n-1} k^{n-1} + \cdots + a_1 k + a_0 = b_m k^m + b_{m-1} k^{m-1} + \cdots + b_1 k + b_0,$$

where $0 \leq a_0, a_1, \ldots, a_n, b_0, b_1, \ldots, b_m \leq k-1$ and $a_n, b_m > 0$ and $n \geq m$, then we first conclude that $n = m$. Indeed, if $n > m$ then we also have

$$\begin{aligned} b_m k^m + b_{m-1}k^{m-1} + \cdots + b_1 k + b_0 &\leq (k-1)k^m + (k-1)k^{m-1} + \cdots + (k-1)k + k - 1 \\ &= k^{m+1} - 1 \\ &< k^{m+1} \leq k^n \leq a_n k^n + a_{n-1}k^{n-1} + \cdots + a_1 k + a_0, \end{aligned}$$

which is a contradiction. So $n = m$, that is, the length of the representation must be unique. Similarly we can conclude that $a_n = b_n$, because if $a_n > b_n$ then

$$\begin{aligned} b_n k^n + b_{n-1}k^{n-1} + \cdots + b_1 k + b_0 &\leq (a_n - 1)k^n + (k-1)k^{n-1} + \cdots + (k-1)k + k - 1 \\ &= a_n k^n - 1 \\ &< a_n k^n + a_{n-1}k^{n-1} + \cdots + a_1 k + a_0, \end{aligned}$$

which is also a contradiction. Again in the same way we can conclude that $a_{n-1} = b_{n-1}$ and so on. □

Representation of the number 0 is basically an empty sequence in every base. This of course creates problems and so we agree on the convention that the representation of 0 is 0. Conversion between base representations, the so-called *change of base* or *radix transformation,* is a basic task concerning integers.

Theorem 2.4. *The length of the base-k representation of a positive integer x is*

$$\lfloor \log_k x \rfloor + 1 = \lceil \log_k (x+1) \rceil$$

where $\log_k$ is the base-k logarithm.[3]

Proof. If the base-k representation of x is $x = a_n k^n + a_{n-1}k^{n-1} + \cdots + a_1 k + a_0$ then its length is $s = n + 1$. It is apparent that $x \geq k^n$, and on the other hand that

$$x \leq (k-1)k^n + (k-1)k^{n-1} + \cdots + (k-1)k + k - 1 = k^{n+1} - 1 < k^{n+1}.$$

Since $k^{s-1} \leq x < k^s$, then $s - 1 \leq \log_k x < s$ and so

$$s = \lfloor \log_k x \rfloor + 1.$$

Then again $k^{s-1} < x + 1 \leq k^s$, whence $s - 1 < \log_k (x+1) \leq s$ and so

$$s = \lceil \log_k (x+1) \rceil.$$

□

2.3 Greatest Common Divisor and Least Common Multiple

The *greatest common divisor* (*g.c.d.*) of the integers x and y is the largest integer d which divides both integers, denoted

$$d = \gcd(x, y).$$

The g.c.d. exists if at least one of the integers x and y is $\neq 0$. Note that the g.c.d. is positive. (It's often agreed, however, that $\gcd(0, 0) = 0$.) If $\gcd(x, y) = 1$ then we say that x *and* y *have no common divisors* or that they are *coprime.*

[3] Remember that change of base of logarithms is done by the formula $\log_k x = \ln x / \ln k$. Here $\lfloor x \rfloor$ denotes the so-called *floor* of x, i.e. the largest integer which is $\leq x$. Correspondingly $\lceil x \rceil$ denotes the so-called *ceiling* of x, i.e. the smallest integer which is $\geq x$. These floor and ceiling functions crop up all over number theory!

Theorem 2.5. (Bézout's theorem) *The g.c.d. d of the integers x and y, at least one of which is $\neq 0$, can be written in the form*

$$d = c_1 x + c_2 y \quad \textit{(the so-called} \text{ Bézout form}\textit{)}$$

where c_1 and c_2 are integers, the so-called Bézout coefficients. *Also, if $x, y \neq 0$, then we may assume that $|c_1| \leq |y|$ and $|c_2| \leq |x|$.*

Proof. Bézout's form and the g.c.d. d are produced by the following so-called *(Generalized) Euclidean algorithm.* Here we may assume that $0 \leq x \leq y$, without loss of generality. Denote $\mathrm{GCD}(x, y) = (d, c_1, c_2)$.

(Generalized) Euclidean algorithm:

1. If $x = 0$ then we come out of the algorithm with $\mathrm{GCD}(x, y) = (y, 0, 1)$ and quit.
2. If $x > 0$ then first we divide y with x: $y = qx + r$, where $0 \leq r < x$. Next we find $\mathrm{GCD}(r, x) = (d, e_1, e_2)$. Now

$$d = e_1 r + e_2 x = e_1(y - qx) + e_2 x = (e_2 - e_1 q)x + e_1 y.$$

   We end the algorithm by returning $\mathrm{GCD}(x, y) = (d, e_2 - e_1 q, e_1)$ and quit.

Since $r = y - qx$, $\gcd(x, y)$ divides r and hence $\gcd(x, y) \leq \gcd(x, r)$. Similarly $\gcd(x, r)$ divides y and thus $\gcd(x, r) \leq \gcd(x, y)$, so $\gcd(x, r) = \gcd(x, y)$. Hence #2 produces the correct result. The recursion ends after a finite number of iterations because $\min(r, x) < \min(x, y)$, and so every time we call GCD (iterate) the minimum value gets smaller and is eventually $= 0$.

If $x, y \neq 0$ then apparently right before stopping in #1 in the recursion we have $y = qx$ and $r = 0$ and $d = x$, whence at that point $c_1 = 1 \leq y$ and $c_2 = 0 \leq x$. On the other hand, every time when in #2 we have $y = qx + r$ and $d = e_1 r + e_2 x$, where $|e_1| \leq x$ and $|e_2| \leq r$, then e_1 and e_2 have opposite signs and thus $|e_2 - e_1 q| = |e_2| + |e_1| q \leq r + xq = y$. So, the new coefficients $c_1 = e_2 - e_2 q$ and $c_2 = e_1$ will then also satisfy the claimed conditions. □

Example. *As a simple example, let's compute* $\gcd(15, 42)$ *and its Bézout form. We use indentation to indicate recursion level:*

$\gcd(15, 42) = ?$
  $42 = 2 \cdot 15 + 12\ ,\ q = 2$
  $\gcd(12, 15) = ?$
    $15 = 1 \cdot 12 + 3\ ,\ q = 1$
    $\gcd(3, 12) = ?$
      $12 = 4 \cdot 3 + 0\ ,\ q = 4$
      $\gcd(0, 3) = ?$
      $\mathrm{GCD}(0, 3) = (3, 0, 1)$
    $\mathrm{GCD}(3, 12) = (3, 1 - 0 \cdot 4, 0) = (3, 1, 0)$
  $\mathrm{GCD}(12, 15) = (3, 0 - 1 \cdot 1, 1) = (3, -1, 1)$
$\mathrm{GCD}(15, 42) = (3, 1 - (-1) \cdot 2, -1) = (3, 3, -1)$

So, the g.c.d. is 3 *and the Bézout form is* $3 = 3 \cdot 15 + (-1) \cdot 42$.

You can get the next result straight from Bézout's theorem:

Corollary. *If the integer z divides the integers x and y, at least one of which is $\neq 0$, then it also divides* $\gcd(x, y)$.

NB. *Due to this corollary* $\gcd(x, y)$ *is often defined as the common divisor of* x *and* y*, which is divisible by every common divisor of these integers. This leads to the same concept of g.c.d. Such a definition is also suitable for the situation* $x = y = 0$ *and gives the formula* $\gcd(0, 0) = 0$ *(mentioned above).*

Another corollary of Bézout's theorem is uniqueness of factorization of integers, see Theorem 2.2.

Theorem 2.6. *Factorization of an integer* $x \neq 0$ *is unique.*

Proof. Assume the contrary: There exists an integer x, which has (at least) two different factorizations. We may assume that x is positive and that x is the smallest positive integer that has this property. Thus $x \geq 2$, since the only factorization of 1 is the empty product. Now we can write x as a product of primes with respect to two different factorizations:

$$x = p_1^{i_1} p_2^{i_2} \cdots p_n^{i_n} = q_1^{j_1} q_2^{j_2} \cdots q_m^{j_m}$$

where $p_1, \ldots, p_n$ are different primes and likewise $q_1, \ldots, q_m$ are different primes and $i_1, \ldots, i_n$ as well as $j_1, \ldots, j_m$ are positive integers. In fact, we also know that the primes $p_1, \ldots, p_n$ differ from the primes $q_1, \ldots, q_m$. If, for example, $p_1 = q_1$, then the integer x/p_1 would have two different factorizations and $x/p_1 < x$, a contradiction. So we know that $\gcd(p_1, q_1) = 1$, in Bézout's form

$$1 = c_1 p_1 + c_2 q_1.$$

But it follows from this that

$$q_1^{j_1-1} q_2^{j_2} \cdots q_m^{j_m} = (c_1 p_1 + c_2 q_1) q_1^{j_1-1} q_2^{j_2} \cdots q_m^{j_m} = c_1 p_1 q_1^{j_1-1} q_2^{j_2} \cdots q_m^{j_m} + c_2 x,$$

from which we see further that p_1 divides the product $q_1^{j_1-1} q_2^{j_2} \cdots q_m^{j_m}$, in other words,

$$q_1^{j_1-1} q_2^{j_2} \cdots q_m^{j_m} = p_1 z.$$

Because z and $q_1^{j_1-1} q_2^{j_2} \cdots q_m^{j_m}$ have unique factorizations (they are both smaller than x), it follows from this that p_1 is one of the primes $q_1, \ldots, q_m$ which is a contradiction. So the contrary is false and factorization is unique. □

When giving a rational number in the form x/y, it is usually assumed that $\gcd(x, y) = 1$, in other words, that the number is *with the smallest terms.* This is very important when calculating with large numbers, to prevent numerators and denominators from growing too large. Such reduced form is naturally obtained by dividing x and y by $\gcd(x, y)$, so in long calculations the g.c.d. must be determined repeatedly.

It's important to notice that the bounds of the coefficients mentioned in Bézout's theorem, i.e. $|c_1| \leq |y|$ and $|c_2| \leq |x|$, are valid in every step of the Euclidean algorithm. This way intermediate results won't get too large. On the other hand, the Euclidean algorithm does not even take too many steps:

Theorem 2.7. *When computing* $\gcd(x, y)$*, where* $0 \leq x \leq y$*, the Euclidean algorithm needs no more than* $\lfloor 2 \log_2 y + 1 \rfloor$ *divisions.*

Proof. If $x = 0$ (no divisions) or $x = y$ (one division) there's nothing to prove, so we can concentrate on the case $0 < x < y$. The proof is based on the following simple observation concerning division: Every time we divide integers a and b, where $0 < a < b$, and write $b = qa + r$ (quotient q, remainder r), we have

$$b = qa + r \geq a + r > 2r.$$

When computing $\gcd(x, y)$ using the Euclidean algorithm we get a sequence

$$\begin{aligned} y &= q_1 x + r_1 \qquad (0 < r_1 < x),\\ x &= q_2 r_1 + r_2 \qquad (0 < r_2 < r_1),\\ r_1 &= q_3 r_2 + r_3 \qquad (0 < r_3 < r_2),\\ &\vdots\\ r_{l-2} &= q_l r_{l-1} + r_l \qquad (0 < r_l < r_{l-1}),\\ r_{l-1} &= q_{l+1} r_l \end{aligned}$$

with $l + 1$ divisions. If $l = 2k + 1$ is odd then by our observation above

$$1 \le r_l < 2^{-1} r_{l-2} < 2^{-2} r_{l-4} < \cdots < 2^{-i} r_{l-2i} < \cdots < 2^{-k} r_1 < 2^{-k-1} y = 2^{-\frac{l+1}{2}} y < 2^{-\frac{l}{2}} y,$$

and if $l = 2k$ is even then

$$1 \le r_l < 2^{-1} r_{l-2} < 2^{-2} r_{l-4} < \cdots < 2^{-k+1} r_2 < 2^{-k} x < 2^{-\frac{l}{2}} y.$$

So, in any case $y > 2^{\frac{l}{2}}$ which means that (taking base-2 logarithms) $2 \log_2 y > l$, and the result follows. □

Thus we see that applying the Euclidean algorithm is not very laborious, $\lfloor 2 \log_2 y + 1 \rfloor$ is proportional to the length of the binary representation of y (Theorem 2.4). If you want to know more about the computational efficiency of the Euclidean algorithm, see e.g. KNUTH.

The greatest common divisor of more than two integers $x_1, x_2, \ldots, x_N$

$$d = \gcd(x_1, x_2, \ldots, x_N)$$

is defined in same way as for two integers, so it's the largest integer which divides all the numbers in the sequence $x_1, x_2, \ldots, x_N$. Again we require that at least one of the numbers is $\neq 0$. We may agree that $x_N \neq 0$. This kind of g.c.d. can be computed by applying the Euclidean algorithm $N - 1$ times, since

Theorem 2.8. $$\begin{aligned} \gcd(x_1, x_2, \ldots, x_N) &= \gcd(x_1, \gcd(x_2, \ldots, x_N))\\ &= \gcd(x_1, \gcd(x_2, \gcd(x_3, \ldots, \gcd(x_{N-1}, x_N) \cdots))) \end{aligned}$$

and furthermore the g.c.d. can be written in Bézout's form

$$\gcd(x_1, x_2, \ldots, x_N) = c_1 x_1 + c_2 x_2 + \cdots + c_N x_N.$$

Proof. For a more concise notation we denote

$$d = \gcd(x_1, x_2, \ldots, x_N) \quad \text{and} \quad d' = \gcd(x_1, \gcd(x_2, \gcd(x_3, \ldots, \gcd(x_{N-1}, x_N) \cdots))).$$

By Bézout's theorem

$$\gcd(x_{N-1}, x_N) = e_1 x_{N-1} + e_2 x_N$$

and further

$$\gcd(x_{N-2}, \gcd(x_{N-1}, x_N)) = e_3 x_{N-2} + e_4 \gcd(x_{N-1}, x_N) = e_3 x_{N-2} + e_4 e_1 x_{N-1} + e_4 e_2 x_N$$

and so on, so eventually we see that for some integers $c_1, \ldots, c_N$

$$d' = c_1 x_1 + c_2 x_2 + \cdots + c_N x_N.$$

From here it follows, that $d \mid d'$ and so $d \leq d'$. On the other hand, d' divides both x_1 and the g.c.d.

$$\gcd(x_2, \gcd(x_3, \ldots, \gcd(x_{N-1}, x_N) \cdots)).$$

The g.c.d. above divides both x_2 and $\gcd(x_3, \ldots, \gcd(x_{N-1}, x_N) \cdots)$. Continuing in this way we see that d' divides each number $x_1, x_2, \ldots, x_N$ and therefore $d' \leq d$. We can thus conclude that $d = d'$. □

If the numbers $x_1, x_2, \ldots, x_N$ are $\neq 0$ then they have factorizations

$$x_i = \pm p_1^{j_{i1}} p_2^{j_{i2}} \cdots p_M^{j_{iM}} \quad (i = 1, 2, \ldots, N),$$

where we agree that $j_{ik} = 0$ whenever the prime p_k is not a factor of x_i. It then becomes apparent that

$$\gcd(x_1, x_2, \ldots, x_N) = p_1^{\min(j_{11}, \ldots, j_{N1})} p_2^{\min(j_{12}, \ldots, j_{N2})} \cdots p_M^{\min(j_{1M}, \ldots, j_{NM})}.$$

The trouble when using this result is that factorizations are not generally known and finding them can be very laborious.

The *least common multiple* (*l.c.m.*) of the integers $x_1, x_2, \ldots, x_N$ is the smallest positive integer that is divisible by every number $x_1, x_2, \ldots, x_N$, we denote it by $\operatorname{lcm}(x_1, x_2, \ldots, x_N)$. For the l.c.m. to exist we must have $x_1, x_2, \ldots, x_N \neq 0$. Remembering the factorizations above, we can see that

$$\operatorname{lcm}(x_1, x_2, \ldots, x_N) = p_1^{\max(j_{11}, \ldots, j_{N1})} p_2^{\max(j_{12}, \ldots, j_{N2})} \cdots p_M^{\max(j_{1M}, \ldots, j_{NM})}.$$

The l.c.m. is also obtained recursively using the Euclidean algorithm, without knowledge of factors, since

Theorem 2.9. $\operatorname{lcm}(x_1, x_2, \ldots, x_N) = \operatorname{lcm}(x_1, \operatorname{lcm}(x_2, \ldots, x_N))$

$$= \operatorname{lcm}(x_1, \operatorname{lcm}(x_2, \operatorname{lcm}(x_3, \ldots, \operatorname{lcm}(x_{N-1}, x_N) \cdots)))$$

and

$$\operatorname{lcm}(x_1, x_2) = \frac{|x_1 x_2|}{\gcd(x_1, x_2)}.$$

Proof. The first formula of the theorem follows from the factorization formula, since the exponent of p_k in $\operatorname{lcm}(x_1, \operatorname{lcm}(x_2, \ldots, x_N))$ is $\max(j_{1k}, \max(j_{2k}, \ldots, j_{Nk}))$ and on the other hand

$$\max(j_{1k}, \max(j_{2k}, \ldots, j_{Nk})) = \max(j_{1k}, j_{2k}, \ldots, j_{Nk}) \quad (k = 1, 2, \ldots, M).$$

The second formula follows from the factorization formula as well, since the exponent of the prime factor p_k in $x_1 x_2$ is $j_{1k} + j_{2k}$ and on the other hand

$$\max(j_{1k}, j_{2k}) = j_{1k} + j_{2k} - \min(j_{1k}, j_{2k}).$$

□

NB. *We see from the factorization formula that the g.c.d. of more than two numbers is also the (positive) common divisor of these numbers that is divisible by every other common divisor and this property is often used as the definition. Correspondingly we can see that the l.c.m. is the (positive) common multiple of these numbers that divides every other common multiple of the numbers and this property is also often used as its definition. By these alternative definitions it is usually agreed that* $\gcd(0, 0, \ldots, 0) = 0$ *and* $\operatorname{lcm}(0, x_2, \ldots, x_N) = 0$.

2.4 Congruence Calculus or Modular Arithmetic

The idea of *congruence calculus* is that you compute only with the remainders of integers using a fixed divisor (or several of them), the so-called *modulus* $m \geq 1$. Congruence calculus is also often called *modular arithmetic.*

We say that integers x and y are *congruent modulo* m, denoted

$$x \equiv y \mod m \quad \text{(a so-called } \textit{congruence}\text{)},$$

if $x-y$ is divisible by m. This might be read as "x is congruent to y modulo m" or just "x equals y modulo m". Then again, if $x - y$ is indivisible by m, it's said that x and y are *incongruent modulo* m and this is denoted by $x \not\equiv y \mod m$. Note that $x \equiv 0 \mod m$ exactly when x is divisible by m, and that every number is congruent to every other number modulo 1.

The congruence $x \equiv y \mod m$ says that when dividing x and y by m the remainder is the same, or in other words, x and y belong to the same *residue class* modulo m. Every integer always belongs to one residue class modulo m and only in one. There are exactly m residue classes modulo m, as there are m different remainders.

Obviously x is always congruent to itself modulo m and if $x \equiv y \mod m$, then also $y \equiv x \mod m$ and $-x \equiv -y \mod m$. Furthermore, if $x \equiv y \mod m$ and $y \equiv z \mod m$ then also $x \equiv z \mod m$, in this case we may write

$$x \equiv y \equiv z \mod m.$$

(Congruence of integers is thus an example of an equivalence relation.) For basic computing of congruences we have the rules

Theorem 2.10. (i) *If* $x \equiv y \mod m$ *and* $u \equiv v \mod m$ *then* $x + u \equiv y + v \mod m$.

(ii) *If* c *is an integer and* $x \equiv y \mod m$ *then* $cx \equiv cy \mod m$.

(iii) *If* $x \equiv y \mod m$ *and* $u \equiv v \mod m$ *then* $xu \equiv yv \mod m$.

(iv) *If* $x \equiv y \mod m$ *and* n *is a positive integer then* $x^n \equiv y^n \mod m$.

Proof. (i) If $x - y = km$ and $u - v = lm$ then $(x + u) - (y + v) = (k + l)m$.
(ii) If $x - y = km$ then $cx - cy = ckm$.
(iii) This follows from (ii), since $xu \equiv yu \equiv yv \mod m$.
(iv) This follows from (iii). □

You can compute with congruences pretty much in the same way as with normal equations, except that division and reduction are not generally allowed (we get back to this soon).

If you think about remainders, in calculations you can use any integer that has the same remainder when divided by the modulus, results will still be the same, in other words, the result is independent of the choice of the representative of the residue class. For simplicity certain sets of representatives, so-called *residue systems*, are however often used:

- *positive residue system* $0, 1, \ldots, m - 1$ (that is, the usual remainders);
- *symmetric residue system* $-(m - 1)/2, \ldots, 0, 1, \ldots, (m - 1)/2$ for odd m;
- *symmetric residue system* $-(m - 2)/2, \ldots, 0, 1, \ldots, m/2$ for even m;
- *negative residue system* $-(m - 1), \ldots, -1, 0$.

The positive residue system is the usual choice. In general, any set of m integers, which are not congruent modulo m, form a residue system modulo m. From now on the residue of a number x modulo m in the positive residue system—in other words, the remainder of x when divided by the module m—is denoted by $(x,\mathrm{mod}\ m)$.

Division (or reduction) of each side of a congruence is not generally allowed and can only be done under the following circumstances.

Theorem 2.11. *$xu \equiv yu \mod m$ is the same as $x \equiv y \mod m/\gcd(u,m)$, so you can divide an integer out of a congruence if you divide the modulus by the g.c.d. of the modulus and the integer that's being divided out. (Note that if m is a factor of u then $m/\gcd(u,m) = 1$.)*

Proof. We first start from the assumption $xu \equiv yu \mod m$ or $(x-y)u = km$. Then we denote $d = \gcd(u,m)$, $u = du'$ and $m = dm'$. We have that $\gcd(u',m') = 1$ and $m' = m/\gcd(u,m)$ and further that $(x-y)u' = km'$. By Bézout's theorem $1 = c_1u' + c_2m'$, from which it follows that

$$x - y = c_1u'(x-y) + c_2m'(x-y) = (c_1k + c_2(x-y))m',$$

or in other words that $x \equiv y \mod m/\gcd(u,m)$, as claimed.

Next we start from the assumption that $x \equiv y \mod m/d$ or that $x - y = km/d$. From this it follows that $(x-y)d = km$ and furthermore $(x-y)u = u'km$. So $xu \equiv yu \mod m$. □

In particular, you can divide an integer that has no common factors with the modulus out of the congruence without dividing the modulus.

Corollary. *If $\gcd(x,m) = 1$ then the numbers $y + kx$ ($k = 0, 1, \ldots, m-1$) form a residue system modulo m, no matter what integer y is.*

Proof. Now we have m numbers. If $y + ix \equiv y + jx \mod m$, where $0 \le i, j \le m-1$, then $ix \equiv jx \mod m$ and by Theorem 2.11 we know that $i \equiv j \mod m$. So $i - j = km$, but because $0 \le i, j \le m-1$ this is possible only when $k = 0$, i.e. when $i = j$. So different numbers are not congruent. □

Using the same kind of technique we see immediately that if $\gcd(x,m) = 1$, then x has an *inverse modulo m,* in other words, there exists an integer y such that

$$xy \equiv 1 \mod m.$$

In this case we also write $x^{-1} \equiv y \mod m$ or $1/x \equiv y \mod m$.[4] This kind of inverse is obtained using the Euclidean algorithm, since by Bézout's theorem $1 = c_1x + c_2m$ and so $x^{-1} \equiv c_1 \mod m$. On the other hand, if $\gcd(x,m) \neq 1$ then x can't have an inverse modulo m, as we can easily see. Note that if $x^{-1} \equiv y \mod m$ then $y^{-1} \equiv x \mod m$ or $(x^{-1})^{-1} \equiv x \mod m$. Inverses modulo m (when they exist) satisfy the usual rules of calculus of powers. For example,

$$(xy)^{-1} \equiv x^{-1}y^{-1} \mod m \quad \text{and} \quad x^{-n} \equiv (x^{-1})^n \equiv (x^n)^{-1} \mod m \quad (n = 1, 2 \ldots).$$

Those numbers x of a residue system for which $\gcd(x,m) = 1$ form the so-called *reduced residue system.* The respective residue classes are called *reduced residue classes* modulo m. We can easily see that if $x \equiv y \mod m$ then $\gcd(x,m) = \gcd(y,m)$. This means there is exactly the same amount of numbers in two reduced residue systems modulo m (they are the numbers coprime to m) and that the numbers of two reduced residue systems can be paired

[4] This inverse must not be confused with the rational number $1/x$.

off by their being congruent modulo m. That is, there is a bijection between any two reduced residue systems modulo m. The amount of numbers in a reduced residue system modulo m is called *Euler's (totient) function,* denoted $\phi(m)$. It's needed for example in RSA cryptosystem. The most common reduced residue system is the one that is formed out of the positive residue system. Also note that if p is a prime then $1, 2, \ldots, p-1$ form a reduced residue system modulo p and $\phi(p) = p - 1$.

2.5 Residue Class Rings and Prime Fields

Integers are divided into m residue classes, according to which number $0, \ldots, m-1$ they are congruent to modulo m. The class that the integer x belongs to is denoted by $\overline{x}$. Note that $\overline{x} = \overline{x + km}$, no matter what integer k is. We can define basic arithmetic operations on residue classes using their "representatives" as follows:

$$\overline{x} \pm \overline{y} = \overline{x \pm y} \quad , \quad \overline{x} \cdot \overline{y} = \overline{x \cdot y} \quad \text{and} \quad \overline{x}^n = \overline{x^n} \quad (n = 0, 1, \ldots).$$

The result of the operation is independent of the choice of the representatives, which is easy to confirm. The operation is thus well-defined. The basic properties of computing with integers will transfer to residue classes:

(1) $+$ and $\cdot$ are associative and commutative.

(2) Distributivity holds.

(3) Every class a has an *opposite class* $-a$, i.e. a class $-a$ such that $a + (-a) = \overline{0}$. If $a = \overline{x}$, then obviously $-a = \overline{-x}$.

(4) $\overline{0}$ and $\overline{1}$ "behave" as they should, i.e. $a + \overline{0} = a$ and $a \cdot \overline{1} = a$. Also $\overline{0} \neq \overline{1}$, if $m > 1$.

In the algebraic sense residue classes modulo m form a so-called *ring,* see Chapter 4 and the course Algebra 1. This *residue class ring modulo* m is denoted by $\mathbb{Z}_m$. $\mathbb{Z}_1$ is singularly uninteresting—and some do not think of it as a ring at all.

If $\gcd(x, m) = 1$ then the residue class $\overline{x}$ has an *inverse class* $\overline{x}^{-1}$ for which $\overline{x} \cdot \overline{x}^{-1} = \overline{1}$. Naturally, if $x^{-1} \equiv y \mod m$ then $\overline{x}^{-1} = \overline{y}$. If $\gcd(x, m) \neq 1$ then there does not exist such an inverse class. We have that every residue class other than $\overline{0}$ has an inverse class exactly when the modulus m is a prime. In this case the residue class ring is also called a *prime field.* So in prime fields division, meaning multiplication by the inverse class, is available. The smallest and most common prime field is the *binary field* $\mathbb{Z}_2$, whose members are the elements $\overline{0}$ and $\overline{1}$ (called *bits*, and mostly written without the overlining as 0 and 1).

Arithmetical operations in residue class rings can be transferred in a natural way to arithmetical operations of matrices and vectors formed of residue classes. This way we get to use the familiar addition, subtraction, multiplication, powers and transposition of matrices. Determinants of square matrices also satisfy the basic calculation rules. Just as in basic courses, we note that a square matrix has an inverse matrix, if its determinant (which is a residue class in $\mathbb{Z}_m$) has an inverse class. Note that it is not enough for the determinant to be $\neq \overline{0}$, because when using Cramer's rule to form the inverse matrix we need division modulo m by the determinant. In prime fields it is of course enough for the determinant to be $\neq \overline{0}$.

2.6 Basic Arithmetic Operations for Large Integers

Operation of modern cryptosystems is based on arithmetical computations of large integers. They must be executable quickly and efficiently. Efficiencies of algorithms are often compared using numbers of basic steps needed to execute the algorithm versus the maximum length N of the input numbers. A basic step could be for example addition, subtraction or multiplication of the decimals $0, 1, \ldots, 9$. The most common of these comparison notations is the so-called *O-notation.* In this case $O(f(N))$ denotes collectively any function $g(N)$ such that starting from some lower limit $N \geq N_0$ we have $|g(N)| \leq Cf(N)$ where C is a constant. Actual computational complexity is discussed in Section 6.1.

The customary functions $\lfloor x \rfloor$ (floor of x, i.e. the largest integer which is $\leq x$) and $\lceil x \rceil$ (ceiling of x, i.e. smallest integer which is $\geq x$) are used for rounding when needed.

Addition and subtraction

The common methods of addition and subtraction by hand that we learn in school can be programmed more or less as they are. Addition and subtraction of numbers of length N and M requires $O(\max(N, M))$ steps, which is easy to confirm.

Multiplication

The usual method of integer multiplication by hand is also suitable for a computer, but it is not nearly the fastest method. In this method multiplication of numbers of lenght N and M requires $O(NM)$ steps, which can be lot.

Karatsuba's algorithm is faster than the traditional algorithm. The algorithm is a kind of "divide and conquer" procedure. For multiplication of positive numbers n and m in decimal representation we first write them in the form

$$n = a10^k + b \quad \text{and} \quad m = c10^k + d$$

where $a, b, c, d < 10^k$ and the maximum length of the numbers is $2k$ or $2k - 1$. One of the numbers a and c can be zero, but not both of them. In other words, at least one of these numbers is written in base-10^k representation. Then

$$nm = (a10^k + b)(c10^k + d) = y10^{2k} + (x - y - z)10^k + z,$$

where

$$x = (a + b)(c + d)\ , \quad y = ac \quad \text{and} \quad z = bd,$$

so we need just three individual "long" multiplications of integers (and not four as you may originally think). When these three multiplications

$$(a + b)(c + d)\ , \quad ac \quad \text{and} \quad bd$$

are performed in the same way by dividing each of them into three shorter multiplications and so on, whereby we eventually end up using a simple multiplication table, we get Karatsuba's algorithm (where we denote $\mathrm{PROD}(n, m) = nm$):

Karatsuba's multiplication algorithm:

1. If $n = 0$ or $m = 0$, we return 0 and quit.

2. We reduce the case to one in which both the multiplier and the multiplicand are positive:

(2.1) If $n < 0$ and $m > 0$, or $n > 0$ and $m < 0$, we compute $t = \mathrm{PROD}(|n|, |m|)$, return $-t$ and quit.

(2.2) If $n < 0$ and $m < 0$, we compute $t = \mathrm{PROD}(-n, -m)$, return t and quit.

3. If $n, m < 10$, we look up $\mathrm{PROD}(n, m)$ in the multiplication table, and quit.

4. If $n \geq 10$ or $m \geq 10$, we write n and m in the form $n = a10^k + b$ and $m = c10^k + d$ where $a, b, c, d < 10^k$, as above. In decimal representation this is easy.

5. We compute $\mathrm{PROD}(a + b, c + d)$, $\mathrm{PROD}(a, c)$ and $\mathrm{PROD}(b, d)$, return (the easily obtained)

$$\begin{aligned}\mathrm{PROD}(n, m) = 10^{2k}\mathrm{PROD}(a, c) + 10^k(\mathrm{PROD}(a + b, c + d)\\ - \mathrm{PROD}(a, c) - \mathrm{PROD}(b, d)) + \mathrm{PROD}(b, d)\end{aligned}$$

and quit.

The procedure ends since the maximum length of the numbers being multiplied is reduced to about half in every iteration.

If we multiply two numbers of length N and denote by $K(N)$ an approximate upper bound on the number of basic arithmetical operations on the numbers $0, 1, \ldots, 9$ needed, then it is apparent that $K(N)$ is obtained using a recursion formula

$$K(N) = \begin{cases}\alpha N + 3K(N/2) \text{ if } N \text{ is even}\\ \alpha N + 3K((N + 1)/2) \text{ if } N \text{ is odd}\end{cases}, \quad K(1) = 1,$$

where the coefficient α is obtained from the number of required additions and subtractions, depending on the algorithm used. A certain approximate bound for the number of required basic operations is given by

Theorem 2.12. *If $N = 2^l$ then $K(N) = (2\alpha + 1)3^l - \alpha 2^{l+1} = (2\alpha + 1)N^{\log_2 3} - 2\alpha N$.*

Proof. The value is correct, when $N = 1$. If the value is correct when $N = 2^l$ then it is also correct when $N = 2^{l+1}$, since

$$K(2^{l+1}) = \alpha 2^{l+1} + 3K(2^l) = \alpha 2^{l+1} + 3(2\alpha + 1)3^l - 3\alpha 2^{l+1} = (2\alpha + 1)3^{l+1} - \alpha 2^{l+2}. \quad \square$$

Naturally the number of basic operations for very large N obtained by the theorem, that is

$$(2\alpha + 1)N^{\log_2 3} - 2\alpha N = O(N^{\log_2 3}) = O(N^{1.585}),$$

is substantially smaller than $O(N^2)$. For example, if $N = 2^{12} = 4\,096$ then $N^2/N^{\log_2 3} \cong 32$. There are even faster variants of Karatsuba's procedure where numbers are divided into more than two parts, see for example MIGNOTTE.

The fastest multiplication algorithms use the so-called fast Fourier transformation (FFT), see for example LIPSON or CRANDALL & POMERANCE. In this case the number of basic operations is $O(N \ln N \ln(\ln N))$. See also the course Fourier Methods.

Division

Common "long division" that is taught in schools can be transferred to a computer, although the guessing phase in it is somewhat hard to execute efficiently if the base number is large, see KNUTH. The number of basic operations is $O(N^2)$ where N is the length of the dividend. Also a division algorithm similar to Karatsuba's algorithm is possible and quite fast.[5]

Division based on Newton's method, familiar from basic courses, is very efficient. First we assume that both the divisor m and the dividend n are positive, and denote the length of the dividend by N and the length of the divisor by M. Since the cases $N < M$ and $N = M$ are easy, we assume that $N > M$. We denote the result of the division $n = qm + r$ (quotient q and remainder r) by $\mathrm{DIV}(n, m) = (q, r)$. Note that then $q = \lfloor n/m \rfloor$.

We start by finding the inverse of the divisor. To find the root of the function $f(x) = m - 1/x$, i.e. $1/m$, we use the Newton iteration

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = 2x_i - mx_i^2.$$

However, since we can only use multiplication of integers, we compute $l = 10^N/m$, i.e. the root of the function $g(x) = m - 10^N/x$, for which we correspondingly get the exact Newton iteration

$$x_{i+1} = 2x_i - \frac{mx_i^2}{10^N} = 2x_i - \frac{x_i^2}{l}.$$

To be able to stay purely among integers, we use a version of this iteration that is rounded to integers:

$$y_{i+1} = 2y_i - \left\lfloor \frac{m}{10^M} \left\lfloor \frac{y_i^2}{10^{N-M}} \right\rfloor \right\rfloor.$$

Divisions by powers of 10 are trivial in the decimal system. The purpose of using this is to calculate $\lfloor l \rfloor$, by taking the floor $\lfloor n10^{-N} \lfloor l \rfloor \rfloor$ we then obtain the quotient by some trial and error, and finally get the remainder using the quotient.

The following properties are easy to confirm:

- $2y - \lfloor m10^{-M} \lfloor y^2 10^{M-N} \rfloor \rfloor \geq 2y - y^2/l$, in other words, rounding to integers does not reduce values of iterants.
- If $x \neq l$ then $2x - x^2/l < l$. So the exact iteration approaches l from below. Because $m/10^M < 1$, for the rounded iteration we correspondingly get

$$\begin{aligned} 2y - \left\lfloor \frac{m}{10^M} \left\lfloor \frac{y^2}{10^{N-M}} \right\rfloor \right\rfloor &\leq 2y - \left\lfloor \frac{m}{10^M} \left(\frac{y^2}{10^{N-M}} - 1 \right) \right\rfloor \leq 2y - \left\lfloor \frac{1}{l} y^2 - 1 \right\rfloor \\ &< 2y - \left(\frac{1}{l} y^2 - 2 \right) \leq l + 2. \end{aligned}$$

- If $x < l$ then $2x - x^2/l > x$. So the exact iteration is strictly growing as long as iterants are $< l$. The same applies for the rounded iteration also.

[5] Such an algorithm is described for example in the book MIGNOTTE and in the old Finnish lecture notes RUOHONEN, K.: *Kryptologia*, and is very well analyzed in the report BURNIKEL, C. & ZIEGLER, J.: Fast Recursive Division. *Max Planck Institut für Informatik. Forschungsbericht* **MPI-I-98-1-022** (1998).

We denote

$$l = y_i + \epsilon_i$$

where ϵ_i is the error. Newton's methods are *quadratic,* i.e. they double the amount of the correct numbers in every step, and so it is here too: If $y_i < l$ then

$$|\epsilon_i| = l - y_i \leq l - 2y_{i-1} + \frac{1}{l}y_{i-1}^2 = \frac{1}{l}\epsilon_{i-1}^2.$$

By repeating this and noting that $l > 10^{N-M}$ we get (assuming again that $y_i < l$)

$$|\epsilon_i| \leq \frac{1}{l}\epsilon_{i-1}^2 \leq \frac{1}{l}\left(\frac{1}{l}\epsilon_{i-2}^2\right)^2 \leq \cdots \leq l^{-(1+2+2^2+\cdots+2^{i-1})}\epsilon_0^{2^i} = l^{1-2^i}\epsilon_0^{2^i} < 10^{(1-2^i)(N-M)}\epsilon_0^{2^i}.$$

Now it is required that $10^{(1-2^i)(N-M)}\epsilon_0^{2^i} \leq 1$. Assuming that $|\epsilon_0| < 10^{N-M}$ this is equivalent to

$$i \geq \left\lceil \log_2 \frac{N-M}{N-M-\log_{10}|\epsilon_0|} \right\rceil$$

(confirm!). We choose then

$$y_0 = 10^{N-M}\left\lfloor \frac{10^M}{m} \right\rfloor \quad \text{or} \quad y_0 = 10^{N-M}\left\lceil \frac{10^M}{m} \right\rceil,$$

depending on which is nearer the number $10^M/m$, the floor or the ceiling, whence $|\epsilon_0| \leq 10^{N-M}/2$. So it suffices to choose

$$I = \left\lceil \log_2 \frac{N-M}{\log_{10} 2} \right\rceil = \lceil \log_2(N-M) - \log_2(\log_{10} 2) \rceil$$

as the number of iterations.

Using the iteration rounded to integers produces a strictly growing sequence of integers, until we obtain a value that is in the interval $[l, l+2)$. Then we can stop and check whether it is the obtained value or some preceding value that is the correct $\lfloor l \rfloor$. The whole procedure is the following (the output is $\mathrm{DIV}(n, m)$):

Division using Newton's method:

1. If $n = 0$, we return $(0, 0)$ and quit.
2. If $m = 1$, we return $(n, 0)$ and quit.
3. If $m < 0$, we compute $\mathrm{DIV}(n, -m) = (q, r)$, return $(-q, r)$ and quit.
4. If $n < 0$, we compute $\mathrm{DIV}(-n, m) = (q, r)$, return $(-q-1, m-r)$, if $r > 0$, or $(-q, 0)$, if $r = 0$, and quit.
5. Set $N \leftarrow$ length of dividend n and $M \leftarrow$ length of divisor m.
6. If $N < M$, we return $(0, n)$ and quit.
7. If $N = M$, we compute the quotient q. This is easy, since now $0 \leq q \leq 9$. (By trying out, if not in some other way.) We return $(q, n - mq)$ and quit.

8. If $N > M$, we compute $\lfloor 10^M/m \rfloor$. Again this is easy, since $1 \leq \lfloor 10^M/m \rfloor \leq 10$. (By trying out or in some other way.)

9. If $10^M/m - \lfloor 10^M/m \rfloor \leq 1/2$, that is, $2 \cdot 10^M - 2m\lfloor 10^M/m \rfloor \leq m$, we set $y_0 \leftarrow 10^{N-M}\lfloor 10^M/m \rfloor$. Otherwise we set $y_0 \leftarrow 10^{N-M}(\lfloor 10^M/m \rfloor + 1)$. Note that in the latter case $y_0 > l$ and at least one iteration must be performed.

10. We iterate the recursion formula

$$y_{i+1} = 2y_i - \left\lfloor \frac{m}{10^M} \left\lfloor \frac{y_i^2}{10^{N-M}} \right\rfloor \right\rfloor$$

starting from the value y_0 until $i \geq 1$ and $y_{i+1} \leq y_i$.

11. We check by multiplications which one of the numbers $y_i, y_i - 1, \ldots$ is the correct $\lfloor l \rfloor$ and set $k \leftarrow \lfloor l \rfloor$.

12. We set $t \leftarrow \lfloor nk/10^N \rfloor$ (essentially just a multiplication) and check by multiplications again which number t or $t+1$ is the correct quotient q in the division $\mathrm{DIV}(n, m) = (q, r)$. We then return $(q, n - mq)$ and quit.

The procedure in #12 produces the correct quotient because first of all $r < m$ and

$$q = \frac{n-r}{m} \leq \frac{n}{m} < \frac{10^N}{m}.$$

Further, if $\mathrm{DIV}(10^N, m) = (k, r')$ then $r' < m$ and

$$\frac{nk}{10^N} = \frac{(qm + r)(10^N - r')}{m10^N} = q - \frac{qr'}{10^N} + \frac{r(10^N - r')}{m10^N}.$$

The middle term on the right hand side is in the interval $(-1, 0]$ and the last term is in the interval $[0, 1)$. So q is either t or $t + 1$.

Because the maximum number I of iterations is very small—about the logarithm of the difference of the length N of the dividend and the length M of the divisor—and in an iteration step there always are three multiplications and one subtraction of integers of maximum length $2M$ (some of which remain constant), division is not essentially more laborious than multiplication. Trying out numbers in #7 and #8 does not take that many steps either.

NB. *There are many different variants of this kind of division.* CRANDALL & POMERANCE *handles the topic with a wider scope and gives more references.*

Powers

Raising the number a to the n^{th} power a^n takes too much time if you just repeatedly multiply by a, since you need then $|n| - 1$ multiplications, while it in fact suffices to use at most $2\lfloor \log_2 |n| \rfloor$ multiplications:

Method of Russian peasants:

1. If $n = 0$ then we return the power 1 and quit.

2. If $n < 0$, we set $a \leftarrow a^{-1}$ and $n \leftarrow -n$.

3. If $n \geq 1$, we compute the binary representation $b_j b_{j-1} \cdots b_0$ of n where $j = \lfloor \log_2 n \rfloor$ (the length of n as binary number minus one, see Theorem 2.4).
4. Set $i \leftarrow 0$ and $x \leftarrow 1$ and $y \leftarrow a$.
5. If $i = j$ then we return the power xy and quit.
6. If $i < j$ and
   6.1 $b_i = 0$ then we set $y \leftarrow y^2$ and $i \leftarrow i + 1$ and go to #5.
   6.2 $b_i = 1$ then we set $x \leftarrow xy$ and $y \leftarrow y^2$ and $i \leftarrow i + 1$ and go to #5.

Correctness of the algorithm is a straightforward consequence of binary representation:

$$|n| = b_j 2^j + b_{j-1} 2^{j-1} + \cdots + b_1 2 + b_0$$

and

$$a^{|n|} = a^{b_j 2^j} a^{b_{j-1} 2^{j-1}} \cdots a^{b_1 2} a^{b_0}.$$

It's convenient to compute bits of the binary representation of n one by one when they are needed, and not all at once. Now, if $i = 0$, only one multiplication is needed in #6 since then $x = 1$. Similarly, when $i = j$, only one multiplication is needed in #5. For other values of i two multiplications may be needed, so the maximum overall number of multiplications is $1 + 1 + 2(j - 1) = 2j$, as claimed.

Actually this procedure works for every kind of power and also when multiplication is not commutative, for example for powers of polynomials and matrices. When calculating powers modulo m products must be reduced to the (positive) residue system modulo m, so that the numbers needed in calculations won't get too large. This way you can quickly compute very high modular powers.

The procedure takes its name from the fact that Russian peasants used this method for multiplication when calculating with an abacus and you can think of $a \cdot n$ as the n^{th} power of a with respect to addition. Apparently the algorithm is very old.

Integral root

The *integral l^{th} root*[6] of a nonnegative integer n is $\lfloor n^{1/l} \rfloor$. The most common of these roots is of course the *integral square root* ($l = 2$). Denote the length of n in binary representation by N.

We can use the same kind of Newton method for computing an integral root as we used for division.[7] For calculating the root of the function $x^l - n$, i.e. $n^{1/l}$, we get the Newton iteration

$$x_{i+1} = \frac{l-1}{l} x_i + \frac{n}{l x_i^{l-1}}.$$

However, because we want to compute using integers, we take an iteration rounded to integers:

$$y_{i+1} = \left\lfloor \frac{1}{l} \left((l-1) y_i + \left\lfloor \frac{n}{y_i^{l-1}} \right\rfloor \right) \right\rfloor,$$

and use addition, multiplication and division of integers.

The following properties are easy to confirm (e.g. by finding extremal values):

[6] In some texts it's $\lceil n^{1/l} \rceil$, and in some texts $n^{1/l}$ rounded to the nearest integer.

[7] It may be noted that the procedure that used to be taught in schools for calculating square roots by hand is also similar to long division.

- $\left\lfloor \frac{1}{l}\left((l-1)y + \left\lfloor \frac{n}{y^{l-1}} \right\rfloor\right)\right\rfloor \le \frac{l-1}{l}y + \frac{n}{ly^{l-1}}$, so rounding to integers does not increase iterant values.

- If $x \neq n^{1/l}$ and $x > 0$ then
$$\frac{l-1}{l}x + \frac{n}{lx^{l-1}} > n^{1/l}.$$
So the exact iteration approaches the root from "above". For the rounded version we get correspondingly
$$\begin{aligned}\left\lfloor \frac{1}{l}\left((l-1)y + \left\lfloor \frac{n}{y^{l-1}} \right\rfloor\right)\right\rfloor &\ge \left\lfloor \frac{1}{l}\left((l-1)y + \frac{n}{y^{l-1}} - 1\right)\right\rfloor \\ &> \frac{l-1}{l}y + \frac{n}{ly^{l-1}} - 2 \ge n^{1/l} - 2.\end{aligned}$$

- If $x > n^{1/l}$ then
$$\frac{l-1}{l}x + \frac{n}{lx^{l-1}} < x.$$
The exact iteration is strictly decreasing. The same is true for the rounded version.

