Computational Methods in Public Key Cryptology

Arjen K. Lenstra

Citibank, N.A. and Technische Universiteit Eindhoven
1 North Gate Road, Mendham, NJ 07945-3104, U.S.A.
arjen.lenstra@citicorp.com
August 13, 2001

Abstract. These notes informally review the most common methods from computational number theory that have applications in public key cryptology.

Table of Contents

1 Introduction

Cryptology consists of cryptography and cryptanalysis. Cryptography refers to the design and application of information protection methods. Cryptanalysis is the evaluation of the strength of cryptographic methods.

Roughly speaking, there are two types of cryptology: symmetric key cryptology and public key cryptology. In the former a single key is shared (and kept secret) by the communicating parties. It is used for encryption by the sender, and for decryption by the recipient. Examples are the Data Encryption Standard (DES) and the Advanced Encryption Standard (AES). In public key cryptology each party has a key that consists of two parts, a public and a secret (or private) part. The public part can be used to encrypt information, the corresponding secret part to decrypt. Alternatively, the secret key is used to sign a document, the corresponding public key to verify the resulting signature. Furthermore, a widely shared public key can be used to establish a common secret among two parties, for instance a key for a symmetric system. Examples of public key cryptosystems are the Diffie-Hellman key agreement protocol and the ElGamal and RSA encryption and signature schemes (Section 2).

The effectiveness – or security – of symmetric key systems relies on the secrecy of the shared symmetric key and the alleged infeasibility to decrypt without access to the key. Similarly, public key systems rely on the secrecy of the secret key and the infeasibility to decrypt (or sign) without access to the secret key. For public key systems this implies that it should be infeasible to derive the secret key from its corresponding public key. On the other hand, it must be possible to derive the public key from the secret key, or else public key systems could not be realized. Thus, public key systems involve some type of function that is effectively one-way. Given the secret key it must be possible to compute the corresponding public key reasonably efficiently. But deriving the secret from the public key must be impossible, practically speaking.

All currently popular public key systems rely on problems from computational number theory that are widely believed to be hard. As a consequence, computational methods involving integers play an important role in public key cryptology. They are used both to obtain efficient implementations (cryptography) and to provide guidance in key size selection (cryptanalysis). These notes review some of the most important methods from computational number theory that have applications in public key cryptology. Section 2 briefly outlines some of the most popular public key cryptosystems. The most important basic arithmetic methods required for efficient implementations of those systems are described in Section 3. Cryptanalytic methods relevant for the methods from Section 2 are sketched in Section 4.

2 Public Key Cryptography

In this section the basic versions of some of the most popular public key encryption and signature schemes are informally sketched. In Sections 2.2 and 2.4 the same public keys are used for the encryption and signature protocols. This is done for expository purposes only. In general it cannot be recommended to use a single key both for encryption and signature purposes. In practical applications many other peculiarities must be dealt with. They are not taken into account in the descriptions below, see [76].

2.1 Problems that are widely believed to be hard

So far only two supposedly hard problems have found widespread applications in public key cryptography:

1. Integer factorization: given a positive composite integer n, find a non-trivial factor of n.
2. Discrete logarithm: given a generator g of an appropriately chosen group and $h \in \langle g \rangle$, find an integer t such that

$$\underbrace{g \times g \times \ldots \times g}_{t} = g^t = h.$$

   Here it is assumed that the group law is written multiplicatively and indicated by the symbol $\times$. The integer t is unique modulo the order of g, referred to as the *discrete logarithm* of h with respect to g, and denoted by $\log_g(h)$. In the literature it is also referred to as the *index* of h with respect to g.

The integer factorization problem is certainly not always hard, not even if n is large. Indeed, most randomly selected integers (say of some fixed large size) have a relatively small non-trivial factor that can quickly be found. The point is, however, that a hard instance of the integer factoring problem can easily be generated: just pick two sufficiently large primes (Section 3.5) and compute their product. Given just the product there does not seem to be an efficient way to retrieve the primes. In Section 4 the question is addressed what is meant by 'sufficiently large'.

Similarly, the discrete logarithm problem is not always hard. Computing discrete logarithms in $\langle g \rangle$ is straightforward if $g = 1$ generates the additive group of integers $\mathbf{Z}/m\mathbf{Z}$, for any positive integer m. A less trivial example is when g generates the multiplicative group $\mathbf{F}_{p^\ell}^*$ of a finite field $\mathbf{F}_{p^\ell}$. In this case, the hardness of the discrete logarithm problem in $\langle g \rangle$ depends on the size of the largest prime factor of $p^\ell - 1$ (Section 4.1), the size of p^ℓ itself, and the characteristic p of $\mathbf{F}_{p^\ell}$ (Section 4.2). Alternatively, g may just generate a subgroup of $\mathbf{F}_{p^\ell}^*$ (see [101]). In that case the security depends on the size of the largest prime factor of the order of g (which divides $p^\ell - 1$), the characteristic p, and the size of p^d for the smallest $d \le \ell$ (and dividing ℓ) such that the order of g divides

$p^d - 1$ (see [61]). It follows that g should be chosen so that $d = \ell$. It is easy to construct finite fields with multiplicative groups (or subgroups thereof) in which the discrete logarithm problem is believed to be intractable. For appropriately chosen subgroups compression methods based on traces such as LUC [109] and XTR [70] can be used to represent the subgroup elements.