Denote
$$n^{1/l} = y_i - \epsilon_i$$
and choose $y_0 = 2^{\lceil N/l \rceil}$ as the starting value. This can be quickly computed using the algorithm of Russian peasants. Since $n < 2^N$ then $y_0 > n^{1/l}$. First we estimate the obtained ϵ_0 as follows:
$$\epsilon_0 = y_0 - n^{1/l} = 2^{\lceil N/l \rceil} - n^{1/l} \le 2^{N/l+1-1/l} - n^{1/l} = 2 \cdot 2^{(N-1)/l} - n^{1/l} \le n^{1/l}.$$

This Newton's method is also quadratic. We only confirm the case $l = 2$. (The general case is more complicated but similar.) If $y_{i-1}, y_i > n^{1/l}$ then
$$\begin{aligned}0 < \epsilon_i = y_i - n^{1/l} &\le \frac{l-1}{l}y_{i-1} + \frac{n}{ly_{i-1}^{l-1}} - n^{1/l} = \frac{1}{ly_{i-1}}\left(y_{i-1}^2 + \frac{n}{y_{i-1}^{l-2}} - ln^{1/l}y_{i-1}\right) \\ &< \frac{1}{ln^{1/l}}\left(y_{i-1}^2 + \frac{n}{n^{(l-2)/l}} - 2n^{1/l}y_{i-1}\right) = \frac{1}{ln^{1/l}}(y_{i-1} - n^{1/l})^2 = \frac{1}{ln^{1/l}}\epsilon_{i-1}^2.\end{aligned}$$

Repeating this estimation we get (denoting $a = \frac{1}{ln^{1/l}}$ for brevity)
$$\begin{aligned}\epsilon_i &< a\epsilon_{i-1}^2 < a \cdot a^2\epsilon_{i-2}^{2^2} < \cdots < a^{1+2+2^2+\cdots+2^{i-1}}\epsilon_0^{2^i} \\ &= a^{2^i-1}\epsilon_0^{2^i} = a^{-1}(a\epsilon_0)^{2^i} = ln^{1/l}\left(\frac{\epsilon_0}{ln^{1/l}}\right)^{2^i} \le n^{1/l}l^{1-2^i}.\end{aligned}$$

If we now want to have $\epsilon_i < 1$ then it's sufficient to take $n^{1/l}l^{1-2^i} \le 1$, so (confirm!) a maximum of
$$I = \left\lceil \log_2\left(1 + \frac{\log_2 n}{l \log_2 l}\right)\right\rceil$$
iterations is needed. Hence the sufficient number of iterations is proportional to $\log_2 N$, which is very little. So, calculation of an integral root is about as demanding as division.

NB. *Because $n^{1/\log_2 n} = 2$, we are only interested in values of l which are at most as large as the length of n, others can be dealt with with little effort.*

Iteration rounded to integers produces a strictly decreasing sequence of integers, until we hit a value in the interval $(n^{1/l} - 2, n^{1/l}]$.

Newton's method for computing integral l^{th} root

1. If $n = 0$ or $n = 1$ then we return n and quit.
2. Set $y_0 \leftarrow 2^{\lceil N/l \rceil}$ where N is the length of n in binary representation.
3. Repeat the iteration
$$y_{i+1} = \left\lfloor \frac{1}{l} \left((l-1)y_i + \left\lfloor \frac{n}{y_i^{l-1}} \right\rfloor \right) \right\rfloor$$
starting from y_0 until $y_{i+1} \geq y_i$.
4. Check which one of the numbers $y_i, y_i + 1, \ldots$ is the correct integral root $\lfloor n^{1/l} \rfloor$, and quit.

Generating a random integer

Random bit sequences are commonly generated using a *shift register* [8] of p^{th} order modulo 2:

$$r_i \equiv a_1 r_{i-1} + a_2 r_{i-2} + \cdots + a_p r_{i-p} \mod 2$$

where $a_1, a_2, \ldots, a_p$ are constant bits (0 or 1, $a_p = 1$). First we need the initial "seed" bits $r_0, r_1, \ldots, r_{p-1}$. Here we calculate using the positive residue system modulo 2 in other words, using bits. Of course the obtained sequence $r_p, r_{p+1}, \ldots$ is not random in any way, indeed, it is obtained using a fully deterministic procedure and is periodic (length of period is at most 2^p). When we choose the coefficients $a_1, a_2, \ldots, a_{p-1}$ conveniently, we get the sequence to behave "randomly" in many senses, the period is long and so on, see for example KNUTH. In the simplest cases almost every coefficient is zero.

Shift registers of the type

$$r_i \equiv r_{i-q} + r_{i-p} \mod 2,$$

where p is a prime and q is chosen conveniently, often produce very good random bits. Some choices, where the number q can be replaced by the number $p - q$, are listed in the table below.

p	q ($p - q$ works also)	p	q ($p - q$ works also)
2	1	1279	216, 418
3	1	2281	715, 915, 1029
5	2	3217	67, 576
7	1, 3	4423	271, 369, 370, 649, 1393, 1419, 2098
17	3, 5, 6	9689	84, 471, 1836, 2444, 4187
31	3, 6, 7, 13	19937	881, 7083, 9842
89	38	23209	1530, 6619, 9739
127	1, 7, 15, 30, 63	44497	8575, 21034
521	32, 48, 158, 168	110503	25230, 53719
607	105, 147, 273	132049	7000, 33912, 41469, 52549, 54454

These values were found via a computer search.[9] Small values of p of course are not very useful.

[8] A classic reference is GOLOMB, S.W.: *Shift Register Sequences.* Aegean Park Press (1982)

[9] The original articles are ZIERLER, N.: On Primitive Trinomials Whose Degree is a Mersenne Exponent. *Information and Control* **15** (1969), 67–69 and HERINGA, J.R. & BLÖTE, H.W.J. & COMPAGNER, A.: New Primitive Trinomials of Mersenne-Exponent Degrees for Random Number Generation. *International Journal of Modern Physics* **C3** (1992), 561–564.

In matrix form in the binary field $\mathbb{Z}_2$, see the previous section, the shift register is the following. Denote

$$\mathbf{r}_i = \begin{pmatrix} r_{i+p-1} \\ r_{i+p-2} \\ \vdots \\ r_i \end{pmatrix} \quad \text{and} \quad \mathbf{A} = \begin{pmatrix} a_1 & a_2 & \cdots & a_{p-1} & a_p \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix}.$$

$\mathbf{A}$ is the so-called *companion matrix* of the shift register. Then

$$\mathbf{r}_{i+1} \equiv \mathbf{A}\mathbf{r}_i \mod 2$$

and hence

$$\mathbf{r}_i \equiv \mathbf{A}^i \mathbf{r}_0 \mod 2 \quad (i = 0, 1, \ldots).$$

The matrix power $\mathbf{A}^i$ can be quickly computed modulo 2 using the method of Russian peasants. So, perhaps a bit surprisingly, we can quite quickly compute terms of the sequence $r_p, r_{p+1}, \ldots$ "ahead of time" without computing that many intermediate terms. Note that for the bitstream to be "random", the matrix $\mathbf{R}_i = (\mathbf{r}_i, \ldots, \mathbf{r}_{i+p-1})$ obtained from p consecutive vectors $\mathbf{r}_i$ should be invertible, i.e. $\det(\mathbf{R}_i) \not\equiv 0 \mod 2$, at some stage. Then you can solve the equation $\mathbf{A}\mathbf{R}_i \equiv \mathbf{R}_{i+1} \mod 2$ for the matrix $\mathbf{A}$. For large values of p all these calculations naturally tend to become difficult.

Random integers are obtained from random bit sequences using binary representation. Random integers $s_0, s_1, \ldots$ of maximum binary length n are obtained by dividing the sequence into consecutive blocks of n bits and interpreting the blocks as binary numbers.

NB. *Generating random bits and numbers needed in encryption is quite demanding. "Badly" generated random bits assist in breaking the encryption a lot. One may say with good reason that generation of random numbers has lately progressed significantly largely due to the needs of encryption.*

The shift register generator above is quite sufficient for "usual" purposes, even for light encrypting, especially for larger values of p. For a shift register generator to be cryptologically strong, it should not be too predictable and for this p must be large, too large for practice. There are better methods, for example the so called Blum–Blum–Shub generator, which we discuss in Section 7.7. See also GOLDREICH.

Another common random number generator is the so-called *linear congruence generator.* It generates a sequence $x_0, x_1, \ldots$ of random integers in the interval $0, 1, \ldots, m$ using the recursion congruence

$$x_{i+1} \equiv ax_i + b \mod m$$

where a and b are given numbers—also the "seed" input x_0 is given. By choosing the numbers a and b conveniently we get good and fast random number generators which are suitable for many purposes. (See for example KNUTH.) The `rand`-operation in Maple used to be based on a linear congruence generator where $m = 999\,999\,999\,989$ (a prime), $a = 427\,419\,669\,081$ and $b = 0$.

Since

$$\begin{pmatrix} x_i \\ 1 \end{pmatrix} \equiv \begin{pmatrix} a & b \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_{i-1} \\ 1 \end{pmatrix} \equiv \cdots \equiv \begin{pmatrix} a & b \\ 0 & 1 \end{pmatrix}^i \begin{pmatrix} x_0 \\ 1 \end{pmatrix} \mod m,$$

the sequence $x_0, x_1, \ldots$ can also be calculated very quickly "in advance" using the method of Russian peasants, even for large numbers m and a. On the other hand, if $\gcd(x_i - x_{i-1}, m) = 1$, as it sooner or later will be, we can solve the congruence $x_{i+1} - x_i \equiv a(x_i - x_{i-1}) \mod m$ for a, and then get $b \equiv x_{i+1} - ax_i \mod m$. For pretty much the same reasons as for the shift register generator, the linear congruence generator is cryptologically very weak.

"Cryptographers on a diet of alphabet soup
would quickly recognise familiar ingredients...
Were there any words forming? Or familiar
combinations of letters? TH, for example, or
ER or ON or AN or RE?"

(MARKS, L.: *Between Silk and Cyanide*)

Chapter 3

SOME CLASSICAL CRYPTOSYSTEMS AND CRYPTANALYSES

3.1 AFFINE. CAESAR

To be able to use the results of number theory from the preceding chapter, symbols of plaintext must be encoded as numbers and residue classes. If there are M symbols to be encoded, we can use residue classes modulo M. In fact, we may think the message to be written using these residue classes or numbers of the positive residue system.

In the *affine cryptosystem* AFFINE a message symbol i (a residue class modulo m represented in the positive residue system) is encrypted in the following way:

$$e_{k_1}(i) = (ai + b, \bmod M).$$

Here a and b are integers and $\overline{a}$ has an inverse class $\overline{c}$ modulo M, in other words $\gcd(a, M) = 1$. The encrypting key k_1 is formed by the pair (a, b) and the decrypting key k_2 by the pair (c, b) (usually represented in the positive residue system). The decrypting function is

$$d_{k_2}(j) = (c(j - b), \bmod M).$$

So the length of the message block is one. Hence affine encrypting is also suitable for stream encryption. When choosing a and b from the positive residue system the number of possible values of a is $\phi(M)$, see Section 2.4, and all in all there are $\phi(M)M$ different encrypting keys. The number of encrypting keys is thus quite small. Some values:

$$\phi(10) = 4\ ,\ \phi(26) = 12\ ,\ \phi(29) = 28\ ,\ \phi(40) = 16.$$

The special case where $a = 1$ is known as the *Caesar cryptosystem* CAESAR. A more general cryptosystem, where

$$e_{k_1}(i) = (p(i), \bmod M)$$

and p is a polynomial with integral coefficients, isn't really much more useful as there are still very few keys (why?).

NB. *AFFINE resembles the linear congruence generator discussed before. The cryptosystem HILL, to be introduced next, resembles the shift register generator. This is not totally coincidental, random number generators and cryptosystems do have a connection: often you can obtain a strong random number generator from a strong cryptosystem, possibly a not too useful such, though.*

3.2 HILL. PERMUTATION. AFFINE-HILL. VIGENÈRE

In *Hill's*[1] *cryptosystem* HILL we use the same encoding of symbols as residue classes modulo M as in AFFINE. However, now the block is formed of d residue classes considered as a d-vector. Hill's original d was 2. The encrypting key is a $d \times d$ matrix $\mathbf{H}$ that has an inverse matrix modulo M, see Section 2.5. This inverse matrix $\mathbf{H}^{-1} = \mathbf{K}$ modulo M is the decrypting key.

A message block

$$\mathbf{i} = (i_1, \ldots, i_d)$$

is encrypted as

$$e_{\mathbf{H}}(\mathbf{i}) = (\mathbf{iH}, \bmod M),$$

and decrypted similarly as

$$e_{\mathbf{K}}(\mathbf{j}) = (\mathbf{jK}, \bmod M).$$

Here we calculate modulo M in the positive residue system.

There are as many encrypting keys as there are invertible $d \times d$ matrices modulo M. This number is quite hard to compute. However, usually there is a relatively large number of keys if d is large.

A special case of HILL is PERMUTATION or the so-called *permutation encryption.* Here $\mathbf{H}$ is a *permutation matrix,* in other words, a matrix that has exactly one element equal to one in every row and in every column all other elements being zeros. Note that in this case $\mathbf{H}^{-1} = \mathbf{H}^{\mathsf{T}}$, or that $\mathbf{H}$ is an orthogonal matrix. In permutation encrypting the symbols of the message block are permutated using the constant permutation given by $\mathbf{H}$.

A more general cryptosystem is AFFINE-HILL or the *affine Hill cryptosystem.* Comparing with HILL, now the encrypting key k_1 is a pair $(\mathbf{H}, \mathbf{b})$, where $\mathbf{b}$ is a fixed d-vector modulo M, and the decrypting key k_2 is the corresponding pair $(\mathbf{K}, \mathbf{b})$. In this case

$$e_{k_1}(\mathbf{i}) = (\mathbf{iH} + \mathbf{b}, \bmod M)$$

and

$$e_{k_2}(\mathbf{j}) = ((\mathbf{j} - \mathbf{b})\mathbf{K}, \bmod M).$$

From this we obtain a special case, the so-called *Vigenère*[2] *encryption* VIGENÈRE by choosing $\mathbf{H} = \mathbf{I}_d$ ($d \times d$ identity matrix). (This choice of $\mathbf{H}$ isn't suitable for HILL!) In Vigenère's encryption we add in the message block symbol by symbol a keyword of length d modulo M.

Other generalizations of HILL are the so-called *rotor cryptosystems,* that are realized using mechanical and electro-mechanical devices. The most familiar example is the famous ENIGMA machine used by Germans in the Second World War. See SALOMAA or BAUER.

3.3 ONE-TIME-PAD

Message symbols are often encoded binary numbers of a certain maximum length, for example ASCII encoding or UNICODE encoding. Hence we may assume that the message is a bit vector of length M. If the maximum length of the message is known in advance and encrypting is needed just once then we may choose a random bit vector $\mathbf{b}$ (or vector modulo 2) of length M as the key, the so-called *one-time-pad*, which we add to the message modulo 2 during the

[1] Lester S. Hill (1929)

[2] Blaise de Vigenère (1523–1596)

encryption. The encrypted message vector obtained as result is also random (why?) and a possible eavesdropper won't get anything out of it without the key. During the decrypting we correspondingly add the same vector $\mathbf{b}$ to the encrypted message, since $2\mathbf{b} \equiv \mathbf{0} \mod 2$. In this way we get the so-called *one-time-pad cryptosystem* ONE-TIME-PAD.

3.4 Cryptanalysis

The purpose of *cryptanalysis* is to break the cryptosystem, in other words, to find the decrypting key or encrypting key, or to at least produce a method which will let us get some information out of encrypted messages. In this case it is usually assumed that the cryptanalyzer is an eavesdropper or some other hostile party and that the cryptanalyzer knows which cryptosystem is being used but does not know the key being used.

A cryptanalyzer may have different information available:

(CO) just some, maybe random, cryptotext (*cryptotext only*),

(KP) some, maybe random, plaintext and the corresponding cryptotext (*known plaintext*),

(CP) a chosen plaintext and the corresponding cryptotext (*chosen plaintext*),

(CC) a chosen cryptotext and the corresponding plaintext (*chosen cryptotext*).

Classical attack methods are often based on *frequency analysis*, that is, knowledge of the fact that in long cryptotexts certain symbols, symbol pairs, symbol triplets and so on, occur at certain frequencies. Frequency tables have been prepared for the ordinary English language, American English and so on.

NB. *If a message is compressed before encrypting, it will lose some of its frequency information, see the course Information Theory.*

We now take as examples cryptanalyses of the cryptosystems discussed above.

AFFINE

In affine encryption the number of the possible keys is usually small, so they can all be checked one by one in a CO attack in order to find the probable plaintext. Apparently this won't work if there is no recognizable structure in the message. On the other hand, we can search for a structure in the cryptotext, in accordance with frequency tables, and in this way find KP data, for example the most common symbol might be recognized.

In a KP attack it is sufficient to find two message-symbol-cryptosymbol pairs (i_1, j_1) and (i_2, j_2) such that $\gcd(i_1 - i_2, M) = 1$. Such a pair is usually found in a long cryptotext. Then the matrix

$$\begin{pmatrix} i_1 & 1 \\ i_2 & 1 \end{pmatrix}$$

is invertible modulo M and the key is easily obtained:

$$\begin{pmatrix} j_1 \\ j_2 \end{pmatrix} \equiv \begin{pmatrix} i_1 & 1 \\ i_2 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} \mod M$$

or

$$\begin{pmatrix} a \\ b \end{pmatrix} \equiv (i_1 - i_2)^{-1} \begin{pmatrix} 1 & -1 \\ -i_2 & i_1 \end{pmatrix} \begin{pmatrix} j_1 \\ j_2 \end{pmatrix} \mod M.$$

In a CP attack the symbol pairs (i_1, j_1) and (i_2, j_2) can actually be chosen. In a CC attack it is sufficient to choose a long cryptotext. Because it is quite easy to break, AFFINE is only suitable for a light covering of information from casual readers.

HILL and AFFINE-HILL

The number of keys in Hill's cryptosystem is usually large, especially if d is large. A CO attack does not work well as such. By applying frequency analysis some KP data can in principle be found, especially if d is relatively small. In a KP attack it is sufficient to find message-block-cryptoblock pairs $(\mathbf{i}_1, \mathbf{j}_1), \ldots, (\mathbf{i}_d, \mathbf{j}_d)$ such that the matrices

$$\mathbf{S} = \begin{pmatrix} \mathbf{i}_1 \\ \vdots \\ \mathbf{i}_d \end{pmatrix} \quad \text{and} \quad \mathbf{R} = \begin{pmatrix} \mathbf{j}_1 \\ \vdots \\ \mathbf{j}_d \end{pmatrix}$$

are invertible modulo M. Note that in fact it is sufficient to know one of these matrices is invertible, the other will then also be invertible. Of course $\mathbf{S}$ can be directly chosen in a CP attack and $\mathbf{R}$ in a CC attack. If $\mathbf{S}$ and $\mathbf{R}$ are known, the key $\mathbf{H}$ is easily obtained:

$$\mathbf{R} \equiv \mathbf{SH} \mod M \quad \text{or} \quad \mathbf{H} \equiv \mathbf{S}^{-1}\mathbf{R} \mod M.$$

HILL is difficult to break, if one doesn't at least have some KP data available, especially if d is large and/or the cryptanalyzer does not know the value of d. On the other hand, a KP attack and especially a CP or a CC attack is easy—very little data is needed—so HILL is not suitable for high-end encryption.

AFFINE-HILL is a little harder to break than HILL. In a KP attack you need message-block-cryptoblock pairs $(\mathbf{i}_1, \mathbf{j}_1), \ldots, (\mathbf{i}_{d+1}, \mathbf{j}_{d+1})$ such that the matrices

$$\mathbf{S} = \begin{pmatrix} \mathbf{i}_1 - \mathbf{i}_{d+1} \\ \vdots \\ \mathbf{i}_d - \mathbf{i}_{d+1} \end{pmatrix} \quad \text{and} \quad \mathbf{R} = \begin{pmatrix} \mathbf{j}_1 - \mathbf{j}_{d+1} \\ \vdots \\ \mathbf{j}_d - \mathbf{j}_{d+1} \end{pmatrix}$$

are invertible modulo M. Note again, that it is actually sufficient to know that one of these matrices is invertible. In a CP attack $\mathbf{S}$ can be directly chosen, as can $\mathbf{R}$ in a CC attack. If $\mathbf{S}$ and $\mathbf{R}$ are known, $\mathbf{H}$ is easily obtained in the same manner as above. When $\mathbf{H}$ is known, $\mathbf{b}$ is easily obtained.

VIGENÈRE

VIGENÈRE was a widely used cryptosystem in its heydays. Its breaking was improved on with time, reaching a quite respectable level of ingenuity. The first step is to find d. There are specific methods for this, and d in VIGENÈRE is usually quite large. After this we can apply frequency analysis. See STINSON or SALOMAA or BAUER.

ONE-TIME-PAD

If the key is not available to the cryptanalyzer, ONE-TIME-PAD is impossible to break in a CO attack. However, if the same key is used many times, we basically come to a VIGENÈRE-encrypting.

"In real life, I assure you, there is
no such thing as algebra."

(FRAN LEBOWITZ)

Chapter 4

ALGEBRA: RINGS AND FIELDS

4.1 Rings and Fields

An *algebraic structure* is formed of a set A. There must be one or more computing operations defined on this set's elements and these operations must follow some calculation rules. Also usually a special role is given to some element(s) of A.

A *ring* is a structure $R = (A, \oplus, \odot, \mathbf{0}, \mathbf{1})$ where $\oplus$ is the ring's *addition operation,* $\odot$ is the ring's *multiplication operation,* $\mathbf{0}$ is the ring's *zero element,* and $\mathbf{1}$ is the ring's *identity element* (and $\mathbf{0} \neq \mathbf{1}$). If $\oplus, \odot, \mathbf{0}$ and $\mathbf{1}$ are obvious within the context then the ring is often simply denoted by A. It is also required that the following conditions hold true:

(1) $\oplus$ and $\odot$ are *commutative* operations, in other words, always

$$a \oplus b = b \oplus a \quad \text{and} \quad a \odot b = b \odot a.$$

(2) $\oplus$ and $\odot$ are *associative* operations, in other words, always

$$(a \oplus b) \oplus c = a \oplus (b \oplus c) \quad \text{and} \quad (a \odot b) \odot c = a \odot (b \odot c).$$

It follows from associativity that long sum and product chains can be written using parentheses in any (allowed) way you like without changing the result. Often they are written completely without parentheses, for example $a_1 \oplus a_2 \oplus \cdots \oplus a_k$ or $a_1 \odot a_2 \odot \cdots \odot a_k$. Especially we get in this way *multiples* and *powers,* that is, expressions

$$ka = \underbrace{a \oplus \cdots \oplus a}_{k \text{ times}} \quad \text{and} \quad a^k = \underbrace{a \odot \cdots \odot a}_{k \text{ times}}$$

and, as special cases, $0a = \mathbf{0}$, $1a = a$, $a^0 = \mathbf{1}$ and $a^1 = a$.

(3) $\mathbf{0} \oplus a = a$ and $\mathbf{1} \odot a = a$ (note how these are compatible with multiples and powers).

(4) $a \odot (b \oplus c) = (a \odot b) \oplus (a \odot c)$ (*distributivity*).

(5) For every element a there is an *additive inverse* or *opposite element* $-a$, which satisfies $(-a) \oplus a = \mathbf{0}$. Using additive inverses we obtain *subtraction* $a \ominus b = a \oplus (-b)$ and *negative multiples* $(-k)a = k(-a)$.

NB. *To be more precise, this kind of ring R is a so-called* commutative ring with identity, *a proper ring is an even more general concept in the algebraic sense. See the course Algebra 1. In future what we mean by a ring is this kind of commutative ring with identity.*

If the following condition (6) is also valid in addition to the above ones, then R is a so-called *field*:

(6) For every element $a \neq \mathbf{0}$ there is a *(multiplicative) inverse* a^{-1}, for which $a \odot a^{-1} = \mathbf{1}$. Using inverses we obtain *division* $a/b = a \odot b^{-1}$ and *negative powers* $a^{-k} = (a^{-1})^k$.

It is usually agreed that multiplication and division must be performed before addition and subtraction, which allows us to leave out a lot of parentheses. From these conditions we can derive many "familiar" calculation rules, for example

$$-(a \odot b) = (-a) \odot b \quad \text{and} \quad \frac{a \odot b}{c \odot d} = \frac{a}{c} \odot \frac{b}{d}.$$

So, every field is also a ring. Familiar rings which are not fields are for example the ring of integers $\mathbb{Z}$ and various *polynomial rings,* e.g. polynomial rings with rational, real, complex or integral coefficients, denoted by $\mathbb{Q}[x]$, $\mathbb{R}[x]$, $\mathbb{C}[x]$ and $\mathbb{Z}[x]$. Computational operations in these rings are the common $+$ and $\cdot$, the zero element is 0, and the identity element is 1. Also $\mathbb{Z}_m$ (residue classes modulo m) forms a ring, so a residue class ring is truly a ring, see Section 2.5.

Familiar fields are *number fields*, the field of real numbers $(\mathbb{R}, +, \cdot, 0, 1)$, the field of rational numbers $(\mathbb{Q}, +, \cdot, 0, 1)$ and the field of complex numbers $(\mathbb{C}, +, \cdot, 0, 1)$, and e.g. the field of rational functions with real coefficients $(\mathbb{R}(x), +, \cdot, 0, 1)$ and the *prime fields* $(\mathbb{Z}_p, +, \cdot, \overline{0}, \overline{1})$ (see Section 2.5). These are usually denoted briefly by $\mathbb{R}$, $\mathbb{Q}$, $\mathbb{C}$, $\mathbb{R}(x)$ and $\mathbb{Z}_p$.

4.2 Polynomial Rings

Polynomials defined formally using the elements of a field F as coefficients, form the so-called *polynomial ring* of F, denoted by $F[x]$. A polynomial is written as the familiar sum expression using a dummy variable (here x):

$$p(x) = a_0 \oplus a_1 x \oplus a_2 x^2 \oplus \cdots \oplus a_n x^n \,, \quad \text{where } a_0, a_1, \ldots, a_n \in F \text{ and } a_n \neq \mathbf{0}.$$

The *zero polynomial* is the empty sum. In the usual way the zero polynomial of $F[x]$ is identified with the zero element $\mathbf{0}$ of F and constant polynomials with the corresponding elements of F. Further, the *degree* of a polynomial $p(x)$, denoted $\deg(p(x))$, is defined in the usual way as the exponent of the highest power of x in the polynomial (the degree above is n). It is agreed that the degree of the zero polynomial is -1 (just for the sake of completeness). The coefficient of the highest power of x in the polynomial is called the *leading coefficient* (above a_n). If the leading coefficient is $= \mathbf{1}$, then the polynomial is a so-called *monic polynomial.* Conventionally the term $\mathbf{1}x^i$ can be replaced by x^i and the term $(-\mathbf{1})x^i$ by $\ominus x^i$, and a term $\mathbf{0}x^i$ can be left out altogether.

Addition, subtraction and *multiplication* of polynomials are defined in the usual way using coefficients and the corresponding computational operations of the field. Let's study these operations on the generic polynomials

$$p_1(x) = a_0 \oplus a_1 x \oplus a_2 x^2 \oplus \cdots \oplus a_n x^n \quad \text{and} \quad p_2(x) = b_0 \oplus b_1 x \oplus b_2 x^2 \oplus \cdots \oplus b_m x^m$$

where $a_n, b_m \neq \mathbf{0}$. (So we assume here that $p_1(x), p_2(x) \neq \mathbf{0}$.) Then

$$p_1(x) \oplus p_2(x) = c_0 \oplus c_1 x \oplus c_2 x^2 \oplus \cdots \oplus c_k x^k$$

where $k = \max(n, m)$ and

$$c_i = \begin{cases} a_i \oplus b_i, \text{ if } i \leq n, m \\ a_i, \text{ if } m < i \leq n \\ b_i, \text{ if } n < i \leq m. \end{cases}$$

Note that if $n = m$ then c_k can be $= \mathbf{0}$, in other words, the degree of the sum can be $< k$. Further, the *opposite polynomial* of $p_2(x)$ is

$$-p_2(x) = (-b_0) \oplus (-b_1)x \oplus (-b_2)x^2 \oplus \cdots \oplus (-b_m)x^m$$

and we get the subtraction in the form

$$p_1(x) \ominus p_2(x) = p_1(x) \oplus (-p_2(x)).$$

Multiplication is defined as follows:

$$p_1(x) \odot p_2(x) = c_0 \oplus c_1 x \oplus c_2 x^2 \oplus \cdots \oplus c_{n+m} x^{n+m}$$

where

$$c_i = \bigoplus_{t+s=i} a_t \odot b_s.$$

Hence

$$\deg(p_1(x) \odot p_2(x)) = \deg(p_1(x)) + \deg(p_2(x)).$$

It is easy, although a bit tedious, to confirm that the $(F[x], \oplus, \odot, \mathbf{0}, \mathbf{1})$ obtained in this way is indeed a ring.

Furthermore, *division* is defined for polynomials $a(x)$ and $m(x) \neq \mathbf{0}$ in the form

$$a(x) = q(x) \odot m(x) \oplus r(x) \quad , \quad \deg(r(x)) < \deg(m(x))$$

(*quotient* $q(x)$ and *remainder* $r(x)$). Remember it was agreed that the degree of the zero polynomial is -1. The result of the division is unambiquous, because if

$$a(x) = q_1(x) \odot m(x) \oplus r_1(x) = q_2(x) \odot m(x) \oplus r_2(x)$$

where $\deg(r_1(x)), \deg(r_2(x)) < \deg(m(x))$ then

$$r_1(x) \ominus r_2(x) = (q_2(x) \ominus q_1(x)) \odot m(x).$$

But $\deg(r_1(x) \ominus r_2(x)) < \deg(m(x))$, so the only possibility is that $q_2(x) \ominus q_1(x)$ is the zero polynomial. i.e. $q_1(x) = q_2(x)$, and further that $r_1(x) = r_2(x)$.

Division can be performed by the following algorithm, which then also shows that division is possible (the output is denoted by $\mathrm{DIV}(a(x), m(x)) = (q(x), r(x))$):

Division of polynomials:

1. Set $q(x) \leftarrow \mathbf{0}$ and $n \leftarrow \deg(a(x))$ and $k \leftarrow \deg(m(x))$. Denote the leading coefficient of $m(x)$ by m_k.
2. If $n < k$, return $(q(x), a(x))$, and quit.
3. Find the leading coefficient a_n of $a(x)$.

4. Set
$$a(x) \leftarrow a(x) \ominus (a_n \odot m_k^{-1}) \odot x^{n-k} \odot m(x)$$
and
$$q(x) \leftarrow q(x) \oplus (a_n \odot m_k^{-1}) \odot x^{n-k}$$
and $n \leftarrow \deg(a(x))$ and go to #2.

Each time we repeat #4 the degree n gets smaller and so eventually we come out at #2.

Further, we can define factors and divisibility as in Section 2.1. If $a(x) = q(x) \odot m(x)$, we say that $a(x)$ is *divisible* by $m(x)$ or that $m(x)$ is a *factor* of $a(x)$. A polynomial which has no factors of lower degree other than constant polynomials is called *irreducible.*

When dividing $a(x)$ by $m(x)$ the remainder $r(x)$ is said to be a residue of $a(x)$ modulo $m(x)$, compare the corresponding concept for integers in Section 2.4. $m(x)$ acts as a *modulus.* Here it is assumed that the modulus is at least of degree 1. The same kind of notation is also used as for integers: If the residues of $a(x)$ and $b(x)$ modulo $m(x)$ are equal, we denote
$$a(x) \equiv b(x) \mod m(x)$$
and say that $a(x)$ *is congruent to* $b(x)$ *modulo* $m(x)$. The same calculation rules apply to polynomial congruences as for integers.

The *residue class* $\overline{a(x)} = \overline{r(x)}$ corresponding to the residue $r(x)$ is formed by all those polynomials $a(x)$ whose residue modulo $m(x)$ is $r(x)$. All residue classes modulo $m(x)$ form the so-called *residue class ring* or *factor ring* or *quotient ring* $F[x]/m(x)$.[1] It is easy to see, in the same way as for integers, that residue classes modulo $m(x)$ can be given and be calculated with by "using representatives", in other words,
$$\begin{aligned}
&\overline{a_1(x)} \oplus \overline{a_2(x)} = \overline{a_1(x) \oplus a_2(x)} \quad , \quad -\overline{a(x)} = \overline{-a(x)}\,, \\
&\overline{a_1(x)} \ominus \overline{a_2(x)} = \overline{a_1(x) \ominus a_2(x)} \quad , \quad k\overline{a(x)} = \overline{ka(x)}\,, \\
&\overline{a_1(x)} \odot \overline{a_2(x)} = \overline{a_1(x) \odot a_2(x)} \quad \text{and} \quad \overline{a(x)}^k = \overline{a(x)^k},
\end{aligned}$$
and the result does not depend on the choice of the representatives. (The operations are thus well-defined.) The most common representative system is the set formed by all possible remainders, or polynomials of at most degree $\deg(m(x)) - 1$. Hence $F[x]/m(x)$ is truly a ring.

Furthermore, just as we showed that every element of $\mathbb{Z}_p$ other than the zero element $\overline{0}$ has an inverse, we can show that every element of $F[x]/p(x)$ other than the zero element $\overline{\mathbf{0}}$ has an inverse, assuming that the modulus $p(x)$ is an irreducible polynomial. For this purpose we need the greatest common divisor of two or more polynomials in $F[x]$ and the Euclidean algorithm for polynomials.

The *greatest common divisor* (g.c.d.) of the polynomials $a(x)$ and $b(x)$ of $F[x]$ (not both the zero polynomial) is a polynomial $d(x)$ of the highest degree that divides both $a(x)$ and $b(x)$, denoted $d(x) = \gcd(a(x), b(x))$. Note that such greatest common divisor is not unique, since if $d(x) = \gcd(a(x), b(x))$ then also $c \odot d(x)$, where $c \neq \mathbf{0}$ is constant polynomial, is $\gcd(a(x), b(x))$. It is therefore often required that $d(x)$ is a monic polynomial.

Theorem 4.1. (Bézout's theorem) *If at least one of the polynomials $a(x)$ and $b(x)$ is nonzero then any g.c.d. of theirs can be written in the form*
$$d(x) = c_1(x) \odot a(x) \oplus c_2(x) \odot b(x) \quad \textit{(}\text{Bézout's form}\textit{).}$$
In addition, if $a(x), b(x) \neq \mathbf{0}$, it may be assumed that $\deg(c_1(x)) \leq \deg(b(x))$ and $\deg(c_2(x)) \leq \deg(a(x))$.

[1] A similar notation is often used for integers: $\mathbb{Z}_m = \mathbb{Z}/m$.

Proof. The proof is quite similar to the proof of Theorem 2.5. We denote $\mathrm{GCD}(a(x), b(x)) = (d(x), c_1(x), c_2(x))$ and assume that $\deg(a(x)) \leq \deg(b(x))$. The *(Generalized) Euclidean algorithm* needed in the proof is the following recursion:

The (Generalized) Euclidean algorithm for polynomials:

1. If $a(x) = \mathbf{0}$ then we return $\mathrm{GCD}(a(x), b(x)) = (b(x), \mathbf{0}, \mathbf{1})$, and quit.
2. If $a(x) \neq \mathbf{0}$ is a constant polynomial, we return $\mathrm{GCD}(a(x), b(x)) = (a(x), \mathbf{1}, \mathbf{0})$, and quit.
3. If $\deg(a(x)) \geq 1$ then we find the residue $r(x)$ of $b(x)$ modulo $a(x)$, in other words, we write $b(x) = q(x) \odot a(x) \oplus r(x)$ where $\deg(r(x)) < \deg(a(x))$. Then we find $\mathrm{GCD}(r(x), a(x)) = (d(x), e_1(x), e_2(x))$. Because $d(x) = e_1(x) \odot r(x) \oplus e_2(x) \odot a(x)$, then $d(x) = \gcd(a(x), b(x))$ and
$$d(x) = (e_2(x) \ominus e_1(x) \odot q(x)) \odot a(x) \oplus e_1(x) \odot b(x).$$
We return $\mathrm{GCD}(a(x), b(x)) = (d(x), e_2(x) \ominus e_1(x) \odot q(x), e_1(x))$, and quit.

The process ends since $\min(\deg(r(x)), \deg(a(x))) < \min(\deg(a(x), \deg(b(x)))$, in other words, each time we call GCD the minimum degree gets lower. □

If $\gcd(a(x), m(x))$ is a constant $f \neq \mathbf{0}$ then by multiplying both sides of Bézout's form by f^{-1} we obtain
$$\mathbf{1} = e_1(x) \odot a(x) \oplus e_2(x) \odot m(x).$$
Hence in this case $a(x)$ has an inverse $e_1(x)$ modulo $m(x)$, i.e. $\overline{a(x)}$ has an inverse $\overline{e_1(x)}$ in $F[x]/m(x)$. (Assuming that $\deg(m(x)) \geq 1$.) At the same time we have a method for finding the inverse.

In the special case where $p(x)$ is an irreducible polynomial of $F[x]$ and its degree is at least 1 the factor ring $F[x]/p(x)$ is a field. Elements of this field are usually written in the residue form
$$c_0 \oplus c_1 x \oplus c_2 x^2 \oplus \cdots \oplus c_{n-1} x^{n-1}$$
where $n = \deg(p(x))$ and the coefficients $c_0, c_1, \ldots, c_{n-1}$ are elements of F, that is, essentially as n-vectors whose components are in F. Note that in this form c_{n-1} can be $= \mathbf{0}$. If $p(x)$ is of the first degree then $F[x]/p(x) = F$, that is, we return to the original field.

Example. *Irreducible polynomials of $\mathbb{R}[x]$ are, except for the constants, either of the first or the second degree. (This statement is equivalent to the Fundamental theorem of algebra, see the course Complex Analysis.) We obtain from the former $\mathbb{R}$ and from the latter $\mathbb{C}$. So for example $\mathbb{C} = \mathbb{R}[x]/(x^2 + 1)$. On the other hand, irreducible polynomials of $\mathbb{C}[x]$ are constants or of the first degree, so that doesn't lead us anywhere.*

A polynomial ring $R[x]$ can also be formed using the elements of the ring R as coefficients, in this way we obtain for example the polynomial ring with integer coefficients $\mathbb{Z}[x]$. In such polynomial rings addition, subtraction and multiplication are defined as usual, but division is not generally possible. By studying the division algorithm it becomes clear that *division is defined if the leading coefficient of the dividing polynomial has an inverse in R.* In the special case where *the divisor is a monic polynomial division is defined in any polynomial ring.* Hence the residue class ring $R[x]/m(x)$ is defined only if the leading coefficient of $m(x)$ has an inverse in R, and always if $m(x)$ is a monic polynomial.

This kind of division is needed for example in the NTRU cryptosystem, see Chapter 11.

4.3 Finite Fields

Prime fields were denoted by $\mathbb{Z}_p$ in Section 2.5 or as residue classes modulo a prime number p. A prime field is one example of a *finite field,* but there are others. To obtain these we choose an irreducible polynomial $P(x)$ from the polynomial ring $\mathbb{Z}_p[x]$ of the prime field $\mathbb{Z}_p$. Residues modulo $P(x)$ form the field $\mathbb{Z}_p[x]/P(x)$ the elements of which are usually expressed in the form

$$c_0 + c_1x + c_2x^2 + \cdots + c_{n-1}x^{n-1}$$

where $n = \deg(P(x))$ and $c_0, \ldots, c_{n-1} \in \mathbb{Z}_p$, or essentially as vectors $(c_0, c_1, \ldots, c_{n-1})$. This field is finite, it has as many elements as there are residues modulo $P(x)$ (that is, p^n).

It can be shown (passed here), that every possible finite field can be obtained in this way—including the prime field $\mathbb{Z}_p$ itself. So the number of elements in a finite field is always a power of a prime number. There are many ways to construct finite fields, in particular, there are usually more than one irreducible polynomial to choose from in $\mathbb{Z}_p[x]$, but all finite fields with p^n elements are structurally the same, that is, they are isomorphic to any field $\mathbb{Z}_p[x]/P(x)$ where $\deg(P(x)) = n$. Hence there is essentially only one finite field with p^n elements, and it's denoted by $\mathbb{F}_{p^n}$ or by $\mathrm{GF}(p^n)$.[2] For each power p^n there exists an $\mathbb{F}_{p^n}$, in other words, you can find irreducible polynomials of all degrees $n \geq 1$ in the polynomial ring $\mathbb{Z}_p[x]$.

NB. *If we take an irreducible polynomial $P(x)$ of degree m with coefficients in the finite field $\mathbb{F}_{p^n}$, i.e. an irreducible element of the polynomial ring $\mathbb{F}_{p^n}[x]$, then—as noted—the factor ring $\mathbb{F}_{p^n}/P(x)$ of residues modulo $P(x)$ is a field that has $(p^n)^m = p^{nm}$ elements. This field must be $\mathbb{F}_{p^{nm}}$, and it is isomorphic to some $\mathbb{Z}_p[x]/Q(x)$ where $Q(x)$ is an irreducible polynomial of degree nm in $\mathbb{Z}_p[x]$.*

In practice calculating in a finite field $\mathbb{F}_{p^n}$ is done by expressing the elements as residue classes modulo some irreducible polynomial $P(x) \in \mathbb{Z}_p[x]$ of degree n. The operations are carried out by using representatives of degree no higher than $n-1$, or residues, to which results are also reduced modulo $P(x)$ by division. If p and/or n is large, these operations are obviously very laborious by hand. There are other representations for finite fields. Representation as powers of primitive elements is used a lot in some cryptosystems (see Chapter 10).

Example. *To construct $\mathbb{F}_{2^8}$ we may choose the irreducible polynomial $P(x) = 1+x+x^3+x^4+x^8$ in $\mathbb{Z}_2[x]$ of degree 8. Let's check that $P(x)$ is indeed irreducible using the Maple program:*

```
> Irreduc(1+x+x^3+x^4+x^8) mod 2;
```

$$true$$

Elements of $\mathbb{F}_{2^8}$ are in the residue form

$$b_0 + b_1x + b_2x^2 + b_3x^3 + b_4x^4 + b_5x^5 + b_6x^6 + b_7x^7$$

where $b_0, \ldots, b_7$ are bits, essentially as bit vectors $(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7)$. Using the `GF` *library of Maple we can calculate in finite fields, although it's a bit clumsy. Let's try the library on $\mathbb{F}_{2^8}$:*

```
> GF256:=GF(2,8,1+x+x^3+x^4+x^8):
> a:=GF256[ConvertIn](x);
```

[2] "GF" = "Galois' field". Of course $\mathbb{Z}_p = \mathbb{F}_p = \mathrm{GF}(p)$.

$$a := x \mod 2$$

```
> GF256[`^`](a,1200);
```

$$(x^7 + x^6 + x^5 + x^3 + x^2 + x + 1) \mod 2$$

```
> c:=GF256[inverse](a);
```

$$c := (x^7 + x^3 + x^2 + 1) \mod 2$$

```
> GF256[`+`](a,GF256[`^`](c,39));
```

$$(x^7 + x^5 + x^3 + 1) \mod 2$$

So here we calculated in residue form the elements $x^{1\,200}, x^{-1}$ *and* $x + x^{39}$. *The command* `ConvertIn` *converts a polynomial to Maple's inner representation.*

If you don't know any suitable irreducible polynomial of $\mathbb{Z}_p[x]$, *Maple will find one for you:*

```
> GF81:=GF(3,4):
> GF81[extension];
```

$$(T^4 + T^3 + 2T + 1) \mod 3$$

The choice can be found by using the `extension` *command. So here we got as a result the irreducible polynomial* $\overline{1} + \overline{2}x + x^3 + x^4$ *of* $\mathbb{Z}_3[x]$.

Matrix and vector operations in finite fields are defined as usual by the operations of their elements. In this way we can apply addition, subtraction, multiplication, powers and transposes of matrices, familiar from basic courses. Also determinants of square matrices follow the familiar calculation rules. Just as in basic courses, we note that a square matrix has an inverse matrix if and only if its determinant is not the zero element of the field.

Besides cryptology, finite fields are very important for error-correcting codes. They are discussed more in the courses Finite Fields and Coding Theory. Good references are McEliece and Lidl & Niederreiter and also Garrett. The mass encryption system AES, which is in general use nowadays, is based on the finite field $\mathbb{F}_{2^8}$, see the next chapter.

"The grand thing is to be able to reason backwards."

(ARTHUR CONAN DOYLE: *A Study in Scarlet*)

Chapter 5

AES

5.1 Background

AES (Advanced Encryption Standard) is a fast symmetric cryptosystem for mass encryption. It was developed through competition, and is based on the *RIJNDAEL system,* published in 1999 by Joan Daemen and Vincent Rijmen from Belgium, see DAEMEN & RIJMEN. AES replaced the old DES system (Data Encryption Standard, see Appendix) published in 1975.

AES works on bit symbols, so the residue classes (bits) 0 and 1 of $\mathbb{Z}_2$ can be considered as plaintext and cryptotext symbols. The workings of RIJNDAEL can be described using the field $\mathbb{F}_{2^8}$ and its polynomial ring $\mathbb{F}_{2^8}[z]$. To avoid confusion we use z as the dummy variable in the polynomial ring and x as the dummy variable for polynomials in $\mathbb{Z}_2$ needed in defining and representing the field $\mathbb{F}_{2^8}$. Furthermore, we denote addition and multiplication in $\mathbb{F}_{2^8}$ by $\oplus$ and $\odot$, the identity element is denoted by $\mathbf{1}$ and the zero element by $\mathbf{0}$. Note that because $1 = -1$ in $\mathbb{Z}_2$, the additional inverse of an element in $\mathbb{Z}_2[x]$, $\mathbb{F}_{2^8}$ and in $\mathbb{F}_{2^8}[z]$ is the element itself. So subtraction $\ominus$ is the same as addition $\oplus$, in this case.

5.2 RIJNDAEL

In the RIJNDAEL system the length l_{B} of the plaintext block and the length l_{K} of the key are independently either 128, 192 or 256 bits. Dividing by 32 we get the numbers

$$N_{\mathrm{B}} = \frac{l_{\mathrm{B}}}{32} \quad \text{and} \quad N_{\mathrm{K}} = \frac{l_{\mathrm{K}}}{32}.$$

Bits are handled as bytes of 8 bits. An 8-bit byte $b_7b_6 \cdots b_0$ can be considered as an element of the finite field $\mathbb{F}_{2^8}$, which has the residue representation $b_0 + b_1x + b_2x^2 + b_3x^3 + b_4x^4 + b_5x^5 + b_6x^6 + b_7x^7$, see the example in Section 4.3 and note the order of terms.

The key is usually expressed as a $4 \times N_{\mathrm{K}}$ matrix whose elements are bytes. If the key is, byte by byte,

$$\mathbf{k} = k_{00}k_{10}k_{20}k_{30}k_{01}k_{11}k_{21} \cdots k_{3,N_{\mathrm{K}}-1}$$

then the corresponding matrix is

$$\mathbf{K} = \begin{pmatrix} k_{00} & k_{01} & k_{02} & \cdots & k_{0,N_{\mathrm{K}}-1} \\ k_{10} & k_{11} & k_{12} & \cdots & k_{1,N_{\mathrm{K}}-1} \\ k_{20} & k_{21} & k_{22} & \cdots & k_{2,N_{\mathrm{K}}-1} \\ k_{30} & k_{31} & k_{32} & \cdots & k_{3,N_{\mathrm{K}}-1} \end{pmatrix}.$$

Note how the elements of the matrix are indexed starting from zero. Similarly, if the input block (plaintext block) is, byte by byte,

$$\mathbf{a} = a_{00}a_{10}a_{20}a_{30}a_{01}a_{11}a_{21}\cdots a_{3,N_\mathrm{B}-1}$$

then the corresponding matrix is

$$\mathbf{A} = \begin{pmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0,N_\mathrm{B}-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1,N_\mathrm{B}-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2,N_\mathrm{B}-1} \\ a_{30} & a_{31} & a_{32} & \cdots & a_{3,N_\mathrm{B}-1} \end{pmatrix}.$$

During encryption we are dealing with a bit sequence of length l_B, the so-called *state*. Like the block, it is also expressed byte by byte in the form of a $4 \times N_\mathrm{B}$ matrix:

$$\mathbf{S} = \begin{pmatrix} s_{00} & s_{01} & s_{02} & \cdots & s_{0,N_\mathrm{B}-1} \\ s_{10} & s_{11} & s_{12} & \cdots & s_{1,N_\mathrm{B}-1} \\ s_{20} & s_{21} & s_{22} & \cdots & s_{2,N_\mathrm{B}-1} \\ s_{30} & s_{31} & s_{32} & \cdots & s_{3,N_\mathrm{B}-1} \end{pmatrix}.$$

Elements of the matrices $\mathbf{K}$, $\mathbf{A}$ and $\mathbf{S}$ are bytes of 8 bits, which can be interpreted as elements of the field $\mathbb{F}_{2^8}$. In this way these matrices are matrices over this field. Another way to interpret the matrices is to consider their columns as sequences of elements of the field $\mathbb{F}_{2^8}$ of length 4. These can be interpreted further, from top to bottom, as coefficients of polynomials with maximum degree 3 from the polynomial ring $\mathbb{F}_{2^8}[z]$. So, the state $\mathbf{S}$ mentioned above would thus correspond to the polynomial sequence

$$s_{00} \oplus s_{10}z \oplus s_{20}z^2 \oplus s_{30}z^3\ ,\ s_{01} \oplus s_{11}z \oplus s_{21}z^2 \oplus s_{31}z^3\ , \ldots ,$$
$$s_{0,N_\mathrm{B}-1} \oplus s_{1,N_\mathrm{B}-1}z \oplus s_{2,N_\mathrm{B}-1}z^2 \oplus s_{3,N_\mathrm{B}-1}z^3.$$

For the representation to be unique, a given fixed irreducible polynomial of degree 8 from $\mathbb{Z}_2[x]$ must be used in the construction of $\mathbb{F}_{2^8}$. In RIJNDAEL it is the so-called *RIJNDAEL polynomial*

$$p(x) = 1 + x + x^3 + x^4 + x^8$$

which, by the way, is the same as in the example in Section 4.3.

5.2.1 Rounds

There is a certain number N_R of so-called *rounds* in RIJNDAEL. The number of rounds is given by the following table:

N_R	$N_\mathrm{B} = 4$	$N_\mathrm{B} = 6$	$N_\mathrm{B} = 8$
$N_\mathrm{K} = 4$	10	12	14
$N_\mathrm{K} = 6$	12	12	14
$N_\mathrm{K} = 8$	14	14	14

The i^{th} round receives as its input the current state $\mathbf{S}$ and its own so-called *round key* $\mathbf{R}_i$. In particular, we need the initial round key $\mathbf{R}_0$. In each round, except for the last one, we go through the following sequence of operations:

$$\mathbf{S} \leftarrow \mathrm{SubBytes}(\mathbf{S})$$
$$\mathbf{S} \leftarrow \mathrm{ShiftRows}(\mathbf{S})$$
$$\mathbf{S} \leftarrow \mathrm{MixColumns}(\mathbf{S})$$
$$\mathbf{S} \leftarrow \mathrm{AddRoundKey}(\mathbf{S}, \mathbf{R}_i)$$

The last round is the same except that we drop $\mathrm{MixColumns}$.

The encrypting key is *expanded* first and then used to distribute round keys to all rounds. This and the different operations in rounds are discussed one by one in the following sections. Encrypting itself then consists of the following steps:

- Initialize the state: $\mathbf{S} \leftarrow \mathrm{AddRoundKey}(\mathbf{A}, \mathbf{R}_0)$.
- $N_R - 1$ "usual" rounds.
- The last round.

When decrypting we go through the inverse steps in reverse order.

5.2.2 Transforming Bytes (SubBytes)

In this operation each byte s_{ij} of the state is transformed in the following way:

1. Interpret s_{ij} as an element of the field $\mathbb{F}_{2^8}$ and compute its inverse s_{ij}^{-1}. It is agreed here that the inverse of the zero element is the element itself.
2. Expand s_{ij}^{-1} in eight bits $b_7b_6b_5b_4b_3b_2b_1b_0$, denote

   $$b(x) = b_0 + b_1x + b_2x^2 + b_3x^3 + b_4x^4 + b_5x^5 + b_6x^6 + b_7x^7 \quad \text{(a polynomial in } \mathbb{Z}_2[x])$$

   and compute

   $$b'(x) \equiv b(x)(1 + x + x^2 + x^3 + x^4) + (1 + x + x^5 + x^6) \mod 1 + x^8.$$

   The result

   $$b'(x) = b'_0 + b'_1x + b'_2x^2 + b'_3x^3 + b'_4x^4 + b'_5x^5 + b'_6x^6 + b'_7x^7$$

   is interpreted as a byte $b'_7b'_6b'_5b'_4b'_3b'_2b'_1b'_0$ or as an element of $\mathbb{F}_{2^8}$. By the way, division by $1 + x^8$ in $\mathbb{Z}_2[x]$ is easy since

   $$x^k \equiv x^{(k, \bmod 8)} \mod 1 + x^8.$$

The operation in #2 may also be done by using matrices. We then apply an affine transformation in $\mathbb{Z}_2$:

$$\begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}.$$

Byte transformation is done in reverse order during the decryption. Because in $\mathbb{Z}_2[x]$

$$1 = \gcd(1 + x + x^2 + x^3 + x^4, 1 + x^8)$$

(easy to verify using the Euclidean algorithm), the polynomial $1 + x + x^2 + x^3 + x^4$ has an inverse modulo $1 + x^8$ and the occuring 8×8 matrix is invertible modulo 2. This inverse is $x + x^3 + x^6$.

Transforming the byte is in all a nonlinear transformation, which can be given in one table, the so-called *RIJNDAEL S-box.* This table can be found for example in MOLLIN and STINSON.

5.2.3 Shifting Rows (ShiftRows)

In this operation the elements of the rows of the matrix representation of the state are shifted left cyclically in the following way:

shift	row 0	row 1	row 2	row 3
$N_{\mathrm{B}} = 4$	no shift	1 element	2 elements	3 elements
$N_{\mathrm{B}} = 6$	no shift	1 element	2 elements	3 elements
$N_{\mathrm{B}} = 8$	no shift	1 element	3 elements	4 elements

While decrypting rows are correspondingly shifted right cyclically.

5.2.4 Mixing Columns (MixColumns)

In this transformation columns of the state matrix are interpreted as polynomials of maximum degree 3 in the polynomial ring $\mathbb{F}_{2^8}[z]$. Each column (polynomial) is multiplied by the fixed polynomial

$$c(z) = c_0 \oplus c_1 z \oplus c_2 z^2 \oplus c_3 z^3 \in \mathbb{F}_{2^8}[z]$$

modulo $\mathbf{1} \oplus z^4$ where

$$c_0 = x \quad , \quad c_1 = c_2 = \mathbf{1} \quad \text{and} \quad c_3 = 1 + x.$$

Dividing by the polynomial $\mathbf{1} \oplus z^4$ in $\mathbb{F}_{2^8}[z]$ is especially easy since

$$z^k \equiv z^{(k,\,\mathrm{mod}\ 4)} \mod \mathbf{1} \oplus z^4.$$

Alternatively the operation can be considered as a linear transformation of $\mathbb{F}_{2^8}$:

$$\begin{pmatrix} s'_{0i} \\ s'_{1i} \\ s'_{2i} \\ s'_{3i} \end{pmatrix} = \begin{pmatrix} c_0 & c_3 & c_2 & c_1 \\ c_1 & c_0 & c_3 & c_2 \\ c_2 & c_1 & c_0 & c_3 \\ c_3 & c_2 & c_1 & c_0 \end{pmatrix} \begin{pmatrix} s_{0i} \\ s_{1i} \\ s_{2i} \\ s_{3i} \end{pmatrix}.$$

When decrypting we divide by the polynomial $c(z)$ modulo $\mathbf{1} \oplus z^4$. Although $\mathbf{1} \oplus z^4$ is not an irreducible polynomial of $\mathbb{F}_{2^8}[z]$[1], $c(z)$ has an inverse modulo $\mathbf{1} \oplus z^4$, because

$$\mathbf{1} = \gcd(c(z), \mathbf{1} \oplus z^4).$$

The inverse is obtained using the Euclidean algorithm (hard to compute!) and it is

$$d(z) = d_0 \oplus d_1 z \oplus d_2 z^2 \oplus d_3 z^3$$

where

$$d_0 = x + x^2 + x^3 \quad , \quad d_1 = 1 + x^3 \quad , \quad d_2 = 1 + x^2 + x^3 \quad \text{and} \quad d_3 = 1 + x + x^3.$$

So, when decrypting the column (polynomial) is multiplied by $d(z)$ modulo $\mathbf{1} \oplus z^4$ and the operation is thus no more complicated than when encrypting. In matrix form in $\mathbb{F}_{2^8}$

$$\begin{pmatrix} s_{0i} \\ s_{1i} \\ s_{2i} \\ s_{3i} \end{pmatrix} = \begin{pmatrix} d_0 & d_3 & d_2 & d_1 \\ d_1 & d_0 & d_3 & d_2 \\ d_2 & d_1 & d_0 & d_3 \\ d_3 & d_2 & d_1 & d_0 \end{pmatrix} \begin{pmatrix} s'_{0i} \\ s'_{1i} \\ s'_{2i} \\ s'_{3i} \end{pmatrix}.$$

[1] It happens to be $= (\mathbf{1} \oplus z)^4$.

5.2.5 Adding Round Keys (AddRoundKey)

The round key is as long as the state. In this operation the round key is added to the state byte by byte modulo 2. The inverse operation is the same.

5.2.6 Expanding the Key

The round keys $\mathbf{R}_0, \mathbf{R}_1, \ldots, \mathbf{R}_{N_\mathrm{R}}$ are obtained from the encrypting key by expanding it and then choosing from the expanded key certain parts for different rounds. The length of the expanded key in bits is $l_\mathrm{B}(N_\mathrm{R}+1)$. Divided into bytes it can be expressed as a $4 \times N_\mathrm{B}(N_\mathrm{R}+1)$ matrix, which has $N_\mathrm{B}(N_\mathrm{R}+1)$ columns of length 4:

$$\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_{N_\mathrm{B}(N_\mathrm{R}+1)-1}.$$

Denote the columns of the key (matrix $\mathbf{K}$) correspondingly:

$$\mathbf{k}_0, \mathbf{k}_1, \ldots, \mathbf{k}_{N_\mathrm{K}-1}.$$

The expanded key is computed using the following method:

1. Set $\mathbf{w}_i \leftarrow \mathbf{k}_i$ ($i = 0, \ldots, N_K - 1$).

2. Define the remaining $\mathbf{w}_i$'s recursively by the following rules where addition of vectors in $\mathbb{F}_{2^8}$ is done elementwise in the usual fashion:

   2.1 If $i \equiv 0 \mod N_\mathrm{K}$ then compute $u = x^{i/N_\mathrm{K}}$ in the field $\mathbb{F}_{2^8}$ and set

   $$\mathbf{w}_i \leftarrow \mathbf{w}_{i-N_\mathrm{K}} \oplus \mathrm{SubByte}(\mathrm{RotByte}(\mathbf{w}_{i-1})) \oplus \begin{pmatrix} u \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{pmatrix}.$$

   Here the operation $\mathrm{SubByte}$ means transforming every element (byte) of the column. Operation $\mathrm{RotByte}$ does a cyclic shift of one element up in a column.

   2.2 If $N_\mathrm{B} = 8$ and $i \equiv 4 \mod N_\mathrm{K}$, set

   $$\mathbf{w}_i \leftarrow \mathbf{w}_{i-N_K} \oplus \mathrm{SubByte}(\mathbf{w}_{i-1})$$

   where the operation $\mathrm{SubByte}$ is the same as in #2.1.

   2.3 Otherwise simply set

   $$\mathbf{w}_i \leftarrow \mathbf{w}_{i-N_\mathrm{K}} \oplus \mathbf{w}_{i-1}.$$

Now the round key $\mathbf{R}_i$ of the i^{th} round is obtained from the columns $\mathbf{w}_{iN_\mathrm{B}}, \ldots, \mathbf{w}_{(i+1)N_\mathrm{B}-1}$ ($i = 0, 1, \ldots, N_\mathrm{R}$). In particular, from the first N_B columns we get the initial round key $\mathbf{R}_0$.

NB. *Expansion of the key can be made in advance, as long as the encrypting key is known. Anyway, the x^{i/N_K}'s can be computed beforehand in the field $\mathbb{F}_{2^8}$.*

5.2.7 A Variant of Decryption

A straightforward procedure for decrypting follows the following chain of operations—they are the inverse operations of the encrypting operations that were introduced before:

$$
\begin{aligned}
&\mathbf{S} \leftarrow \mathrm{AddRoundKey}(\mathbf{S}, \mathbf{R}_{N_\mathrm{R}}) \\
&\mathbf{S} \leftarrow \mathrm{ShiftRows}^{-1}(\mathbf{S}) \\
&\mathbf{S} \leftarrow \mathrm{SubBytes}^{-1}(\mathbf{S}) \\
&\\
&\mathbf{S} \leftarrow \mathrm{AddRoundKey}(\mathbf{S}, \mathbf{R}_{N_\mathrm{R}-1}) \\
&\mathbf{S} \leftarrow \mathrm{MixColumns}^{-1}(\mathbf{S}) \\
&\mathbf{S} \leftarrow \mathrm{ShiftRows}^{-1}(\mathbf{S}) \\
&\mathbf{S} \leftarrow \mathrm{SubBytes}^{-1}(\mathbf{S}) \\
&\qquad\vdots \\
&\mathbf{S} \leftarrow \mathrm{AddRoundKey}(\mathbf{S}, \mathbf{R}_1) \\
&\mathbf{S} \leftarrow \mathrm{MixColumns}^{-1}(\mathbf{S}) \\
&\mathbf{S} \leftarrow \mathrm{ShiftRows}^{-1}(\mathbf{S}) \\
&\mathbf{S} \leftarrow \mathrm{SubBytes}^{-1}(\mathbf{S}) \\
&\\
&\mathbf{S} \leftarrow \mathrm{AddRoundKey}(\mathbf{S}, \mathbf{R}_0)
\end{aligned}
$$

The order of the operations can, however, also be inverted. First, the order of row shifting and transforming bytes does not matter, the former operates on rows and the latter on bytes. The same goes for the inverted operations. Second, the operations

$$
\begin{aligned}
&\mathbf{S} \leftarrow \mathrm{AddRoundKey}(\mathbf{S}, \mathbf{R}_i) \\
&\mathbf{S} \leftarrow \mathrm{MixColumns}^{-1}(\mathbf{S})
\end{aligned}
$$

can be replaced by the operations

$$
\begin{aligned}
&\mathbf{S} \leftarrow \mathrm{MixColumns}^{-1}(\mathbf{S}) \\
&\mathbf{S} \leftarrow \mathrm{AddRoundKey}(\mathbf{S}, \mathrm{MixColumns}^{-1}(\mathbf{R}_i))
\end{aligned}
$$

In this way decrypting can also follow the chain

$$
\begin{aligned}
&\mathbf{S} \leftarrow \mathrm{AddRoundKey}(\mathbf{S}, \mathbf{R}_{N_\mathrm{R}}) \\
&\\
&\mathbf{S} \leftarrow \mathrm{SubBytes}^{-1}(\mathbf{S}) \\
&\mathbf{S} \leftarrow \mathrm{ShiftRows}^{-1}(\mathbf{S}) \\
&\mathbf{S} \leftarrow \mathrm{MixColumns}^{-1}(\mathbf{S}) \\
&\mathbf{S} \leftarrow \mathrm{AddRoundKey}(\mathbf{S}, \mathrm{MixColumns}^{-1}(\mathbf{R}_{N_\mathrm{R}-1}))
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{S} \leftarrow \mathrm{SubBytes}^{-1}(\mathbf{S})\\
&\mathbf{S} \leftarrow \mathrm{ShiftRows}^{-1}(\mathbf{S})\\
&\mathbf{S} \leftarrow \mathrm{MixColumns}^{-1}(\mathbf{S})\\
&\mathbf{S} \leftarrow \mathrm{AddRoundKey}(\mathbf{S}, \mathrm{MixColumns}^{-1}(\mathbf{R}_{N_\mathrm{R}-2}))\\
&\qquad\vdots\\
&\mathbf{S} \leftarrow \mathrm{SubBytes}^{-1}(\mathbf{S})\\
&\mathbf{S} \leftarrow \mathrm{ShiftRows}^{-1}(\mathbf{S})\\
&\mathbf{S} \leftarrow \mathrm{MixColumns}^{-1}(\mathbf{S})\\
&\mathbf{S} \leftarrow \mathrm{AddRoundKey}(\mathbf{S}, \mathrm{MixColumns}^{-1}(\mathbf{R}_1))\\
&\\
&\mathbf{S} \leftarrow \mathrm{SubBytes}^{-1}(\mathbf{S})\\
&\mathbf{S} \leftarrow \mathrm{ShiftRows}^{-1}(\mathbf{S})\\
&\mathbf{S} \leftarrow \mathrm{AddRoundKey}(\mathbf{S}, \mathbf{R}_0)
\end{aligned}
$$

which reminds us very much of the encrypting process. Hence RIJNDAEL encrypting and decrypting are very similar operations.

5.3 RIJNDAEL's Cryptanalysis

RIJNDAEL is built to withstand just about every known attack on this kind of cryptosystem.[2] Its designers Joan Daemen and Vincent Rijmen gave an extensive description of the construction principles in a public document DAEMEN, J. & RIJMEN, V.: *AES Proposal: Rijndael* (1999), which they later expanded to the book DAEMEN & RIJMEN. It should be mentioned that linear cryptanalysis and differential cryptanalysis, that were much investigated in connection with DES, are efficiently prevented in RIJNDAEL in their various forms. These cryptanalyses are explained e.g. in STINSON (see also Appendix).

On the other hand, RIJNDAEL is actually the only "better" cryptosystem where the (single) S-box can be written in a comparatively simple algebraic form in $\mathbb{F}_{2^8}$:

$$S(b) = s_0 \oplus \bigoplus_{i=1}^{8} (s_i \odot b^{255-2^{i-1}})$$

for suitable elements $s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8$ of $\mathbb{F}_{2^8}$. Continuing from here it is relatively easy to derive an explicit algebraic formula for the whole encryption process! This has raised the question whether such formulas can be inverted efficiently. If the answer is yes, it would seem that RIJNDAEL can be broken after all. This is a matter of lively investigation, so far no weaknesses have been found.[3]

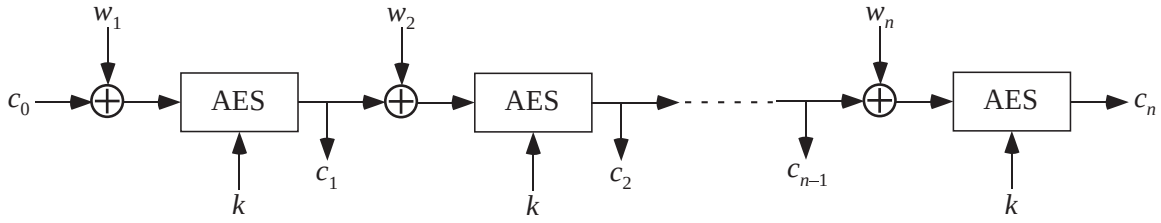
[2] Here among other things ideas of the Finnish mathematician Kaisa Nyberg were used. See NYBERG, K.: Differentially Uniform Mappings for Cryptography. *Proceedings of EuroCrypt '93. Lecture Notes in Computer Science* **765**. Springer–Verlag (1994), 55–64.

[3] See for example FERGUSON, N. & SCHROEPPEL, R. & WHITING, D.: A Simple Algebraic Representation of Rijndael. *Proceedings of SAC '01. Lecture Notes in Computer Science* **2259**. Springer–Verlag (2001), 103–111 and MURPHY, S. & ROBSHAW, M.J.B.: Essential Algebraic Structure Within the AES. *Proceedings of Crypto '02. Lecture Notes in Computer Science* **2442**. Springer–Verlag (2002), 1–16 and COURTOIS, N. & PIEPRZYK, J.: Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. *Proceedings of AsiaCrypt '02. Lecture Notes in Computer Science* **2501**. Springer–Verlag (2002), 267–287.

5.4 Operating Modes of AES

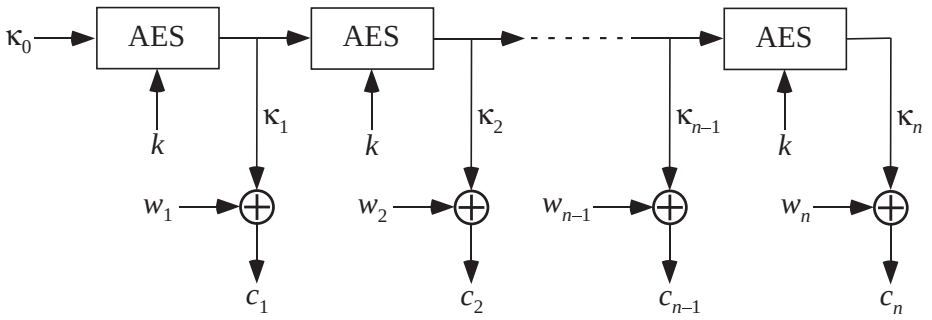
The usual way of using AES is to encrypt one long message block at a time with the same key, the so-called *ECB mode* (electronic codebook).

Another way, the so-called *CBC mode* (cipher block chaining), is to always form a sum of a message block w_i and the preceding cryptoblock c_{i-1} bit by bit modulo 2, i.e. $w_i \oplus c_{i-1}$, and encrypt it, using the same key k all the time. In the beginning we need an initial (crypto)block. Schematically CBC mode is the following operation:

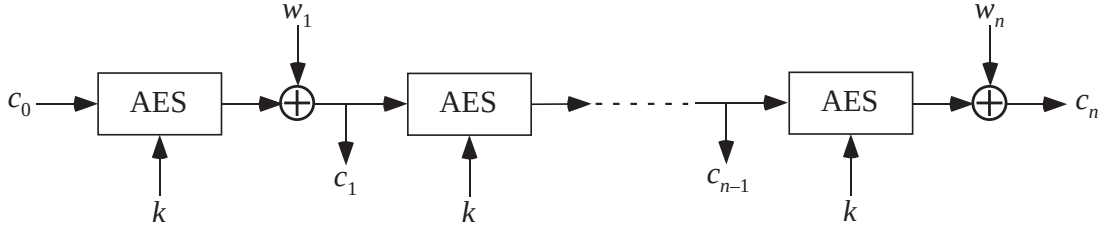


A change in a message block causes changes in the following cryptoblocks in CBC mode. This way CBC mode can be used for *authentication* or the so-called *MAC* (message authentication code) in the following way. The initial block can e.g. be formed of just 0-bits. The sender has a message that is formed of message blocks $w_1, \ldots, w_n$ and he/she computes, using CBC mode, the corresponding cryptoblocks $c_1, \ldots, c_n$ applying a secret key k. The sender sends the message blocks and c_n to the receiver. The receiver also has the key k and he/she can check whether the c_n is valid by using the key.

In the so-called *OFB mode* (output feedback) AES is used to transform the key in a procedure similar to ONE-TIME-PAD encrypting. Starting from a certain "initial key" κ_0 we get a key stream $\kappa_1, \ldots, \kappa_n$ by encrypting this key over and over using AES, κ_1 is obtained by encrypting κ_0. Again, when encrypting we use the same secret key k all the time. Schematically:



OFB mode gives rise to a variant, the so-called *CFB mode* (cipher feedback), where the key κ_i of the key stream is formed by encrypting the preceding cryptoblock. Again κ_1 is obtained by encrypting the initial block c_0.



This variant can be used for authentication much as the CBC-mode, which it also otherwise resembles.

There are also other modes, for example the so-called *CTR mode* (counter mode).

"An algorithm is a finite answer to
an infinite number of questions."

(STEPHEN KLEENE)

Chapter 6

PUBLIC-KEY ENCRYPTION

6.1 Complexity Theory of Algorithms

Computational complexity is about the resources needed for computational solving of a problem versus the size of the problem. Size of the problem is measured by the *length* N of the input, resources are usually *time,* that is, the number of computational steps required, and *space,* that is, the maximum memory capacity needed for the computation. Many problems are so-called *recognition problems* where the solution is a yes-answer. A nice reference concerning classical complexity theory is HOPCROFT & ULLMAN, later results are discussed e.g. in DU & KO.

To make complexity commensurable, we must agree on a mathemathical model for algorithms, for example computing with Turing machines, see the course Theory of Automata, Formal Languages or Mathematical Logic. There is a *deterministic* version of the algorithm model, where the algorithm does not have the possibility to choose, and a *nondeterministic* version, where the next step of the algorithm may be chosen from finitely many possible steps. To be able to say that a nondeterministic algorithm does solve a problem we must make the following assumptions:

- The algorithm stops, no matter what steps are chosen.
- The algorithm can stop in a state, where it has not solved the problem.
- When the algorithm stops in a state where it has solved the problem, then the solution must be correct. The solution is not necessarily unique.
- In recognition problems, a situation where the algorithm does not give any yes-answers is interpreted as a no-answer.
- In problems other than the recognition problems, every input of a nondeterministic algorithm must lead to a solution (output) by some choice of steps.

It is often a good idea to consider a nondeterministic algortihm as a verifying method for a solution, not a method for producing it.

Complexity is mostly examined as asymptotic, in other words, considering sufficiently large problems, and not separating time/space complexities that differ only by a constant multiplier. After all, linear acceleration and space compression are easy in any algorithm model. Although choice of the algorithm model has a clear effect on complexity, it does not have any essential meaning, in other words, it does not change the complexity classes into which problems are divided according to their complexity. Complexity is often given using the O-notation $O(f(N))$,

see Section 2.6. Without going any further into algorithm models, we define a few important complexity classes.

The time complexity class $\mathcal{P}$ (*deterministic-polynomial-time problems*) is composed of the problems, where using a deterministic algorithm solving the problem with input of length N takes a maximum of $p(N)$ steps, and p is a polynomial which depends on the problem. For example, basic computational operations on integers and computing g.c.d. are in $\mathcal{P}$, see Chapter 2.

The time complexity class $\mathcal{NP}$ (*nondeterministic-polynomial-time problems*) is composed of the problems, where using a nondeterministic algorithm solving the problem with input of the length N takes a maximum of $p(N)$ steps, and again p is a polynomial depending on the problem. For example compositeness of integers is in $\mathcal{NP}$: Just guess (nondeterminism!) two factors ($\neq 1$) and check by multiplication whether the guess was correct.

The time complexity class co–$\mathcal{NP}$ (*complementary-nondeterministic-polynomial-time problems*) is formed of those recognition problems that have their complement in $\mathcal{NP}$. The *complement* of a problem is obtained when the yes- and no-answers are interchanged. For example, recognition of primes is in co–$\mathcal{NP}$, since its complement is testing compositeness, which is in $\mathcal{NP}$. It is not very hard to show that primality testing is in $\mathcal{NP}$, but it is much more difficult to show that it is in $\mathcal{P}$, see Section 7.4.

Apparently $\mathcal{P} \subseteq \mathcal{NP}$ and for recognition problems also $\mathcal{P} \subseteq$ co–$\mathcal{NP}$. Is either of these a proper inclusion? This is an open problem and a very famous one! It is commonly believed that both inclusions are proper. Neither is it known whether either of the equations $\mathcal{NP} =$ co–$\mathcal{NP}$ and $\mathcal{P} = \mathcal{NP} \cap$ co–$\mathcal{NP}$ holds for recognition problems. The prevalent belief is that they do not.

The space complexity class $\mathcal{PSPACE}$ (*deterministic-polynomial-space problems*) is formed of those problems, where using a deterministic algorithm solving the problem with input of length of N takes a maximum of $p(N)$ memory units, and p is a polynomial depending on the problem. For example, basic computational operations of integers and computing g.c.d. are in $\mathcal{PSPACE}$.

The space complexity class $\mathcal{NPSPACE}$ (*nondeterministic-polynomial-space problems*) comprises those problems, where using a nondeterministic algorithm solving the problem with input of length N takes a maximum of $p(N)$ memory units, and p is a polynomial, again depending on the problem. It is not very difficult to conclude that

$$\mathcal{NP} \subseteq \mathcal{PSPACE} = \mathcal{NPSPACE},$$

but it is not known whether or not the inclusion is proper.

An algorithm may contain generation of ideal random numbers, which makes it *probabilistic* or *stochastic.* A stochastic algorithm may fail from time to time, in other words, it may not produce a result at all and gives up on solving the problem. Such algorithms are called *Las Vegas algorithms.* On the other hand, a stochastic algorithm may sometimes produce a wrong answer. These algorithms are called *Monte Carlo algorithms.* Note that every Las Vegas algorithm is easily transformed into a Monte Carlo algorithm (how?).

The polynomial time complexity class corresponding to Monte Carlo algorithms is $\mathcal{BPP}$ (*bounded-probability-polynomial-time problems*). In this case the algorithm must produce a correct result with probability at least p, where $p > 1/2$ is a fixed number not depending on the input. The relationship between classes $\mathcal{BPP}$ and $\mathcal{NP}$ is pretty much open—for example it is not known whether one is a subset of the other.

Thinking about the future quantum computers we may define the polynomial time complexity class $\mathcal{BQP}$ (*bounded-error-quantum-polynomial-time problems*). Considering applications

to encrypting, it is interesting to notice that factorization of numbers and computing discrete logarithms belong to this class (the so-called *Shor algorithms,* see Section 15.3).

The function of the algorithm may sometimes be just to convert one problem to another, in this case we are talking about *reduction.* If problem A can be reduced to another problem B using reduction operating in deterministic polynomial time, we get a deterministic polynomial-time algorithm for A from a deterministic polynomial-time algorithm for B.[1] A problem is said to be $\mathcal{NP}$*-hard,* if every problem in $\mathcal{NP}$ can be reduced to it using a deterministic polynomial-time algorithm. An $\mathcal{NP}$-hard problem is $\mathcal{NP}$*-complete,* if it is itself in $\mathcal{NP}$. An $\mathcal{NP}$-complete problem is the "worst kind" of problem in the sense that if it could be shown to be in deterministic polynomial time then every problem in $\mathcal{NP}$ would be in $\mathcal{P}$ and $\mathcal{NP} = \mathcal{P}$. Nowadays over a thousand $\mathcal{NP}$-complete problems are known, and, depending on how they are counted, maybe even more.

Theorem 6.1. *If some $\mathcal{NP}$-complete recognition problem is in $\mathcal{NP} \cap$ co–$\mathcal{NP}$ then for recognition problems $\mathcal{NP} =$ co–$\mathcal{NP}$.*

Proof. Assume that some $\mathcal{NP}$-complete recognition problem C is in $\mathcal{NP} \cap$ co–$\mathcal{NP}$. Now we shall examine an arbitrary recognition problem A in $\mathcal{NP}$. Since C is $\mathcal{NP}$-complete, A can be reduced to C in deterministic polynomial time. Hence the complement of A can be reduced to the complement of C, which is also in $\mathcal{NP}$, in deterministic polynomial time. So A is in co–$\mathcal{NP}$. A was arbitrary and so $\mathcal{NP} \subseteq$ co–$\mathcal{NP}$. As an immediate consequence also co–$\mathcal{NP} \subseteq \mathcal{NP}$, and thus $\mathcal{NP} =$ co–$\mathcal{NP}$. □

Because it is commonly believed that $\mathcal{NP} \neq$ co–$\mathcal{NP}$, no $\mathcal{NP}$-complete recognition problem would thus be in $\mathcal{NP} \cap$ co–$\mathcal{NP}$.

The old division of problems based on computing time is into the practically possible ones (*tractable problems*) and to ones that take too much computing time (*intractable problems*). Problems in $\mathcal{P}$ are tractable and the others are intractable. Since it is a common belief that $\mathcal{NP} \neq \mathcal{P}$, $\mathcal{NP}$-complete problems should be intractable. In practice even problems in the class $\mathcal{BPP}$ are possible to solve: just apply the algorithm on the problem so many times that the probability of half of these producing wrong results is negligible. Hence it is natural to demand in cryptology that encrypting and decrypting functions are in $\mathcal{P}$. It is, however, important to remember that encrypting may include stochastic elements.

6.2 Public-Key Cryptosystems

There are at least two keys in a public-key cryptosystem or nonsymmetric cryptosystem: the public key and the secret key, or several of them. For the secret key to remain a secret it must be computationally very challenging to calculate the secret key starting from the public key. The public key can be left in a "place" where anyone who wants to can take it and use it to send encrypted messages to the owner of the secret key. This seemingly simple idea was first announced by Whitfield Diffie and Martin Hellman and independently by Ralph Merkle in 1976.[2]

[1] Note that even if the output of the polynomial-time reduction is longer than its input, the length of the output is still polynomially bounded by the length of the input, and that composition of two polynomials is a polynomial. A similar phenomenon hardly ever occurs in other function classes. For example, the composition of two exponential functions is not an exponential function.

[2] The original reference is DIFFIE, W. & HELLMAN, M.: New Directions in Cryptography. *IEEE Transactions on Information Theory* **IT–22** (1976), 644–654. It became known later that James Ellis, Clifford Cocks and Malcolm Williamson came up with the same idea a bit earlier, but they worked for the British intelligence organization

It might seem a good idea to arrange the keys so that cryptanalysis using CO data and the public key would be computationally very demanding, e.g. $\mathcal{NP}$-complete. Quite obviously such cryptanalysis is in $\mathcal{NP}$: Just guess the plaintext and encrypt it using the public key. Even if there are stochastic elements in the encrypting this works since the random choices can be guessed, too.

This cryptanalysis problem may also be considered as a recognition problem, the so-called *cryptorecognition:* "Is w the plaintext corresponding to the cryptotext c in the triple (w, k, c) where k is the public key?" Cryptorecognition is in $\mathcal{P}$ if encrypting is deterministic, so making it more complex requires stochastic encrypting. We won't however get very far this way either, because

Theorem 6.2. *If for some cryptosystem cryptorecognition is $\mathcal{NP}$-complete, then $\mathcal{NP} =$ co–$\mathcal{NP}$.*

Proof. The cryptorecognition problem is obviously in $\mathcal{NP}$ since the stochastic parts can be guessed. On the other hand, it is also in co–$\mathcal{NP}$ because if c is a cryptotext then there is just one plaintext corresponding to it, otherwise decrypting won't succeed. Now let's guess some plaintext w' and encrypt it using the public key k. If the result is c then compare w with w', and accept the triple (w, k, c) if $w \neq w'$. If the encrypting of w' does not give c or $w = w'$, the procedure will end without giving a result. So cryptorecognition is in $\mathcal{NP} \cap$ co–$\mathcal{NP}$ and the result follows from Theorem 6.1. □

Hence it would seem that cryptorecognition cannot be $\mathcal{NP}$-complete in practice. The result also shows that stochastic cryptosystems are not that much better than deterministic ones.

Usually when we speak about public-key systems we also mention so-called *one-way functions:* A function $y = f(x)$ is one-way if computing y from x is tractable but computing x from y is intractable, possibly even $\mathcal{NP}$-complete. If the encrypting function of a public-key system is e_k then the function $(c, k) = (e_k(w), k) = f(w, k)$ is ideally one-way. Note that because the public key k is always available, it is included in the value of the function. On the other hand, for a fixed public key k the corresponding secret key gives a so called *trap door* which can be used to compute w from c very fast. Existence of the trap door of course means that the encrypting function is not really one-way for a fixed k.

NB. *Connecting trap doors to $\mathcal{NP}$-complete problems has proved to be difficult. In practice having the trap door restricts an otherwise $\mathcal{NP}$-complete problem to a subproblem that is not $\mathcal{NP}$-complete, and usually not even very demanding. In fact, it has not been proved of any cryptosystem-related function, that should ideally be one-way, that it is really one-way. There is the $\mathcal{P} = \mathcal{NP}$ problem haunting in background, of course. Problems on which good cryptosystems can be based are ones with open complexity statuses. In this case breaking the system would also mean a theoretical breakthrough in complexity theory and algorithm development. All this, and also Theorem 6.2, means that complexity theory does not quite have such an important role in cryptology as it is often given, viz. cryptography is often mentioned as the practical application of complexity theory 'par excellence'.*

Protocols which cannot be executed by secret-key systems are often possible when public-key cryptosystems are used. As examples we take *verification* and *signature.* If B wants to verify that a message is sent by A, the message must contain information that sufficiently unambiguously specifies A as its sender. In this case the following requirements are natural:

GCHQ (Government Communications Headquarters) which is why their ideas remained classified and were not published until 1997.

(i) Both A and B must be able to protect themselves against fake messages. An outside agent C must not be able to pose as A.

(ii) A must be able to protect herself against B's fake messages, which he claims to be sent and signed by A.

(iii) A must not be able to deny sending a message she in fact did send.

Denote by e_{A} and e_{B} the public-key encrypting functions of A and B, and by d_{A} and d_{B} the corresponding decrypting functions. Here it is assumed that encrypting is deterministic. The procedure is the following:

1. A sends the message w to B in the form $c = e_{\mathrm{B}}(d_{\mathrm{A}}(w))$.

2. B computes $e_{\mathrm{A}}(d_{\mathrm{B}}(c)) = e_{\mathrm{A}}(d_{\mathrm{A}}(w)) = w$. Note that e_{A} and d_{A} are inverse functions.

Conditions (i) and (iii) are satisfied since only A knows d_{A}. There must be some recognizable content of correct type in the message, otherwise the message might be totally meaningless. Condition (ii) is also valid since it would be practically impossible for B to generate the right kind of message because he does not know d_{A}. If the signature is all that matters and not keeping the message safe, it is enough for A to send B the pair $(w, d_{\mathrm{A}}(w))$. This simplest version of verification/signature is vulnerable and there are better protocols, see Chapter 13.

6.3 Rise and Fall of Knapsack Cryptosystems

An example of the effects of the preceding section's complexity considerations is the fate of the well-known public-key system KNAPSACK[3] or the *knapsack system.*

The knapsack system is based on the so-called *knapsack problem.* Its input is $(\mathbf{a}, m)$ where $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ is a vector of positive integers and m is a positive integer, represented in some base. The problem is to write m as a sum of (some of) the components of $\mathbf{a}$, or then state that this is not possible. In other words, the problem is to choose bits $c_1, c_2, \ldots, c_n$ such that

$$\sum_{i=1}^{n} c_i a_i = m,$$

or then state that this is not possible at all. In the corresponding recognition problem it is sufficient just to state whether or not the choice is possible. The knapsack problem is clearly in $\mathcal{NP}$: Just guess $c_1, c_2, \ldots, c_n$ and test whether the guess is correct. It is in fact known to be $\mathcal{NP}$-complete.

KNAPSACK-encrypting is done in the following way. The message symbols are bits and the length of the message block is n. A message block $w = b_1 b_2 \cdots b_n$ (bit sequence) is encrypted as the number

$$c = e_k(w) = \sum_{i=1}^{n} b_i a_i.$$

The public key k is $\mathbf{a}$. Apparently this kind of encrypting is in $\mathcal{P}$. Cryptanalysis starting from c and $\mathbf{a}$ is $\mathcal{NP}$-complete.

[3]KNAPSACK is "historically" remarkable as it is one of the first public-key crypto systems, the original reference is MERKLE, R. & HELLMAN, M.: Hiding Information and Signatures in Trapdoor Knapsacks. *IEEE Transactions in Information Theory* **IT–24** (1978), 525–530.

Without any help KNAPSACK decrypting would also be $\mathcal{NP}$-complete. The trap door is gotten by starting from some simple knapsack problem which can be solved in $\mathcal{P}$, and then disguising it as an ordinary arbitrary knapsack problem. The $\mathbf{a}$ of the latter knapsack problem is then published as the public key. Using the trap door information the knapsack problem $(\mathbf{a}, c)$ can be restored to its original easily solved form, and in this way the encrypted message can be decrypted. But this does not lead to a strong cryptosystem, in other words, by using the trap door we don't obtain a disguised knapsack system, whose cryptanalysis would be $\mathcal{NP}$-complete, or even very difficult. In fact different variants of KNAPSACK have been noticed to be dangerously weak and so they are not used anymore. A well-known attack against basic KNAPSACK is the so-called *Shamir attack,* see e.g. SALOMAA.

6.4 Problems Suitable for Public-Key Encryption

As the knapsack problem, the types of problems found useful in public-key encryption are usually problems of number theory or algebra, often originally of merely theoretical interest and quite abstract. This has brought many problems that earlier were considered to be purely mathematical to serve as bases of practical cryptosystems. In particular, results of algebraic number theory and theory of algebraic curves have become concretely and widely used, to the amazement of mathematicians who believed they were working in a very theoretical and "useless" field.

Some examples:

Cryptosystem	Problem type
RSA, RABIN	Factoring the product of two large primes.
ELGAMAL, DIFFIE–HELLMAN, XTR	Computing discrete logarithm in a cyclic group
MENEZES–VANSTONE, CRANDALL	Computing logarithm in a cyclic group determined by an elliptic curve
ARITHMETICA	Conjugate problem in a group
NTRU	Finding the smallest vector of a number lattice
MCELIECE, NIEDERREITER	Decoding an algebraic-geometric linear code (Goppa's code)

The exact complexity of the first four of these is not known, however the problems are in $\mathcal{NP}$. Finding the smallest vector of a number lattice and decoding a linear code (see the course Coding Theory) are known to be $\mathcal{NP}$-complete problems, so considering NTRU, MCELIECE and NIEDERREITER the situation should be similar to KNAPSACK, which for that matter they distantly resemble. Indeed, some weaknesses are found in these systems.[4] The large size of keys needed in MCELIECE has seriously limited its use. NTRU is however in use, to some extent. The drawback of ARITHMETICA is in the difficulty of finding a suitable group—all choices so far have turned out to be bad in one way or in another.

In the sequel we will discuss the systems RSA, ELGAMAL, DIFFIE–HELLMAN, XTR, MENEZES–VANSTONE and NTRU. A good general presentation can be found e.g. in the book GARRETT.

[4]See for example CANTEAUT, A. & SENDRIER, N.: Cryptanalysis of the Original McEliece Cryptosystem. *Proceedings of AsiaCrypt '98. Lecture Notes in Computer Science* **1514**. Springer–Verlag (2000).

"The largest known prime number is $2^{32582657} - 1$.
I am proud to say that I memorized all its digits – in binary."

(CARL POMERANCE: MAA talk, August 4^{th} 2007)

Chapter 7

NUMBER THEORY. PART 2

7.1 Euler's Function and Euler's Theorem

We return to Euler's function $\phi(m)$, already mentioned in Section 2.4, which gives the count of those numbers x in the interval $1 \leq x \leq m$, for which $\gcd(x, m) = 1$, or the number of reduced residue classes modulo m. Note that $\phi(1) = 1$.

Theorem 7.1. (i) *If p is a prime and $k \geq 1$ then*

$$\phi(p^k) = p^{k-1}(p-1).$$

In particular, $\phi(p) = p - 1$.

(ii) *If* $\gcd(m, n) = 1$ *then*

$$\phi(mn) = \phi(m)\phi(n)$$

*(*multiplicativity of ϕ*).*

Proof. (i) Every p^{th} of the numbers $1, 2, \ldots, p^k$ is divisible by p. Hence there are $p^k - p^k/p = p^{k-1}(p-1)$ numbers that are coprime to p.

(ii) Write the numbers $1, 2, \ldots, mn$ in an array as follows:

$$\begin{array}{ccccc} 1 & 2 & 3 & \cdots & n \\ n+1 & n+2 & n+3 & \cdots & 2n \\ 2n+1 & 2n+2 & 2n+3 & \cdots & 3n \\ \vdots & \vdots & \vdots & & \vdots \\ (m-1)n+1 & (m-1)n+2 & (m-1)n+3 & \cdots & mn \end{array}$$

The cases $n = 1$ and $m = 1$ are trivial so we may assume that $n, m \geq 2$. Numbers in any column are mutually congruent modulo n. On the other hand, by the Corollary of Theorem 2.11 numbers in any column form a residue system modulo m. There are $\phi(n)$ columns with numbers coprime to n. (Remember that if $x \equiv y \mod n$ then $\gcd(x, n) = \gcd(y, n)$.) Each of these columns has $\phi(m)$ numbers coprime to m. These are the numbers coprime to mn, and there are $\phi(m)\phi(n)$ of them. □

Using the factorization

$$x = p_1^{i_1} p_2^{i_2} \cdots p_N^{i_N}$$

(see Theorems 2.2 and 2.6) we obtain, using the theorem,

$$\phi(x) = \phi(p_1^{i_1})\phi(p_2^{i_2}) \cdots \phi(p_N^{i_N}) = p_1^{i_1-1} p_2^{i_2-1} \cdots p_N^{i_N-1}(p_1 - 1)(p_2 - 1) \cdots (p_N - 1).$$

Because factorization is a computationally demanding operation, $\phi(x)$ is not practically computable in this way unless the factorization is given beforehand. However, we can see from this fairly easily that if x is a composite number, then $\phi(x) < x - 1$, and that $\phi(x) \geq \sqrt{x}$ when $x > 6$.

An essential result e.g. in defining the cryptosystem RSA is

Theorem 7.2. (Euler's theorem) *If* $\gcd(x, m) = 1$ *then*

$$x^{\phi(m)} \equiv 1 \mod m.$$

Proof. Choose the reduced residue system $j_1, j_2, \ldots, j_{\phi(m)}$ from the positive residue system modulo m. Then the numbers $xj_1, xj_2, \ldots, xj_{\phi(m)}$ also form a reduced residue system since by the Corollary of Theorem 2.11 they are not congruent and are all coprime to m. So, the numbers $xj_1, xj_2, \ldots, xj_{\phi(m)}$ and $j_1, j_2, \ldots, j_{\phi(m)}$ are pairwise congruent in some order:

$$xj_k \equiv j_{i_k} \mod m \quad (k = 1, 2, \ldots, \phi(m)).$$

By multiplying both sides of these congruences we obtain

$$x^{\phi(m)} j_1 j_2 \cdots j_{\phi(m)} \equiv j_1 j_2 \cdots j_{\phi(m)} \mod m$$

and since $\gcd(j_1 j_2 \cdots j_{\phi(m)}, m) = 1$, by dividing out $j_1 j_2 \cdots j_{\phi(m)}$, further $x^{\phi(m)} \equiv 1 \mod m$. □

As an immediate consequence we get

Theorem 7.3. (Fermat's little theorem) *If* p *is a prime and* x *is not divisible by* p *then*

$$x^{p-1} \equiv 1 \mod p.$$

Euler's theorem is often useful when we compute powers modulo m. In addition to using the algorithm of Russian peasants, we first reduce the exponent modulo $\phi(m)$. If $k = q\phi(m) + r$ (division) then

$$x^k = x^{q\phi(m)+r} = (x^{\phi(m)})^q x^r \equiv 1^q \cdot x^r = x^r \mod m.$$

Furthermore, it is immediately noticed that

$$x^{-1} \equiv x^{\phi(m)-1} \mod m$$

and that if $k \equiv l \mod \phi(m)$ then $x^k \equiv x^l \mod m$. (Assuming of course all the time that $\gcd(x, m) = 1$.) Fermat's little theorem is especially useful when computing powers modulo a prime. For instance, if p is prime then always

$$x^p \equiv x \mod p.$$

7.2 Order and Discrete Logarithm

The smallest number $i \geq 1$ (if one exists) such that $x^i \equiv 1 \mod m$, is called the *order* of x modulo m. Basic properties of order are the following:

Theorem 7.4. (i) *The order exists exactly when* $\gcd(x, m) = 1$.