Another popular group where the discrete logarithm problem may be sufficiently hard is the group of points of a properly chosen elliptic curve over a finite field [56, 77]. This has the advantage that the methods from Section 4.2 do not seem to apply. Therefore it is generally believed that the finite field can be chosen much smaller than in the earlier example where $g \in \mathbf{F}_{p^\ell}^*$. In particular for high security applications this should lead to more manageable public key cryptosystems than the ones based on factoring, multiplicative groups, or anything else that is known to be susceptible to the methods from Section 4.2. Appropriate elliptic curves are much harder to find than hard to factor integers or good multiplicative groups. This is a serious obstacle to widespread deployment of elliptic curves in public key cryptography.

A proof that integer factorization is indeed a hard problem has never been published. The alleged difficulty of integer factorization is just a belief and, for the moment at least, nothing more than that. Computing discrete logarithms, on the other hand, can be proved to be hard. The proof, however, applies only to an abstract setting without practical relevance [84, 104]. Thus, also the alleged difficulty of the discrete logarithm problems referred to above is just a belief.

On a quantum computer, factoring and computing discrete logarithms can be done in polynomial time [103]. Believing that factoring and computing discrete logarithms are hard problems implies belief in the impossibility of quantum computing [38].

2.2 RSA

The RSA cryptosystem [98] is named after its inventors Rivest, Shamir, and Adleman. It relies for its security on the difficulty of the integer factoring problem. Each user generates two distinct large primes p and q (Section 3.5), integers e and d such that $ed \equiv 1 \bmod (p-1)(q-1)$ (Section 3.3), and computes $n = pq$. The pair of integers (n, e) is made public. The corresponding p, q, and d are kept secret. This construction satisfies the requirement mentioned in Section 1 that the public key can efficiently be derived from the secret key. The primes p and q can be derived from (n, e) and d. After generation of (n, e) and d, they are in principle no longer needed (but see 3.3.7).

Encryption. A message $m \in \mathbf{Z}/n\mathbf{Z}$ intended for the owner of public key (n, e) is encrypted as $E = m^e \in \mathbf{Z}/n\mathbf{Z}$. The resulting E can be decrypted by computing $D = E^d \in \mathbf{Z}/n\mathbf{Z}$ (see also 3.3.7). It follows from $ed \equiv 1 \bmod (p-1)(q-1)$, Fermat's little theorem, and the Chinese remainder theorem that $D = m$.

Signature generation. A message $m \in \mathbf{Z}/n\mathbf{Z}$ can be signed as $S = m^d \in \mathbf{Z}/n\mathbf{Z}$. The signature S on m can be verified by checking that $m = S^e \in \mathbf{Z}/n\mathbf{Z}$.

In practice the public exponent e is usually chosen to be small. This is done to make the public operations (encryption and signature verification) fast. Care should be taken with the use of small public exponents, as shown in [29, 30, 28, 47]. The secret exponent d corresponding to a small e is in general of the same order of magnitude as n. There are applications for which a small d (and thus large e) would be attractive. However, small private exponents have been shown to make RSA susceptible to attacks [11, 113].

A computational error made in the RSA private operation (decryption and signature generation) may reveal the secret key [10]. These operations should therefore always be checked for correctness. This can be done by applying the corresponding public operation to the result and checking that the outcome is as expected. So-called fault attacks are rendered mostly ineffective by this simple precaution. So-called timing attacks [57] can be made less effective by 'blinding' the private operation. Blinding factors b and b_e are selected by randomly generating $b \in \mathbf{Z}/n\mathbf{Z}$ and computing $b_e = b^{-e}$. Direct computation of x^d for some $x \in \mathbf{Z}/n\mathbf{Z}$, is replaced by the computations of the blinded value $b_e x$, the private operation $y = (b_e x)^d$, and the final outcome $by = x^d$.

2.3 Diffie-Hellman protocol

A key agreement protocol is carried out by two communicating parties to create a shared key. This must be done in such a way that an eavesdropper does not gain any information about the key being generated. The Diffie-Hellman protocol [36] is a key agreement protocol. It is not an encryption or signature scheme.