(ii) *If* $x^j \equiv 1 \mod m$ *and the order of* x *modulo* m *is* i *then* i *divides* j. *In particular, as a consequence of Euler's theorem,* i *divides* $\phi(m)$.

(iii) *If the order of x modulo m is i then the order of x^j modulo m is*

$$\frac{\operatorname{lcm}(i,j)}{j} = \frac{i}{\gcd(i,j)}$$

(see Theorem 2.9).

(iv) *If the order of x modulo m is i and the order of y modulo m is j and $\gcd(i,j) = 1$ then the order of xy modulo m is ij.*

Proof. (i) When $\gcd(x,m) = 1$ then at least $x^{\phi(m)} \equiv 1 \mod m$ (Euler's theorem). On the other hand, if $\gcd(x,m) \neq 1$ then obviously also $\gcd(x^i,m) \neq 1$, and hence $x^i \not\equiv 1 \mod m$ when $i \geq 1$.

(ii) If $x^j \equiv 1 \mod m$ but the order i of x does not divide j then $j = qi + r$ where $1 \leq r < i$ (division) and

$$x^r = x^r \cdot 1^q \equiv x^r (x^i)^q = x^{qi+r} = x^j \equiv 1 \mod m,$$

and i would not be the smallest possible.

(iii) If the order of x modulo m is i and the order of x^j modulo m is l then first of all $i \mid jl$ (item (ii)) and $j \mid jl$, so $\operatorname{lcm}(i,j) \mid jl$, i.e. $\operatorname{lcm}(i,j)/j$ is a factor of l. Secondly, $(x^j)^{\operatorname{lcm}(i,j)/j} \equiv 1 \mod m$, so l divides $\operatorname{lcm}(i,j)/j$ (item (ii) again). Therefore $l = \operatorname{lcm}(i,j)/j$.

(iv) If the order of x modulo m is i and the order y modulo m is j and $\gcd(i,j) = 1$ then first of all

$$(xy)^i = x^i y^i \equiv y^i \mod m,$$

so the order of $(xy)^i$ modulo m is the same as the order of y^i, which is j (item (iii)). But if the order of xy modulo m is k then the order of $(xy)^i$ modulo m is $k/\gcd(i,k)$ (item (iii) again). Hence $j \mid k$. It is shown similarly that $i \mid k$. Because $\gcd(i,j) = 1$, it must be that $ij \mid k$. On the other hand,

$$(xy)^{ij} = (x^i)^j (y^j)^i \equiv 1 \mod m,$$

whence it follows that $k \mid ij$ (item (ii)). Therefore $k = ij$. □

If the order of g modulo m is the largest possible, i.e. $\phi(m)$, and $1 \leq g < m$ then g is a so-called *primitive root* of m or a *primitive root modulo* m. Of course, in this case necessarily $\gcd(g,m) = 1$. Since then the powers

$$1, g, g^2, \ldots, g^{\phi(m)-1}$$

are not congruent—otherwise the smaller power could be divided out from the congruence and a lower order for g would be obtained—and there are $\phi(m)$ of them, they actually form a reduced residue system. The following property of primitive roots is given without proof.[1].

Theorem 7.5. *A number $m \geq 2$ has primitive roots if and only if it is either 2 or 4 or of the form p^k or $2p^k$ where p is an odd prime. In particular, every prime has primitive roots.*

[1]The proof is not very difficult but quite long—the cases $m = 2$ and $m = 4$ are of course trivial. It can be found in almost every elementary number theory book, see for example SIERPINSKI. Some cryptology books contain this proof as well, see for example KRANAKIS or GARRETT.

On the other hand, it is easy to deduce the number of different primitive roots, when they exist:

Theorem 7.6. *If there are primitive roots modulo m then there are $\phi(\phi(m))$ of them.*[2] *In particular, a prime p has $\phi(p-1)$ primitive roots.*

Proof. If g is a primitive root of m then those numbers

$$(g^i, \bmod m) \quad (i = 1, 2, \ldots, \phi(m) - 1)$$

for which $\gcd(i, \phi(m)) = 1$ are primitive roots of m, and in fact exactly all of them (Theorem 7.4 (iii)). Hence, if the number m has primitive roots at all, there are $\phi(\phi(m))$ of them. □

The following well-known characterization of primes is obtained immediately from the above.

Theorem 7.7. (Lucas' criterion for primality) *A number $p \geq 2$ is a prime if and only if there exists a number whose order modulo p is $p-1$.*

Proof. If p is prime, it has a primitive root of order $p-1$.

Then again, if there exists a number x of order $p-1$ modulo p then p must be prime. Otherwise $\phi(p) < p-1$ and hence the order of x cannot be $p-1$ because $p-1 \mid \phi(p)$ (Theorem 7.4 (ii)). □

It might be mentioned that no powerful general algorithms are known for finding primitive roots, not even for primes. On the other hand, if the factors of $\phi(m)$ are known then the following result gives a useful test for a primitive root of m. Such a test is needed e.g. in setting up certain cryptosystems, see Section 10.1. In the general case even computing $\phi(m)$ is a very demanding task for large values of m, not to mention its factorization.

Theorem 7.8. (Lucas's criterion for primitive root) *A number $1 \leq g < m$ is a primitive root of m if and only if $\gcd(g, m) = 1$ and $g^{\phi(m)/q} \not\equiv 1 \mod m$ for every prime factor q of $\phi(m)$.*

Proof. If g is a primitive root of m then apparently $\gcd(g, m) = 1$ and $g^{\phi(m)/q} \not\equiv 1 \mod m$ for every prime factor q of $\phi(m)$, since the order of g is $\phi(m)$.

Then again, if $\gcd(g, m) = 1$ and $g^{\phi(m)/q} \not\equiv 1 \mod m$ for every prime factor q of $\phi(m)$, the order i of g divides $\phi(m)$ (Theorem 7.4 (ii)), in other words, $\phi(m) = il$. If $l = 1$ then $i = \phi(m)$ and g is a primitive root. Anything else is out of the question, since if $l > 1$ then l would have a prime factor q' and $l = q't$ and

$$g^{\phi(m)/q'} = g^{il/q'} = g^{it} = (g^i)^t \equiv 1^t = 1 \mod m.$$ □

Furthermore, combining these two Lucas' criteria we obtain

Theorem 7.9. (Lucas–Lehmer criterion for primality) *A number $p \geq 2$ is a prime if and only if there exists a number g such that $g^{p-1} \equiv 1 \mod p$ and $g^{(p-1)/q} \not\equiv 1 \mod p$ for every prime factor q of $p-1$.*

Proof. If p is a prime then we take a primitive root modulo m as g.

Now let's assume that for a number g we have $g^{p-1} \equiv 1 \mod p$ and $g^{(p-1)/q} \not\equiv 1 \mod p$ for every prime factor q of $p-1$. Then $p \mid g^{p-1} - 1$, so $\gcd(g, p) = 1$. Further, if j is the order of g modulo p then $j \mid p-1$ (Theorem 7.4. (ii)). Now we conclude, just as in the preceding proof, that $j = p-1$ and further, by Lucas' criterion, that p is a prime. □

[2] This is the reason why the odd-looking expression $\phi(\phi(m))$ appears in cryptography here and there.

Because, for a primitive root g of m, the numbers $1, g, g^2, \ldots, g^{\phi(m)-1}$ form a reduced residue system modulo m, then for every number x coprime to m there exists exactly one exponent in the interval $0 \leq y < \phi(m)$ for which $g^y \equiv x \mod m$. This exponent is called the *discrete logarithm* or the *index* of x modulo m in base g. No efficient algorithms for calculating discrete logarithms are known, e.g. the cryptosystem ELGAMAL is based on this. We get back to this later. There is of course a nondeterministic polynomial-time algorithm starting from the input (m, g, x): First just guess an index y and then check whether it is correct. Exponentiation using the algorithm of Russian peasants and reducing the result modulo m is in polynomial time.

7.3 Chinese Remainder Theorem

If factors of the modulus m are known, i.e. we can write

$$m = m_1 m_2 \cdots m_k,$$

the congruences $x \equiv y \mod m_i$ $(i = 1, 2, \ldots, k)$ naturally follow from $x \equiv y \mod m$. If the modulus is a large number, it may often be easier to compute using these smaller moduli. This can be done very generally, if the factors $m_1, m_2, \ldots, m_k$ are pairwise coprime, in other words, if $\gcd(m_i, m_j) = 1$ when $i \neq j$:

Theorem 7.10. (Chinese remainder theorem[3]) *If the numbers $y_1, y_2, \ldots, y_k$ are given and the moduli $m_1, m_2, \ldots, m_k$ are pairwise coprime then there is a unique integer x modulo $m_1 m_2 \cdots m_k$ that satisfies the k congruences*

$$x \equiv y_i \mod m_i \quad (i = 1, 2, \ldots, k).$$

Proof. Denote $M = m_1 m_2 \cdots m_k$ and $M_i = M/m_i$ $(i = 1, 2, \ldots, k)$. Since the m_i's are pairwise coprime, $\gcd(M_1, M_2, \ldots, M_k) = 1$ and $\gcd(m_i, M_i) = 1$ $(i = 1, 2, \ldots, k)$. The following procedure produces a solution x (if there is one!), and also shows that the solution is unique modulo M:

1. CRT algorithm:

1. Using the Euclidean algorithm we write $\gcd(M_1, M_2, \ldots, M_k) = 1$ in Bézout's form (see Theorem 2.8)
$$1 = c_1 M_1 + c_2 M_2 + \cdots + c_k M_k.$$
2. Return $x \equiv c_1 M_1 y_1 + c_2 M_2 y_2 + \cdots + c_k M_k y_k \mod M$, e.g. in the positive residue system.

The procedure works if a solution exists, because it follows immediately from the congruences $x \equiv y_i \mod m_i$ that $c_i M_i x \equiv c_i M_i y_i \mod M$ $(i = 1, 2, \ldots, k)$, and by addition we obtain further

$$x = 1 \cdot x = (c_1 M_1 + c_2 M_2 + \cdots + c_k M_k)x \equiv c_1 M_1 y_1 + c_2 M_2 y_2 + \cdots + c_k M_k y_k \mod M.$$

[3]The name "Chinese remainder theorem" (CRT) comes from the fact that Chinese mathematicians knew this result a long time ago, at least in the case $k = 2$.

It still must be shown that a solution exists. Because apparently $M_i \equiv 0 \mod m_j$ if $i \neq j$, and on the other hand $1 = c_1M_1 + c_2M_2 + \cdots + c_kM_k$, we have $c_iM_i \equiv 1 \mod m_i$ ($i = 1, 2, \ldots, k$). Therefore

$$x \equiv c_1M_1y_1 + c_2M_2y_2 + \cdots + c_kM_ky_k \equiv y_i \mod m_i \quad (i = 1, 2, \ldots, k).$$

Because now $c_i \equiv M_i^{-1} \mod m_i$, we can moreover conclude that the solution can also be obtained in another way:

2. CRT algorithm:

1. Compute $N_i \equiv M_i^{-1} \mod m_i$ ($i = 1, 2, \ldots, k$) by the Euclidean algorithm.
2. Return $x \equiv y_1M_1N_1 + y_2M_2N_2 + \cdots + y_kM_kN_k \mod M$ (in the positive residue system).

□

The proof gives an algorithm (actually two of them) for finding the number x mentioned in the theorem. Apparently this algorithm is polynomial-time when the input consists of the numbers $y_1, y_2, \ldots, y_k$ and $m_1, m_2, \ldots, m_k$. Other algorithms are known, for example the so-called *Garner algorithm* which is even faster, see e.g. CRANDALL & POMERANCE.

NB. *In a way the Chinese remainder theorem gives a fitting (interpolation) of functions of the form*

$$y = f_x(m) = (x, \bmod m)$$

through the "points" (m_i, y_i)*, something that can be used in certain cryptoprotocols. The Chinese remainder theorem is very useful in many contexts. A good reference is* DING & PEI & SALOMAA.

7.4 Testing and Generating Primes

It took a long time before the first nondeterministic polynomial-time algorithm for primality testing was found. It is the so-called *Pratt algorithm.*[4] The algorithm is based on Lucas' criteria. The input is a number $n \geq 2$ whose binary length is N. Denote the number of the steps of the algorithm by $T(n)$ and

$$\mathrm{PRATT}(n) = \begin{cases} \text{YES if } n \text{ is prime} \\ \text{FAIL if the test does not produce a result with the choices made.} \end{cases}$$

From Section 6.1 we recall that if the algorithm works then the input n is a composite number if and only if $\mathrm{PRATT}(n) = \mathrm{FAIL}$ for every possible choice.

Pratt's algorithm:

1. If $n = 2$ or $n = 3$, return YES and quit (0 test steps).
2. If n is > 3 and even (division by 2), the algorithm gives up and $\mathrm{PRATT}(n) = \mathrm{FAIL}$ (0 test steps).

[4]The original reference is PRATT, V.R.: Every Prime has a Succint Certificate. *SIAM Journal on Computing* **4** (1976), 198–221.

3. Guess (nondeterminism) an integer x in the interval $1 \leq x \leq n-1$.

4. Check whether $x^{n-1} \equiv 1 \mod n$ using the algorithm of Russian peasants and reducing modulo n by divisions (1 test step). If this is not so then the algorithm gives up and $\mathrm{PRATT}(n) = \mathrm{FAIL}$.

5. Guess (nondeterminism) prime factors $p_1, \ldots, p_k$ of $n-1$, where each assumed prime factor may occur several times (0 test steps). Lengths of these numbers in the binary representation are $P_1, \ldots, P_k$. Note that $P_1 + \cdots + P_k \leq N + k - 1$ and that $2 \leq k \leq N$.

6. Check by multiplication whether $p_1 \cdots p_k = n-1$ (1 test step). If this is not so, the algorithm gives up and $\mathrm{PRATT}(n) = \mathrm{FAIL}$.

7. Check, by calling Pratt's algorithm recursively, whether the numbers $p_1, \ldots, p_k$ are truly primes (a maximum of $T(p_1) + \cdots + T(p_k)$ test steps). If some $\mathrm{PRATT}(p_i) = \mathrm{FAIL}$ then the algorithm gives up and $\mathrm{PRATT}(n) = \mathrm{FAIL}$.

8. Check whether $x^{(n-1)/p_i} \not\equiv 1 \mod n$ $(i = 1, \ldots, k)$ by the algorithm of Russian peasants and divisions (a maximum of k test steps). If this is true, return YES, otherwise the algorithm gives up and $\mathrm{PRATT}(n) = \mathrm{FAIL}$.

Now we get following recursion inequality for $T(n)$:

$$T(n) \leq 2 + k + \sum_{i=1}^{k} T(p_i)\,, \quad T(2) = 0\,, \quad T(3) = 0.$$

Using this we can find an upper bound for $T(n)$. It is easy to see recursively that for example $L(n) = 4\log_2 n - 4$ is such an upper bound, since $L(2) = 0$ and $L(3) > 0$ and

$$\begin{aligned} T(n) &\leq 2 + k + \sum_{i=1}^{k} L(p_i) = 2 + k + \sum_{i=1}^{k} (4\log_2 p_i - 4) \\ &= 2 + k + 4\log_2(p_1 \cdots p_k) - 4k = 2 - 3k + 4\log_2(n-1) \\ &< -4 + 4\log_2 n = L(n). \end{aligned}$$

On the other hand, it takes $O(N^3)$ steps to perform each test step (there are better estimates) and $L(n)$ is proportional to N (Theorem 2.4). So, the overall time is $O(N^4)$.

In the "old aristocracy" of primality testing are the *Adleman–Pomerance–Rumely test*[5] and its variants. The test is based on some quite advanced algebraic number theory, it is deterministic and fast. Testing a number n for primality takes at most

$$O((\ln n)^{c \ln(\ln(\ln n))})$$

steps where c is (small) constant, and hence it is not quite in $\mathcal{P}$—but almost, since the $\ln(\ln(\ln n))$ does grow very slowly. On the other hand, both theoretically and considering implementation, it is hard to handle. See for example KRANAKIS.

A recent celebrated result in number theory is the fact that primality testing is in $\mathcal{P}$. This was proved by the Indians Manindra Agrawal, Neeraj Kayal and Nitin Saxena in 2002.[6] The proved complexity of the algorithm is $O((\ln n)^8)$ but heuristically a complexity $O((\ln n)^6)$ is obtained. However, as of yet there are no very fast implementations, although the algorithm is quite short to present (the input is $n \geq 2$):

[5]The original reference is ADLEMAN, L. & POMERANCE, C. & RUMELY, R.: On Distinguishing Prime Numbers from Composite Numbers. *Annals of Mathematics* **117** (1983), 173–206.

[6]The article reference is AGRAWAL, M. & KAYAL, N. & SAXENA, N.: PRIMES is in P. *Annals of Mathematics* **160** (2004), 781–793.

Agrawal–Kayal–Saxena algorithm:

1. Find out whether n is a higher power of an integer r, in other words, whether it can be expressed as $n = r^l$ where $l \geq 2$. (Because then $l = \log_2 n / \log_2 r \leq \log_2 n$, the number of possible values of l we must try out is proportional to the length of n. After finding these we compute the integral l^{th} root of n for every candidate l using Newton's algorithm from Section 2.6 and see if its l^{th} power is $= n$.) If n is such a power, return "NO" and quit.

2. Find an integer m such that the order of n modulo m is $> (\log_2 n)^2$. (This can be done by trying out numbers. A much more difficult thing is to show that such an m need not be too large.)

3. Check whether n has a prime factor in the interval $2, 3, \ldots, m$ (perhaps by trying out numbers and using the Euclidean algorithm). If it has, return "NO" and quit.

4. Examine whether the congruences

$$(x + \overline{i}\,)^n \equiv x^n + \overline{i} \mod x^m - \overline{1} \quad (i = 1, 2, \ldots, \lfloor \sqrt{m} \log_2 n \rfloor)$$

   hold in the polynomial ring $\mathbb{Z}_n[x]$. (For this we need the algorithm of Russian peasants and divisions. Note that regardless of the value of n division by the monic polynomial $x^m - \overline{1}$ is defined in $\mathbb{Z}_n[x]$. See Section 4.2.) If they do not all hold true, return "NO" and quit.

5. Return "YES" and quit.

A nice exposition of the algorithm and its working is in the article GRANVILLE, A.: It Is Easy to Determine Whether a Given Integer Is Prime. *Bulletin of the American Mathematical Society* **42** (New Series) (2004), 3–38.

Some very useful primality tests are probabilistic, in other words, they produce the correct result with high probability. For example the so-called *Miller–Rabin test*[7] is such a test. The test is based on Fermat's little theorem, according to which, if n is a prime and x is an integer such that $\gcd(x, n) = 1$ then $x^{n-1} \equiv 1 \mod n$. Let's write n in the form

$$n = 1 + 2^l m,$$

where m is odd. If n is odd then $l \geq 1$ and

$$0 \equiv x^{n-1} - 1 = x^{2^l m} - 1 = (x^{2^{l-1} m} - 1)(x^{2^{l-1} m} + 1) \mod n,$$

and because n is a prime it divides either $x^{2^{l-1} m} - 1$ or $x^{2^{l-1} m} + 1$, but not both of them (why?). If n divides $x^{2^{l-1} m} - 1$ then we can go through the same operation again. And so on. From this we conclude that either for some number $i = 0, 1, \ldots, l - 1$ we have

$$x^{2^i m} \equiv -1 \mod n,$$

or if this is not true, eventually

$$x^m \equiv 1 \mod n.$$

[7] The original references are MILLER, G.L.: Riemann's Hypothesis and Tests for Primality. *Journal of Computer and System Sciences* **13** (1976), 300–317 and RABIN, M.O.: Probability Algorithms. *Algorithms and Complexity* (J.F. TRAUB, Ed.). Academic Press (1976), 35–36. The algorithm is sometimes also known as *Selfridge's test.*

If it now happens for an integer x such that $\gcd(x, n) = 1$ and $x^m \not\equiv \pm 1 \mod n$, that for all numbers $i = 1, 2, \ldots, l-1$

$$x^{2^i m} \equiv 1 \mod n$$

then we can only conclude that n is not a prime after all. Similarly if we run into an $i > 0$ such that $x^{2^i m} \not\equiv \pm 1 \mod n$. On the other hand, when we try out several numbers, for example certain "small" primes $x = 2, 3, 5, 7, 11, \ldots$, we obtain evidence of a kind for the primality of n. As a matter of fact, this evidence can be made very strong by using several well-chosen numbers x. This is so also in a probabilistic sense, with a random choice of the number x in the interval $1 < x < n - 1$.

In the following it is assumed that given or randomly chosen test numbers $x_1, x_2, \ldots, x_k$ are available.

Miller–Rabin primality test:

1. If n is even, the case is clear, return the result and quit.
2. If n is odd, set $l \leftarrow 0$ and $m \leftarrow n - 1$.
3. Set $l \leftarrow l + 1$ and $m \leftarrow m/2$.
4. If m is even, go to #3. (The maximum number of these rounds is $\lfloor \log_2 n \rfloor$.)
5. Set $j \leftarrow 0$.
6. If $j < k$, set $j \leftarrow j + 1$ and $x \leftarrow x_j$. Otherwise return "PRIME" (supposed information) and quit.
7. If $x^m \equiv 1 \mod n$ or $\gcd(x, n) = n$ then go to #6. Then again, if $1 < \gcd(x, n) < n$, return "COMPOSITE" (certain information) and quit. (Compute powers using the algorithm of Russian peasants, the g.c.d. using the Euclidean algorithm.)
8. Set $i \leftarrow 0$.
9. If $x^{2^i m} \equiv 1 \mod n$, return "COMPOSITE" (certain information) and quit. (Compute powers by repeated squarings starting from the power in #7, be sure to keep the intermediate results!)
10. If $x^{2^i m} \equiv -1 \mod n$, go to #6.
11. If $i = l-1$, return "COMPOSITE" (certain information) and quit. Otherwise set $i \leftarrow i+1$ and go to #9.

NB. *This is the so-called "bottom-up" version of the test. There is also a "top-down" version, where i is decreased, see e.g. the lecture notes* RUOHONEN, K.: Symbolinen analyysi. *There appears to be no significant difference in speed between these two versions.*

So, the test is not "rock-solid". There are composite numbers that it returns as primes, these are called *strong pseudoprimes* for the test numbers $x_1, x_2, \ldots, x_k$. For example, $25\,326\,001 = 2\,251 \cdot 11\,251$ is a strong pseudoprime for the test numbers 3 and 5. For a fixed value of k the time complexity of the test is $O(N^3)$, as it is easy to see (again N is the length of n). As a probabilistic algorithm the Miller–Rabin test is of the Monte Carlo type. It can be shown that for a single randomly chosen x from the interval $1 < x < n - 1$ the test produces the wrong result with a probability no higher than $1/4$, see the original reference RABIN or e.g. CRANDALL &

POMERANCE or KRANAKIS or GARRETT. By repeating the test we get a certainty as good as we want.[8]

Besides primality testing, generating primes of a given length is an essential task. A prime of length N can be chosen randomly by first choosing a random integer of length N, see Section 2.6, and then testing it for primality by the Miller–Rabin test. This prime generation is quite fast. If we denote by $\pi(x)$ the number of the primes less than or equal to x, we get a famous asymptotic estimate:

Theorem 7.11. (Prime number theorem) $\lim_{x\to\infty} \dfrac{\pi(x)}{x/\ln x} = 1$

The proof is difficult! Hence, of the numbers of magnitude n approximately one in every $\ln n$ is a prime. This is enough for random search of primes to go quickly. The random number generators of Section 2.6 are good enough for this purpose. An older result

Theorem 7.12. (Chebychev's theorem) $\dfrac{7}{8} < \dfrac{\pi(x)}{x/\ln x} < \dfrac{9}{8}$ *when* $x \geq 5$.

gives rough quantitative bounds. It guarantees that there are at least

$$\left\lceil \frac{7n}{8\ln n} \right\rceil$$

primes among the numbers $1, 2, \ldots, n$, and that in the interval $(m, n]$ there are at least

$$\left\lceil \frac{7n}{8\ln n} \right\rceil - \left\lfloor \frac{9m}{8\ln m} \right\rfloor$$

primes. For example, in the interval $(10^{150}, 10^{151}]$ there are thus at least something like

$$\frac{7\cdot 10^{151}}{1\,208\ln 10} - \frac{9\cdot 10^{150}}{1\,200\ln 10} \cong 2.19\cdot 10^{148}$$

primes, much more actually. Primes also occur fairly uniformly:

Theorem 7.13. (Bertrand's postulate[9]) *When* $n \geq 2$*, there is at least one prime* p *in the interval* $n < p < 2n$.

Theorem 7.14. (Dirichlet–de la Vallée-Poussin theorem) *If* $m \geq 2$ *then primes are distributed asymptotically equally among the reduced residue classes modulo* m.

Primes and primality testing are widely discussed in CRANDALL & POMERANCE.

7.5 Factorization of Integers

From the fact that primality testing is in $\mathcal{P}$ it follows immediately that factorization of integers is in $\mathcal{NP}$: just guess the prime factors and test their primality. Although primality testing is in $\mathcal{P}$ and also quite fast in practice, factorization appears to be a highly demanding task. It is enough to give a method that finds a nontrivial factor d of an integer $n \geq 2$, or then confirms

[8] There are other Monte Carlo type primality tests, for example the so-called *Solovay–Strassen algorithm,* see e.g. SALOMAA or KRANAKIS.

[9] The postulate was actually proved by Chebychev.

that n itself is a prime. After that we can continue recursively from the numbers d and n/d. Of course, we should start with primality testing, after which we may assume that n is not a prime.

The following well-known algorithm often finds a factor for an odd composite number n, assuming that for some prime factor p of n there are no prime powers dividing $p-1$ larger than b. From this condition it follows that $p-1$ is a factor of $b!$ (doesn't it?).

Pollard's $p-1$-algorithm[10]**:**

1. Set $a \leftarrow 2$.
2. Iterate setting $a \leftarrow (a^j, \mathrm{mod}\ n)$ for $j = 2, \ldots, b$.
3. Compute $d = \gcd(a-1, n)$.
4. If $1 < d < n$, return the factor d, otherwise give up.

Assume that p is a prime factor of n which satisfies the given condition. After #2 apparently $a \equiv 2^{b!} \mod n$ and thus also $a \equiv 2^{b!} \mod p$. By Fermat's little theorem $2^{p-1} \equiv 1 \mod p$. As was noted, $p-1 \mid b!$ whence $a \equiv 1 \mod p$. So, $p \mid a-1$ and thus $p \mid d$. It is possible that $a = 1$, though, in which case a factor cannot be found.

The time complexity of the algorithm is

$$O(bBN^2 + N^3)$$

where N and B are the binary lengths of the numbers n and b, respectively. From this it is seen that b should be kept as small as possible compared with n, for the algorithm to work fast. On the other hand, if b is too small, too many prime factors are precluded and the algorithm does not produce a result.

More exact presentation and analysis of Pollard's $p-1$-algorithm and many other algorithms can be found in the references RIESEL and CRANDALL & POMERANCE. Pollard's $p-1$-algorithm has been generalized in many ways, for example to the so-called *method of elliptic curves* and to *Williams' $p+1$-algorithm.*

A very classical algorithm for finding factors is the so-called *test division algorithm.* In this algorithm we first try out factors 2 and 3 and after that factors of form $6k \pm 1$ up to $\lfloor \sqrt{n} \rfloor$. Integral square root can be computed fast, as was noted. Of course this procedure is rather time-consuming. Test division is a so-called *sieve method.* There are much more powerful sieve methods, for instance the *quadratic sieve* and the *number field sieve.* The estimated time complexities for the fastest algorithms at the moment are given in the following table. Shor's algorithm, see Section 15.3, is not included, since quantum computers do not really exist yet.

Algorithm	**Time complexity** *
Quadratic sieve	$O\left(e^{(1+o(1))\sqrt{\ln n \ln(\ln n)}}\right)$
Method of elliptic curves	$O\left(e^{(1+o(1))\sqrt{2 \ln p \ln(\ln p)}}\right)$ (p is the smallest prime factor of n)
Number field sieve	$O\left(e^{(1.92+o(1))(\ln n)^{1/3}(\ln(\ln n))^{2/3}}\right)$

* The notation $f(n) = o(1)$ means that $\lim_{n\to\infty} f(n) = 0$. More generally, the notation $f(n) = o(g(n))$ means that $\lim_{n\to\infty} f(n)/g(n) = 0$.

[10]The original reference is POLLARD, J.M.: Theorems on Factorization and Primality Testing. *Proceedings of the Cambridge Philosophical Society* **76** (1975), 521–528. The algorithm can be varied in a number ways in order to make it more powerful, this is just a basic version.

7.6 Modular Square Root

The number x is called a *square root of y modulo m* or a so-called *modular square root* if

$$x^2 \equiv y \mod m.$$

Usually this square root is represented in the positive residue system. We see immediately that if x is a square root of y modulo m then so is $(-x,\mod m)$. Thus there are usually at least two modular square roots, often many more.

There does not necessarily have to be any square root modulo m. A number y that has square root(s) modulo m is called a *quadratic residue* modulo m, and a number y that has no square roots modulo m is called a *quadratic nonresidue* modulo m. Apparently at least the numbers 0 and 1 are quadratic residues. In the general case testing quadratic residuosity or quadratic nonresiduosity modulo m is a difficult computational task.

If the number y is a quadratic residue modulo m and the factorization of m is

$$m = p_1^{i_1} p_2^{i_2} \cdots p_M^{i_M}$$

and some square root x_j of y modulo $p_j^{i_j}$ $(j = 1, 2, \ldots, M)$ is known, then we can obtain more square roots of y modulo m using the Chinese remainder theorem. Note that if y is a quadratic residue modulo m then it is also quadratic residue modulo every $p_j^{i_j}$, since every square root x of y modulo m is also its square root modulo $p_j^{i_j}$. Solve for x modulo m the congruence system

$$\begin{cases} x \equiv \pm x_1 \mod p_1^{i_1} \\ x \equiv \pm x_2 \mod p_2^{i_2} \\ \quad \vdots \\ x \equiv \pm x_M \mod p_M^{i_M} \end{cases}$$

by using the CRT algorithm. The solution is uniquely determined modulo $m = p_1^{i_1} p_2^{i_2} \cdots p_M^{i_M}$. Any of the 2^M combinations of the signs $\pm$ may be chosen. Then

$$x^2 \equiv (\pm x_j)^2 \equiv y \mod p_j^{i_j}$$

and so $p_j^{i_j} \mid x^2 - y$ $(j = 1, 2, \ldots, M)$. Since the $p_j^{i_j}$ are coprime we have $m \mid x^2 - y$, i.e. $x^2 \equiv y \mod m$. By going through all choices for the square roots x_j—there may well be several of them—and all $\pm$-sign combinations we actually obtain every square root of y modulo m.

So, the situation is reduced to computing square roots modulo primes or prime powers. Computing square roots modulo higher powers of primes is a bit more difficult and it is not discussed here.[11] On the other hand, square roots modulo a prime p can be computed fast by the so-called *Shanks algorithm.* There are always exactly two square roots of y modulo p, unless $y \equiv 0 \mod p$, since if x is a square root and x' is another then

$$x^2 \equiv y \equiv x'^2 \mod p \quad \text{or} \quad (x - x')(x + x') \equiv 0 \mod p$$

and either $p \mid x - x'$, i.e. $x \equiv x' \mod p$, or $p \mid x + x'$, i.e. $x' \equiv -x \mod p$. And if $y \equiv 0 \mod p$, then the only square root is 0, as it is easy to see.

If $p > 2$ then apparently all quadratic residues modulo p are obtained when we take the squares of the numbers $0, 1, \ldots, (p-1)/2$ modulo p. These squares are not congruent modulo p (why?), so there is one more of the quadratic residues than the quadratic nonresidues, and this one quadratic residue is 0. Whether y is a quadratic residue or a quadratic nonresidue modulo p can be decided quickly, the cases $p = 2$ and $y \equiv 0 \mod p$ being of course trivial.

[11] A so-called Hensel lifting, much as the one in Section 11.3, is needed there, see e.g. GARRETT.

Theorem 7.15. (Euler's criterion) *If p is an odd prime and $y \not\equiv 0 \mod p$ then y is a quadratic residue modulo p if and only if*

$$y^{\frac{p-1}{2}} \equiv 1 \mod p.$$

(Modular powers are computed quickly using the algorithm of Russian peasants.)

Proof. If y is a quadratic residue, that is, for some x we have $y \equiv x^2 \mod p$, then by Fermat's little theorem $x^{p-1} \equiv 1 \mod p$ (note that $\gcd(x,p) = 1$ since $y \not\equiv 0 \mod p$). So

$$y^{\frac{p-1}{2}} \equiv x^{p-1} \equiv 1 \mod p.$$

Conversely, if $y^{(p-1)/2} \equiv 1 \mod p$ then we take a primitive root g modulo p. In this case we have $y \equiv g^i \mod p$ for some i because $\gcd(y,p) = 1$, and

$$g^{\frac{p-1}{2}i} \equiv y^{\frac{p-1}{2}} \equiv 1 \mod p.$$

But since the order of g is $p-1$, $(p-1)i/2$ must be divisible by $p-1$. Hence i is even and y has the square roots $(\pm g^{i/2}, \mod p)$ modulo p. □

If p is of the form $p = 4l - 1$, i.e. $p \equiv 3 \mod 4$, then by using Euler's criterion we immediately get those two square roots of y—assuming of course that $y \not\equiv 0 \mod p$. They are $(\pm y^{(p+1)/4}, \mod p)$, since

$$\left(\pm y^{\frac{p+1}{4}}\right)^2 = y^{\frac{p+1}{2}} = y^{\frac{p-1}{2}} y \equiv y \mod p.$$

One of these two modular square roots is actually a quadratic residue itself, this is the so-called *principal square root,* and the other is a quadratic nonresidue. To see this, first of all, if x is both a square root of y and a quadratic residue modulo p then $-x$ cannot be a quadratic residue. Otherwise $x \equiv z_1^2 \equiv -z_2^2 \mod p$ for some numbers z_1 and z_2, and $-1 \equiv (z_1 z_2^{-1})^2 \mod p$, i.e. -1 is a quadratic residue modulo p. However this is not possible by Euler's criterion since $(-1)^{(p-1)/2} = (-1)^{2l-1} = -1$. On the other hand these modular square roots cannot both be quadratic nonresidues, otherwise there will be too many of them.

The case $p = 4l + 1$ is much more complicated, oddly enough, and we need Shanks' algorithm to deal with it.

Before we go to Shanks' algorithm, we can now state that if m does not have higher powers of primes as factors—in other words, m is *square-free*—and the factorization

$$m = p_1 p_2 \cdots p_M$$

is known then the situation concerning quadratic residues and square roots modulo m is quite simple:

- y is a quadratic residue modulo m if and only if it is a quadratic residue modulo each p_j ($j = 1, 2, \ldots, M$), and this is very quickly decided using Euler's criterion.
- After computing the square roots x_j of y modulo p_j using Shanks' algorithm, we obtain all 2^M square roots of y modulo m applying the CRT algorithm as above.

Furthermore we obtain

Theorem 7.16. *If m is odd and square-free, $\gcd(y,m) = 1$, i.e. y is not divisible by any of the primes p_j, and y is a quadratic residue modulo m then there are exactly 2^M square roots of y modulo m where M is the number of prime factors of m.*

Proof. Otherwise for some p_j we have $x_j \equiv -x_j \mod p_j$, i.e. $2x_j \equiv 0 \mod p_j$. Thus, because p_j is odd, $x_j \equiv 0 \mod p_j$ and further $y \equiv x_j^2 \equiv 0 \mod p_j$. □

If the primes p_j are all $\equiv 3 \mod 4$ then exactly one of these 2^M square roots of y modulo m in the theorem is obtained by the CRT algorithm choosing principal square roots of y modulo each p_j. This square root is the *principal square root* of y modulo m.

Corollary. *If m is odd and square-free, y is a quadratic residue modulo m, and x is a square root of y modulo m then the square roots of y modulo m are exactly $(x\omega_i, \mod m)$ ($i = 1, 2, \ldots, 2^M$) where M is the number of prime factors of m and $\omega_1, \omega_2, \ldots, \omega_{2^M}$ are the square roots of 1 modulo m.*

NB. *All this depends very much on the factorization of m being available. Already in the case where $M = 2$ and the factors are not known deciding whether y is quadratic residue modulo m or not, and in the positive case finding its square roots modulo m, is very laborious. Even knowing one of the square root pairs does not help. As a matter of fact, if we know square roots x_1 and x_2 of y modulo $m = p_1p_2$ such that $x_1 \not\equiv \pm x_2 \mod m$ then the numbers $\gcd(m, x_1 \pm x_2)$ are the primes p_1 and p_2. Many cryptosystems and protocols, e.g. RSA, are based on these observations.*

And then the Shanks algorithm:

Shanks' algorithm:

1. If $p = 2$, return $(y, \mod 2)$ and quit. If $y \equiv 0 \mod p$, return 0 and quit.
2. If $y^{(p-1)/2} \not\equiv 1 \mod p$ then y does not have square roots modulo p by Euler's criterion. Return this information and quit.
3. If $p \equiv 3 \mod 4$, return $(\pm y^{(p+1)/4}, \mod p)$ and quit.
4. Then again if $p \equiv 1 \mod 4$, write $p - 1 = 2^s t$ where t is odd and $s \geq 2$. This is accomplished by repeated divisions by 2, and no more than $\lfloor \log_2(p-1) \rfloor$ of them are needed.
5. Randomly choose a number u from the interval $1 \leq u < p$. Now if $u^{(p-1)/2} \equiv 1 \mod p$, give up and quit. By Euler's criterion u is in this case a quadratic residue modulo p and for the sequel a quadratic nonresidue will be needed. Hence the choice of u succeeds with a probability of 50%.
6. Set $v \leftarrow (u^t, \mod p)$. Then the order of v modulo p is 2^s. This is because if i is this order, then $it \mid 2^s t$ and so $i \mid 2^s$. On the other hand, $u^{t2^k} \not\equiv 1 \mod p$ for $k < s$, otherwise $u^{(p-1)/2} \equiv 1 \mod p$.
7. Set $z \leftarrow (y^{(t+1)/2}, \mod p)$. Then $z^2 \equiv y^t y \mod p$. In a sense z is an "approximate" square root of y modulo p, and using it we can find the correct square root in the form $x = (zv^{-l}, \mod p)$.
8. Find the said correct square root, in other words, a number l such that

$$x^2 \equiv (zv^{-l})^2 \equiv y \mod p\,, \quad \text{i.e.} \quad v^{2l} \equiv z^2 y^{-1} \equiv y^t \mod p.$$

Such a number exists because the modular equation $w^{2^{s-1}} \equiv 1 \mod p$ has 2^{s-1} roots[12] (solving for w) and they are $(v^{2j},\mathrm{mod}\ p)$ $(j = 0, 1, \ldots, 2^{s-1} - 1)$. Since $(y^t,\mathrm{mod}\ p)$ is one of the roots, the number l can be found recursively in the binary form

$$l = b_{s-2}2^{s-2} + b_{s-3}2^{s-3} + \cdots + b_1 2 + b_0$$

as follows:

8.1 The bit b_0 is found when both sides of the congruence $v^{2l} \equiv y^t \mod p$ are raised to the $(2^{s-2})^{\text{th}}$ power since

$$b_0 = \begin{cases} 0 \text{ if } (y^{t2^{s-2}}, \mathrm{mod}\ p) = 1 \\ 1 \text{ otherwise.} \end{cases}$$

8.2 The bit b_1 is found, when both sides of the congruence $v^{2l} \equiv y^t \mod p$ are raised to the $(2^{s-3})^{\text{th}}$ power since

$$b_1 = \begin{cases} 0 \text{ if } (y^{t2^{s-3}}v^{-b_0 2^{s-2}}, \mathrm{mod}\ p) = 1 \\ 1 \text{ otherwise.} \end{cases}$$

Note that here we need the already obtained b_0.

8.3 Using the obtained bits b_0 and b_1 we similarly find the following bit b_2, and so on.

9. Return $(\pm zv^{-l},\mathrm{mod}\ p)$ and quit.

It is quite easy to see that the algorithm is polynomial-time and produces the correct result with an approximate probability of 50%. It is a Las Vegas type stochastic algorithm.

7.7 Strong Random Numbers

Cryptologically strong random numbers are needed for example in probabilistic cryptosystems where random numbers are used in the encryption. Encrypting one and the same message can then produce different results at different times. Many protocols also use random numbers.

Many otherwise quite good traditional random number generators, such as the shift register generator introduced in Section 2.6, have proved to be dangerously weak in cryptography. The specific needs of cryptology started an extensive research of pseudorandom numbers, theoretically as well as in practice.

The *Blum–Blum–Shub generator*[13] is a simple random number generator, whose strength is in its connections to quadratic residuosity testing. Since, as of now, no fast algorithms are known for the testing, even probabilistic ones not to mention deterministic, the BBS generator is thought to be strong in the cryptological sense, see e.g. GARRETT or STINSON.

Squaring a quadratic residue x modulo n produces a new quadratic residue y. Now if y has a principal square root, it must be x, and so in this case we are actually talking about permuting quadratic residues. This permutation is so powerfully randomizing that it can be used as a random number generator.

[12] Here we need from polynomial algebra the result that an algebraic equation of d^{th} degree has at most d different roots. See for example the course Algebra 1 or Symbolic Computing or some elementary algebra book.

[13] The original reference is BLUM, L. & BLUM, M. & SHUB, M.: A Simple Unpredictable Random Number Generator. *SIAM Journal on Computing* **15** (1986), 364–383.

The BBS generator produces a sequence of random bits. The generator needs two primes p and q, kept secret, of approximately same length. The condition $p \equiv q \equiv 3 \mod 4$ must be satisfied, too, for the principal square roots to exist. Denote $n = pq$. If the goal is to produce l random bits, the procedure is the following:

Blum–Blum–Shub generator:

1. Choose a random number s_0 from the interval $1 \leq s_0 < n$. Randomness is very important here, and for that the random number generators introduced in Section 2.6 are quite sufficient. Indeed, some choices lead to very short sequences, and the random number generator starts repeating itself quite soon, which is of course a serious deficiency. This is discussed thoroughly in the original article.

2. Repeat the recursion
$$s_i = (s_{i-1}^2, \mod n)$$
l times and compute the bits
$$b_i = (s_i, \mod 2) \quad (i = 1, 2, \ldots, l).$$

3. Return $(b_1, b_2, \ldots, b_l)$ and quit.

NB. *Cryptologically strong random number generators and good cryptosystems have a lot in common, as a matter of fact, many cryptosystems can be transformed to cryptologically strong random number generators, see e.g.* GOLDREICH *and* SHPARLINSKI *and the article* AIELLO, W. & RAJAGOPALAN, S.R. & VENKATESAN, R.: Design of Practical and Provably Good Random Number Generators. *Journal of Algorithms* **29** (1998), 358–389.

7.8 Lattices. LLL Algorithm

If $\mathbf{v}_1, \ldots, \mathbf{v}_k$ are linearly independent vectors of $\mathbb{R}^k$ then the *lattice*[14] generated by them is the set of the points
$$\langle \mathbf{v}_1, \ldots, \mathbf{v}_k \rangle = \{c_1\mathbf{v}_1 + \cdots + c_k\mathbf{v}_k \mid c_1, \ldots, c_k \in \mathbb{Z}\}$$
of $\mathbb{R}^k$. The vectors $\mathbf{v}_1, \ldots, \mathbf{v}_k$ are called the *base vectors* or the *basis* of the lattice, and k is the *dimension* of the lattice. A lattice has infinitely many bases if $k > 1$. So, a central task considering lattices is to find a "good" basis which includes at least one short vector and whose vectors do not meet at very sharp angles. Such a basis resembles the natural basis of $\mathbb{R}^k$.

The *discriminant* of the lattice is $D = |\det(\mathbf{V})|$ where $\mathbf{V}$ is the matrix whose columns are $\mathbf{v}_1, \ldots, \mathbf{v}_k$. D is the volume of the k-dimensional parallelepiped spanned by the base vectors, and does not depend on the choice of the basis of the lattice. This is because a matrix $\mathbf{C}$, used for changing the basis, and its inverse $\mathbf{C}^{-1}$ must have integral elements, in which case both $\det(\mathbf{C})$ and $\det(\mathbf{C}^{-1}) = \det(\mathbf{C})^{-1}$ are also integers and hence $\det(\mathbf{C}) = \pm 1$. After the change of basis the discriminant is $|\det(\mathbf{CV})| = |\det(\mathbf{C})\det(\mathbf{V})| = D$. The discriminant offers a measure to which other quantities of the lattice can be compared.

The celebrated *Lenstra–Lenstra–Lovász algorithm*[15] (*LLL algorithm*) gives a procedure for constructing a good basis for a lattice, in the above mentioned sense, starting from a given basis. The resulting basis is a so-called *LLL reduced base.* After getting the base vectors $\mathbf{v}_1, \ldots, \mathbf{v}_k$ as an input, the algorithm produces a new basis $\mathbf{u}_1, \ldots, \mathbf{u}_k$ for the lattice $\langle \mathbf{v}_1, \ldots, \mathbf{v}_k \rangle$, for which

[14] Research of lattices belongs to the so-called *geometric number theory* or *Minkowski's geometry.*

[15] The original reference is LENSTRA, A.K. & LENSTRA JR., H.W. & LOVÁSZ, L: Factoring Polynomials with Rational Coefficients. *Mathematische Annalen* **261** (1982), 515–534.

1. $\|\mathbf{u}_1\| \leq 2^{\frac{k-1}{4}} D^{\frac{1}{k}}$,
2. $\|\mathbf{u}_1\| \leq 2^{\frac{k-1}{2}} \lambda$ where λ is the length of the shortest nonzero vector of the lattice, and
3. $\|\mathbf{u}_1\| \cdots \|\mathbf{u}_k\| \leq 2^{\frac{k(k-1)}{4}} D$.

Items 1. and 2. guarantee that the new base vector $\mathbf{u}_1$ is short both compared with the discriminant and with the shortest nonzero vector of lattice. Item 3. guarantees that the angles spanned by the new vectors are not too small. A measure of approximate orthogonality of the basis $\mathbf{u}_1, \ldots, \mathbf{u}_k$ is how close $\|\mathbf{u}_1\| \cdots \|\mathbf{u}_k\|$ is to D, since $\|\mathbf{u}_1\| \cdots \|\mathbf{u}_k\| = D$ for orthogonal vectors $\mathbf{u}_1, \ldots, \mathbf{u}_k$.

For the time complexity of the LLL algorithm there is the estimate

$$O(k^6(\ln \max(\|\mathbf{v}_1\|, \ldots, \|\mathbf{v}_k\|))^3),$$

but usually it is a lot faster in practice. However, note that time is polynomial only in the size of the vectors, not in the size of the dimension. Performance of the algorithm depends also on how the vectors $\mathbf{v}_1, \ldots, \mathbf{v}_k$ are given and how you compute with them. Naturally, an easy case is when the vectors have integral elements.

The LLL algorithm won't be discussed any further here, it is treated in much more detail for example in COHEN. Suffice it to say that it is extremely useful in a number of contexts.

"The obvious mathematical breakthrough
would be development of an easy way
to factor large prime numbers."

(BILL GATES: *The Road Ahead*)

Chapter 8

RSA

8.1 Defining RSA

RSA's[1] secret key k_2 consists of two large primes p and q of approximately equal length, and a number b (the so-called *decrypting exponent*) such that

$$\gcd(b, \phi(pq)) = \gcd(b, (p-1)(q-1)) = 1.$$

The public key k_1 is formed of the number $n = pq$ (multiplied out), and the number a (the so-called *encrypting exponent*) such that

$$ab \equiv 1 \mod \phi(n).$$

Note that b does have an inverse modulo $\phi(n)$. The encrypting function is

$$e_{k_1}(w) = (w^a, \bmod n),$$

and the decrypting function is

$$e_{k_2}(c) = (c^b, \bmod n).$$

For encrypting to work, a message block must be coded as an integer in the interval $0 \leq w \leq n-1$. Both encrypting and decrypting are done quickly using the algorithm of Russian peasants. The following small special case of the Chinese remainder theorem will be very useful:

Lemma. $x \equiv y \mod n$ *if and only if both* $x \equiv y \mod p$ *and* $x \equiv y \mod q$.

When setting up an RSA cryptosystem, we go through the following steps:

1. Generate random primes p and q of desired length, see Section 7.4.
2. Multiply p and q to get the number $n = pq$, and compute $\phi(n) = (p-1)(q-1)$ as well.
3. Find a random number b from the interval $1 \leq b \leq \phi(n) - 1$ such that $\gcd(b, \phi(n)) = 1$, by generating numbers randomly from this interval and computing the g.c.d.
4. Compute the inverse a of b modulo $\phi(n)$ using the Euclidean algorithm.
5. Publish the pair $k_1 = (n, a)$.

[1]The original reference is RIVEST, R.L. & SHAMIR, A. & ADLEMAN, L.: A Method for Obtaining Digital Signatures and Public Key Cryptosystems. *Communications of the Association for Computing Machinery* **21** (1978), 120–126.

Now let's verify that decrypting works. First of all, if $\gcd(w, n) = 1$ then by Euler's theorem for some number l we have

$$c^b \equiv (w^a)^b = w^{ab} = w^{1+l\phi(n)} = w(w^{\phi(n)})^l \equiv w \cdot 1 = w \mod n.$$

Then again, if $\gcd(w, n) \neq 1$, we have three cases:

- $w = 0$. Now apparently
$$c^b \equiv (w^a)^b = 0^b = 0 \mod n.$$
- $p \mid w$ but $w \neq 0$. Now $w = pt$ where $\gcd(q, t) = 1$. Clearly
$$c^b \equiv w^{ab} \equiv w \mod p.$$
On the other hand, by Fermat's little theorem for some number l we have
$$w^{ab} = w^{1+l\phi(n)} = w(w^{\phi(n)})^l = w(w^{(p-1)(q-1)})^l = w(w^{q-1})^{l(p-1)} \equiv w \cdot 1 = w \mod q.$$
By the lemma $c^b \equiv w^{ab} \equiv w \mod n$.
- $q \mid w$ but $w \neq 0$. We handle this just as we did the previous case.

NB. *The above mentioned condition* $\gcd(w, n) \neq 1$ *does not bode well: Either the message is directly readable or it has* p *or* q *as a factor, in which case using the Euclidean algorithm* $\gcd(w, n)$ *can be obtained and thus the whole system can be broken. Of course, this also happens if* $\gcd(c, n) \neq 1$*, but because* n *does not have higher powers of primes as factors and* $c \equiv w^a \mod n$*, in fact*

$$\gcd(c, n) = \gcd(w^a, n) = \gcd(w, n).$$

8.2 Attacks and Defences

RSA can be made very safe but this requires that certain dangerous choices are avoided. Note that KP data is always available in public-key systems. One case to be avoided was already indicated in the note above, but it is very rare. Other things that should be kept in mind are the following:

(A) The absolute value of the difference $p-q$ must not be small! Namely, if $p-q > 0$ is small then $(p - q)/2$ is small too, and $(p + q)/2$ is just a bit larger than $\sqrt{pq} = \sqrt{n}$ (check!). On the other hand,
$$n = \left(\frac{p+q}{2}\right)^2 - \left(\frac{p-q}{2}\right)^2.$$
To find the factors p and q of n we try out integers one by one starting from $\lceil \sqrt{n} \rceil$ until we hit a number x such that $x^2 - n = y^2$ is a square. When this x is found, we immediately obtain $p = x + y$ and $q = x - y$. Because n itself is not square, $\lceil \sqrt{n} \rceil = \lfloor \sqrt{n} \rfloor + 1$. Computing the integral square root is quite fast, see Section 2.6.

(B) We must keep an eye on the factor structure of $\phi(n)$ when choosing the primes p and q. If $\gcd(p - 1, q - 1)$ is large then
$$u = \operatorname{lcm}(p - 1, q - 1) = \frac{(p-1)(q-1)}{\gcd(p-1, q-1)}$$

is small (see Theorem 2.9). On the other hand, $\gcd(a, u) = 1$ (why?) and a has an inverse b' modulo u. This b' will also work as a decrypting exponent because we can now write $ab' = 1 + lu$ and $u = t(p-1) = s(q-1)$ for some numbers l, t and s, and by Fermat's little theorem

$$c^{b'} \equiv w^{ab'} = w^{1+lu} = w(w^u)^l = w(w^{p-1})^{lt} \equiv w \cdot 1 = w \mod p.$$

(Here of course $c \equiv w^a \mod p$.) Similarly $c^{b'} \equiv w \mod q$ and by the lemma also $c^{b'} \equiv w \mod n$. If u is much smaller than $\phi(n)$ then b' can be found by trying out numbers. The conclusion is that $p-1$ and $q-1$ should not have a large common divisor.

(C) A situation where $\phi(n)$ has only small prime factors must be avoided, too. Except that in this situation we can try to factor n by Pollard's $p-1$-algorithm and similar algorithms, it may also be possible to go through all candidates f for $\phi(n)$, for which $\gcd(f, a) = 1$, compute the inverse of a modulo f, decrypt some cryptotext, and in this way find $\phi(n)$ by trial and error. Note that if $\phi(n) = (p-1)(q-1)$ and n are known we can easily obtain p and q as the roots of the second degree equation

$$(x-p)(x-q) = x^2 + (\phi(n) - n - 1)x + n = 0.$$

The roots

$$x_{1,2} = \frac{-\phi(n) + n + 1 \pm \sqrt{(\phi(n) - n - 1)^2 - 4n}}{2}$$

can be computed quite quickly using integral square root.

(D) Using *iterated encrypting* we can either factor n or find the plaintext w, when the corresponding cryptotext c is available. Compute the sequence

$$c_i = (c_{i-1}^a, \mod n) = (c^{a^i}, \mod n) = (w^{a^{i+1}}, \mod n) \quad , \quad c_0 = c,$$

recursively until $\gcd(c_i - c, n) \neq 1$. If this succeeds, there are two possibilities:

- $\gcd(c_i - c, n) = p$ or $\gcd(c_i - c, n) = q$: In this case p and q are found and the system is broken.
- $\gcd(c_i - c, n) = n$: In this case necessarily $w = c_{i-1}$ and the plaintext is found. If w has a recognizable content, it will be found already in the preceding iteration round!

Does the procedure succeed every time? By Euler's theorem

$$a^{\phi(\phi(n))} \equiv 1 \mod \phi(n),$$

i.e. we can write $a^{\phi(\phi(n))} - 1 = l\phi(n)$, and further

$$c_{\phi(\phi(n))-1} \equiv w^{a^{\phi(\phi(n))}} = w^{1+l\phi(n)} = w(w^{\phi(n)})^l \equiv w \cdot 1 = w \mod n,$$

so at least $i = \phi(\phi(n))$ suffices. On the other hand, $\phi(\phi(n)) \geq \sqrt[4]{n}$, so that this bound for the number of iterations is not very interesting.

(E) Apparently very small decrypting exponents must be avoided, since they can be found by trying out numbers. As a matter of fact, certain methods make it possible to find even fairly large decrypting exponents. For example, if $b < n^{0.292}$, it can be found using the LLL algorithm.[2]

[2]See BONEH, D. & DURFEE, G.: Cryptanalysis of RSA with Private Key d Less Than $n^{0.292}$. *Proceedings of EuroCrypt '99. Lecture Notes in Computer Science* **1592**. Springer–Verlag (1999), 1–11.

A small encrypting exponent can also do harm, even if the decrypting exponent is large. If for example $w^a < n$ then w can be easily obtained from c by taking the integral a^{th} root. See also Section 8.5.

(F) It goes without saying that if there is such a small number of possible messages that they can be checked out one by one then the encrypting can be broken. If all messages are "small" then this can be done quite conveniently by the so-called *meet-in-the-middle procedure.* Here we assume that $w < 2^l$, in other words, that the length of the message in binary representation is $\leq l$. Because by the Prime number theorem there are only few possible large prime factors of w, it is fairly likely that w will be of form

$$w = w_1 w_2 \quad \text{where } w_1, w_2 \leq \lceil 2^{l/2} \rceil$$

(at least for large enough l), in which case the corresponding encrypted message is

$$c \equiv w_1^a w_2^a \mod n.$$

$\lceil 2^{l/2} \rceil$ is obtained by the algorithm of Russian peasants and by extracting the integral square root if needed. The procedure is the following:

1. Sort the numbers $(i^a, \mod n)$ $(i = 1, 2, 3, \ldots, \lceil 2^{l/2} \rceil)$ according to magnitude, including the i's in the list $\mathcal{L}$ obtained. Computing the numbers $(i^a, \mod n)$ by the algorithm of Russian peasants takes time $O(2^{l/2} N^3)$ where N is the length of n, and sorting with quicksort takes $O(l 2^{l/2})$ time steps.
2. Go through the numbers $(cj^{-a}, \mod n)$ $(j = 1, 2, 3, \ldots, \lceil 2^{l/2} \rceil)$ checking them against the list $\mathcal{L}$—this is easy, since the list is in order of magnitude. If we find a j such that
$$cj^{-a} \equiv i^a \mod n$$
then we have found $w = ij$ (meeting in the middle). Using binary search and computing powers by the algorithm of Russian peasants takes time $O(2^{l/2}(l + N^3))$. If it so happens that $j^{-1} \mod n$ does not exist then $\gcd(j, n) \neq 1$ and a factor of n is found.

The overall time is $O(2^{l/2}(l + N^3))$, which is a lot less than 2^l, assuming of course that the list $\mathcal{L}$ can be stored in a quickly accessible form.

The problem of small messages can be solved using *padding*, in other words by adding random decimals (or bits) in the beginning of the decimal (or binary) representation of the message, so that the message becomes sufficiently long. Of course a new padding needs to be taken every time. In this way even single bits can be messages and safely encrypted.

NB. *In items (B) and (C) safety can be increased by confining to the so-called* safe primes *or* Germain's numbers *p and q, i.e. to primes p and q such that $(p-1)/2$ and $(q-1)/2$ are primes. Unfortunately finding such primes is difficult—and it is not even known whether or not there infinitely many of them. Some cryptologists even think there are so few Germain numbers it is not actually safe to use them!*

A particularly unfortunate possibility in item (D) is that the iteration succeeds right away. Then it can happen that $p \mid c$ or $q \mid c$, but what is much more likely is that the message is a so-called *fixed-point message*, in other words, a message w such that

$$c = e_{k_1}(w) = w.$$

Apparently 0, 1 and $n - 1$ are such messages. But there are usually many more of them!

Theorem 8.1. *There are exactly*

$$(1+\gcd(a-1,p-1))(1+\gcd(a-1,q-1))$$

fixed-point messages.

Proof. Denote $l=\gcd(a-1,p-1)$ and $k=\gcd(a-1,q-1)$ and take some primitive roots g_1 and g_2 modulo p and q, respectively. Then the order of g_1^{a-1} modulo p is $(p-1)/l$ and the order of g_2^{a-1} modulo q is $(q-1)/k$, see Theorem 7.4 (iii). Hence the only numbers i in the interval $0\le i<p-1$ such that

$$(g_1^{a-1})^i\equiv 1 \mod p \quad \text{or} \quad (g_1^i)^a\equiv g_1^i \mod p,$$

are the numbers

$$i_j=j\frac{p-1}{l} \quad (j=0,1,\ldots,l-1).$$

Similarly the only numbers i in the interval $0\le i<q-1$ such that $(g_2^i)^a\equiv g_2^i \mod q$, are the numbers

$$h_m=m\frac{q-1}{k} \quad (m=0,1,\ldots,k-1).$$

Apparently every fixed-point message w satisfies the congruences $w^a\equiv w \mod p,q$, and vice versa. Hence exactly all fixed-point messages are obtained by the Chinese remainder theorem from the $(l+1)(k+1)$ congruence pairs

$$\begin{cases} x\equiv 0 \mod p \\ x\equiv 0 \mod q \end{cases}, \quad \begin{cases} x\equiv 0 \mod p \\ x\equiv g_2^{h_m} \mod q \end{cases}, \quad \begin{cases} x\equiv g_1^{i_j} \mod p \\ x\equiv 0 \mod q \end{cases}, \quad \begin{cases} x\equiv g_1^{i_j} \mod p \\ x\equiv g_2^{h_m} \mod q \end{cases}$$

$$(j=0,1,\ldots,l-1 \text{ and } m=0,1,\ldots,k-1). \qquad \square$$

Of course, there should not be many fixed-point messages. Because in practice a and both p and q are odd, generally there are at least $(1+2)(1+2)=9$ fixed-point messages. Especially difficult is the situation where $p-1\mid a-1$ and $q-1\mid a-1$. In this case there are $(1+p-1)(1+q-1)=n$ fixed-point messages, that is, *all messages* are fixed-point messages. If g_1 and g_2 are known and the number of fixed-point messages is relatively small, they can be found in advance and avoided later.

Some much more complicated ideas have been invented for breaking RSA. These are introduced for example in MOLLIN. None of these has turned out to be a real threat so far.

8.3 Cryptanalysis and Factorization

Breaking RSA is hard because the factors of n cannot be computed in any easy way. In the public key there is also the encrypting exponent a. The following result shows that there is no easy way to obtain additional information out of a, either. In other words, an algorithm A, which computes b from n and a, can be transformed to a probabilistic algorithm, which can be used to quickly factor n.

If a square root ω of 1 modulo n is known somehow and $\omega\not\equiv\pm 1 \mod n$, then the factors of n can be quickly computed using this square root, because then $(\omega-1)(\omega+1)\equiv 0 \mod n$ and one of the numbers $\gcd(\omega\pm 1,n)$ equals p. The following algorithm uses this idea and the assumed algorithm A trying to factor n. In a way the algorithm resembles the Miller–Rabin algorithm.

Exponent algorithm:

1. Choose a random message w, $1 \leq w < n$.
2. Compute $d = \gcd(w, n)$ using the Euclidean algorithm.
3. If $1 < d < n$, return d and n/d and quit.
4. Compute b using the algorithm A and set $y \leftarrow ab - 1$.
5. If y is now odd go to #7.
6. If y is even, set $y \leftarrow y/2$ and go to #5. If $ab - 1 = 2^s r$ where r is odd, we cycle this loop s times. Note that in this case $s \leq \log_2(ab - 1) < 2 \log_2 n$, i.e. s is comparable to the length of n.
7. Compute $\omega = (w^y, \mod n)$ by the algorithm of Russian peasants.
8. If $\omega \equiv 1 \mod n$, we give up and quit.
9. If $\omega \not\equiv 1 \mod n$, set $\omega' \leftarrow \omega$ and $\omega \leftarrow (\omega^2, \mod n)$ and go to #9. This loop will be cycled no more than s times, since $ab - 1 = 2^s r$ is divisible by $\phi(n)$ and on the other hand by Euler's theorem $w^{\phi(n)} \equiv 1 \mod n$.
10. Eventually we obtain a square root ω' of 1 modulo n such that $\omega' \not\equiv 1 \mod n$. Now if $\omega' \equiv -1 \mod n$, we give up and quit. Otherwise we compute $t = \gcd(\omega' - 1, n)$, return t and n/t, and quit.

The procedure is a probabilistic Las Vegas type algorithm where #1 is random. It may be shown that it produces the correct result at least with probability $1/2$, see for example STINSON or SALOMAA.

Despite the above results it has not been shown that breaking RSA would necessarily lead to factorization of n. On the other hand, this would make RSA vulnerable to attacks using CC data, indeed CC data may be thought of as random broken cryptotexts.

8.4 Obtaining Partial Information about Bits

Even if finding the message itself would seem to be difficult, could it be possible to find some partial information about the message, such as whether the message is even or odd, or in which of the intervals $0 \leq w < n/2$ or $n/2 < w < n$ it is? Here we assume of course that n is odd. If for example we encrypt a single bit by adding a random padding to the binary representation, parity of the message would give away the bit immediately.

In this way we obtain two problems:

(1) Compute the *parity* of w

$$\mathrm{par}(c) = (w, \mod 2)$$

starting from the cryptotext $c = e_{k_1}(w)$.

(2) Compute the *half* of w

$$\mathrm{half}(c) = \left\lfloor \frac{2w}{n} \right\rfloor$$

starting from the cryptotext $c = e_{k_1}(w)$.

These two problems are not independent:

Lemma. *The functions* par *and* half *are connected by the equations*

$$\mathrm{half}(c) = \mathrm{par}((2^a c, \bmod n)) \quad \textit{and} \quad \mathrm{par}(c) = \mathrm{half}((2^{-a} c, \bmod n)).$$

Proof. First we denote

$$c' = (2^a c, \bmod n) = ((2w)^a, \bmod n).$$

If now $\mathrm{half}(c) = 0$ then $0 \leq 2w < n$, i.e. $2w$ is the plaintext corresponding to c', and $\mathrm{par}(c') = 0$. Again, if $\mathrm{half}(c) = 1$ then $n/2 < w < n$, i.e. $0 < 2w - n < n$. Thus in this case $2w - n$ is the plaintext corresponding to c' and it is odd so $\mathrm{par}(c') = 1$.

The latter equality follows from the former. If we denote $c'' = (2^{-a}c, \bmod n)$ then by the above

$$\mathrm{half}(c'') = \mathrm{par}((2^a c'', \bmod n)) = \mathrm{par}((2^a 2^{-a} w^a, \bmod n)) = \mathrm{par}(c). \qquad \square$$

Hence it suffices to consider the function half. Now let's compute the numbers

$$c_i = \mathrm{half}(((2^i w)^a, \bmod n)) \quad (0 \leq i \leq \lfloor \log_2 n \rfloor).$$

Here of course $2^i w$ can be replaced by the "correct" message $(2^i w, \bmod n)$ if needed. Hence $c_i = 0$ exactly when dividing $2^i w$ by n the remainder is in the interval $[0, n/2)$, in other words, exactly when w is in one of the intervals

$$\frac{jn}{2^i} \leq w < \frac{jn}{2^i} + \frac{n}{2^{i+1}} \quad (j = 0, 1, \ldots, 2^i - 1).$$

Because n is odd, the following logical equivalences hold:

$$\begin{aligned}
c_0 = 0 &\iff 0 \leq w < \frac{n}{2} \\
c_1 = 0 &\iff 0 \leq w < \frac{n}{4} \text{ or } \frac{n}{2} < w < \frac{3n}{4} \\
c_2 = 0 &\iff 0 \leq w < \frac{n}{8} \text{ or } \frac{n}{4} < w < \frac{3n}{8} \text{ or } \frac{n}{2} < w < \frac{5n}{8} \text{ or } \frac{3n}{4} < w < \frac{7n}{8} \\
&\vdots
\end{aligned}$$

Thus w can be found in $\lfloor \log_2 n \rfloor + 1$ steps by binary search.

All in all we can conclude by this that an algorithm, which computes one of the functions par or half, can be transformed to an algorithm for decrypting an arbitrary message in polynomial time. So, the information about a message carried by these functions cannot be found in any easy way.

NB. *On the other hand, if we know some number of decimals/bits of the decrypting key or of the primes p or q, we can compute the rest of them quickly, see* COPPERSMITH, D.: Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities. *Journal of Cryptology* **10** (1997), 233–260.

8.5 Attack by LLL Algorithm

Very often the beginning of a plaintext is fixed and the variable extension is short. In such situations one should not use a very small encrypting exponent a. In this case the plaintext is of form

$$w = x + y$$

where x remains always the same and y is the small variable part. Let's agree that $|y| \leq Y$. The choice of Y is revealed later, of course Y is an integer. A negative y is also possible here, whatever that might mean! The corresponding cryptotext is

$$c = ((x+y)^a, \bmod n).$$

A hostile outside party now knows the public key (n, a), c, x and Y and wants to find y. For this the polynomial

$$P(t) = (\overline{x} + t)^a - \overline{c} = \sum_{i=0}^{a} \overline{d}_i t^i$$

of $\mathbb{Z}_n[t]$ is used, where the coefficients d_i are represented in the positive residue system and $d_a = 1$. So, we are seeking a number y such that $|y| \leq Y$ and $P(y) \equiv 0 \mod n$.

Consider then the $a+1$-dimensional lattice $\langle \mathbf{v}_1, \ldots, \mathbf{v}_{a+1} \rangle$ where

$$\begin{aligned} &\mathbf{v}_1 = (n, 0, \ldots, 0) \quad , \quad \mathbf{v}_2 = (0, nY, 0, \ldots, 0) \quad , \quad \mathbf{v}_3 = (0, 0, nY^2, 0, \ldots, 0) \quad , \ldots , \\ &\mathbf{v}_a = (0, \ldots, 0, nY^{a-1}, 0) \quad , \quad \mathbf{v}_{a+1} = (d_0, d_1 Y, d_2 Y^2, \ldots, d_{a-1} Y^{a-1}, Y^a). \end{aligned}$$

See Section 7.8. When the LLL algorithm is applied to this we obtain a new basis $\mathbf{u}_1, \ldots, \mathbf{u}_{a+1}$, from which we only need $\mathbf{u}_1$. Now the discriminant of the lattice is

$$D = \begin{vmatrix} n & 0 & 0 & \cdots & 0 & 0 \\ 0 & nY & 0 & \cdots & 0 & 0 \\ 0 & 0 & nY^2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & nY^{a-1} & 0 \\ d_0 & d_1 Y & d_2 Y^2 & \cdots & d_{a-1} Y^{a-1} & Y^a \end{vmatrix} = n^a Y^{1+2+\cdots+a} = n^a Y^{\frac{a(a+1)}{2}},$$

so

$$\|\mathbf{u}_1\| \leq 2^{\frac{a}{4}} D^{\frac{1}{a+1}} = 2^{\frac{a}{4}} n^{\frac{a}{a+1}} Y^{\frac{a}{2}}.$$

$\mathbf{u}_1$ can naturally be written as a linear combination of the original base vectors with integer coefficients:

$$\mathbf{u}_1 = e_1 \mathbf{v}_1 + \cdots + e_{a+1} \mathbf{v}_{a+1} = (f_0, f_1 Y, f_2 Y^2, \ldots, f_a Y^a)$$

where

$$f_i = e_{i+1} n + e_{a+1} d_i \quad (i = 0, 1, \ldots, a-1) \quad \text{and} \quad f_a = e_{a+1}.$$

Hence

$$f_i \equiv e_{a+1} d_i \mod n \quad (i = 0, 1, \ldots, a).$$

Now we take the polynomial

$$Q(t) = \sum_{i=0}^{a} f_i t^i.$$

Because $P(y) \equiv 0 \mod n$, we have also

$$Q(y) = \sum_{i=0}^{a} f_i y^i \equiv \sum_{i=0}^{a} e_{a+1} d_i y^i = e_{a+1} \sum_{i=0}^{a} d_i y^i = e_{a+1} P(y) \equiv 0 \mod n.$$

Furthermore, by the triangle inequality, the estimate $|y| \leq Y$ and the Cauchy–Schwarz inequality,

$$|Q(y)| \leq \sum_{i=0}^{a} |f_i y^i| \leq \sum_{i=0}^{a} |f_i| Y^i = \sum_{i=0}^{a} 1 \cdot |f_i| Y^i \leq (a+1)^{\frac{1}{2}} \|\mathbf{u}_1\|.$$

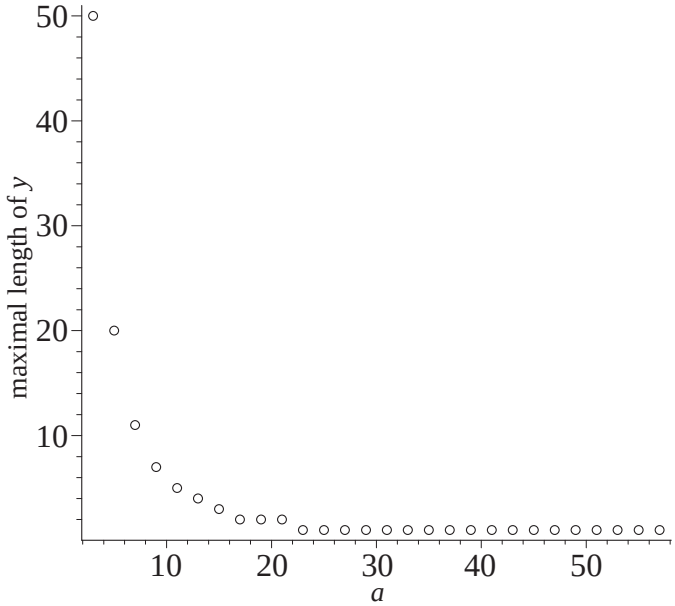
At this point we can give an estimate for Y. Choose a Y such that

$$(a+1)^{\frac{1}{2}} 2^{\frac{a}{4}} n^{\frac{a}{a+1}} Y^{\frac{a}{2}} < n\,, \quad \text{i.e. (check!)} \quad Y < 2^{-\frac{1}{2}} (a+1)^{-\frac{1}{a}} n^{\frac{2}{a(a+1)}}.$$

Hence $|Q(y)| < n$. Because, on the other hand, $Q(y) \equiv 0 \mod n$ it must be that $Q(y) = 0$. So, the desired y can also be found by any numerical algorithm for finding the roots of the polynomial equation $Q(y) = 0$ with integral coefficients. There may be several alternatives, hopefully one of them will turn out to be the correct one.

The method is fast if a is small enough. The maximum length of the vectors $\mathbf{v}_1, \ldots, \mathbf{v}_{a+1}$ is proportional to the length of Y^a and the LLL algorithm is polynomial-time in this length. On the other hand, the LLL algorithm is slow for large values of a—remember it wasn't polynomial-time in the length of the dimension—and the numerical search of roots is then laborious also.

On the other hand, for large values of a, a rather small Y and hence y must be chosen, which further limits usefulness. If n is of order 10^{300}, we obtain the following connection between the decimal length of y and a using the choice of Y above:



”Algebra is numbers forgetting why they’re letters.”

(JAROD KINTZ: *This Book is Not FOR SALE*)

Chapter 9

ALGEBRA: GROUPS

9.1 Groups

A *group* is an algebraic structure $G = (A, \odot, \mathbf{1})$ where $\odot$ is a binary computational operation, the so-called *group operation,* and $\mathbf{1}$ is the so-called *identity element* of the group. In addition it is required that the following conditions hold:

(1) $(a \odot b) \odot c = a \odot (b \odot c)$ ($\odot$ is associative).

(2) $a \odot \mathbf{1} = \mathbf{1} \odot a = a$.

(3) For every element a there exists a unique element a^{-1}, the so-called *inverse* of a, for which $a \odot a^{-1} = a^{-1} \odot a = \mathbf{1}$.

Furthermore, it is naturally assumed that $a \odot b$ is defined for all elements a and b, and that the result is unique. The group operation is often read ”times” and called *product*. If in addition

(4) $a \odot b = b \odot a$ ($\odot$ is commutative)

then we say that G is a *commutative group.*[1]

Because of the associativity we can write

$$a_1 \odot a_2 \odot \cdots \odot a_n$$

without parentheses, the result does not depend on how the parentheses are set. Furthermore we denote, as in Section 4.1,

$$a^n = \underbrace{a \odot \cdots \odot a}_{n \text{ copies}}, \quad a^{-n} = \underbrace{a^{-1} \odot \cdots \odot a^{-1}}_{n \text{ copies}} \quad \text{and} \quad a^0 = \mathbf{1}$$

and the usual rules of power calculus hold. Powers can also be computed using the algorithm of Russian peasants.

NB. *Commutative groups are also often called* additive groups. *In this case the following additive notation and nomenclature is commonly used: The group operation is denoted by* $\oplus$ *or* $+$ *etc. and called* sum. *It is often read ”plus”. The identity element is called* zero element *and denoted by* $\mathbf{0}$ *or* 0 *etc. The inverse* a^{-1} *is called* opposite element *and denoted by* $-a$. *A power* a^n *is called* multiple *and denoted by* na. *Compare with the notations in Section 4.1.*

[1] A commutative group is also called *Abelian group.*

The simplest group is of course the *trivial group* where there is only one element (the identity element). Other examples of groups are:

- The familiar group $(\mathbb{Z}, +, 0)$ (integers and addition) is usually denoted briefly just by $\mathbb{Z}$. Inverses are opposite numbers and the group is commutative.
- $(\mathbb{Z}_m, +, \overline{0})$ (residue classes modulo m and addition) is also a commutative group, inverses are opposite residue classes. This is a called the *residue class group* modulo m, and denoted briefly by $\mathbb{Z}_m$.
- Nonsingular $n \times n$ matrices with real elements form the group $(\mathbb{R}^{n\times n}, \cdot, \mathbf{I}_n)$ with respect to matrix multiplication. This group is not commutative (unless $n = 1$). The identity element is the $n \times n$ identity matrix $\mathbf{I}_n$ and inverses are inverse matrices.
- If we denote reduced residue classes modulo m by $\mathbb{Z}_m^*$, see Section 2.4, then $(\mathbb{Z}_m^*, \cdot, \overline{1})$ is a commutative group, inverses are inverse classes. Note that the product of two reduced residue classes is also a reduced residue class. This is called the *group of units* of $\mathbb{Z}_m$, denoted briefly by just $\mathbb{Z}_m^*$, and it has $\phi(m)$ elements (reduced residue classes).
- From every ring $R = (A, \oplus, \odot, \mathbf{0}, \mathbf{1})$, see Section 4.1, its *additive group* $R^+ = (A, \oplus, \mathbf{0})$ can be extracted. Moreover, from every field $F = (A, \oplus, \odot, \mathbf{0}, \mathbf{1})$ also its *multiplicative group* $F^* = (A - \{\mathbf{0}\}, \odot, \mathbf{1})$ can be extracted, it is also called group of units of F.

For an element a of a group $(A, \odot, \mathbf{1})$ the smallest number $i \geq 1$ (if one exists) such that $a^i = \mathbf{1}$ is called the *order* of a. Basic properties of order are same as for the order of a number modulo m in Section 7.2, and the proofs are also the same (indeed, order modulo m is the same as order in the group $\mathbb{Z}_m^*$):

- If $a^j = \mathbf{1}$ then the order of a divides j.
- If the order of a is i then the order of a^j is
$$\frac{i}{\gcd(i,j)} = \frac{\mathrm{lcm}(i,j)}{j}.$$
- If the order of a is i then $a^{-1} = a^{i-1}$.
- If, in a commutative group, the order of a is i and the order of b is j and $\gcd(i,j) = 1$ then the order of $a \odot b$ is ij.
- Elements of finite groups always have orders.

If the size of a finite group $G = (A, \odot, \mathbf{1})$ is N and for some element g

$$A = \{\mathbf{1}, g, g^2, \ldots, g^{N-1}\},$$

in other words, all elements of the group are powers of g then the group is called a *cyclic group* and g is called its *generator*. In this case we often write $G = \langle g \rangle$. Note that the order of g then must be N (why?). An infinite group can also be cyclic, we then require that

$$A = \{\mathbf{1}, g^{\pm 1}, g^{\pm 2}, \ldots\}.$$

A cyclic group is naturally always commutative.

Apparently for instance $\mathbb{Z}$ and $\mathbb{Z}_m$ are cyclic with 1 and $\overline{1}$ as their generators. If there exists a primitive root modulo m then $\mathbb{Z}_m^*$ is cyclic with the primitive root as its generator.

NB. *A finite cyclic group $\langle g \rangle$ with N elements has a structure equal (or isomorphic) to that of $\mathbb{Z}_N$:*

$$g^i \odot g^j = g^{(i+j, \bmod N)} \quad \textit{and} \quad (g^i)^{-1} = g^{(-i, \bmod N)}.$$

Computing in $\mathbb{Z}_N$ is easy and fast, as we have seen. On the other hand, computing in $\langle g \rangle$ is not necessarily easy at all if the connection between g^i and i is not easy to compute. This is used in numerous cryptosystems, see the next chapter. We get back to this when considering discrete logarithms.

The multiplicative group $\mathbb{F}_{p^n}^*$ of the finite field $\mathbb{F}_{p^n}$ is always cyclic. Its generators are called *primitive elements.* This was already stated in Theorem 6.4 for the prime field $\mathbb{Z}_p$, whose generators are also called primitive roots modulo p. If $G = (A, \odot, \mathbf{1})$ is a group and $H = (B, \odot, \mathbf{1})$, where B is subset of A, is also group then H is a so-called *subgroup* of G. For example, $(2\mathbb{Z}, +, 0)$, where $2\mathbb{Z}$ is the set of even integers, is a subgroup of $\mathbb{Z}$. *Cyclic subgroups,* that is, *subgroups generated by single elements,* are important subgroups: If the order of a is i then in the subgroup $\langle a \rangle$ generated by a we take

$$B = \{\mathbf{1}, a, a^2, \ldots, a^{i-1}\}.$$

And if a does not have an order then

$$B = \{\mathbf{1}, a^{\pm 1}, a^{\pm 2}, \ldots\}.$$

It is easy to see that this is a subgroup. A basic property of subgroups of finite groups is the following divisibility property. Denote the cardinality of a set C by $|C|$.

Theorem 9.1. (Lagrange's theorem) *If $G = (A, \odot, \mathbf{1})$ is a finite group and $H = (B, \odot, \mathbf{1})$ is its subgroup then $|B|$ divides $|A|$. In particular, the order of every element of G divides $|A|$.*

Proof. Consider the sets

$$a \odot H = \{a \odot b \mid b \in B\},$$

the so-called *left cosets.* If c is in the left coset $a \odot H$ then $c = a \odot b$ and $a = c \odot b^{-1}$ where $b \in B$. Hence $c \odot H \subseteq a \odot H$ and $a \odot H \subseteq c \odot H$, so $a \odot H = c \odot H$. Thus two left cosets are always either exactly the same or completely disjoint. So A is partitioned into a number of mutually disjoint left cosets, each of which has $|B|$ elements. Note that B itself is the left coset $\mathbf{1} \odot H$. □

If $G_1 = (A_1, \odot_1, \mathbf{1}_1)$ and $G_2 = (A_2, \odot_2, \mathbf{1}_2)$ are groups then their *direct product* is the group

$$G_1 \times G_2 = (C, \otimes, (\mathbf{1}_1, \mathbf{1}_2))$$

where the set of elements is the Cartesian product

$$C = A_1 \times A_2 = \{(a_1, a_2) \mid a_1 \in A_1 \text{ ja } a_2 \in A_2\}$$

and the operation $\otimes$ and inverses are defined by

$$(a_1, a_2) \otimes (b_1, b_2) = (a_1 \odot_1 b_1, a_2 \odot_2 b_2) \quad \text{and} \quad (a_1, a_2)^{-1} = (a_1^{-1}, a_2^{-1}).$$

It is easy to see that the $G_1 \times G_2$ defined in this way is truly a group. The idea can extended, direct products $G_1 \times G_2 \times G_3$ of three groups can be defined, and so on. Without proofs we now present the following classical result, which shows that the groups $\mathbb{Z}_m$ can be used to essentially chacterize every finite commutative group using direct products:

Theorem 9.2. (Kronecker's decomposition) *Every commutative finite group is structurally identical (or isomorphic) to some direct product*

$$\mathbb{Z}_{p_1^{i_1}} \times \mathbb{Z}_{p_2^{i_2}} \times \cdots \times \mathbb{Z}_{p_k^{i_k}}$$

where $p_1, \ldots, p_k$ are different primes and $i_1, \ldots, i_k \geq 1$. Here we may agree that the empty direct product corresponds to the trivial group $\{\mathbf{1}\}$, so that it is included, too.

9.2 Discrete Logarithm

In a cyclic goup $\langle g \rangle$ we define the *discrete logarithm* in the base g by

$$\log_g a = j \quad \text{exactly when} \quad a = g^j.$$

Furthermore we will assume that in a finite cyclic group with N elements, $0 \leq \log_g a \leq N-1$.

For example in $\mathbb{Z}$ the logarithm is trivial: The only bases are ± 1 and $\log_{\pm 1} a = \pm a$. It is also quite easy in the group $\mathbb{Z}_m$: The base is some $\overline{i}$ where $\gcd(i, m) = 1$, and $\log_{\overline{i}} \overline{j} = (ji^{-1}, \mathrm{mod}\ m)$. But already discrete logarithms in $\mathbb{Z}_p^*$ are anything but trivial for a large prime p, and have proved to be very laborious to compute. Also discrete logarithms in many other groups are difficult to compute. Even if the group G itself is not cyclic, and discrete logarithm is not defined in G itself, in any case discrete logarithms are defined in its cyclic subgroups.

Now let's take a closer look at the logarithm in $\mathbb{Z}_p^*$, also often called *index*. The problem is to find a number j in the interval $0 \leq j \leq p-2$ such that $g^j \equiv b \mod p$, when the generator (primitive root) g and b are given e.g. as decimal numbers in the positive residue system. Clearly this problem is in $\mathcal{NP}$: Guess j and test its correctness by exponentiation using the algorithm of Russian peasants. On the other hand, deterministically j can be computed by simple search and the algorithm of Russian peasants in estimated time $O(p(\ln p)^3)$ and in polynomial space. By computing in advance as preprocessing the so-called *index table*, in other words, the pairs

$$(i, (g^i, \mathrm{mod}\ p)) \quad (i = 0, 1, \ldots, p-2)$$

sorted by the second component, the problem can be solved in polynomial time and space, excluding the index table, but then there is an overhead of superpolynomial time and space. A sort of intermediate form is given by

Shanks's baby-step-giant-step algorithm:

1. Set $m \leftarrow \lceil \sqrt{p-1}\, \rceil$. The integral square root $\lfloor \sqrt{p-1} \rfloor$ is quick to compute and

$$\lceil \sqrt{p-1}\, \rceil = \begin{cases} \lfloor \sqrt{p-1} \rfloor \text{ if } p-1 \text{ is a square, i.e. } p-1 = \lfloor \sqrt{p-1} \rfloor^2 \\ \lfloor \sqrt{p-1} \rfloor + 1 \text{ otherwise.} \end{cases}$$

2. Compute the pairs

$$(i, (g^{mi}, \mathrm{mod}\ p)) \quad (i = 0, 1, \ldots, m-1) \quad \text{(the } \textit{giant steps}\text{)}$$

   and sort them by the second component. As a result we have the list $\mathcal{L}_1$. In this we need the algorithm of Russian peasants and a fast sorting algorithm, for example quicksort.

3. Compute the pairs

$$(k, (bg^{-k}, \bmod p)) \quad (k = 0, 1, \ldots, m-1) \quad \text{(the } \textit{baby steps}\text{)}$$

and sort them by the second component, as well. In this way we obtain the list $\mathcal{L}_2$.

4. Find a pair (i, y) from the list $\mathcal{L}_1$ and a pair (k, z) from the list $\mathcal{L}_2$ such that $y = z$.

5. Return $(mi + k, \bmod p - 1)$ and quit.

If these pairs can be found, the obtained number $j = (mi+k, \bmod p-1)$ is the correct logarithm, since in this case we can write $mi + k = t(p - 1) + j$ and

$$g^{mi} \equiv bg^{-k} \mod p\,, \quad \text{i.e.} \quad b \equiv g^{mi+k} = (g^{p-1})^t g^j \equiv 1 \cdot g^j \equiv g^j \mod p.$$

On the other hand, the algorithm always returns a result, since if $b \equiv g^j \mod p$ and $0 \le j \le p-2$ then using division j can be expressed in the form $j = mi + k$ where $0 \le k < m$, whence also

$$i = \frac{j-k}{m} \le \frac{j}{m} < \frac{p-1}{m} \le \frac{p-1}{\sqrt{p-1}} = \sqrt{p-1} \le m.$$

The baby-step-giant-step algorithm can be implemented in time $O(m)$ and space $O(m)$.

Other algorithms for computing discrete logarithm in $\mathbb{Z}_p^*$ are for example Pollard's kangaroo algorithm, see Section 12.2, the *Pohlig–Hellman algorithm* and the so-called *index calculus method,* see for example STINSON and SALOMAA. The Pohlig–Hellman algorithm is reasonably fast if $p - 1$ has only small prime factors. All these algorithms can be generalized to computing discrete logarithms of $\mathbb{F}_{p^n}^*$, also a very laborious task.

9.3 Elliptic Curves

Geometrically an *elliptic*[2] *curve* means a curve of third degree, satisfying the implicit equation

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

Note the special indexing of coefficients, which is traditional. An additional requirement is that the curve is smooth, in other words, that the equations

$$\begin{aligned} a_1y &= 3x^2 + 2a_2x + a_4 \\ 2y + a_1x + a_3 &= 0 \end{aligned}$$

obtained by differentiating both sides, are not both simultaneously satisfied in the curve. Geometrically this guarantees that the curve has a tangent in every point. Using implicit derivation, familiar from basic courses,

$$\frac{dy}{dx} = \frac{3x^2 + 2a_2x + a_4 - a_1y}{2y + a_1x + a_3} \quad \text{and} \quad \frac{dx}{dy} = \frac{2y + a_1x + a_3}{3x^2 + 2a_2x + a_4 - a_1y}.$$

When both horizontal and vertical tangents are allowed, the only situation where a tangent may not exist is when the numerator and the denominator both vanish.

[2]The name comes from the fact that certain algebraic functions $y = f(x)$, related to computing lengths of arcs of ellipses by integration, satisfy such third degree equation.

Originally an elliptic curve was of course real, or in $\mathbb{R}^2$. The curve can be considered in any field $\mathbb{F}$ (the so-called *field of constants*), which the coefficients come from, however. In this case the curve is the set of all pairs (x, y), which satisfy the defining equation. Although the smoothness condition does not necessarily have any "geometric" meaning in this case, it turns out to be very important.

Quite generally we can confine ourselves to simpler elliptic curves of the form

$$y^2 = x^3 \oplus ax \oplus b$$

(the so-called *Weierstraß short form*) where the equations

$$\begin{aligned} \mathbf{0} &= \mathbf{3}x^2 \oplus a \\ \mathbf{2}y &= \mathbf{0} \end{aligned}$$

are not simultaneously satisfied (the smoothness condition). Here the notations $\mathbf{2} = 2\mathbf{1}$ and $\mathbf{3} = 3\mathbf{1}$ are used. Assuming that $\mathbf{2} \neq \mathbf{0}$ and $\mathbf{3} \neq \mathbf{0}$, eliminating x and y from the equations

$$\begin{aligned} y^2 &= x^3 \oplus ax \oplus b \\ \mathbf{0} &= \mathbf{3}x^2 \oplus a \\ \mathbf{2}y &= \mathbf{0} \end{aligned}$$

(which is not very difficult, try it) we see that this corresponds to the condition

$$4a^3 \oplus 27b^2 \neq \mathbf{0}.$$

A special property of this simpler type of curves is that they are symmetric with respect to the x-axis, in other words, if a point (x, y) is in the curve then so is the point $(x, -y)$.

So, exceptions will be fields where $\mathbf{2} = \mathbf{0}$ (for example the fields $\mathbb{F}_{2^n}$) or where $\mathbf{3} = \mathbf{0}$ (for example $\mathbb{F}_{3^n}$). In the former the equations are of the form

$$y^2 \oplus ay = x^3 \oplus bx \oplus c \quad \text{(the } \textit{supersingular} \text{ case)}$$

and

$$y^2 \oplus xy = x^3 \oplus ax^2 \oplus b \quad \text{(the } \textit{nonsupersingular} \text{ case)},$$

and in the latter

$$y^2 = x^3 \oplus ax^2 \oplus bx \oplus c.$$

In addition, the corresponding smoothness conditions will be needed, too. Even though for instance the fields $\mathbb{F}_{2^n}$ are very important in cryptography, in what follows we will for simplicity confine ourselves only to fields for which the above-mentioned short form $y^2 = x^3 \oplus ax \oplus b$, where $4a^3 \oplus 27b^2 \neq \mathbf{0}$, is possible. Other forms are considered e.g. by WASHINGTON and BLAKE & SEROUSSI & SMART.

For geometric reasons it has been known for a long time that for a real elliptic curve, or rather for its points, a computational operation can be defined, which makes it a commutative group. The corresponding definition can also be made in other fields, in which case we also obtain a commutative group. These groups are simply called just *elliptic curves.* Because there are a lot of elliptic curves, we obtain in this way abundant cyclic subgroups, convenient for cryptosystems based on discrete logarithms.

Now let's first consider the group operation in $\mathbb{R}^2$ for the sake of illustration. The identity element of the group is somewhat artificial, it is a "point" O in infinity in the direction of the y-axis. Positive and negative infinities are identified. It is agreed that all lines parallel to the y-axis intersect at this point O. Geometrically the group operation $\boxplus$ for the points P and Q produces the point $R = P \boxplus Q$, and the opposite point $-P$ by the following rule:

1. Draw a line through the points P and Q. If $P = Q$, this line is the tangent line at the point P. Smoothness guarantees that a tangent exists.
2. If the drawn line is parallel to the y-axis then $R = O$.
3. Otherwise R is the reflection of the point of intersection of the line and the curve, with respect to the x-axis. It is possible that the line is tangent to the curve in P (when the point of intersection and P merge), in which case R is the reflection of P, or in Q (the point of intersection and Q merge), in which case R is the reflection of Q.
4. $-P$ is the reflection of P with respect to the x-axis. In particular, $-O = O$.

Apparently the operation $\boxplus$ is commutative. Interpreting this rule suitably we see immediately that $P \boxplus O = O \boxplus P = P$ (in particular, $O \boxplus O = O$) and that $P \boxplus -P = -P \boxplus P = O$, as in a group it should be.