Let g be a publicly known generator of an appropriately chosen group of known order. To create a shared key, parties A and B proceed as follows:

1. A picks an integer $a \in \{2, 3, \ldots, \mathrm{order}(g) - 1\}$ at random, computes g^a, and sends g^a to B.
2. B receives g^a, picks an integer $b \in \{2, 3, \ldots, \mathrm{order}(g) - 1\}$ at random, computes g^b and g^{ab}, and sends g^b to A.
3. A receives g^b and computes g^{ba}.
4. The shared key is $g^{ab} = g^{ba}$.

The Diffie-Hellman problem is the problem of deriving g^{ab} from g, g^a, and g^b. It can be solved if the discrete logarithm problem in $\langle g \rangle$ can be solved: an eavesdropper can find a from the transmitted value g^a and the publicly known g, and compute g^{ab} based on a and g^b. Conversely, it has not been proved in generality that the Diffie-Hellman problem is as hard as solving the discrete logarithm problem. If g generates a subgroup of the multiplicative groups of a finite field the problems are not known to be equivalent. Equivalence has been proved for the group of points of an elliptic curve over a finite field [12].

A related problem is the Decision Diffie-Hellman problem of efficiently deciding if $g^c = g^{ab}$, given g, g^a, g^b, and g^c. For multiplicative groups of finite fields this problem is believed to be hard. It is known to be easy for at least some elliptic curve groups [52].

2.4 The ElGamal family of protocols

Let g be a publicly known generator of an appropriately chosen group of known order. Let $h = g^t$ for a publicly known h and secret integer t. ElGamal encryption and signature protocols [39] based on a public key consisting of (g, h) come in many different variations. Basic variants are as follows.

Encryption. A message $m \in \langle g \rangle$ intended for the owner of public key (g, h) is encrypted as $(g^k, m \times h^k)$ for a randomly selected $k \in \{2, 3, \ldots, \mathrm{order}(g) - 1\}$. It follows that the party $t = \log_g(h)$ can compute m as

$$m = \frac{m \times h^k}{(g^k)^t}.$$

Signature generation (based on DSA, the Digital Signature Algorithm). Let $\mathrm{order}(g) = q$ for a publicly known prime number q and $t = t \bmod q \in \mathbf{Z}/q\mathbf{Z}$. Let f be a publicly known function from $\langle g \rangle$ to $\mathbf{Z}/q\mathbf{Z}$. To sign a message $m \in \mathbf{Z}/q\mathbf{Z}$, compute $r = f(g^k) \in \mathbf{Z}/q\mathbf{Z}$ for a randomly selected $k \in \{2, 3, \ldots, q - 1\}$ with $f(g^k) \neq 0$ and

$$s = \frac{m + tr}{k} \in \mathbf{Z}/q\mathbf{Z}.$$

The signature consists of $(r, s) \in (\mathbf{Z}/q\mathbf{Z})^2$, unless $s = 0$ in which case another k is selected. Signature $(r, s) \in (\mathbf{Z}/q\mathbf{Z})^2$ is accepted if both r and s are non-zero and $r = f(v)$ where

$$v = g^e \times h^d,$$

with $w = s^{-1} \bmod q$, $e = mw$, and $d = rw$, all in $\mathbf{Z}/q\mathbf{Z}$. This follows from

$$\frac{m + tr}{s} = k \text{ in } \mathbf{Z}/q\mathbf{Z}$$

and

$$g^e \times h^d = g^{mw} \times g^{trw} = g^{\frac{m+tr}{s} \bmod q} = g^k.$$

The encryption requirement that $m \in \langle g \rangle$ can be relaxed by replacing $m \times h^k$ by the encryption of m using some symmetric key system with key h^k (mapped to the proper key space). In the signature scheme, the usage of m in the definition of s can be replaced by the hash of m (mapped to $\mathbf{Z}/q\mathbf{Z}$), so that m does not have to be in $\mathbf{Z}/q\mathbf{Z}$.

2.5 Other supposedly hard problems

Most practical public key system are based on one of only two supposedly hard problems, integer factoring and discrete logarithms. As shown in Section 4 many integer factoring methods allow variants that solve the discrete logarithm problem. The complexity of the two problems may therefore be related. From a practical point of view this is an undesirable situation with a potential for disastrous consequences. If factoring integers and computing discrete logarithms turn out to be easier than anticipated, most existing security solutions can no longer be used.

For that reason there is great interest in hard problems that are suitable for public key cryptography and that are independent of integer factoring and discrete logarithms. Candidates are not easy to find. Once they have been found it is not easy to convince the user community that they are indeed sufficiently hard. Suitable problems for which the hardness can be guaranteed have not been proposed, so that security is a matter of perception only: the longer a system survives, i.e., no serious weaknesses are detected, the more users are willing to believe its security. It will be hard for any newcomer to acquire the level of trust currently enjoyed by the integer factoring and discrete logarithm problems. Some of the latest proposals that are currently under scrutiny are the following.