Example. *On the right there is the elliptic curve*

$$y^2 = x^3 - 5x + 1$$

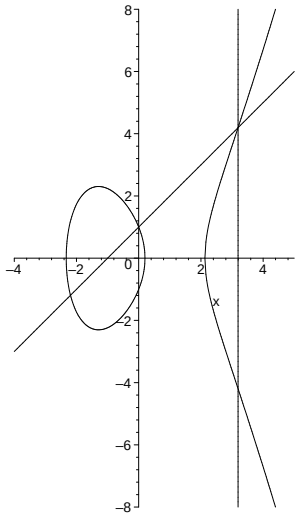
in $\mathbb{R}^2$ drawn by the Maple program. Also shown is the group operation of the points

$$P = ((1 - \sqrt{29})/2, (3 - \sqrt{29})/2) \quad \textit{and} \quad Q = (0, 1)$$

of the curve. The result is

$$R = ((1 + \sqrt{29}/2, -(3 + \sqrt{29})/2).$$

Note how the curve has two separate parts, of which one is closed and the other infinite. Not all elliptic curves are bipartite in this way.



We will now compute the result of the operation $P \boxplus Q = R$ in general. The cases $P = O$ and/or $Q = O$ are easy. If the points are $P = (x_1, y_1)$ and $Q = (x_2, y_2)$, $P \neq Q$ and $x_1 = x_2$ then apparently $y_1 = -y_2$, so $R = O$ or $P = -Q$. Hence we move on to cases in which either $x_1 \neq x_2$ or $P = Q$. First let's deal with the former case. A parametric representation of the line through P and Q is then

$$\begin{cases} x = x_1 + (x_2 - x_1)t \\ y = y_1 + (y_2 - y_1)t. \end{cases}$$

Let's substitute these into equation $y^2 - x^3 - ax - b = 0$ of the elliptic curve:

$$(y_1 + (y_2 - y_1)t)^2 - (x_1 + (x_2 - x_1)t)^3 - a(x_1 + (x_2 - x_1)t) - b = 0.$$

The left side is a third-degree polynomial $p(t)$ in the variable t. Since the point P is in the curve (corresponding to $t = 0$) and so is the point Q (corresponding to $t = 1$), the polynomial $p(t)$ is divisible by $t(t - 1)$, i.e. $p(t) = q(t)t(t - 1)$ for some first-degree polynomial $q(t)$. Furthermore we obtain from the equation $q(t) = 0$ the parameter value t_3 corresponding to the third intersection point (x_3, y_3). A division shows that

$$q(t) = (y_2 - y_1)^2 - 3x_1(x_2 - x_1)^2 - (x_2 - x_1)^3(t + 1)$$

and so

$$t_3 = \frac{(y_2 - y_1)^2}{(x_2 - x_1)^3} - \frac{2x_1 + x_2}{x_2 - x_1}.$$

Substituting these to the parametric representation of the line we obtain

$$\begin{cases} x_3 = \lambda^2 - x_1 - x_2 \\ y_3 = \lambda(x_3 - x_1) + y_1 \end{cases}$$

where

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1}$$

(slope of the line), and finally

$$P \boxplus Q = R = (x_3, -y_3).$$

Here it may be that $(x_3, y_3) = P$ or $(x_3, y_3) = Q$. Note that (x_3, y_3) is always defined.

We still need to consider the case $P = Q = (x_1, y_1)$, and compute

$$P \boxplus P = 2P = R.$$

If $y_1 = 0$, the tangent of the curve is apparently parallel to the y-axis and $R = O$ or $-P = P$. Thus we move on to the case $y_1 \neq 0$. The slope of the tangent is

$$\frac{dy}{dx} = \frac{3x^2 + a}{2y}.$$

Hence a parametric representation of tangent line drawn in the point P is

$$\begin{cases} x = x_1 + 2y_1 t \\ y = y_1 + (3x_1^2 + a)t. \end{cases}$$

Substituting these into the equation of the curve as before we obtain the polynomial

$$p(x) = (y_1 + (3x_1^2 + a)t)^2 - (x_1 + 2y_1 t)^3 - a(x_1 + 2y_1 t) - b.$$

Since the point P is in the curve (corresponding to $t = 0$), $p(t)$ is divisible by t, in other words, $p(t) = q(t)t$. By division we obtain

$$q(t) = ((3x_1^2 + a)^2 - 12x_1 y_1^2)t - 8y_1^3 t^2.$$

One root of the equation $q(t) = 0$ is $t = 0$ and the other is

$$t_2 = \frac{(3x_1^2 + a)^2}{8y_1^3} - \frac{3x_1}{2y_1}.$$

The intersection point (x_2, y_2) is obtained by substituting this into the parametric representation:

$$\begin{cases} x_2 = \lambda^2 - 2x_1 \\ y_2 = \lambda(x_2 - x_1) + y_1 \end{cases}$$

where

$$\lambda = \frac{3x_1^2 + a}{2y_1}$$

(slope of the line). Finally we obtain

$$2P = R = (x_2, -y_2).$$

Again it can be that $P = (x_2, y_2)$. Also in this case (x_2, y_2) is always defined.

These computational formulas can be used in any field in which the elliptic curve can be written in the short form $y^2 = x^3 \oplus ax \oplus b$ where $4a^3 \oplus 27b^2 \neq \mathbf{0}$. In other fields somewhat different formulas are needed, see KOBLITZ or WASHINGTON or BLAKE & SEROUSSI & SMART.

All in all we conclude that forming the opposite element is easy (reflection), the group operation is commutative and quite easy to compute. However, associativity of the operation is difficult to prove starting from the formulas above. The correct world, thinking about properties of elliptic curves, is the so-called *projective geometry,* in which the group operation itself occurs naturally. Associativity in $\mathbb{R}^2$ follows fairly directly from classical results of projective geometry for curves of the third degree. The following result (translated) can be found in an old Finnish classic[3] of projective geometry, from which associativity follows easily:

> *"If two lines a and b intersect a third-degree curve in the points A_1, A_2, A_3; B_1, B_2, B_3, respectively, the third intersection points C_1, C_2, C_3 of the lines A_1B_1, A_2B_2, A_3B_3 and the curve are collinear."*

In other fields associativity must be proved separately and it is quite an elaborate task, see for example WASHINGTON. Note that in other fields also commutativity must be proved separately, but this is fairly easy. Both laws are symbolic identities, so they can be verified symbolically. Let's do it by using the Maple program. Apparently cases in which at least one of the elements is O are trivial, so they can be ignored.

Let's begin with commutativity. First we define the group operation by

```
> eco:=proc(u,v)
  local lambda,xx,yy;
  lambda:=(v[2]-u[2])/(v[1]-u[1]);
  xx:=lambda^2-u[1]-v[1];
  yy:=lambda*(xx-u[1])+u[2];
  [xx,-yy];
  end:
```

and then check the commutative law:

```
> A:=eco([x[1],y[1]],[x[2],y[2]]);
```

$$[\frac{(y_2-y_1)^2}{(x_2-x_1)^2}-x_1-x_2,-(y_2-y_1)\left(\frac{(y_2-y_1)^2}{(x_2-x_1)^2}-2\,x_1-x_2\right)(x_2-x_1)^{-1}-y_1]$$

```
> B:=eco([x[2],y[2]],[x[1],y[1]]);
```

$$[\frac{(y_1-y_2)^2}{(x_1-x_2)^2}-x_2-x_1,-(y_1-y_2)\left(\frac{(y_1-y_2)^2}{(x_1-x_2)^2}-2\,x_2-x_1\right)(x_1-x_2)^{-1}-y_2]$$

```
> normal(A-B);
```

$$[0,0]$$

Let's then verify associativity in the case of no doublings.

```
> A:=eco([x[1],y[1]],eco([x[2],y[2]],[x[3],y[3]])):
> B:=eco(eco([x[1],y[1]],[x[2],y[2]]),[x[3],y[3]]):
> C:=numer(normal(A-B)):
> max(degree(C[1],y[1]),degree(C[1],y[2]),degree(C[1],y[3]),
      degree(C[2],y[1]),degree(C[2],y[2]),degree(C[2],y[3]));
```

$$11$$

We need to substitute the equation of the curve raised to higher powers:

[3] NYSTRÖM, E.J.: *Korkeamman geometrian alkeet sovellutuksineen.* Otava (1948).

```
> yhtalot:={seq(y[1]^(2*i)=(x[1]^3+a*x[1]+b)^i,i=1..5),
            seq(y[2]^(2*i)=(x[2]^3+a*x[2]+b)^i,i=1..5),
            seq(y[3]^(2*i)=(x[3]^3+a*x[3]+b)^i,i=1..5),
            seq(y[1]^(2*i+1)=y[1]*(x[1]^3+a*x[1]+b)^i,i=1..5),
            seq(y[2]^(2*i+1)=y[2]*(x[2]^3+a*x[2]+b)^i,i=1..5),
            seq(y[3]^(2*i+1)=y[3]*(x[3]^3+a*x[3]+b)^i,i=1..5)}:
> normal(subs(yhtalot,C));
```

$$[0, 0]$$

Numbers of terms are pretty large:

```
> nops(C[1]),nops(C[2]);
```

$$1082, 6448$$

Verification by hand would thus be quite tedious, but associativity can also be proved mathematically using some ingenuity. Let's then check associativity in a remaining case which has one doubling:

$$P \boxplus (Q \boxplus Q) = (P \boxplus Q) \boxplus Q.$$

(The other cases are checked similarly.) First we define the doubling by

```
> ecs:=proc(u)
  local lambda,xx,yy;
  lambda:=(3*u[1]^2+a)/2/u[2];
  xx:=lambda^2-2*u[1];
  yy:=lambda*(xx-u[1])+u[2];
  [xx,-yy];
  end:
> A:=eco([x[1],y[1]],ecs([x[2],y[2]])):
> B:=eco(eco([x[1],y[1]],[x[2],y[2]]),[x[2],y[2]]):
> C:=numer(normal(A-B)):
> max(degree(C[1],y[1]),degree(C[1],y[2]),
      degree(C[2],y[1]),degree(C[2],y[2]));
```

$$15$$

Again we need to substitute the equation of the curve raised to higher powers:

```
> yhtalot:={seq(y[1]^(2*i)=(x[1]^3+a*x[1]+b)^i,i=1..7),
            seq(y[2]^(2*i)=(x[2]^3+a*x[2]+b)^i,i=1..7),
            seq(y[1]^(2*i+1)=y[1]*(x[1]^3+a*x[1]+b)^i,i=1..7),
            seq(y[2]^(2*i+1)=y[2]*(x[2]^3+a*x[2]+b)^i,i=1..7)}:
> normal(subs(yhtalot,C));
```

$$[0, 0]$$

Elliptic curves are very variable as groups. However, Kronecker's decomposition tells us that finite elliptic curves are direct products of residue class groups. In fact, we get an even more accurate result:

Theorem 9.3. (Cassels' theorem) *An elliptic curve over the finite field $\mathbb{F}_q$ is either cyclic or structurally identical (i.e. isomorphic) to a direct product $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2}$ of two residue class groups such that $n_1 \mid n_2, q-1$.*

Considering the size of the group we know that

Theorem 9.4. (Hasse's theorem) *If there are N elements in an elliptic curve over the finite field $\mathbb{F}_q$ then*

$$q + 1 - 2\sqrt{q} \leq N \leq q + 1 + 2\sqrt{q}.$$

Astonishingly enough, if the coefficients of an elliptic curve are in some subfield, it is enough to know how many of its elements are in this subfield:

Theorem 9.5. *Assume that E is an elliptic curve over the field $\mathbb{F}_q$, that there are $q+1-a$ elements in it (cf. Hasse's theorem), and that the roots of the equation $x^2 - ax + q = 0$ are α and β. Then, if we consider E as an elliptic curve over the field $\mathbb{F}_{q^m}$, there are exactly $q^m + 1 - \alpha^m - \beta^m$ elements in it. Note that because $\mathbb{F}_q$ is a subfield of $\mathbb{F}_{q^m}$, E can also be interpreted as an elliptic curve over $\mathbb{F}_{q^m}$. See Section 4.3.*

Proofs of these theorems require some fairly deep algebraic number theory![4] Hence there are approximately as many elements in an elliptic curve over the field $\mathbb{F}_q$ as there are in $\mathbb{F}_q$. Some quite powerful algorithms are known for computing the exact number of the elements, the so-called *Schoof algorithm*[5] and its followers, see WASHINGTON or BLAKE & SEROUSSI & SMART.

It is not easy to find even one of these many elements. As a matter of fact, we do not know any polynomial-time deterministic algorithm for generating elements of elliptic curves over finite fields. If $q = p^k$, one (slow) way is of course to generate random pairs (x, y), where $x, y \in \mathbb{F}_q$, using the representation of the field $\mathbb{F}_q$ as residue classes of polynomials in $\mathbb{Z}_p[x]$ modulo some k^{th}-degree indivisible polynomial of $\mathbb{Z}_p[x]$—see Section 4.3—and test whether the pair satisfies the equation of the elliptic curve. By Hasse's theorem, an element is found by a single guess with an approximate probability of $1/q$. The following Las Vegas type algorithm produces an element of the curve in the positive residue system, in a prime field $\mathbb{Z}_p$ where $p > 3$:

1. Choose a random number x from the interval $0 \leq x < p$ and set
$$z \leftarrow (x^3 + ax + b, \bmod p).$$
By Hasse's theorem this produces a quadratic residue z with an approximate probability of 50%, since from each z we obtain two values of y, unless $z = 0$.
2. If $z = 0$, return $(x, 0)$ and quit.
3. If $z^{(p-1)/2} \not\equiv 1 \mod p$, give up and quit. By Euler's criterion z is then a quadratic nonresidue modulo p.
4. Compute the square roots y_1 and y_2 of z modulo p by Shanks' algorithm, return (x, y_1) and (x, y_2) and quit.

The algorithm is apparently polynomial-time and produces a result with an approximate probability of 25%. Recall that Shanks' algorithm produces a result with an approximate probability of 50%.

NB. *By random search we can now find e.g. an element $P \neq O$ of the elliptic curve and a (large) prime r such that $rP = O$, whence the order of P is r (the order of P must divide r anyway). The cyclic subgroup $\langle P \rangle$ is then sufficient for the needs of cryptography. Another (slow) way is to choose a random element P and test its order, which of course should be large. For this we can use a version of Shanks' baby-step-giant-step algorithm. By iterating and using properties of order—see Section 9.1—elements of even higher order may then be found.*

Nevertheless, the issue is quite complicated and use of elliptic curves in cryptography is not straightforward. See for example ROSING *or* BLAKE & SEROUSSI & SMART.

Good references are KOBLITZ and WASHINGTON and e.g. SILVERMAN & TATE or COHEN or CRANDALL & POMERANCE.

[4] See for example WASHINGTON or CRANDALL & POMERANCE.

[5] The original reference is SCHOOF, R.: Elliptic Curves over Finite Fields and the Computation of Square Roots mod p. *Mathematics of Computation* **44** (1985), 483–494. The algorithm is difficult and also difficult to implement.

"In highschool I was very excited that
$a^{\log b} = b^{\log a}$, and still find it useful today."

(NOGA ALON)

Chapter 10

ELGAMAL. DIFFIE–HELLMAN

10.1 Elgamal's Cryptosystem

Elgamal's cryptosystem[1] ELGAMAL can be based on any finite group $G = (A, \odot, \mathbf{1})$ in whose large cyclic subgroups $\langle a \rangle$ discrete logarithm $\log_a$ is difficult to compute. Such groups are for instance $\mathbb{Z}_p^*$ and more generally $\mathbb{F}_{p^n}^*$, in particular $\mathbb{F}_{2^n}^*$, and elliptic curves over finite fields.

The public key is the triple

$$k_1 = (G, a, b)$$

where $b = a^y$. The secret key is $k_2 = y$. Note that the public key holds the information of the secret key because $y = \log_a b$, but it is not easy to obtain it from the public key. Encrypting is nondeterministic. For that we randomly choose a number x from the interval $0 \leq x < l$ where l is the order of a. If it is not wished for l to be published, or it is not known, we can alternatively give some larger upper bound, for example the number of elements G, which has l as a factor, see Lagrange's theorem. The encrypting function is

$$e_{k_1}(w, x) = (a^x, w \odot b^x) = (c_1, c_2).$$

Thus the message block must be interpreted as an element of G. The decrypting function is

$$d_{k_2}(c_1, c_2) = c_2 \odot c_1^{-y}.$$

Decrypting works since

$$d_{k_2}(a^x, w \odot b^x) = w \odot b^x \odot (a^x)^{-y} = w \odot a^{xy} \odot a^{-xy} = w.$$

The idea is to "mask" w by multiplying it by b^x, x is supplied via a^x.

For setting up ELGAMAL in the multiplicative group $\mathbb{Z}_p^*$ of a prime field we choose both p and the primitive root a modulo p simultaneously. Moreover, it is to be kept in mind that $p - 1$ should have a large prime factor so that discrete logarithm cannot be quickly computed (see Section 7.2) e.g. by the Pohlig–Hellman algorithm. This goes in the following way:

1. Choose a large random prime q, and a smaller random number r which can be factored.

2. If $2qr + 1$ is a prime, set $p \leftarrow 2qr + 1$. Note that in this case $p - 1$ has a large prime factor q. Otherwise we return to #1.

[1] The system was developed by Taher Elgamal in 1984. The original reference is ELGAMAL, T.: A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Transactions on Information Theory* **IT–31** (1985), 469–472. Discrete logarithms in $\mathbb{Z}_p^*$ were used in this cryptosystem.

3. Randomly choose a number a from the interval $1 \leq a < p$.

4. Test by Lucas' criterion whether a is a primitive root modulo p. The prime factors of $p-1$ needed here, that is, 2 and q and the known prime factors of r, are now easy to obtain.

5. If a is a primitive root modulo p, choose a random number y from the interval $1 \leq y < p$, return p, a and y, and quit. Otherwise return to #3.

NB. *In a group $\mathbb{Z}_p^*$, using an element b of order much lower than p must be avoided. Otherwise it is easy to try out candidate values r for the order and compute*

$$c_2^r \equiv (wb^x)^r \equiv w^r (b^r)^x \equiv w^r \cdot 1 = w^r \mod p.$$

If the candidate happens to be the correct order of b, the whole cryptosystem is transformed into a deterministic system resembling RSA, possibly easily broken by e.g. the meet-in-the-middle attack, see Section 8.2. An exception is the case where $w \equiv b^i \mod p$ for some i and $c_2^r \equiv 1 \mod p$, but there are very few of these choices if r is small.

10.2 Diffie–Hellman Key-Exchange

ELGAMAL allows many parties to publish their public keys within the same system: Each party just chooses its own y and publishes the corresponding a^y. ELGAMAL is in fact a later modification of one of the oldest public-key systems, the *Diffie-Hellman key-exchange system* DIFFIE–HELLMAN.

The setting here is the same as in ELGAMAL. Each party i again chooses a random number x_i from the interval $0 \leq x_i < l$ or from some larger interval, and publishes a^{x_i}. The common key of the parties i and j is in that case $a^{x_i x_j}$, which they both can compute quickly from the published information and from their own secret numbes.

Breaking DIFFIE–HELLMAN consists of the following two operations. First, compute x_i from a^{x_i}. Second, compute $(a^{x_j})^{x_i} = a^{x_i x_j}$. In this way it is equivalent to solving the following problem:

DHP: Given (G, a, b, c), compute $b^{\log_a c}$.

This problem is the so-called *Diffie–Hellman problem.* The complexity of the Diffie–Hellman problem is not known, computing discrete logarithms naturally solves that too. Note that the order of appearance of b and c does not actually matter since

$$b^{\log_a c} = (a^{\log_a b})^{\log_a c} = (a^{\log_a c})^{\log_a b} = c^{\log_a b}.$$

ELGAMAL's decrypting is also equivalent to the Diffie–Hellman problem. If DHP can be quickly solved, we can first compute

$$b^x = b^{\log_a a^x} = b^{\log_a c_1}$$

quickly and then

$$c_2 \odot b^{-x} = w,$$

and ELGAMAL is broken. On the other hand, if ELGAMAL is broken, we can quickly compute $w = c_2 \odot b^{-x}$ from the cryptotext (c_1, c_2) and the public information, in which case we can also quickly compute

$$b^{\log_a c_1} = b^x = (c_2^{-1} \odot w)^{-1}.$$

Because c_1 is random element of $\langle a \rangle$ this means that DHP can be solved quickly.

10.3 Cryptosystems Based on Elliptic Curves

A finite cyclic subgroup of an elliptic curve can be used to set up Elgamal's cryptosystem. Naturally in this cyclic group discrete logarithm must be difficult to compute or the Diffie-Hellman problem must be difficult to solve. Unfortunately in certain elliptic curves (supersingular elliptic curves) over finite fields these problems are solved relatively quickly by the so-called *Menezes–Okamoto–Vanstone algorithm,* and these must be avoided, see KOBLITZ or WASHINGTON or BLAKE & SEROUSSI & SMART.[2] It might be mentioned that Shanks' baby-step-giant-step algorithm is suitable for computing discrete logarithms in elliptic curves, and so is the Pohlig–Hellman algorithm, but they are not always fast.

One difficulty naturally is that construction of cyclic subgroups of elliptic curves is laborious. Another difficulty is that when ELGAMAL for finite fields approximately doubles the length of message (the pair construction), ELGAMAL for elliptic curves approximately quadruples it. Recall that, by Hasse's theorem, there are approximately as many points in an elliptic curve as there are elements in the field. This is avoided by using a more powerful variant of ELGAMAL, the so-called *Menezes–Vanstone system* MENEZES–VANSTONE. The public key of the system is a triple $k_1 = (E, \alpha, \beta)$ where E is an elliptic curve over a prime field $\mathbb{Z}_p$ where $p > 3$, α is the generating element in a cyclic subgroup of E, and $\beta = a\alpha$. The secret key is $k_2 = a$. A message block is a pair (w_1, w_2) of elements of $\mathbb{Z}_p$ represented in the positive residue system.

The encrypting function is defined in the following way:

$$e_{k_1}((w_1, w_2), x) = (y_0, y_1, y_2)$$

where

$$y_0 = x\alpha \quad , \quad y_1 = (c_1 w_1, \bmod p) \quad , \quad y_2 = (c_2 w_2, \bmod p),$$

x is a random number—compare to ELGAMAL—and the numbers c_1 and c_2 are obtained by representing the point $x\beta = (c_1, c_2)$ of the elliptic curve in the positive residue system. x must be chosen so that $c_1, c_2 \not\equiv 0 \mod p$. The decrypting function is

$$d_{k_2}(y_0, y_1, y_2) = ((y_1 c_2^{-1}, \bmod p), (y_2 c_2^{-1}, \bmod p)).$$

Note that c_1 and c_2 are obtained by a from y_0, since

$$ay_0 = a(x\alpha) = (ax)\alpha = x(a\alpha) = x\beta = (c_1, c_2).$$

The idea is, as in ELGAMAL, to use the elliptic curve to "mask" the message. Like ELGAMAL MENEZES-VANSTONE also approximately doubles the length of message, two elements of $\mathbb{Z}_p$ are encrypted to four.

NB. *Space can also be saved by "compressing" elements of the elliptic curve into smaller space. Compressing and decompressing take more time, though. For example, in the prime field $\mathbb{Z}_p$ an element (point) (x, y) of an elliptic curve can be compressed into (x, i) where $i = (y, \bmod 2)$, since y can be computed from $x^3 + ax + b$ by Shanks' algorithm and choice of sign is determined by i. (If (x, y) is a point of the curve then so is $(x, p - y)$, and $p - y \equiv 1 - y \equiv 1 - i \bmod 2$.)*

[2] It is also an unfortunate feature that the most convenient bit-based finite fields $\mathbb{F}_{2^n}$ seem to be worse than the others. See for example GAUDRY, P. & HESS, F. & SMART, N.P.: Constructive and Destructive Facets of Weil Descent on Elliptic Curves. *Journal of Cryptology* **15** (2002), 19–46. The further we get in the mathemathically quite demanding theory of elliptic curves, the more such weaknesses seem to be revealed.

A third difficulty in using elliptic curves is in encoding messages to points of the curve. One way to do this is the following. We confine ourselves to elliptic curves over the prime field $\mathbb{Z}_p$ here for simplicity, the procedure generalizes to other finite fields, too.

1. Encode the message block first to a number m such that $m + 1 \leq p/100$.
2. Check in the same way as in the algorithm of Section 9.3 whether the elliptic curve has a point (x, y) such that $100m \leq x \leq 100m + 99$.
3. If such a point (x, y) is found, choose it to serve as the counterpart of the message m. Otherwise give up. It may be noted that giving up here is very rare, since it has been shown that the algorithm does it with an approximate probability of $2^{-100} \cong 10^{-30}$.

Of course this procedure slows the encrypting process a notch. Note that decoding is quite fast, though: $m = \lfloor x/100 \rfloor$.

NB. *An advantage of cryptosystems based on elliptic curves, when compared to RSA, is that the currently recommended key-size is much smaller. A "fast" cryptosystem CRANDALL using elliptic curves, patented by Richard Crandall, might be mentioned here, too. It is based on the use of special primes, so-called* Mersenne numbers.

10.4 XTR

A newer quite fast variant of DIFFIE–HELLMAN or ELGAMAL type cryptosystem is obtained in the unit groups of certain finite fields, the socalled *XTR system.*[3] In XTR we work in a cyclic subgroup (of a large size r) of $\mathbb{F}_{p^6}^*$ where p is a large prime and $r \mid p^2 - p + 1$. In such subgroups we can represent the elements in a small space and fast implementations of computing operations are possible. So, the question is mostly just of a suitable choice of the group, regarding implementation. There are other similar procedures, for example the so-called CEILIDH system.

[3]The original reference is LENSTRA, A.K. & VERHEUL, E.R.: The XTR Public Key System. *Proceedings of Crypto '00. Lecture Notes in Computer Science* **1880**. Springer–Verlag (2000), 1–19. The name originates from the words "Efficient Compact Subgroup Trace Representation", got it?

"I had a polynomial once. My doctor removed it."

(MICHAEL GRANT: *Gone*)

Chapter 11

NTRU

11.1 Definition

The *NTRU cryptosystem*[1] is a cryptosystem based on polynomial rings and their residue class rings, which in a way resembles RIJNDAEL. Like RIJNDAEL, it is mostly inspired by the so-called cyclic codes in coding theory, see the course Coding Theory. The construction of NTRU is a bit more technical than that of RSA or ELGAMAL.

In NTRU we first choose positive integers n, p and q where p is much smaller than q and $\gcd(p, q) = 1$. One example choice is $n = 107$, $p = 3$ and $q = 64$. The system is based on the polynomial rings $\mathbb{Z}_p[x]$ and $\mathbb{Z}_q[x]$, and especially on the residue class rings $\mathbb{Z}_p[x]/(x^n - \overline{1})$ and $\mathbb{Z}_q[x]/(x^n - \overline{1})$. See Section 4.2 and note that $x^n - \overline{1}$ is a monic polynomial in both polynomial rings, so we can divide by it.

So, remainders are important when dividing by $x^n - \overline{1}$, that is, polynomials of $\mathbb{Z}_p[x]$ and $\mathbb{Z}_q[x]$ of maximum degree $n-1$. Computing with these in $\mathbb{Z}_p[x]/(x^n - \overline{1})$ and in $\mathbb{Z}_q[x]/(x^n - \overline{1})$ is easy since addition is the usual addition of polynomials and in multiplication

$$x^k \equiv x^{(k, \bmod n)} \mod x^n - \overline{1}.$$

In the sequel we use the following notation. If $P(x)$ is a polynomial with integral coefficients then the polynomial $P_{(m)}(x)$ of $\mathbb{Z}_m[x]$ is obtained from $P(x)$ by reducing its coefficients modulo m. Moreover, such a $P_{(m)}(x)$ —or rather its coefficients—*is represented in the symmetric residue system,* see Section 2.4. Considering addition and multiplication of polynomials we see quite easily that if $R(x) = P(x) + Q(x)$ and $S(x) = P(x)Q(x)$ in $\mathbb{Z}[x]$ then $R_{(m)}(x) = P_{(m)}(x) + Q_{(m)}(x)$ and $S_{(m)}(x) = P_{(m)}(x)Q_{(m)}(x)$ in $\mathbb{Z}_m[x]$. Furthermore, we see that if $P(x) \in \mathbb{Z}[x]$ is of degree no higher than $n-1$ then so is $P_{(m)}(x) \in \mathbb{Z}_m[x]$. In this case the polynomial $P_{(m)}(x)$ can be considered as a polynomial of the residue class ring $\mathbb{Z}_m[x]/(x^n - \overline{1})$.

For setting up the system we choose two secret polynomials $f(x)$ and $g(x)$ of $\mathbb{Z}[x]$, of degree no higher than $n - 1$. From these we get the polynomials $f_{(p)}(x)$ and $g_{(p)}(x)$ of $\mathbb{Z}_p[x]$, and the polynomials $f_{(q)}(x)$ and $g_{(q)}(x)$ of $\mathbb{Z}_q[x]$. As noted, $f_{(p)}(x)$ and $g_{(p)}(x)$ can also be interpreted as polynomials of the residue class ring $\mathbb{Z}_p[x]/(x^n - \overline{1})$. Similarly the polynomials $f_{(q)}(x)$ and $g_{(q)}(x)$ can be interpreted as polynomials of the residue class ring $\mathbb{Z}_q[x]/(x^n - \overline{1})$. Interpreted this way we also require from the polynomials $f_{(p)}(x)$ and $f_{(q)}(x)$ —or from the original polynomial $f(x)$ —that there are polynomials $F_p(x) \in \mathbb{Z}_p[x]$ and $F_q(x) \in \mathbb{Z}_q[x]$ of degree no higher than $n - 1$ such that

$$F_p(x)f_{(p)}(x) \equiv \overline{1} \mod x^n - \overline{1} \quad \text{and} \quad F_q(x)f_{(q)}(x) \equiv \overline{1} \mod x^n - \overline{1}.$$

[1] The origin of the name is unclear, the original reference is HOFFSTEIN, J. & PIPHER, J. & SILVERMAN, J.H.: NTRU: A Ring-Based Public Key Cryptosystem. *Proceedings of ANTS III. Lecture Notes in Computer Science* **1423**. Springer–Verlag (1998), 267–288. The idea is a couple of years older.

In other words, $F_p(x)$ is the inverse of $f_{(p)}(x)$ in $\mathbb{Z}_p[x]/(x^n - \overline{1})$ and $F_q(x)$ is correspondingly the inverse of $f_{(q)}(x)$ in $\mathbb{Z}_q[x]/(x^n - \overline{1})$. Further we compute in $\mathbb{Z}_q[x]$

$$h(x) \equiv F_q(x)g_{(q)}(x) \mod x^n - \overline{1}.$$

Apparently we may assume that the degree of $h(x)$ is at most $n-1$, so it can also be interpreted as a polynomial of the residue class ring $\mathbb{Z}_q[x]/(x^n - \overline{1})$.

Now, the public key is $(n, p, q, h(x))$ and the secret key is $(f_{(p)}(x), F_p(x))$. A message is encoded as an element of $\mathbb{Z}_p[x]/(x^n - \overline{1})$, i.e.. the message is a polynomial $w(x)$ of $\mathbb{Z}_p[x]$ of degree no higher than $n-1$. In particular, $w(x)$ *is represented using the symmetric residue system* modulo p. If $p = 3$ then the coefficients of $w(x)$ are -1, 0 and 1. A $w(x)$ represented this way can be transformed to a polynomial $w_{(q)}(x)$ of $\mathbb{Z}_q[x]$, just reduce the coefficients modulo q. Note that this expressly requires a fixed representation of coefficients!

11.2 Encrypting and Decrypting

For encrypting we choose a random polynomial $\phi(x)$ of maximum degree $n-1$. From this we get the polynomial $\phi_{(p)}(x)$ in the polynomial ring $\mathbb{Z}_p[x]$ and the polynomial $\phi_{(q)}(x)$ in the polynomial ring $\mathbb{Z}_p[x]$, which can be interpreted further as polynomials of the residue class rings $\mathbb{Z}_p[x]/(x^n - \overline{1})$ and $\mathbb{Z}_q[x]/(x^n - \overline{1})$, respectively. Encrypting is performed in $\mathbb{Z}_q[x]/(x^n - \overline{1})$ in the following way:

$$c(x) \equiv p\phi_{(q)}(x)h(x) + w_{(q)}(x) \mod x^n - \overline{1}.$$

In decrypting we first compute

$$a(x) \equiv f_{(q)}(x)c(x) \mod x^n - \overline{1}$$

in $\mathbb{Z}_q[x]/(x^n - \overline{1})$, and *represent the coefficients of* $a(x)$ *in the symmetric residue system modulo* q. Again in this representation $a(x)$ can be transformed to the polynomial $a_{(p)}(x)$ of $\mathbb{Z}_p[x]$ by reducing the coefficients modulo p. After this the message itself is ideally obtained by computing

$$w'(x) \equiv F_p(x)a_{(p)}(x) \mod x^n - \overline{1}$$

in $\mathbb{Z}_p[x]/(x^n - \overline{1})$, and by representing the coefficients of $w'(x)$ using the symmetric residue system modulo p.

But it is not necessarily true that $w'(x) = w(x)$! Decrypting works only for a suitable choice of the polynomials used—at least with high probability. First of all, we note that in $\mathbb{Z}_q[x]/(x^n - \overline{1})$

$$\begin{aligned} a(x) &\equiv f_{(q)}(x)c(x) \equiv f_{(q)}(x)(p\phi_{(q)}(x)h(x) + w_{(q)}(x)) \\ &\equiv pf_{(q)}(x)F_q(x)\phi_{(q)}(x)g_{(q)}(x) + f_{(q)}(x)w_{(q)}(x) \\ &\equiv p\phi_{(q)}(x)g_{(q)}(x) + f_{(q)}(x)w_{(q)}(x) \mod x^n - \overline{1}. \end{aligned}$$

If now p is much smaller than q and the absolute values of the coefficients of the polynomials $\phi(x)$, $g(x)$, $f(x)$ and $w(x)$ are small, it is highly probable that in computing $p\phi_{(q)}(x)g_{(q)}(x) + f_{(q)}(x)w_{(q)}(x) \mod x^n - \overline{1}$ coefficients need not be reduced modulo q at all when representig them in the symmetric residue system modulo q. (Recall the "easy" multiplication above!) From this it follows that the polynomials $\phi_{(p)}(x)$, $g_{(p)}(x)$ and $f_{(p)}(x)$ are also obtained from the polynomials $\phi_{(q)}(x)$, $g_{(q)}(x)$ and $f_{(q)}(x)$ by just taking their coefficients modulo p—all coefficients being again represented in the symmetric residue system—and that

$$a_{(p)}(x) \equiv p\phi_{(p)}(x)g_{(p)}(x) + f_{(p)}(x)w(x) \equiv f_{(p)}(x)w(x) \mod x^n - \overline{1}$$

in $\mathbb{Z}_p[x]/(x^n - \overline{1})$. Hence (again in $\mathbb{Z}_p[x]/(x^n - \overline{1})$) it is very probable that

$$w'(x) \equiv F_p(x)a_{(p)}(x) \equiv F_p(x)f_{(p)}(x)w(x) \equiv w(x) \mod x^n - \overline{1},$$

i.e. decrypting succeeds.

11.3 Setting up the System

So, errorless decrypting is not automatic but requires that the parameters and polynomials used are chosen conveniently, and even then only with high probability. Denote by $\mathcal{P}_{n,i,j}$ the set of the polynomials of degree no higher than $n-1$ such that i coefficients are $= 1$, j coefficients are $= -1$ and the remaining coefficients are all $= 0$. The following choices are recommended:

n	p	q	$f(x)$	$g(x)$	$\phi(x)$
107	3	64	$\in \mathcal{P}_{107,15,14}$	$\in \mathcal{P}_{107,12,12}$	$\in \mathcal{P}_{107,5,5}$
167	3	128	$\in \mathcal{P}_{167,61,60}$	$\in \mathcal{P}_{167,20,20}$	$\in \mathcal{P}_{167,18,18}$
503	3	256	$\in \mathcal{P}_{503,216,215}$	$\in \mathcal{P}_{503,72,72}$	$\in \mathcal{P}_{503,55,55}$

If—as above—$p = r_1^{i_1}$ and $q = r_2^{i_2}$ where r_1 and r_2 are different primes, the polynomial $f(x)$ and its inverses $F_p(x)$ and $F_q(x)$ can be found by the following procedure. (Otherwise the procedure is further complicated by use the Chinese remainder theorem.)

1. Take a random polynomial $f(x)$ with integral coefficients whose degree is at most $n-1$ (possibly as indicated in the table above).

2. Check using the Euclidean algorithm that $\gcd(f_{(r_1)}(x), x^n - \overline{1}) = \overline{1}$ in $\mathbb{Z}_{r_1}[x]$ and that $\gcd(f_{(r_2)}(x), x^n - \overline{1}) = \overline{1}$ in $\mathbb{Z}_{r_2}[x]$, see Section 4.2. If this is not true, give up.

3. Then by Bézout's theorem we get, by using the Euclidean algorithm, polynomials $h_1(x)$, $k_1(x), l_1(x)$ and $h_2(x), k_2(x), l_2(x)$ with integral coefficients such that

   $$1 = h_1(x)f(x)+k_1(x)(x^n-1)+r_1l_1(x) \quad \text{and} \quad 1 = h_2(x)f(x)+k_2(x)(x^n-1)+r_2l_2(x)$$

   where $h_1(x)$ are $h_2(x)$ of maximum degree k, $k_1(x)$, and $k_2(x)$ of maximum degree $n-1$, and $l_1(x)$ and $l_2(x)$ of maximum degree $2n-1$. In addition we may apparently assume that the coefficients of the polynomials $h_1(x), k_1(x)$ and $h_2(x), k_2(x)$ are in the symmetric residue systems modulo r_1 and r_2, respectively.

4. Denote $j_1 = \lceil \log_2 i_1 \rceil$ and $j_2 = \lceil \log_2 i_2 \rceil$, whence $2^{j_1} \geq i_1$ and $2^{j_2} \geq i_2$.

5. Compute[2]

   $$F_p(x) \equiv h_1(x) \prod_{m=0}^{j_1-1} (\overline{1} + r_1^{2^m} l_1(x)^{2^m}) \mod x^n - \overline{1} \qquad \text{in } \mathbb{Z}_p[x]/(x^n-1)$$

   and

   $$F_q(x) \equiv h_2(x) \prod_{m=0}^{j_2-1} (\overline{1} + r_2^{2^m} l_2(x)^{2^m}) \mod x^n - \overline{1} \qquad \text{in } \mathbb{Z}_q[x]/(x^n-1),$$

   return the results and $f(x)$ and quit.

[2]This operation is the so-called *Hensel lift*. The empty products occuring in the cases $j_1 = 0$ and $j_2 = 0$ are $\equiv \overline{1}$.

The procedure usually produces a result immediately. The result is correct, since (verify!)

$$F_p(x)f_{(p)}(x) \equiv \overline{1} - r_1^{2^{j_1}} l_1(x)^{2^{j_1}} \equiv \overline{1} \mod x^n - \overline{1} \qquad \text{in } \mathbb{Z}_p[x]/(x^n-1)$$

and

$$F_q(x)f_{(q)}(x) \equiv \overline{1} - r_2^{2^{j_2}} l_1(x)^{2^{j_2}} \equiv \overline{1} \mod x^n - \overline{1} \qquad \text{in } \mathbb{Z}_q[x]/(x^n-1).$$

The polynomial $g(x)$ is chosen randomly (say, within the limits allowed by the table).

11.4 Attack Using LLL Algorithm

NTRU uses polynomials of degree no higher than $n-1$, which can be interpreted as n vectors (here column vectors). For these polynomials

$$\begin{aligned} f(x) &= f_0 + f_1 x + \cdots + f_{n-1}x^{n-1}\,, \\ g(x) &= g_0 + g_1 x + \cdots + g_{n-1}x^{n-1} \quad \text{and} \\ h(x) &= h_0 + h_1 x + \cdots + h_{n-1}x^{n-1} \end{aligned}$$

the vectors are

$$\mathbf{f} = (f_0, f_1, \ldots, f_{n-1}) \quad , \quad \mathbf{g} = (g_0, g_1, \ldots, g_{n-1}) \quad \text{and} \quad \mathbf{h} = (h_0, h_1, \ldots, h_{n-1}).$$

As above

$$h(x) \equiv F_q(x)g_{(q)}(x) \mod x^n - \overline{1}\,, \quad \text{i.e.} \quad f_{(q)}(x)h(x) \equiv g_{(q)}(x) \mod x^n - \overline{1}$$

in $\mathbb{Z}_q[x]/(x^n-\overline{1})$. Remember that $F_q(x)$ is the inverse of $f_{(q)}(x)$ in $\mathbb{Z}_q[x]/(x^n-\overline{1})$. If we take the matrix

$$\mathbf{H} = \begin{pmatrix} h_0 & h_1 & \cdots & h_{n-1} \\ h_{n-1} & h_0 & \cdots & h_{n-2} \\ \vdots & \vdots & \ddots & \vdots \\ h_1 & h_2 & \cdots & h_0 \end{pmatrix}$$

then the above equation can be written in the form

$$\mathbf{fH} \equiv \mathbf{g} \mod q.$$

Note how the structure of the matrix $\mathbf{H}$ nicely handles reduction modulo $x^n - \overline{1}$.

The vectors above bring to mind lattices. The dimension of a suitable lattice is however $2n$. Now let's take the $2n \times 2n$ matrix

$$\mathbf{M} = \left(\begin{array}{c|c} \delta\mathbf{I}_n & \mathbf{H} \\ \hline \mathbf{O}_n & -q\mathbf{I}_n \end{array}\right)$$

(in block form) where $\mathbf{I}_n$ is the $n \times n$ indentity matrix , $\mathbf{O}_n$ is the $n \times n$ zero matrix and $\delta \neq 0$ is a real number. Clearly $\mathbf{M}$ is nonsingular, denote the lattice generated by its rows by $\langle \mathbf{M} \rangle$. Note that $\mathbf{M}$ is obtained from the public key.

Because $f_{(q)}(x)h(x) \equiv g_{(q)}(x) \mod x^n - \overline{1}$, then in $\mathbb{Z}[x]/(x^n-1)$

$$f(x)h(x) \equiv g(x) + qk(x) \mod x^n - 1$$

for some polynomial $k(x)$ with integral coefficients of degree at most $n-1$. When $k(x)$ is represented as above as an $n-1$-dimensional column vector $\mathbf{k}$, this equation can also be written in the form

$$\mathbf{fH} = \mathbf{g} + q\mathbf{k}.$$

Furthermore in matrix form we get the equation

$$\left(\mathbf{f}\middle|\mathbf{k}\right)\mathbf{M} = \left(\delta\mathbf{f}\middle|\mathbf{g}\right).$$

This shows that the $2n$-vector $\left(\delta\mathbf{f}\middle|\mathbf{g}\right)$ is in the lattice $\langle\mathbf{M}\rangle$. Because the coefficients of $f(x)$ and $g(x)$ are small, we are talking about a short vector of the lattice. By a convenient choice of the number δ we can make it even shorter. If $\left(\delta\mathbf{f}\middle|\mathbf{g}\right)$ is short enough, it can often be found by the LLL algorithm and used to break the system.

NB. *The recommended parameters of NTRU above are chosen precisely to prevent this kind of attacks by the LLL algorithm. As of now no serious weaknesses in NTRU have been found, despite some claims to the opposite. It should be mentioned that, unlike RSA and ELGAMAL, it is not known either that NTRU could be broken using quantum computing, see Chapter 15.*

”There’s just one thing,” said Ginger.
”What’s that?” ”I might make hash of it.”
”Heavens Ginger! There must be something
in this world that you wouldn’t make hash of.”

(P.G. WODEHOUSE: *The Adventures of Sally*)

Chapter 12

HASH FUNCTIONS AND HASHES

12.1 Definitions

A *hash* is a word of fixed length that describes a message ”accurately enough”. The message can then be quite long. The procedure which gives the hashing is called a *hash function.* Because the number of possible hashes is smaller than the number of messages, a hash fucntion is not one-to-one, in other words, in some cases it gives the same hash for several messages. This is called *collision.* For a hash function to be usable it should naturally be quickly computable from the message, but also such that a hostile party cannot efficiently take advantage of collisions in any way. Bearing this in mind we define several different concepts:

- A hash function h is *weakly collision-free for the message* w if it is computationally hard to find another message w' such that $h(w) = h(w')$.
- A hash function h is *weakly collision-free* if for any given message w it is computationally hard to find another message w' such that $h(w) = h(w')$.
- A hash function h is *strongly collision-free* if it is computationally hard to find messages w and w' such that $h(w) = h(w')$, in other words, if it is hard to find a message w for which h is *not* weakly collision-free.
- A hash function h is *one-way* if for any given hash t it is hard to find a message w such that $h(w) = t$.

These definitions are not quite exact in that we do not consider computational complexity here. If the message space is finite—as it usually is—complexity, being an asymptotic concept, cannot really be defined at all.

NB. *Other nomenclatures are used too. Weakly collision-free hash functions are also called* second preimage resistant, *strongly collision-free hash functions are also called just* collision-free, *and one-way hash functions are also called* preimage resistant.

There is a connection between one-way and strongly collision-free hashing:

Theorem 12.1. *If the message space W is finite and the hash space is T and $|W| \geq 2|T|$, where $|\cdot|$ denotes cardinality of sets, then a strongly collision-free hash function h is one-way. To put it more exactly, an algorithm A which inverts h can be transformed to a Las Vegas type probabilistic algorithm which finds a collision with at least probability $1/2$.*

Proof. Denote by M_w the set of the messages with the same hash as w, and by $\mathcal{D}$ the family of all these sets. Then

$$|\mathcal{D}| = |T| \quad \text{and} \quad \sum_{D\in\mathcal{D}} |D| = |W|.$$

The following Las Vegas algorithm finds a collision or gives up.

1. Choose a random message $w \in W$.
2. Compute the hash $t = h(w)$.
3. Find a message w' such that $h(w') = t$ using the algorithm A.
4. If $w' \neq w$, return w and w' and quit. Otherwise give up and quit.

We just need to show that the algorithm gives a result with at least probability $1/2$:

$$\begin{aligned}
\mathrm{P}(\text{A collision is found.}) &= \sum_{w\in W} \frac{|M_w|-1}{|M_w|}\frac{1}{|W|} = \frac{1}{|W|}\sum_{D\in\mathcal{D}}\sum_{w\in D}\frac{|D|-1}{|D|} \\
&= \frac{1}{|W|}\sum_{D\in\mathcal{D}}(|D|-1) = \frac{1}{|W|}\left(\sum_{D\in\mathcal{D}}|D| - \sum_{D\in\mathcal{D}}1\right) = \frac{|W|-|T|}{|W|} \\
&\geq \frac{|W|-|W|/2}{|W|} = \frac{1}{2}.
\end{aligned}$$

□

It is obvious that for extensively and continuously used hash functions strong collisions should not occur essentially at all. Because of this it was quite a surprise, when in 2004 the Chinese Xiaoyun Wang, Dengguo Feng, Xuejia Lai and Hongbo Yu found collisions in many commonly used hash functions. In addition to that, Wang, Yiqun Lisa Yin and Yu noted that collisions can be found relatively easily even in SHA-1[1], the "flagship" of hash functions. Developing good hash functions appears to be even more difficult than it was thought.

12.2 Birthday Attack

If the number of possible hashes is small, collisions can be found by trying out: Just choose k random messages $w_1, \ldots, w_k$, compute the hashes $t_i = h(w_i)$, and check whether collisions occur. This simple procedure is called the *birthday attack*[2]. Now let's estimate probabilities for the birthday attack to succeed. In this case we may assume that different hashes occur

[1]This "Chinese attack" is discussed in many talks in the references *Proceedings of Crypto '05. Lecture Notes in Computer Science* **3621**. Springer–Verlag (2005) ja *Proceedings of EuroCrypt '05. Lecture Notes in Computer Science* **3494**. Springer–Verlag (2005).

[2]The name comes from the fact that if we have large enough group of people then the probability of at least two of them having the same birthday (day of the year) is high. Using approximation and noting that $1.177\sqrt{365} \cong 22.49$ it is seen that it suffices to have at least 23 people in the group for the probability of same birthdays to be at least $1/2$. In this case the exact computation gives

$$P = \left(1-\frac{1}{365}\right)\left(1-\frac{2}{365}\right)\cdots\left(1-\frac{23-1}{365}\right) \cong 0.493.$$

with at least approximately equal frequency. Otherwise the probability of finding collisions just increases. The probability for no collisions to occur is apparently

$$P_{n,k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{n^k} = \left(1-\frac{1}{n}\right)\left(1-\frac{2}{n}\right)\cdots\left(1-\frac{k-1}{n}\right)$$

where n is the number of hashes. Since it is well-known that $\lim_{n\to\infty}\left(1+\frac{a}{n}\right)^n = e^a$, we obtain further the estimate

$$P_{n,k}^{\frac{1}{n}} \cong e^{-1-2-\cdots-(k-1)} = e^{-\frac{k(k-1)}{2}}.$$

(Here n is of course large and much larger than k.) Hence the probability of finding at least one collision is

$$Q_{n,k} = 1 - P_{n,k} \cong 1 - e^{-\frac{k(k-1)}{2n}}.$$

This way we get an estimate for k when $Q_{n,k} = Q$ is given:

$$-\frac{k(k-1)}{2n} \cong \ln(1-Q)$$

or

$$k^2 - k + 2n\ln(1-Q) \cong 0$$

or

$$k \cong \frac{1}{2}\left(1+\sqrt{1-8n\ln(1-Q)}\right).$$

By choosing $Q = 1/2$ we conclude that a collision is found with probability $1/2$ if

$$k \cong \frac{1}{2}\left(1+\sqrt{1+8n\ln 2}\right) \cong \sqrt{2n\ln 2} \cong 1.177\sqrt{n}.$$

Thus for example for a 40-bit hash the birthday attack succeeds with probability $1/2$ if k is slightly larger than $2^{20} = 1\,048\,576$. Consequently, hashes should be significantly longer, for instance in SHA-1 hash length is 160 bits, and then k should be slightly larger than $2^{80} \cong 1.2 \cdot 10^{24}$ for the birthday attack to succeed. On the other hand, the "Chinese attack" shows somewhat amazingly that a k of order $2^{69} \cong 5.9 \cdot 10^{20}$ may already suffice.

Birthday attacks sometimes occur in a bit different form, which goes as follows. We first choose k_1 messages $w_1, \ldots, w_{k_1}$ randomly, and then independently another k_2 random messages $w'_1, \ldots, w'_{k_2}$, and seek collisions of the form $h(w_i) = h(w'_j)$, so-called *cross-collisions.* Denote the possible cases by the symbols

$$\begin{aligned} T_1 &= \text{"There is a collision in the messages } w_1, \ldots, w_{k_1}\text{."} \\ T_2 &= \text{"There is a collision in the messages } w'_1, \ldots, w'_{k_2}\text{."} \\ T_{12} &= \text{"There is a cross-collision."} \end{aligned}$$

and the complementary cases by overlining as usual. Apparently then for example

$$\mathrm{P}(T_1) = Q_{n,k_1} \quad , \quad \mathrm{P}(\overline{T}_2) = P_{n,k_2} \quad , \quad \mathrm{P}(\overline{T}_1 \text{ and } \overline{T}_2) = P_{n,k_1}P_{n,k_2} \quad \text{etc.}$$

Further, apparently

$$\mathrm{P}(\overline{T}_1 \text{ and } \overline{T}_2 \text{ and } \overline{T}_{12}) = P_{n,k_1+k_2}.$$

By the rules for probabilities, from this we get the conditional probability

$$\mathrm{P}(\overline{T}_{12} \mid \overline{T}_1 \text{ and } \overline{T}_2) = \frac{P_{n,k_1+k_2}}{P_{n,k_1}P_{n,k_2}} \cong \frac{e^{-\frac{(k_1+k_2)(k_1+k_2+1)}{2n}}}{e^{-\frac{k_1(k_1+1)+k_2(k_2+1)}{2n}}} = e^{-\frac{k_1k_2}{n}}.$$

On the other hand, it is very unlikely that many collisions occur, and a few collisions really do not change the probability of a cross-collision by much, as compared to the situation where no collisions occur. (Remember that n is large and that k_1 and k_2 are small compared to it.) Hence

$$\mathrm{P}(\overline{T}_{12} \mid T_1 \text{ or/and } T_2) \cong e^{-\frac{k_1 k_2}{n}} \quad \text{and so} \quad \mathrm{P}(\overline{T}_{12}) \cong e^{-\frac{k_1 k_2}{n}}.$$

So, if we want the probability of cross-collision to be $1/2$ we should choose (verify!)

$$k_1 k_2 \cong n \ln 2.$$

Hence it is enough to choose

$$k_1, k_2 \cong \sqrt{n \ln 2} \cong 0.833\sqrt{n}.$$

This latter type birthday attack resembles Shanks' baby-step-giant-step algorithm in some ways, see Section 9.2. As a matter of fact, a very similar probabilistic algorithm for computing discrete logarithms can be derived from it. The baby-step-giant-step algorithm of course has the advantage of being deterministic, and even somewhat faster. On the other hand, modular exponentiation is a randomizing operation, so it can be used in the random choices, and we get a powerful and very space-efficient probabilistic algorithm for computing the discrete logarithm $b = \log_g a$ modulo p (a prime):

Pollard's kangaroo algorithm:

1. Denote $J = \lfloor \log_2 p \rfloor$ and $N = \lfloor \sqrt{p} \rfloor$, and choose the numbers c and c' randomly from the interval $0, 1, \ldots, p-1$. (Note that J and N are quickly computed.)

2. Compute the number t_N using the recursion

   $$t_i = (t_{i-1} g^{2^{(t_{i-1}, \bmod J)}}, \bmod p) \quad , \quad t_0 = (g^c, \bmod p).$$

   (Because c is known, these recursion steps are called jumps of a *tame kangaroo.*) If we denote

   $$d = \sum_{i=0}^{N} 2^{(t_i, \bmod J)}$$

   then $t_N = (g^{c+d}, \bmod p)$.

3. Compute the numbers

   $$w_j = (w_{j-1} g^{2^{(w_{j-1}, \bmod J)}}, \bmod p) \quad , \quad w_0 = (g^{b+c'}, \bmod p) = (a g^{c'}, \bmod p)$$

   one by one using recursion. (Because b is not known, these steps are called jumps of a *wild kangaroo.*) Simultaneously we compute the numbers

   $$D_j = D_{j-1} + 2^{(w_j, \bmod J)} \quad , \quad D_0 = 0,$$

   recursively, whence $w_j = (g^{b+c'+D_j}, \bmod p)$.

4. If we find a value $l \leq N$ such that $w_l = t_N$ (cross-collision) then

   $$g^{c+d} \equiv g^{b+c'+D_l} \mod p \,, \quad \text{i.e.} \quad g^{c+d-c'-D_l} \equiv a \mod p.$$

   In this case we return $b = (c + d - c' - D_l, \bmod p - 1)$ and quit. Then again, if we have computed all the numbers $w_0, w_1, \ldots, w_N$ without any cross-collisions occuring, we give up and quit.

By the birthday attack principle, a cross-collision is found in this situation with at least probability $1/2$. Note that if a cross-collision is found already for some t_i and w_l, where $l \leq i < N$, then it is also found for t_N because the recursions are identical. By repeating the algorithm many times choosing a new random c' each time, but not a new c, it is very likely that we will eventually be able to compute b. However, because the number of steps needed is $O(\sqrt{p}\,)$, this is not a polynomial-time algorithm, although it is fast. On the other hand, no lists are stored—compare to the baby-step-giant-step algorithm—so the space needed is very small.

12.3 Chaum–van Heijst–Pfitzmann Hash

As an example of a simple hash function we consider the *Chaum–van Heijst–Pfitzmann hash function* h_{CHP}. For this we need a prime p such that $q = (p-1)/2$ is also a prime, i.e. a Germain number, see Section 8.2. Furthermore we need two different primitive roots α and β modulo p. In addition we assume that the discrete logarithm $a = \log_\alpha \beta$ cannot be computed easily. A message (w_1, w_2) consists of two numbers w_1 and w_2 in the interval $0, 1, \ldots, q-1$, and

$$h_{\mathrm{CHP}}(w_1, w_2) = (\alpha^{w_1}\beta^{w_2}, \bmod p).$$

Finding even one collision of h_{CHP} makes it possible to compute the discrete logarithm $\log_\alpha \beta$ fast:

Theorem 12.2. *If different messages (w_1, w_2) and (w_1', w_2') are known such that $h_{\mathrm{CHP}}(w_1, w_2) = h_{\mathrm{CHP}}(w_1', w_2')$ then the discrete logarithm a can be computed fast.*

Proof. The hashes are the same, that is,

$$\alpha^{w_1}\beta^{w_2} \equiv \alpha^{w_1'}\beta^{w_2'} \mod p.$$

Because $\beta \equiv \alpha^a \mod p$, this is equal to

$$\alpha^{a(w_2-w_2')-(w_1'-w_1)} \equiv 1 \mod p.$$

α is a primitive root modulo p, so $a(w_2 - w_2') - (w_1' - w_1)$ is divisible by its order modulo p, i.e. by $p-1$, see Theorem 7.4 (ii). Therefore

$$a(w_2 - w_2') \equiv w_1' - w_1 \mod p-1.$$

Now let's denote $d = \gcd(w_2 - w_2', p-1)$. Then, by the above congruence, d is also a factor of $w_1 - w_1'$. From this it follows that $w_2 \neq w_2'$. Namely, if $w_2 = w_2'$ then $w_1 \neq w_1'$ and $d = p-1$. This is however impossible since $|w_1 - w_1'| < q < p-1$.

We denote further

$$u = \frac{w_2 - w_2'}{d} \quad , \quad v = \frac{w_1' - w_1}{d} \quad \text{and} \quad r = \frac{p-1}{d}.$$

Then $\gcd(u, r) = 1$ and, by Theorem 2.11,

$$au \equiv v \mod r\,, \quad \text{i.e.} \quad a \equiv u^{-1}v \mod r.$$

Thus the possible values of a in the positive residue system modulo $p-1$ are

$$a = (u^{-1}v, \bmod r) + ir \quad (i = 0, 1, \ldots, d-1).$$

On the other hand, the possible values of d are 1, 2, q and $p-1$. Because $w_2 \neq w_2'$ and $|w_2 - w_2'| < q < p-1$, either $d = 1$ or $d = 2$. So the discrete logarithm a is easy to find, it is either $(u^{-1}v, \bmod r)$ or $(u^{-1}v, \bmod r) + r$. □

Thus h_{CHP} is strongly collision-free and by Theorem 12.1 it is also one-way.

NB. *The CHP hash function is too slow to be very useful, many other hash functions are much faster to compute. Another problem lies in the difficulty of finding enough Germain's numbers. On the other hand, as the "Chinese attack" shows, more and more weaknesses are found in fast hash functions.*

"Have you ever observed that we pay much more attention to a wise passage when it is quoted than when we read it in the original author?"

(PHILIP G. HAMERTON)

Chapter 13

SIGNATURE

13.1 Signature System

A *signature system* is a quintet (P, A, K, S, V), where

- P is the finite *message space.*
- A is the finite *signature space.*
- K is the finite *key space.* Each *key* is a pair $(k_{\mathrm{s}}, k_{\mathrm{v}})$ where k_{s} is the secret *signing key* and k_{v} is the public *verifying key.*
- For each signing key k_{s} there is a *signing function* $s_{k_{\mathrm{s}}} \in S$. For a message w we have $s_{k_{\mathrm{s}}}(w) = (w, u)$ where u is the *signature* of the message w. S is the space of all possible signing functions.
- For each verifying key k_{v} there is a *verifying function* $v_{k_{\mathrm{v}}} \in V$. V is the space of all possible verifying functions.
- For each message w and for a key $(k_{\mathrm{s}}, k_{\mathrm{v}})$ we have

$$v_{k_{\mathrm{v}}}(w, u) = \begin{cases} \text{CORRECT if } s_{k_{\mathrm{s}}}(w) = (w, u) \\ \text{FALSE otherwise.} \end{cases}$$

The public verifying key is left available for everyone to use, the secret signing key is personal and only the signer has it. The signed message is $s_{k_{\mathrm{s}}}(w) = (w, u)$. If a receiver wants he/she can verify the signature by the verifying function. Usually a suitable hashing $h(w)$ of the message w is used when signing. This has the advantage of allowing the message to be quite long.

The signature must satisfy the following basic conditions:

- An outside party who does not know the signing key, cannot send a signed message that can be verified in the name of a real signer, or at least such a message should not contain any meaningful information. In particular, an outside party cannot detach a signature from a real signed message and use it as the signature of another message.
- The signer cannot later on deny having signed a correctly signed message.

Many cryptosystems can immediately be transformed to signature systems, and have in fact originally been signature systems.

13.2 RSA Signature

A signature system is obtained from RSA by defining

$$k_{\mathrm{s}} = (n, b) \quad \text{and} \quad k_{\mathrm{v}} = (n, a),$$

and

$$s_{k_{\mathrm{s}}}(w) = (w, (w^b, \bmod n)) \quad \text{and} \quad v_{k_{\mathrm{v}}}(w, u) = \begin{cases} \text{CORRECT, if } w \equiv u^a \mod n \\ \text{FALSE otherwise.} \end{cases}$$

Apparently faking this signature in one way or another is equivalent to breaking RSA. An outside party can however choose a signature u by taking $w = (u^a, \bmod n)$ as the message. Such a message does not contain any information, though. Even this does not work if an one-way hash function h is used. In that case $k_{\mathrm{v}} = (n, a, h)$ and

$$s_{k_{\mathrm{s}}}(w) = (w, (h(w)^b, \bmod n)) \quad \text{and} \quad v_{k_{\mathrm{v}}}(w, u) = \begin{cases} \text{CORRECT if } h(w) \equiv u^a \mod n \\ \text{FALSE otherwise.} \end{cases}$$

RSA can also be used to get a so-called *blind signature.* If A wishes to sign a message w of B, without knowing its content, the procedure is the following:

1. B chooses a random number l such that $\gcd(l, n) = 1$, computes the number $t = (l^a w, \bmod n)$ and sends it to A.
2. A computes the signature $u' = (t^b, \bmod n)$ as if the message would be t, and sends it to B.
3. B computes the number $u = (l^{-1}u', \bmod n)$.

Because A does not know the number l, he/she does not get any information about the message w. On the other hand, u is the correct signature of the message w, since

$$l^{-1}u' \equiv l^{-1}t^b \equiv l^{-1}l^{ab}w^b \equiv l^{-1}lw^b \equiv w^b \mod n.$$

13.3 Elgamal's Signature

Elgamal's cryptosystem can be transformed into a signature system by choosing the group $G = \mathbb{Z}_p^*$, where p is large prime, a primitive root a modulo p and $b = (a^y, \bmod p)$. The verifying key is now $k_{\mathrm{v}} = (p, a, b)$ and the signing key is $k_{\mathrm{s}} = (p, a, y)$. The signing function is $s_{k_{\mathrm{s}}}(w) = (w, c, d)$ where

$$c = (a^x, \bmod p) \quad \text{and} \quad d = ((w - yc)x^{-1}, \bmod p - 1)$$

and x is a random number, chosen from the interval $1 \leq x < p-1$, such that $\gcd(x, p-1) = 1$. Now $xd = w - yc + k(p-1)$ for some number k. The verifying function is

$$v_{k_{\mathrm{v}}}(w, c, d) = \begin{cases} \text{CORRECT if } b^c c^d \equiv a^w \mod p \\ \text{FALSE otherwise.} \end{cases}$$

Verifying a correct signature will then succeed, since by Fermat's little theorem

$$b^c c^d \equiv a^{yc} a^{xd} = a^{yc+w-yc+k(p-1)} = a^w (a^{p-1})^k \equiv a^w \cdot 1 = a^w \mod p.$$

To forge a signature one should be able to compute c and d without knowing y and x. We then note the following:

- If the forger first chooses some c and then tries to obtain the corresponding d, he/she must compute $\log_c(a^w b^{-c})$ modulo p. This is essentially computing the discrete logarithm in G. Note that because $\gcd(x, p-1) = 1$, also c is a primitive root modulo p, see Theorem 7.4 (iii).

- Then again, if the forger chooses first some d and then tries to find the corresponding c, he/she must solve the equation

$$b^c c^d \equiv a^w \mod p.$$

  No fast algorithms are known for solving such equations.

- If the forger tries to send a signed message, even a random one, he/she might try to first choose c and d and then find some suitable w. But in this case he/she must compute $\log_a(b^c c^d)$ modulo p.[1]

NB. *DSS (Digital Signature Standard), a modification of Elgamal's signature, is quite extensively used, see e.g.* STINSON *or* MENEZES & VAN OORSCHOT & VANSTONE.

13.4 Birthday Attack Against Signature

If hashing is used in signing and it is possible to change the message a little bit here and there without essentially altering its meaning, it is also possible to apply a birthday attack to get cross-collisions in the following way, see Section 12.2:

1. If the length of the hashes used in signing is B bits, the forger finds, say, $B/2 + 2$ places where the message to be signed can be changed without really changing it essentially—for example adding or removing commas and spaces, making small innocent mistakes and so on. This way $2^{B/2+2}$ versions of the correct message are obtained, the hashes of which the forger then computes.

2. Correspondingly, the forger finds $B/2 + 2$ places in the fake message he/she chooses, where it can be varied without changing the meaning, and computes the $2^{B/2+2}$ hashes of the fake messages obtained this way.

3. The forger seeks a possible cross-collision in these two hash sets by sorting in the same way as in the baby-step-giant-step algorithm. It can be found very certainly, if the hashes of the messages may be considered as having been born randomly, since the probability of success is in this case approximately

$$1 - e^{-\frac{2^{B/2+2} 2^{B/2+2}}{2^B}} = 1 - e^{-16} =\cong 0.999\,999\,887.$$

   The condition considering randomness is not very demanding, since a good hash function is already randomizing and small differences in messages cause large differences in hashes.

4. The forger leaves the version of the correct message occurring in the cross-collision to be signed. If the signer does not notice the difference or simply does not care, the forger now has a version of the fake message he/she chose which has the very same hash, and gets it signed by the signer as well!

[1] There are however other ways for obtaining a random signed message! It is also possible to sign some other random messages by using a single received signature. See STINSON.

”Using encryption on the Internet is the equivalent of arranging an armored car to deliver credit card information from someone living in a cardboard box to someone living on a park bench.”

(GENE SPAFFORD)

Chapter 14

TRANSFERRING SECRET INFORMATION

14.1 Bit-Flipping and Random Choices

Generating of random bit (”bit-flipping”) is easy, if we have trusted party to perform it. If such a party is not available, bit-flipping is still possible by a proper method. In what follows in the bit-flipping procedure[1] A flips a random bit for B. At first only B knows the result but if he chooses to do so, he can tell it to A. Even if B does not tell the result to A, he still can't change the bit he got and this way he can't cheat by telling the wrong bit to A, without it being revealed to A at some point. This way B is *committed to the bit* that he got.

The procedure works in the following way, see Section 7.6:

1. A chooses two different large primes p and q and sends the product $n = pq$ to B.

2. B randomly chooses a number u from the interval $1 < u < n/2$ and sends the modular square
$$z = (u^2, \bmod n)$$
to A.

3. A computes the four square roots of z modulo n:
$$(\pm x, \bmod n) \quad \text{and} \quad (\pm y, \bmod n).$$
This is possible since A knows the factors of n. Denote the smaller of the numbers $(\pm x, \bmod n)$ by x', and correspondingly the smaller of the numbers $(\pm y, \bmod n)$ by y'. Then u is one of the numbers x' and y'.

4. A cannot know which of the numbers x' and y' is u, so she guesses. It is of no use for A to send B the number she guessed, because if it happens not to be u then B can factor n. Instead A finds the first bit on the right in which the binary representations of x' and y' differ, and sends this bit to B in the form ”The j^{th} bit of your number is ...”.

5. B tells A if the guess was correct (the flipped bit is 1) or incorrect (the flipped bit is 0). Even if B does not tell the result to A, he is still bound to it and cannot change it.

[1] The original reference is BLUM, M.: Coin Flipping by Telephone. A Protocol for Solving Impossible Problems. *SIGACT News* (1981), 23–27.

6. Finally B reveals u to A and A reveals the factorization of n. B cannot fool A, since he only knows one of the square roots x' and y', otherwise B would be able to factor n.

NB. *As is usual, it is here assumed that when choosing a number u randomly we won't get a number such that $\gcd(u, n) \neq 1$. Indeed, this is highly unlikely if n is large.*

Generalizing, we can choose a random integer from a given interval by flipping the bits of its binary representation one by one, and removing initial zeros if needed.

Another random choice situation is when, for both A and B, k numbers from the numbers $1, 2, \ldots, N$ are chosen randomly such that both know their own numbers but not the numbers of the other. Furthermore, it is required that A and B don't share any of the numbers. If the above bit-flipping might be thought of as "coin tossing" then this could be thought of as "card dealing". The procedure is following:

1. A and B agree on a large prime p.

2. A chooses a secret number a from the interval $1 \leq a < p-1$ such that $\gcd(a, p-1) = 1$, and computes the number $a' = (a^{-1}, \mathrm{mod}\ p-1)$.

3. B chooses a secret number b from the interval $1 \leq b < p-1$ such that $\gcd(b, p-1) = 1$, and computes the number $b' = (b^{-1}, \mathrm{mod}\ p-1)$.

4. The numbers i are encoded as the numbers $c_i = (g^{2i+1}, \mathrm{mod}\ p)$ $(i = 1, 2, \ldots, N)$ where g is a primitive root modulo p. g and p can be found in the same way as in setting up ELGAMAL, see Section 10.1. The numbers c_i are all quadratic nonresidues modulo p, since exponents of quadratic residues are even.

5. B computes the numbers $\beta_i = (c_i^b, \mathrm{mod}\ p)$ $(i = 1, 2, \ldots, N)$, permutes them randomly and sends them to A. Note that because b is odd, information of a number c_i being a quadratic residue modulo p passes this encoding process by Euler's criterion, since by Fermat's little theorem $c_i^{p-1} \equiv 1 \mod p$ and hence $c_i^{(p-1)/2} \equiv \pm 1 \mod p$. Because of this, all c_i's were chosen to be quadratic nonresidues modulo p to start with. On the other hand, obtaining c_i from β_i would require computing a discrete logarithm in $\mathbb{Z}_p$.

6. A chooses $2k$ of these numbers, say $\beta_{i_1}, \ldots, \beta_{i_{2k}}$, computes the numbers

$$\alpha_j = (\beta_{i_j}^a, \mathrm{mod}\ p) = (c_{i_j}^{ab}, \mathrm{mod}\ p) \quad (j = 1, 2, \ldots, k),$$

   and sends them and the numbers $\beta_{i_{k+1}}, \ldots, \beta_{i_{2k}}$ to B. Again obtaining β_{i_j} from α_j would require computing a discrete logarithm.

7. B computes the numbers

$$\gamma_j = (\alpha_j^{b'}, \mathrm{mod}\ p) = (c_{i_j}^a, \mathrm{mod}\ p) \quad (j = 1, 2, \ldots, k)$$

   and sends them to A. Compare this to decrypting of RSA.

8. A computes her numbers $c_{i_j} = (\gamma_j^{a'}, \mathrm{mod}\ p)$ $(j = 1, 2, \ldots, k)$.

9. B computes his numbers $c_{i_j} = (\beta_{i_j}^{b'}, \mathrm{mod}\ p)$ $(j = k+1, \ldots, 2k)$.

14.2 Sharing Secrets

If t and v are positive integers and $t \leq v$ then a (t, v)-*threshold scheme* is a procedure which is used to distribute a secret S to v parties so that any $t-1$ parties won't get anything out of the secret but any t parties get to know it in full (the *threshold*).

Threshold schemes are usually carried out by some kind of interpolation. A certain function $f_{p_1,\ldots,p_t}$, the so-called *interpolant,* is defined fully when its parameters $p_1, \ldots, p_t$ are known. The parameters themselves are obtained if we know the values of the function in at least t different points:

$$f_{p_1,\ldots,p_t}(x_i) = y_i \quad (i = 1, 2, \ldots, v \text{ where } v \geq t).$$

On the other hand, values in any $t-1$ points do not define the parameters unambiguously. The secret S is the function $f_{p_1,\ldots,p_t}$, or its parameters $p_1, \ldots, p_t$ or just some of them. Each party is given a value of the function, the so-called *share.* This is done secretly by a trusted outside party, the so-called *distributor* D.

One way to get an interpolant is to use a polynomial

$$p(x) = S \oplus \bigoplus_{j=1}^{t-1} p_{j+1} x^j.$$

This is called *Shamir's threshold scheme.*[2] It can be carried out in any field F with more than v elements. The most common choice is a prime field $\mathbb{Z}_q$ where $q > v$. The secret is the constant term $S = p_1$ of $p(x)$. It is known that a polynomial of degree no higher than $t-1$ is fully determined when its values are known in t different points. On the other hand, a polynomial won't be determined unambiguously, if the degree is $t-1$ and there are less than t points. In particular, the polynomial's constant term is not determined in this way, unless a value is specifically given in the point $x = \mathbf{0}$. This is because if the constant term S were uniquely determined by $t-1$ values $y_i = p(x_i)$ in different points $x_i \neq \mathbf{0}$ $(i = 1, 2, \ldots, t-1)$ then the remaining parameters $p_2, \ldots, p_t$ would be determined by the equations

$$x_i^{-1} \odot (y_i \ominus S) = \bigoplus_{j=1}^{t-1} p_{j+1} x_i^{j-1} \quad (i = 1, 2, \ldots, t-1).$$

As is seen, S can be anything, so no information about S is revealed.

The interpolation itself can be carried out using a linear system of equations—the matrix of which is a so-called Vandermonde matrix—or for example Lagrange's interpolation (see the basic courses):

$$p(x) = \bigoplus_{j=1}^{t} y_j \odot \bigodot_{\substack{k=1 \\ k \neq j}}^{t} (x_j \ominus x_k)^{-1} \odot (x \ominus x_k).$$

In this case

$$S = p(\mathbf{0}) = \bigoplus_{j=1}^{t} y_j \odot \bigodot_{\substack{k=1 \\ k \neq j}}^{t} (x_k \ominus x_j)^{-1} \odot x_k.$$

Points where values of $p(x)$ are computed can be public, in which case the shares would be just these values. Then computation of S is just computation of a linear combination of the shares with known coefficients, possibly precomputed.

The scheme itself is the following:

[2]The original reference is SHAMIR, A.: How to Share a Secret. *Communications of the Association for Computing Machinery* **22** (1979), 612–613.

Shamir's threshold schme:

1. D chooses a field F and v different elements $u_1, u_2, \ldots, u_v \neq \mathbf{0}$ of F, and communicates u_i to the i^{th} party ($i = 1, 2, \ldots, v$). The secret S is an element of F.

2. D secretly and randomly chooses $t-1$ elements $p_2, \ldots, p_t$ of the field F.

3. D computes the shares

$$w_i = S \oplus \bigoplus_{j=1}^{t-1} p_{j+1} \odot u_i^j \quad (i = 1, 2, \ldots, v),$$

and communicates to each party its share, without letting the other parties know anything about it.

4. When the parties $i_1, i_2, \ldots, i_t$ want to know the secret, they interpolate and compute S. For example, using Lagrange's interpolation

$$S = \bigoplus_{j=1}^{t} w_{i_j} \odot \bigodot_{\substack{k=1 \\ k \neq j}}^{t} (u_{i_k} \ominus u_{i_j})^{-1} \odot u_{i_k}.$$

NB. *Sharing secrets must not be confused with a very similar procedure, the so-called* dispersal of information, *where you disperse a file into v pieces, any t of which suffice to reconstruct the file quickly. The difference is that $t-1$ pieces can now perfectly well give a lot of information about the file, possibly not the whole file, however. Dispersal of information has to do with error-correcting codes (see the course Coding Theory), and the dispersed parts are usually much smaller than the shares above. The original reference is* RABIN, M.O.: Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance. *Journal of the Association for Computing Machinery* **36** (1989), 335–348.

There are other ideas for sharing secrets. Many secret sharing schemes are based on coding theory. The Chinese remainder theorem can be used in the interpolation, too, e.g. in the so-called *Mignotte threshold scheme,* see for example DING & PEI & SALOMAA.

14.3 Oblivious Data Transfer

The party A wants to transfer a secret to the party B, but in such a way that the secret may or may not be transferred. Of course B knows whether the secret was transferred or not, but A should not know this. In fact, from A's point of view, the secret is transferred with probability $1/2$. A simple procedure for this would be the following. Here, as usual, n is a product of two different large primes p and q. The secret may be thought to be these two primes, the real secret could then e.g. be encrypted by RSA using n. So, in the beginning A knows p and q while B does not.

1. B chooses a number x from the interval $1 \leq x < n$, computes $(x^2, \mathrm{mod}\ n)$, and sends it to A.

2. A computes the four square roots

$$(\pm x, \mathrm{mod}\ n) \quad \text{and} \quad (\pm y, \mathrm{mod}\ n)$$

of $(x^2,\mathrm{mod}\ n)$ modulo n, and sends one of them to B. Because A knows the factors of n, she can do this quite quickly. A cannot however know which of the square roots x is. See Section 7.6.

3. B checks whether the square root he got from A is $\equiv \pm x \mod n$. In the positive case B does not get the secret. Otherwise B gets to know numbers x and y such that $x^2 \equiv y^2 \mod n$ and $x \not\equiv \pm y \mod n$, and is able to factor n and in this way learns the secret. A cannot know whether or not B got the secret, unless B chooses to tell this to A.

14.4 Zero-Knowledge Proofs

There are two parties in an *interactive proof system*, the *prover* P and the *verifier* V. They send messages to each other and perform computations based on the messages they receive, including random number generating if necessary. The goal of P is to convince V that he knows some property of some object. The object could be e.g. a mathematical result and the property its truth, but of course it could be something quite different. Another goal of P is not to transmit to V any other information than that he knows this property. This is called *zero-knowledge proof.*

The basic requirements of a zero-knowledge proof are the following:

(I) The probability of P successfully fooling V is very small.

If, for example, P does not know the proof of a mathematical result, but claims to do so, then his chances of fooling V should be minuscule.

(II) If P truly knows the property, he can prove this to V beyond any reasonable doubt.

(III) V won't get from P any information that he could not obtain himself without P, computing in polynomial time if needed.

In this case V could actually simulate the proof protocol in polynomial time as if P would participate in it, but without P. Note that there are no restrictions on the complexity of computations of P. The simulation must be exact enough to make it impossible to tell it apart from the "real" one, computing in polynomial time.

Despite condition (III), V might, after some very long computations, be able to get more information, possibly the whole property. So, instead of (III), a stronger condition is required in the so-called *perfect zero-knowledge proof:*

(III′) V won't get from P any information that he could not get by himself without P.

Here too V computes in polynomial time, but the simulation must now be fully identical to the "real" one.

Sometimes the zero-knowledge proof defined by the conditions (I)–(III) above is called *computational zero-knowledge proof,* to distinguish it from perfect zero-knowledge proof. It should be noted that the above conditions do not really give exact definitions. These definitions are actually much more complicated, see for example STINSON or GOLDREICH. The difference between computational and perfect zero-knowledge proofs is in the comparison of stochastic distributions: In perfect zero-knowledge proofs "real" and simulated distributions must be identical, in computational zero-knowledge proofs it is only required that the distributions cannot be separated by polynomial-time computations.

The following protocol[3] gives a perfect zero-knowledge proof of the fact that x is a quadratic residue modulo n where $n = pq$ and p and q are two different large primes, assuming that $\gcd(x, n) = 1$. Here the problem is QUADRATICRESIDUES, and the proof is a square root of x modulo n.

1. Repeat the following k times:

   1.1 P chooses a random number v from the interval $1 \leq v < n$ such that $\gcd(v, n) = 1$, computes the number $y = (v^2, \mathrm{mod}\ n)$, and sends it to V.

   1.2 V chooses randomly a bit b (0 or 1) and sends it to P.

   1.3 P computes the number $z = (u^b v, \mathrm{mod}\ n)$ where u is a square root of x modulo n, and sends it to V.

   1.4 V checks that $z^2 \equiv x^b y \mod n$.

2. If the check passes every time for each of the k rounds, V concludes that P really knows x is a quadratic residue modulo n.

Theorem 14.1. *The above protocol gives a perfect zero-knowledge proof for the problem* QUADRATICRESIDUES.

Proof. If P does not know a square root of x, he must cheat and send to B the number $z = v$, and either the number $y = (z^2, \mathrm{mod}\ n)$ (exposed if $b = 1$) or the number $y = (z^2 x^{-1}, \mathrm{mod}\ n)$ (exposed if $b = 0$). Thus the probability for P to cheat without getting caught is $1/2^k$, which can be made as small as wanted. Then again, if P really knows a square root u, he of course passes the test every time.

V can simulate P's part perfectly in this protocol. The idea is that V generates triples (y, b, z) where

$$y \equiv z^2 x^{-b} \mod n.$$

Let's show that if V chooses the bit b and the number z completely randomly, these triples have a distribution identical to the "right" one, where P is involved and chooses a random v.

We say that the triple (y, b, z) is *feasible,* if

- $1 \leq y < n$ and $\gcd(y, n) = 1$,
- b is 0 or 1, and
- $1 \leq z < n$ and $z^2 \equiv x^b y \mod n$.

There are $2\phi(n)$ feasible triples, because there are $\phi(n)$ possible choices of z and b can be chosen in two different ways, and these choices determine y. Note that since $\gcd(x, n) = 1$ and $\gcd(y, n) = 1$, then $\gcd(z, n) = 1$ also.

Feasible triples occur in the protocol equally probably when P is involved, since P chooses v from among $\phi(n)$ different alternatives, and four possible square roots v correspond to one y. When y and b have been chosen, there are four possible choices for z. Also in the simulation performed by V feasible triples are equiprobable when V chooses z randomly from the interval $1 \leq z < n$ and $\gcd(z, n) = 1$, and b is chosen randomly. □

[3]The original reference is GOLDWASSER, S. & MICALI, S. & RACKOFF, C.: The Knowledge Complexity of Interactive Proof Systems. *SIAM Journal on Computing* **18** (1989), 186–208.

Let's also take an example of a (computational) zero-knowledge proof. The problem is to prove that there is a so-called Hamiltonian circuit in a graph. A *graph* consists of *vertices* and *edges* that connect vertices. Usually not all vertices are connected by edges. A *Hamiltonian circuit* is a path which forms a circuit through all vertices of the graph visiting each vertex exactly once and returning to the starting vertex. The path proceeds via the edges. (See the course Graph Theory.) Finding out whether or not there is a Hamiltonian circuit in a suitably encoded graph is known to be an $\mathcal{NP}$-complete recognition problem HAMILTONCIRCUIT. The following protocol[4] gives a zero-knowledge proof to this problem.

1. Repeat the following k times. The input is the graph G where the vertices are denoted by $1, 2, \ldots, n$.

   1.1 P arranges the vertices in a random order and sends the list $v_1, v_2, \ldots, v_n$ obtained this way (encoded in bits) *encrypted* to V. P also sends to V the $n \times n$ matrix $\mathbf{D} = (d_{ij})$ (the so-called *adjacency matrix*) *encrypted element by element* where the diagonal elements are $= 0$ and

   $$d_{ij} = \begin{cases} 1 \text{ if there is an edge connecting the vertices } v_i \text{ and } v_j \\ 0 \text{ otherwise,} \end{cases}$$

   when $i \neq j$. Because of the symmetry it is enough to send only the upper triangle. Each element of the matrix is encrypted by its own key. The encryption must lead to commitment, that is, P must not be able to change the graph later by changing keys, compare with bit-flipping. Naturally, the encryption is assumed to be strong enough, in other words nothing can be got from an encrypted bit in polynomial time.

   1.2 V chooces a bit b randomly and sends it to P.

   1.3 If $b = 0$, P decrypts the list $v_1, v_2, \ldots, v_n$ and the whole matrix $\mathbf{D}$ for V by sending her the decrypting keys. Then again, if $b = 1$, P decrypts for V only the n elements $d_{i_1 i_2}, d_{i_2 i_3}, \ldots, d_{i_n i_1}$ of the matrix $\mathbf{D}$ where the vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_n}$ in this order form a Hamiltonian circuit (in which case the elements are all $= 1$).

   1.4 If $b = 0$, V checks whether he got the correct graph. The decrypted list $v_1, v_2, \ldots, v_n$ gives the order of the vertices and $\mathbf{D}$ gives the edges. Then again, if $b = 1$, V checks whether the obtained elements of the matrix are $= 1$.

2. If the check passes in each of the k rounds, V concludes that P really does know a Hamiltonian circuit of G.

The commitment mentioned in #1.1 is obtained for example in the following way. Here the large prime p and the primitive root g modulo p are made public.

1. In the beginning V chooses and then sends to P a random number r from the interval $1 < r < p$. P cannot quickly compute the discrete logarithm $\log_g r$ modulo p.

2. P randomly chooses a number y from the interval $0 \leq y < p - 1$ (the secret key) and sends to V the number $c = (r^b g^y, \mathrm{mod}\ p)$ where b is the bit to be encrypted. Each element of $\mathbb{Z}_p^*$ is in the positive residue system both of the form $(g^y, \mathrm{mod}\ p)$ and of the form

[4] The original reference seems to be BLUM, M: How to Prove a Theorem So No One Else Can Claim It. *Proceedings of the International Congress of Mathematicians 1986.* American Mathematical Society (1988), 1444–1451.

$(rg^y, \mod p)$, so c does not reveal anything of the bit b. Whichever the bit is, the distribution of c remains the same. On the other hand, P cannot change the bit b by changing y to some y', otherwise

$$g^y \equiv rg^{y'} \mod p \quad \text{or} \quad rg^y \equiv g^{y'} \mod p,$$

i.e.

$$r \equiv g^{\pm(y-y')} \mod p,$$

and P would immediately obtain $\log_g r$ modulo p from this.

Theorem 14.2. *The above protocol gives a zero-knowledge proof for the problem* HAMILTON-CIRCUIT.

Proof. If P does not know a Hamiltonian circuit, he is able to cheat if he receives the bit $b = 0$, but not if he receives the bit $b = 1$. Then again, if P knows a Hamiltonian circuit of some other graph G' with n vertices, he can cheat if he receives the bit $b = 1$, but not if he receives the bit $b = 0$. So, the probability for P to succesfully cheat all the time is $1/2^k$, which can be made as small as we want. Then again, if P knows a Hamiltonian circuit of G, he of course passes the test every time.

V can simulate the protocol in polynomial time also without P. What V does is the following. V chooses a random bit b. If $b = 0$, V orders the vertices randomly and encrypts the list obtained this way. Further, V gets the adjacency matrix $\mathbf{D}$ and encrypts it. Then again, if $b = 1$, V encrypts only some random elements $d_{i_1 i_2}, d_{i_2 i_3}, \ldots, d_{i_n i_1}$ where the indexing is cyclic each index occurring exactly two times, and each element is $= 1$. For the sake of completeness, V can encrypt something else to obtain the right amount of encrypted data. Because the encryption used is strong, the encrypted element sequences are very "similar" whether they come from the correct adjacency matrix or not. In other words, computing in polynomial time the difference cannot be seen, and the occuring distributions cannot be separated. This does not mean that the distributions should be exactly the same! □

HAMILTONCIRCUIT is an $\mathcal{NP}$-complete problem to which other recognition problems in $\mathcal{NP}$ can be reduced, see Section 6.1. Hence V can always perform such a reduction, if needed, and we have

Theorem 14.3. *Zero-knowledge proofs can be given to all positive solutions of recognition problems in $\mathcal{NP}$.*

A perfect zero-knowledge proof of an $\mathcal{NP}$-complete recognition problem is however thought to be impossible, in other words, the theorem is expected to be false for perfect zero-knowledge proofs. Actually, a result much more general than Theorem 14.3 is known:

Theorem 14.4. (Shamir's theorem[5]) *Recognition problems for whose positive solutions there are zero-knowledge proofs are exactly the recognition problems in $\mathcal{PSPACE}$.*

[5]The original reference is SHAMIR, A.: IP = PSPACE. *Journal of the Association for Computing Machinery* **39** (1992), 869–877.

"Quantum mechanics makes absolutely no sense."

(SIR ROGER PENROSE)

"If quantum physics hasn't profoundly shocked you, you haven't understood it yet."

(NIELS BOHR)

"No one understands quantum physics."

(RICHARD FEYNMAN)

Chapter 15

QUANTUM CRYPTOLOGY

15.1 Quantum Bit

The values 0 and 1 of the classical bit correspond in quantum physics to complex orthonormal base vectors, denoted traditionally by $|0\rangle$ and $|1\rangle$. We can think then that we operate in $\mathbb{C}^2$ considered as a Hilbert space. A *quantum bit* or *qubit* is a linear combination of the form

$$\mathbf{b} = \alpha_0|0\rangle + \alpha_1|1\rangle$$

(a so-called *superposition*) where α_0 and α_1 are complex numbers and

$$\|\mathbf{b}\|^2 = |\alpha_0|^2 + |\alpha_1|^2 = 1.$$

In particular, $|0\rangle$ and $|1\rangle$ themselves are quantum bits, the so-called *pure quantum bits.* It is important that physically a quantum bit can be initialized to one of them.

A quantum physical *measurement* of $\mathbf{b}$ results either in $|0\rangle$ or in $|1\rangle$ —denoted briefly just by 0 and 1. So, the measurement always involves the basis used. According to the probabilistic interpretation of quantum physics, the result 0 is obtained with probability $|\alpha_0|^2$ and the result 1 with probability $|\alpha_1|^2$.

A quantum bit is a quantum physical *state* and it can be transformed to another state in one time step, provided that the transformation is linear and its matrix $\mathbf{U}$ is unitary, i.e. $\mathbf{U}^{-1}$ is the conjugate transpose $\mathbf{U}^\dagger$ of $\mathbf{U}$. Hence also

$$\mathbf{U}\mathbf{b} = \beta_0|0\rangle + \beta_1|1\rangle\,, \quad \text{where} \quad \begin{pmatrix}\beta_0\\ \beta_1\end{pmatrix} = \mathbf{U}\begin{pmatrix}\alpha_0\\ \alpha_1\end{pmatrix},$$

is a quantum bit (state). Note in particular that

$$|\beta_0|^2 + |\beta_1|^2 = \begin{pmatrix}\beta_0^* & \beta_1^*\end{pmatrix}\begin{pmatrix}\beta_0\\ \beta_1\end{pmatrix} = \begin{pmatrix}\alpha_0^* & \alpha_1^*\end{pmatrix}\mathbf{U}^\dagger\mathbf{U}\begin{pmatrix}\alpha_0\\ \alpha_1\end{pmatrix} = \begin{pmatrix}\alpha_0^* & \alpha_1^*\end{pmatrix}\begin{pmatrix}\alpha_0\\ \alpha_1\end{pmatrix} = 1.$$

(Complex conjugation is here denoted by an asterisk.) Now let's recall some basic properties of unitary matrices:

1. The identity matrix $\mathbf{I}_2$ is unitary. It is not necessary to do anything in a time step.
2. If $\mathbf{U}_1$ and $\mathbf{U}_2$ are unitary then $\mathbf{U}_1\mathbf{U}_2$is also unitary. This means a quantum bit can be operated on several times in consecutive time steps, possibly using different operations, and the result is always a legitimate quantum bit. This is exactly how a quantum computer handles quantum bits.

3. If $\mathbf{U}$ is unitary then $\mathbf{U}^\dagger$ is also unitary. When a quantum bit is operated on and another quantum bit is obtained, then the reverse operation is always legitimate, too. A quantum computer does not lose information, and is thus *reversible.* It has been known long that every algorithm can be replaced by a reversible algorithm. This was first proved by the French mathematician Yves Lecerf in 1962. Later it was shown that this does not even increase complexity very much.[1] Hence reversibility is not a real restriction considering computation, of course it makes designing quantum algorithms more difficult.

15.2 Quantum Registers and Quantum Algorithms

Quantum bits can be merged into quantum registers of a given length. The mathemathical operation used to do this is the so-called *Kronecker product* or *tensor product.* Kronecker's product of the matrices $\mathbf{A} = (a_{ij})$ (an $n_1 \times m_1$ matrix) and $\mathbf{B} = (b_{ij})$ (an $n_2 \times m_2$ matrix) is the $n_1 n_2 \times m_1 m_2$ matrix

$$\mathbf{A} \otimes \mathbf{B} = \left(\begin{array}{c|c|c|c} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \cdots & a_{1m_1}\mathbf{B} \\ \hline a_{21}\mathbf{B} & a_{22}\mathbf{B} & \cdots & a_{2m_1}\mathbf{B} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline a_{n_1 1}\mathbf{B} & a_{n_1 2}\mathbf{B} & \cdots & a_{n_1 m_2}\mathbf{B} \end{array}\right)$$

(in block form). As a special case we get Kronecker's product of two vectors ($m_1 = m_2 = 1$). The following basic properties of Kronecker's product are quite easy to prove. Here it is assumed that the occurring matrix operations are well-defined.

1. Distributivity:
$$(\mathbf{A}_1 + \mathbf{A}_2) \otimes \mathbf{B} = \mathbf{A}_1 \otimes \mathbf{B} + \mathbf{A}_2 \otimes \mathbf{B}$$
$$\mathbf{A} \otimes (\mathbf{B}_1 + \mathbf{B}_2) = \mathbf{A} \otimes \mathbf{B}_1 + \mathbf{A} \otimes \mathbf{B}_2$$

2. Associativity:
$$(\mathbf{A} \otimes \mathbf{B}) \otimes \mathbf{C} = \mathbf{A} \otimes (\mathbf{B} \otimes \mathbf{C})$$

   As a consequence of this a chain of consecutive Kronecker's products can be written without parentheses.

3. Multiplication by a scalar: $(c\mathbf{A}) \otimes \mathbf{B} = \mathbf{A} \otimes (c\mathbf{B}) = c(\mathbf{A} \otimes \mathbf{B})$

4. Matrix multiplication of Kronecker's products (this pretty much follows directly from multiplication of block matrices):
$$(\mathbf{A}_1 \otimes \mathbf{B}_1)(\mathbf{A}_2 \otimes \mathbf{B}_2) = (\mathbf{A}_1\mathbf{A}_2) \otimes (\mathbf{B}_1\mathbf{B}_2)$$

5. Matrix inverse of Kronecker's product (follows from the multiplication law):
$$(\mathbf{A} \otimes \mathbf{B})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{B}^{-1}$$

6. Conjugate transpose of Kronecker's product (follows directly from conjugate transposition of block matrices):
$$(\mathbf{A} \otimes \mathbf{B})^\dagger = \mathbf{A}^\dagger \otimes \mathbf{B}^\dagger$$

[1] The original references are LECERF, M.Y.: Machines de Turing réversibles. Récursive insolubilité en $n \in N$ de l'équation $u = \theta^n u$, où θ est un "isomorphisme de codes". *Comptes Rendus* **257** (1963), 2597–2600 and LEVIN, R.Y. & SHERMAN, A.T.: A Note on Bennett's Time-Space Tradeoff for Reversible Computation. *SIAM Journal on Computing* **19** (1990), 673–677.

7. Kronecker's products of unitary matrices are also unitary. (Follows from the above.)

When two quantum bits $\mathbf{b}_1 = \alpha_0|0\rangle + \alpha_1|1\rangle$ and $\mathbf{b}_2 = \beta_0|0\rangle + \beta_1|1\rangle$ are to be combined to a two-qubit *register,* it is done by taking Kronecker's product:

$$\mathbf{b}_1 \otimes \mathbf{b}_2 = \alpha_0\beta_0(|0\rangle \otimes |0\rangle) + \alpha_0\beta_1(|0\rangle \otimes |1\rangle) + \alpha_1\beta_0(|1\rangle \otimes |0\rangle) + \alpha_1\beta_1(|1\rangle \otimes |1\rangle).$$

(More exactly, it is the register's contents that is defined here.) A traditional notation convention here is

$$|0\rangle \otimes |0\rangle = |00\rangle \quad , \quad |0\rangle \otimes |1\rangle = |01\rangle \quad \text{etc.}$$

It is easy to see that $|00\rangle, |01\rangle, |10\rangle, |11\rangle$ is an orthonormal basis, in other words, the register's dimension is four. If we wish to operate on the register's first quantum bit by $\mathbf{U}_1$ and to second by $\mathbf{U}_2$ (both unitary matrices) then this is done by the unitary matrix $\mathbf{U}_1 \otimes \mathbf{U}_2$, because by the multiplication law

$$(\mathbf{U}_1 \otimes \mathbf{U}_2)(\mathbf{b}_1 \otimes \mathbf{b}_2) = (\mathbf{U}_1\mathbf{b}_1) \otimes (\mathbf{U}_2\mathbf{b}_2).$$

In particular, if we want to operate only on the first quantum bit by the matrix $\mathbf{U}$, it is done by choosing $\mathbf{U}_1 = \mathbf{U}$ and $\mathbf{U}_2 = \mathbf{I}_2$. In the same way we can operate only on the second quantum bit. But in a two-qubit register we can operate also by a general unitary 4×4 matrix, since the register is a legitimate quantum physical state. With this kind of operating we can link the quantum bits of the registers. Quantum physical linking is called *entanglement,* and it is a computational resource expressly typical of quantum computation, such a resource does not exist in classical computation.

In a similar way we can form registers of three or more quantum bits, operate on its quantum bits, either on all of them or just one, and so on. Generally the dimension of a register of m quantum bits is 2^m. Base vectors can then be thought to correspond, via binary representation, to integers in the interval $0, \ldots, 2^m - 1$, and we adopt the notation

$$|k\rangle = |b_{m-1}b_{m-2}\cdots b_1b_0\rangle$$

when the binary representation of k is $b_{m-1}b_{m-2}\cdots b_1b_0$, possibly after adding initial zeros. Several registers can be combined to longer registers using Kronecker's products, and we can operate on these either all together or only one and so on.

Despite the register's dimension 2^m being possibly very high, many operations on its quantum bits are physically performable, possibly in several steps, and the huge unitary matrices are not needed in practice. In this case the step sequence is called a *quantum algorithm.* It is important that entanglements too are possible and useful in quantum algorithms.

In the the sequel the following operations are central. Showing that they can be performed by using quantum algorithms is somewhat difficult.[2] Here k is as above.

- From the input $|k\rangle \otimes |0\cdots0\rangle$ we compute $|k\rangle \otimes |(w^k, \mathrm{mod}\ n)\rangle$ where w and $n \leq 2^m$ are given fixed integers. (Essentially by the Russian peasants' method.)

- From the input $|k\rangle$ we compute its so-called *quantum Fourier transformation*

$$\mathcal{F}_\mathrm{Q}(|k\rangle) = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} e^{\frac{2\pi ijk}{n}} |j\rangle$$

  where i is the imaginary unit. Quantum Fourier transformation works much as the "ordinary" discrete Fourier transformation, in other words, it picks periodic parts from the input sequence, see the course Fourier Methods.

[2] See for example SHOR, P.W.: Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Journal on Computing* **26** (1997), 1484–1509 or NIELSEN & CHUANG.

15.3 Shor's Algorithm

Today's quantum computers are very small and have no practical meaning. Handling bigger quantum registers with quantum computers would however mean that procedures central for the safety of for example RSA and ELGAMAL, such as factorization and computing discrete logarithms modulo a prime, could be performed in polynomial time. Indeed, these problems are in the class $\mathcal{BQP}$. This was shown by Peter Shor in 1994. Let's see Shor's factorization algorithm here. See the reference SHOR mentioned in Footnote 2.

Shor's factorization algorithm is very similar to the exponent algorithm for cryptanalysis of RSA in Section 8.3. The mysterious algorithm A, that appeared there, is just replaced by a quantum algorithm. Of course, the number n to be factored can here have many more prime factors than just two. The "classical part" of the algorithm is the following when the input is the integer $n \geq 2$:

Shor's factorization algorithm:

1. Check whether n is a prime. If it is then return n and quit.
2. Check whether n is a higher power of some integer, compare to the Agrawal–Kayal–Saxena algorithm in Section 7.4. If $n = u^t$, where $t \geq 2$, we continue by finding the prime factors of u from which we then easily obtain the factors of n. This part, as the previous one, is included only to take care of some "easy" situations quickly.
3. Choose randomly a number w from the interval $1 \leq w < n$.
4. Compute $d = \gcd(w, n)$ by the Euclidean algorithm.
5. If $1 < d < n$, continue from d and n/d.
6. If $d = 1$, compute with the quantum computer a number $r > 0$ such that $w^r \equiv 1 \mod n$.
7. If r is odd, go to #9.
8. If r is even, set $r \leftarrow r/2$ and go to #7.
9. Compute $\omega = (w^r, \mod n)$ by the algorithm of Russian peasants.
10. If $\omega \equiv 1 \mod n$, give up and quit.
11. If $\omega \not\equiv 1 \mod n$, set $\omega' \leftarrow \omega$ and $\omega \leftarrow (\omega^2, \mod n)$, and go to #11.
12. Eventually we obtain a square root ω' of 1 modulo n such that $\omega' \not\equiv 1 \mod n$. If now $\omega' \equiv -1 \mod n$, give up and quit. Otherwise compute $t = \gcd(\omega' - 1, n)$ and continue from t and n/t. Note that because $\omega' + 1 \not\equiv 0 \mod n$ and on the other hand $\omega'^2 - 1 = (\omega' + 1)(\omega' - 1) \equiv 0 \mod n$, some prime factor of n is a factor of $\omega' - 1$.

As in Section 8.3, it can be proved that if n is composite, the algorithm finds a factor with at least probability $1/2$.

So, #6 is left to be performed with the quantum computer. This can be done based on the fact that $(w^j, \mod n)$ is periodic with respect to j and a period r can be found by a quantum Fourier transformation. The procedure itself is the following:

6.1 Choose a number 2^m such that $n^2 \leq 2^m < 2n^2$.

6.2 Initialize two registers of length m to zeros: $|0\cdots0\rangle \otimes |0\cdots0\rangle$.

6.3 Apply the quantum Fourier transformation to the first register:

$$\begin{aligned}\mathcal{F}_{\mathrm{Q}}(|0\cdots0\rangle) \otimes |0\cdots0\rangle &= \left(\frac{1}{2^{m/2}} \sum_{j=0}^{2^m-1} e^{\frac{2\pi i j \cdot 0}{2^m}} |j\rangle\right) \otimes |0\cdots0\rangle \\ &= \frac{1}{2^{m/2}} \sum_{j=0}^{2^m-1} |j\rangle \otimes |0\cdots0\rangle.\end{aligned}$$

Now we have a uniform superposition of the integers $0, \ldots, 2^m - 1$ in the first register. The quantum computer is ready to handle them all simultaneously!

6.4 Compute by a suitable operation (see the previous section) simultaneously

$$\frac{1}{2^{m/2}} \sum_{j=0}^{2^m-1} |j\rangle \otimes |(w^j, \bmod n)\rangle.$$

The registers are now entangled in the quantum physical sense.

6.5 Measuring the second register we obtain the integer v, and the registers are

$$\gamma \sum_{\substack{j=0 \\ w^j \equiv v \bmod n}}^{2^m-1} |j\rangle \otimes |v\rangle$$

where γ is a scaling constant and the indices j occur periodically. Scaling is needed because after the measuring we must have a quantum physical state.

6.6 Apply the quantum Fourier transformation to the first register:

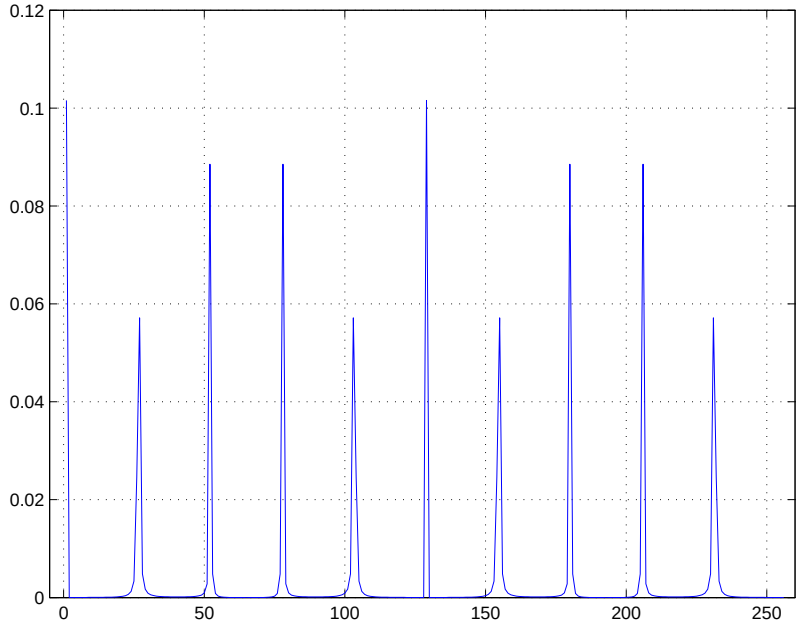
$$\frac{\gamma}{2^{m/2}} \sum_{\substack{j=0 \\ w^j \equiv v \bmod n}}^{2^m-1} \sum_{l=0}^{2^m-1} e^{\frac{2\pi i l j}{2^m}} |l\rangle \otimes |v\rangle.$$

6.7 Measure the first register. The result l is then obtained with probability $|g(l)|^2$ where

$$g(l) = \frac{\gamma}{2^{m/2}} \sum_{\substack{j=0 \\ w^j \equiv v \bmod n}}^{2^m-1} e^{\frac{2\pi i l j}{2^m}}.$$

But $g(l)$ is, ignoring the coefficient, a discrete Fourier transformation of a sequence in which 1 occurs with the same period as j in #6.5, other elements being zeros.

The above-mentioned probability is illustrated below, when $m = 8$ and $r = 10$. These values are of course far too small to be very interesting in practice. r corresponds to the frequency $2^8/10 = 25.6$, which can be seen very clearly together with its multiples. It is very likely that the measured l will be near one of these.



6.8 In this way we obtain a value l, which is an approximate multiple of the frequency $2^m/r$, i.e. there is a j such that

$$\frac{j}{r} \cong \frac{l}{2^m}.$$

Because $r \leq \phi(n) < n - 1$, r might be found by trying out numbers around the rational number $l/2^m$. In any case, using the condition for m in #1, we can very probably find the correct r using so-called Diophantine approximation, see the reference SHOR in Footnote 2.

All in all, we are talking about a kind of probabilistic polynomial-time algorithm using which we can find periods of quite long sequences. Such an algorithm would have a lot of applications, e.g. in group theory, if only we had large quantum computers.

15.4 Grover's Search Algorithm

We now adopt a further feature of the notation already used (often called the *bra-ket notation*). The notation $|\psi\rangle$ (the *ket*) is a column vector and $\langle\psi|$ (the *bra*) is its conjugate transpose (a row vector). Thus $\langle\psi||\phi\rangle$ is the inner product of $|\psi\rangle$ and $|\phi\rangle$, usually written as $\langle\psi|\phi\rangle$. Moreover then $|\psi\rangle\langle\psi|$ is a self-adjoint operator (matrix).

The task is to find among n register states $|j\rangle$ (as above) a state $|q\rangle$ satisfying a given condition. The condition is specified using a black box unitary (check!) operation (matrix) $\mathbf{U}_q$ given by

$$\mathbf{U}_q|j\rangle = \begin{cases} -|q\rangle, \text{ if } j = q \\ |j\rangle, \text{ if } j \neq q, \end{cases}$$

i.e.

$$\mathbf{U}_q = \mathbf{I}_n - 2|q\rangle\langle q|.$$

Thus $\mathbf{U}_q$ flips the state $|q\rangle$ leaving the other states unchanged.

Using the quantum Fourier transform as above the state

$$|\boldsymbol{\beta}\rangle = \frac{1}{\sqrt{n}} \sum_{k=1}^{n} |k\rangle.$$

can be initialized. Another unitary operator (matrix) needed in the algorithm is the so-called *Grover diffusion*

$$\mathbf{U}_{\boldsymbol{\beta}} = 2|\boldsymbol{\beta}\rangle\langle\boldsymbol{\beta}| - \mathbf{I}_n.$$

Operating with $\mathbf{U}_q$ and $\mathbf{U}_{\boldsymbol{\beta}}$ takes place in the *real* 2-dimensional subspace (plane) spanned by $|q\rangle$ and $|\boldsymbol{\beta}\rangle$. Geometrically $\mathbf{U}_q$ is a reflection with respect to the coordinate hyperplane perpendicular to $|q\rangle$. As was seen in the basic courses, this operation transforms a vector $|\boldsymbol{\psi}\rangle$ to $|\boldsymbol{\psi}\rangle$ minus twice the projection of $|\boldsymbol{\psi}\rangle$ on $|q\rangle$, i.e., to

$$|\boldsymbol{\psi}\rangle - 2\langle q|\boldsymbol{\psi}\rangle|q\rangle = |\boldsymbol{\psi}\rangle - 2|q\rangle\langle q||\boldsymbol{\psi}\rangle = \mathbf{U}_q|\boldsymbol{\psi}\rangle.$$

Similarly, $\mathbf{U}_{\boldsymbol{\beta}}$ is a reflection with respect to the line given by $|\boldsymbol{\beta}\rangle$. This transforms a vector $|\boldsymbol{\psi}\rangle$ to $|\boldsymbol{\psi}\rangle$ + twice the difference of the projection of $|\boldsymbol{\psi}\rangle$ on $|\boldsymbol{\beta}\rangle$ and $|\boldsymbol{\psi}\rangle$, i.e., to

$$|\boldsymbol{\psi}\rangle - 2(\langle\boldsymbol{\beta}|\boldsymbol{\psi}\rangle|\boldsymbol{\beta}\rangle - |\boldsymbol{\psi}\rangle) = 2|\boldsymbol{\beta}\rangle\langle\boldsymbol{\beta}||\boldsymbol{\psi}\rangle - |\boldsymbol{\psi}\rangle = \mathbf{U}_{\boldsymbol{\beta}}|\boldsymbol{\psi}\rangle.$$

The algorithm is simply the following. Starting from the initialized state $|\boldsymbol{\beta}\rangle$ iterate r times the operation $\mathbf{U}_{\boldsymbol{\beta}}\mathbf{U}_q$, for a certain number r. During this the result remains in the real plane spanned by $|q\rangle$ and $|\boldsymbol{\beta}\rangle$. Apparently

$$\langle q|\boldsymbol{\beta}\rangle = \frac{1}{\sqrt{n}} \quad \text{and so} \quad \cos\gamma = \frac{1}{\sqrt{n}},$$

where γ is the angle spanned by $|q\rangle$ and $|\boldsymbol{\beta}\rangle$.

Now $\mathbf{U}_{\boldsymbol{\beta}}\mathbf{U}_q$ is actually a rotation in the (2-dimensional) plane spanned by $|q\rangle$ and $|\boldsymbol{\beta}\rangle$ through the angle

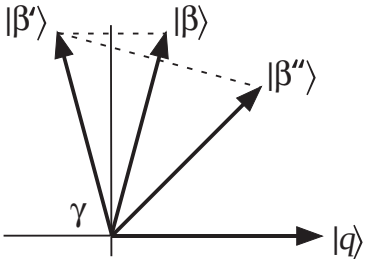
$$\pi - 2\gamma = 2\arcsin\frac{1}{\sqrt{n}} \cong \frac{2}{\sqrt{n}}.$$

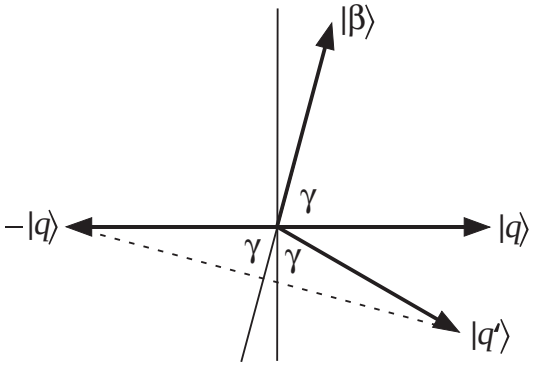
(The approximation is valid for large n.) One only needs to verify this for $|q\rangle$ and $|\boldsymbol{\beta}\rangle$. When we denote

$$|q'\rangle = \mathbf{U}_{\boldsymbol{\beta}}\mathbf{U}_q|q\rangle = -\mathbf{U}_{\boldsymbol{\beta}}|q\rangle \quad \text{and}$$

$$|\boldsymbol{\beta}''\rangle = \mathbf{U}_{\boldsymbol{\beta}}\mathbf{U}_q|\boldsymbol{\beta}\rangle = \mathbf{U}_{\boldsymbol{\beta}}|\boldsymbol{\beta}'\rangle,$$

then this is seen for $|\boldsymbol{\beta}\rangle$ in the upper figure on the right—in an exaggerated fashion because in reality n is large and γ is very close to $\pi/2$—and for $|q\rangle$ in the lower figure.





Thus the number r of iterations needed to reach $|q\rangle$ starting from $|\boldsymbol{\beta}\rangle$ is of the order $\sqrt{n}$. The probability of getting $|q\rangle$ as a result of a measurement is then

$$\cos^2(\gamma - r(\pi - 2\gamma)),$$

which has the maximum value 1 when

$$r = \frac{\gamma}{\pi - 2\gamma} \cong \frac{\pi\sqrt{n}}{4}.$$

This should be compared with the classical case where in the worst case n steps are needed, and $n/2$ even in the average.

Using Grover's algorithm key spaces can be searched quadratically faster, which reduces the effective key lengths to one half of what they are classically.

15.5 No-Cloning Theorem

Classically, if nothing else, an eavesdropper may at least be able to make a copy of a cryptotext for later sinister purposes without being caught at it. This is not possible for quantum states, because

Theorem 15.1. (No-Cloning Theorem) *It is not possible to copy a general quantum state.*

Proof. If it is possible to copy an arbitrary quantum state then there is a unitary operation (matrix) $\mathbf{C}$ such that

$$\mathbf{C}(|\psi\rangle \otimes |0\cdots0\rangle) = |\psi\rangle \otimes |\psi\rangle$$

for all states $|\psi\rangle$. For any state $|\phi\rangle$ then (using the properties of Kronecker's product)

$$\begin{aligned}
\langle\psi|\phi\rangle &= \langle\psi|\phi\rangle \cdot 1 = \langle\psi|\phi\rangle\langle 0\cdots0|0\cdots0\rangle \\
&= \langle\psi||\phi\rangle \otimes \langle 0\cdots0||0\cdots0\rangle \\
&= (\langle\psi| \otimes \langle 0\cdots0|)(|\phi\rangle \otimes |0\cdots0\rangle) \\
&= (\langle\psi| \otimes \langle 0\cdots0|)\mathbf{I}(|\phi\rangle \otimes |0\cdots0\rangle) \\
&= (\langle\psi| \otimes \langle 0\cdots0|)\mathbf{C}^\dagger\mathbf{C}(|\phi\rangle \otimes |0\cdots0\rangle) \\
&= (\langle\psi| \otimes \langle\psi|)(|\phi\rangle \otimes |\phi\rangle) \\
&= \langle\psi||\phi\rangle \otimes \langle\psi||\phi\rangle = \langle\psi|\phi\rangle\langle\psi|\phi\rangle \\
&= \langle\psi|\phi\rangle^2.
\end{aligned}$$

Thus either $\langle\psi|\phi\rangle = 0$ or $\langle\psi|\phi\rangle = 1$ which clearly is not true in general (take e.g. $|\phi\rangle = -|\psi\rangle$). □

Note that it is however possible to initialize two or more copies of a known register state, such as $|0\cdots0\rangle$, and even some other states, such as the $|\beta\rangle$ in the previous section.

Because of reversibility, a consequence is the *No-Deleting Theorem* according to which there is no unitary operator (matrix) $\mathbf{D}$ such that

$$\mathbf{D}(|\psi\rangle \otimes |\psi\rangle) = |\psi\rangle \otimes |0\cdots0\rangle$$

for all states $|\psi\rangle$. An eavesdropper cannot destroy (replace by known content) one part of an entangled state leaving the other part untouched without this being noticed.

15.6 Quantum Key-Exchange

A quantum bit can be represented in many orthonormal bases. Because measuring is always connected to an orthonormal basis and results in one of the base vectors, we can measure a quantum bit, pure in one basis, in another basis and get any one of the latter basis' vectors.

First let's take an orthonormal basis $|0\rangle, |1\rangle$, denoted $\mathcal{B}_1$, and then another basis $|+\rangle, |-\rangle$, denoted $\mathcal{B}_2$, where

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \quad \text{and} \quad |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle).$$

$\mathcal{B}_2$ is then orthonormal too. The measurer can decide in which basis he/she measures. For example, when measuring the quantum bit

$$|0\rangle = \frac{1}{\sqrt{2}}|+\rangle + \frac{1}{\sqrt{2}}|-\rangle$$

in the basis $\mathcal{B}_2$, the measurer gets $|+\rangle$ with probability $1/2$.

Quantum key-exchange can be done in many ways. One way to get a secret key for two parties A and B is the following:

1. A sends a sequence of bits to B, interpreting them as pure quantum bits and choosing for each bit the basis she uses, $\mathcal{B}_1$ or $\mathcal{B}_2$, and when using $\mathcal{B}_2$ identifying, say, 0 with $|-\rangle$ and 1 with $|+\rangle$. A also remembers her choices of bases.

2. After obtaining the quantum bits sent by A, B measures them choosing randomly a basis, $\mathcal{B}_1$ or $\mathcal{B}_2$, for each received quantum bit, and remembers his choices of bases and the measured results.

3. B sends to A the sequence of bases he chose using a classical channel.

4. A tells B which of their choices of bases were the same using a classical channel.

5. A and B use only those bits for the key, which are obtained from these common choices of bases. Indeed, these are the bases where B's measurement gives pure quantum bits identical to the ones sent by A. About half of the bits sent will thus be used.

If an outside party C tries to interfere in the key-exchange, either by trying to obtain the key by measuring quantum bits sent by A or by trying to send B quantum bits of his own, he is very likely caught. (As a consequence of the No-Cloning Theorem, C cannot copy quantum bits for later use.) First of all, when measuring the quantum bits sent by A, C must choose the basis $\mathcal{B}_1$ or $\mathcal{B}_2$. This choice is the same as A's in about half of the cases. C sends these quantum bits to B, who believes they came from A. Then a lot of the bits chosen by A and B for their secret key in #5 will be different. This is naturally revealed later, say, by using AES and letting the first encrypted messages sent be equipped with parity checks or some other test sequences. The same will be true, of course, if C tries to send B quantum bits of his own choice instead of A's quantum bits.

Another key-exchange procedure based on a somewhat different principle is the following:

1. Both A and B initialize a set of registers of length two, each to the state

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \quad \text{(a so-called } \textit{Bell state}\text{)}.$$

This can be done (verify!) by first initializing the registers to the state $|00\rangle = |0\rangle \otimes |0\rangle$ and then applying the unitary matrix

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & -1 & 0 \\ 1 & 0 & 0 & -1 \end{pmatrix}.$$

The basis $\mathcal{B}_1$ is used in all registers, but also for the basis $\mathcal{B}_2$ we have a Bell state, since computing the Kronecker products it is easy to see that

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) = \frac{1}{\sqrt{2}}(|--\rangle + |++\rangle).$$

In Bell's state both positions contain the same pure quantum bit, in other words, the quantum bits are entangled. Physically the quantum bits can be separated and taken very far from each other without destroying the entanglement. A takes the first quantum bits, and B the second, remembering their order. Another possibility is that a trusted third party initializes the Bell states and then distributes their quantum bits to A and B. Ideally all this happens hidden from outsiders. The "halves" of the Bell states reside with A and B, waiting to be taken into use. If A and B can be absolutely sure that they received their "halves" of the Bell states without any outside disturbing, they get their secret key-bits simply by measuring their quantum bits in the same basis (agreed on beforehand). Because of the entanglement they will get the same bits, even though these are random. This happens even if A and B do their measurements so closely following each other that the information cannot be exchanged with the speed of light![3] Otherwise a procedure similar to the one above should be used as follows.

2. When A and B need the key, A measures her quantum bits (the first quantum bits) and chooses randomly the basis, $\mathcal{B}_1$ or $\mathcal{B}_2$, for each quantum bit. After this, B measures his quantum bits and chooses the basis randomly for each quantum bit. Because the quantum bits are entangled, they get the same results if they are using the same basis.

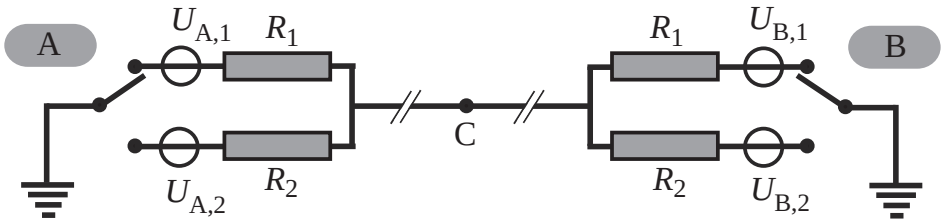
3. A tells B her choices of bases using a classical channel, thus announcing that the key-exchange began. This way the actual key distribution cannot proceed faster than light. B then tells A which of their choices of bases were the same again using a classical channel. An outside party cannot use this information, since he does not know the measured quantum bits. An outside party can however try to mess things up e.g. by sending B faked choices of bases in A's name. This will be revealed eventually as pointed out earlier. That will also happen if an outside party succeeded in meddling with A's or B's quantum bits (the No-Deleting Theorem).

4. A and B choose their key-bits from those quantum bits that they measured in the same bases. This way they get the same bits. About half of the measured quantum bits are then included in the key.

NB. *Nowadays quantum key-echange is used for some quite long distances, and it is thought to be absolutely safe. There are other, different protocols, see e.g.* NIELSEN & CHUANG.

[3]This is the so-called Einstein–Podolsky–Rosen paradox. Actual classical information is not transferred with a speed higher than that of light, since A cannot choose her measurement results and thus she cannot transmit to B any message she chose in advance. Moreover, A's quantum bits are already fixed by the first measurement, so she is not able to try it again either.

It is interesting to note that a key-exchange procedure similar to the first one above can be accomplished using "classical electricity" as well, as the so-called *Kish cipher,* see the figure below.



The two parties A and B both have two resistors with (different) resistances R_1 and R_2 (exactly the same for each). Resistance R_i is connected in series with noise voltage $U_{\mathrm{A},i}$ or $U_{\mathrm{B},i}$. The intensities (power spectral densities) of these noises are of the same form as that of the thermal noises of the resistors[4], i.e., combined these intensities are of the form ER_i where E is a constant. Using switches A and B randomly connect one of these resistor + noise generator units. When both A and B do this, a circuit is closed with current intensity $I = E/(R_{\mathrm{A}} + R_{\mathrm{B}})$ (Ohm's law) where R_{A} and R_{B} are the resistances chosen by A and B, respectively. A and B measure the current, so they know both resistances. If A and B choose the same resistance, either R_1 or R_2, no bit is determined. This happens approximately half the time. On the other hand, each time they choose different resistances, a key bit is determined (say, 0 if A chooses R_1 and 1 otherwise). An outside party C may then measure the current but this gives no information of the bit. Similarly C may measure voltage against ground without getting any information, the intensity of this voltage is $ER_{\mathrm{A}}R_{\mathrm{B}}/(R_{\mathrm{A}} + R_{\mathrm{B}})$. And there is not much anything else C can do.

This procedure works perfectly in an ideal situation and if A and B do the switching at exactly the same time. On the other hand, if e.g. they agree that A switches first and B after that, it may be possible for C to quickly measure the resistance A chose without her noticing this. C may then act as a "man-in-the-middle" posing as A for B and as B for A and finally get the whole key. This "man-in-the-middle" attack, as well as other attacks, can be made considerably more difficult by certain additional arrangements.[5]

[4] According to the so-called Johnson–Nyquist formula the intensity of the thermal noise of a resistance R in temperature T is $4kTR$ where k is Boltzmann's constant. This procedure is also called *Kirchhoff-Law–Johnson-Noise encryption* or *KLJN encryption.*

[5] See the original reference Kish, L.B.: Totally Secure Classical Communication Utilizing Johnson(-like) Noise and Kirchhoff's Law. *Physics Letters A* **352** (2006), 178–182. The procedure has been strongly criticized on various physical grounds, yet it has been physically implemented as well.

”Due to the suspicious nature of crypto users I have
a feeling DES will be with us forever, we will just
keep adding keys and cycles...”

(COLIN DOOLEY)

Appendix:

DES

A.1 General Information

DES (Data Encryption Standard) is a symmetric cryptosystem developed by IBM in the early 1970's. It is based on the LUCIFER system developed earlier by IBM. DES was published in 1975 and was certified as an encryption standard for ”unclassified” documents in USA in 1977. After this it has been used a lot in different circumstances, also as the triple system 3-DES. Many cryptosystems similar to DES are known: SAFER, RC5, BLOWFISH etc.

Mainly because of its far too small keysize DES is now mostly abandoned and replaced by AES.

A.2 Defining DES

DES operates with bit symbols, so the residue classes (bits) 0 and 1 of $\mathbb{Z}_2$ can be considered as the plaintext and cryptotext symbols. The length of the plaintext block is 64. The key k is 56 bits long. It is used in both encrypting and decrypting. In broad lines DES operates in the following way:

1. The bit sequence x_0 is formed of the plaintext x by permutating the bits of x by a certain fixed permutation (the so-called *initial permutation*) π_{ini}. Then we write

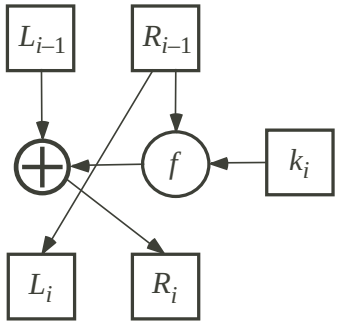
$$x_0 = \pi_{\text{ini}}(x) = L_0 R_0$$

   where L_0 contains the first 32 bits of x_0 and R_0 the rest.

2. Compute the sequence $L_1R_1, L_2R_2, \ldots, L_{16}R_{16}$ by iterating the following procedure 16 times:

$$\begin{cases} L_i = R_{i-1} \\ R_i = L_{i-1} \oplus f(R_{i-1}, k_i) \end{cases}$$

   where $\oplus$ is bitwise addition modulo 2 (known also by the name XOR), f is a function which is given later, and k_i is the key of the i^{th} iteration, obtained from k by permuting 48 of its bits into a certain order. An iteration step is depicted on the right.



3. Apply the inverse permutation π_{ini}^{-1} (the so-called *final permutation*) to the bit sequence $R_{16}L_{16}$.

We still need to give the permutation π_{ini}, define the function f, and give the key sequence $k_1, k_2, \ldots, k_{16}$, for encrypting to be defined.

First let's see the definition of the function f. The first argument R of f is a bit sequence of length 32 and the second argument K is a bit sequence of length 48. The procedure for computing f is the following:

1. The first argument R is *expanded* using the *expanding function* E. We take the first 32 bits of R into $E(R)$, duplicate half of them and then permute them. Bits are taken according to the table on the right, read from left to right and from top to bottom.

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

2. Compute $E(R) \oplus K = B$ and write the result as a catenation of eight 6-bit bit sequences:

$$B = B_1B_2B_3B_4B_5B_6B_7B_8.$$

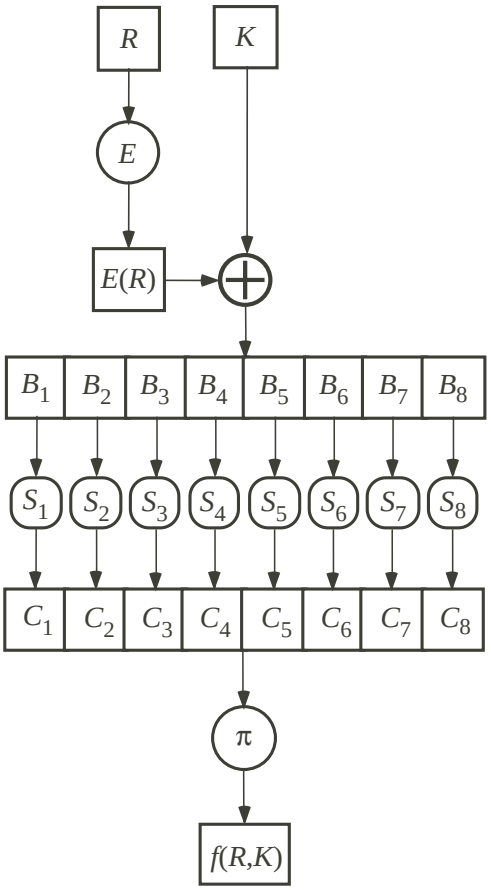
3. Next we use eight so-called *S-boxes* $S_1, \ldots, S_8$. Each S_i is a fixed 4×16 table, formed of the numbers $0, 1, \ldots, 15$. When a bit sequence of length of 6

$$B_i = b_1b_2b_3b_4b_5b_6$$

is obtained, $S_i(B_i) = C_i$ is computed in the following way. The bits b_1b_6 give the binary representation of the index r ($r = 0, 1, 2, 3$) of a certain row. The remaining bits $b_2b_3b_4b_5$ give the binary representation s ($s = 0, 1, \ldots, 15$) of a certain column. (The rows and columns of S_i are indexed starting from zero.) Now $S_i(B_i)$ is the binary representation of the number in the intersection of the r^{th} row and the s^{th} column of S_i, initial zeros added if needed to get four bits. The bit sequences C_i are catenated to the bit sequence

$$C = C_1C_2C_3C_4C_5C_6C_7C_8.$$

4. The bit sequence C of length of 32 is permuted using the fixed permutation π. The bit sequence $\pi(C)$ obtained this way is then $f(R, K)$.



The operation is illustrated above. We may note that E and π are linear operations, in other words, they could be replaced by multiplication of a bit vector by a matrix. On the other hand, S-boxes are highly non-linear. The definitions of S-boxes can be found in the literature (for example STINSON). On the right is S_2, given as an example, and below the permutations π_{ini} and π (c.f. E):

15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

π_{ini} :

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

π :

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

The key sequence $k_1, k_2, \ldots, k_{16}$ can be computed iteratively in the following way:

1. The key k is given in an expanded form such that every eigth bit is a parity-check bit. So there is always an odd number of 1's in a byte and the length of the key is 64 bits. If the parity check shows that there are errors in the key, it will not be taken into use. Then again, if there are no errors in the key, the parity check bits are removed, and we come to original 56-bit key. First a fixed bit permutation π_{K1} is applied to the key. Write
$$\pi_{\text{K1}}(k) = C_0 D_0$$
where C_0 and D_0 are bit sequences of length 28.

2. Compute the sequence $C_1D_1, C_2D_2, \ldots, D_{16}D_{16}$ by iterating the following procedure 16 times:
$$\begin{cases} C_i = \sigma_i(C_{i-1}) \\ D_i = \sigma_i(D_{i-1}) \end{cases}$$
where σ_i is a cyclic shift of the bit sequence by 1 or 2 bits to the left. If $i = 1, 2, 9, 16$ then the shift is 1 bit, otherwise it is 2 bits.

3. Apply the fixed variation π_{K2} of 48 bits to C_iD_i. In this way we obtain $k_i = \pi_{\text{K2}}(C_iD_i)$.

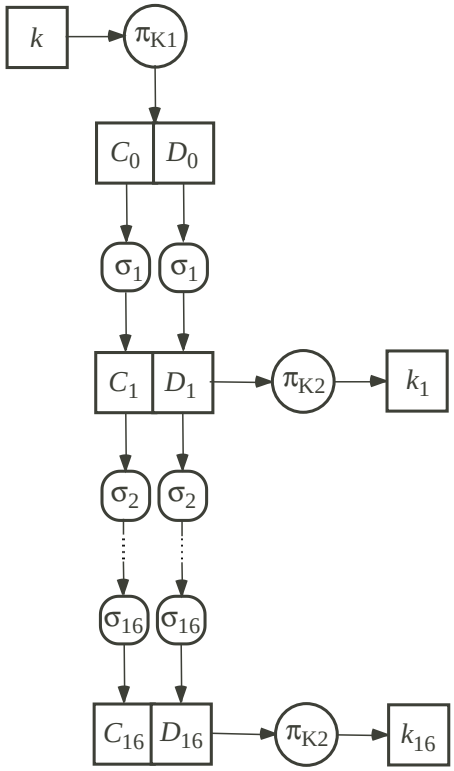
We must still give the permutation π_{K1} and the variation π_{K2}:

π_{K1} :

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

π_{K2} :

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32



The key generating process is illustrated in the above figure.

Decrypting goes essentially by same system but using the key sequence $k_1, k_2, \ldots, k_{16}$ in reverse order and inverting the permutations. Then

$$\begin{cases} L_{i-1} = R_i \oplus f(L_i, k_i) \\ R_{i-1} = L_i. \end{cases}$$

The modes of operation of DES are the same as for AES, see Section 5.4.

A.3 DES' Cryptanalysis

Everything else in DES' structure is linear—that is, doable by matrix multiplications—except the S-boxes. If the S-boxes were affine, i.e. if they could be replaced by matrix multiplications and addition of vectors, DES would essentially be some form of AFFINE-HILL and therefore easy to break. S-boxes are not however affine. Some of the design principles of DES' S-boxes were made public later:

(1) Each row of an S-box is a permutation of the numbers $0, 1, \ldots, 15$.

(2) An S-box is not an affine function of its inputs (and so not a linear function, either). Actually it is required that no output bit of an S-box is "near" a linear function of the input bits.

(3) Changing one bit in the input of an S-box changes at least two bits in the output.

(4) The outputs of an S-box with inputs x and $x \oplus 001100$ differ by at least two bits, no matter what 6-bit sequence x is.

(5) The outputs of an S-box with inputs x and $x \oplus 11b_1b_200$ differ, no matter what 6-bit sequence x is and no matter what bits b_1 and b_2 are.

(6) For each 6-bit sequence $B = b_1b_2b_3b_4b_5b_6 \neq 000000$ there are 32 ($= 2^6/2$) different input pairs x_1, x_2 such that $x_1 \oplus x_2 = B$. Of the corresponding 32 output pairs y_1, y_2 no more than two can have the same sum $y_1 \oplus y_2$.

There are

$$2^{56} = 72\,057\,594\,037\,927\,936$$

keys of DES, a fairly small number by modern standards. This makes it possible to use the following simple KP attack. If the plaintext w and the corresponding cryptotext c are known, we go through the keys until we find a key with which this encrypting can be done. There may, however, be several applicable keys. The procedure does not require anything in addition to time and fast processors, and it is easily parallelized, the memory requirements are minimal, too. DES can be installed in very fast hardware, and processors specifically designed to break DES are possible.

A CP attack is obtained in the following way. Choose a plaintext w and encrypt it using all possible keys of the key space. Tabulate the results. Now, if by the DES to be broken we can encrypt w and obtain the corresponding cryptotext, then by a table search we find a key. This method is of course useful only if it is used for finding several keys, in which case the table can be used repeatedly. The procedure does not require much additional time (after preparing the table), but it does require a great deal of memory space.

There are also procedures where there is a trade-off between time and memory space, sort of intermediate forms of the procedures above. In AES there are at least

$$2^{128} = 340\,282\,366\,920\,938\,463\,463\,374\,607\,431\,768\,211\,456$$

keys which is thought to prevent the above attacks well enough.

The KP attack on AFFINE-HILL introduced in Section 3.4—and actually on AFFINE also—used differences of plaintexts and the corresponding cryptotexts modulo M to break the system, by removing the nonlinearity caused by affinity. Such a procedure is called *differential cryptanalysis.* A similar procedure can be applied to DES in KP and CP attacks to remove some of the effects of nonlinearity of S-boxes. The minus-side of this is the large number of plaintext-cryptotext pairs needed. *Linear cryptanalysis* tries to use linear dependences between some input and output bits, that may appear in certain inputs. These do exist in DES, and it seems that originally they went totally unnoticed! AES is built to to withstand all these cryptanalyses.

References

1. BAUER, F.L.: *Decrypted Secrets. Methods and Maxims of Cryptography.* Springer–Verlag (2006)

2. BLAKE, I. & SEROUSSI, G. & SMART, N.: *Elliptic Curves in Cryptography.* Cambridge University Press (2000)

3. BUCHMANN, J.: *Introduction to Cryptography.* Springer–Verlag (2004)

4. COHEN, H.: *A Course in Computational Algebraic Number Theory.* Springer–Verlag (2000)

5. CRANDALL, R. & POMERANCE, C.: *Prime Numbers. A Computational Perspective.* Springer–Verlag (2005)

6. DAEMEN, J. & RIJMEN, V.: *Design of Rijndael. AES—The Advanced Encryption Standard.* Springer–Verlag (2002)

7. DING, C. & PEI, D. & SALOMAA, A: *Chinese Remainder Theorem. Applications in Computing, Coding, Cryptography.* World Scientific (1999)

8. DU, D.-Z. & KO, K.-I: *Theory of Computational Complexity.* Wiley (2000)

9. GARRETT, P.: *Making, Breaking Codes. An Introduction to Cryptology.* Prentice–Hall (2007)

10. GOLDREICH, O.: *Modern Cryptography, Probabilistic Proofs, and Pseudorandomness.* Springer–Verlag (2001)

11. GOLDREICH, O.: *Foundations of Cryptography. Basic Tools.* Cambridge University Press (2007)

12. GOLDREICH, O.: *Foundations of Cryptography. Basic Applications.* Cambridge University Press (2009)

13. HOFFSTEIN, J. & PIPHER, J. & SILVERMAN, J.H.: *An Introduction to Mathematical Cryptography.* Springer–Verlag (2008)

14. HOPCROFT, J.E. & ULLMAN, J.D.: *Introduction to Automata Theory, Languages, and Computation.* Addison–Wesley (1979)

15. KATZ, J. & LINDELL, Y.: *Introduction to Modern Cryptography.* Chapman & Hall / CRC (2008)

16. KNUTH, D.E.: *The Art of Computer Programming Vol. 2: Seminumerical Algorithms.* Addison–Wesley (1998)

17. KOBLITZ, N.: *A Course in Number Theory and Cryptography.* Springer–Verlag (2001)

18. KOBLITZ, N.: *Algebraic Aspects of Cryptography.* Springer–Verlag (2004)

19. KONHEIM, A.G.: *Cryptography. A Primer.* Wiley (1981)

20. KRANAKIS, E.: *Primality and Cryptography.* Wiley (1991)

21. LIDL, R. & NIEDERREITER, H.: *Finite Fields.* Cambridge University Press (2008)

22. LIPSON, J.D.: *Elements of Algebra and Algebraic Computing.* Addison–Wesley (1981)

23. MAO, W.: *Modern Cryptography. Theory and Practice.* Pearson Education (2004)

24. MCELIECE, R.J.: *Finite Fields for Computer Scientists and Engineers.* Kluwer (1987)

25. MENEZES, A. & VAN OORSCHOT, P. & VANSTONE, S.: *Handbook of Applied Cryptography.* CRC Press (2001)

26. MIGNOTTE, M.: *Mathematics for Computer Algebra.* Springer–Verlag (1991)

27. MOLLIN, R.A.: *An Introduction to Cryptography.* Chapman & Hall / CRC (2006)

28. MOLLIN, R.A.: *RSA and Public-Key Cryptography.* Chapman & Hall / CRC (2003)

29. MOLLIN, R.A.: *Codes. The Guide to Secrecy from Ancient to Modern Times.* Chapman & Hall / CRC (2005)

30. NIELSEN, M.A. & CHUANG, I.L.: *Quantum Computation and Quantum Information.* Cambridge University Press (2000)

31. PAAR, C. & PELZL, J.: *Understanding Cryptography. A Textbook for Students and Practitioners.* Springer–Verlag (2009)

32. RIESEL, H.: *Prime Numbers and Computer Methods for Factorization.* Birkhäuser (1994)

33. ROSEN, K.H..: *Elementary Number Theory.* Longman (2010)

34. ROSING, M.: *Implementing Elliptic Curve Cryptography.* Manning Publications (1998)

35. SALOMAA, A.: *Public-Key Cryptography.* Springer–Verlag (1998)

36. SCHNEIER, B.: *Applied Cryptography. Protocols, Algorithms, and Source Code in C.* Wiley (1996)

37. SHOUP, V.: *A Computational Introduction to Number Theory and Algebra.* Cambridge University Press (2005)

38. SHPARLINSKI, I.: *Cryptographic Applications of Analytic Number Theory. Complexity Lower Bounds and Pseurandomness.* Birkhäuser (2003)

39. SIERPINSKI, W.: *Elementary Theory of Numbers.* Elsevier (1988)

40. SILVERMAN, J.H. & TATE, J.: *Rational Points on Elliptic Curves.* Springer–Verlag (1992)

41. STINSON, D.R.: *Cryptography. Theory and Practice.* Chapman & Hall / CRC (2006)

42. TRAPPE, W. & WASHINGTON, L.C.: *Introduction to Cryptography with Coding Theory.* Pearson Education (2006)

43. WAGSTAFF, S.S.: *Cryptanalysis of Number Theoretic Ciphers.* Chapman & Hall / CRC (2003)

44. WASHINGTON, L.C.: *Elliptic Curves. Number Theory and Cryptography.* Chapman & Hall / CRC (2008)

Index