Lattice based systems. A lattice is a discrete additive subgroup of a fixed dimensional real vector space. Over the past two decades, lattices have been used to great effect in cryptanalysis, most notably to attack so-called knapsack based cryptosystems. More recently, their cryptographic potential is under investigation. The hard problems one tries to exploit are the following:

- The shortest vector problem: find a shortest non-zero element of the lattice, with respect to an appropriately chosen norm.
- The closest vector problem: given an element of the vector space, find a lattice element closest to it, under an appropriately chosen norm.

See [85] for an overview of lattice based cryptology and references to relevant literature.

NTRU and NSS. Let $N > 0$, $q > 0$, and $d < N/2$ be three appropriately chosen integer security parameters. Given a polynomial $h \in (\mathbf{Z}/q\mathbf{Z})[X]$ of degree $< N$, it is supposedly hard to find polynomials $f, g \in (\mathbf{Z}/q\mathbf{Z})[X]$ of degrees $< N$, such that

$$hf = g \bmod (X^N - 1).$$

Here both f and g have d coefficients equal to 1, another d coefficients equal to $(-1 \bmod q)$, and $N-2d$ zero coefficients. Due to the way h is constructed, it is known that such f and g exist. The encryption scheme NTRU [48] and the signature scheme NSS [49] are meant to rely on this problem of recovering f and g from h. This problem can be seen as a lattice shortest vector problem [32]. It was not designed as such. NSS as proposed in [49] is known to have several independent weaknesses [41, 110].

Braid groups. Let Σ_n be the group of n-permutations, for an integer $n > 0$. Let $\sigma_i \in \Sigma_n$ with $i \in \{1, 2, \ldots, n-1\}$ be the permutation that swaps the ith and $(i+1)$st element. The n-braid group B_n is the subgroup of Σ_n generated by σ_1, σ_2, …, σ_{n-1}. In [54] public key systems are described that rely for their security on the difficulty of the following conjugacy problem in B_n: given $x, y \in B_n$ such that $y = bxb^{-1}$ for some unknown $b \in B_m$ with $m \leq n$, find $a \in B_m$ such that $y = axa^{-1}$. See also [5].

3 Basic Computational Methods

This section reviews some of the computational tools for arithmetic on integers that are required to implement the methods from Sections 2.2, 2.3, and 2.4. For a more complete treatment, including runtime analyses, refer to [8, 53, 72, 76].

3.1 Integer arithmetic

The primes used in realistic implementations of number theoretic cryptographic protocols (Section 2) may be many hundreds, and even thousands, of bits long. Thus, the elements of the various groups involved in those protocols cannot be represented by ordinary 32-bit or 64-bit values as available on current computers. Neither can they directly be operated upon using commonly available instructions, such as addition or multiplication modulo 2^{32} on 32-bit integer operands (as common in a programming language such as C).

Implementations of many public key cryptosystems therefore have to rely on a set of multiprecision integer operations that allow arithmetic operations on integers of arbitrary or sufficiently large fixed sizes. It is assumed that routines are available implementing addition, subtraction, multiplication, and remainder with division on integer operands of any size. The design of these basic routines is not the subject of these notes, because most design decisions will, to a large extent, depend on the hardware one intends to use. A variety of different packages is available on the Internet. A basic software only package, written in C, is freely available from the present author [60]. See [105] for a more advanced, but also free, C++ package.

Throughout this section it is assumed that non-negative integers are represented on a computer (or other hardware platform) by their *radix B* representation. Here B is usually a power of 2 that is determined by the wordsize of the computer one is using. On a 32-bit processor $B = 2^{32}$ is a possible, but not always the best, choice. Thus, an integer $m \geq 0$ is represented as

$$m = \sum_{i=0}^{s-1} m_i B^i \text{ for some } s > 0 \text{ and } m_i \in \{0, 1, \ldots, B-1\}.$$

The m_i are referred to as the *blocks* of m. The least and most significant blocks are m_0 and m_{s-1}, respectively. The representation is called *normalized* if $m = m_0 = 0$ and $s = 1$ or if $m \neq 0$ and $m_{s-1} \neq 0$. If a representation is normalized, s is called its *block-length*. A normalized representation is unique.

For integers $m > 0$ and n, the quotient $q = [n/m]$ and the remainder $r = n \bmod m$ are the unique integers with $n = qm + r$ and $r \in \{0, 1, \ldots, m-1\}$. The ring of integers $\mathbf{Z}/m\mathbf{Z}$ is represented by and identified with the set $\{0, 1, \ldots, m-1\}$ of least non-negative residues modulo m. For any integer n, its remainder modulo m is regarded as an element of $\mathbf{Z}/m\mathbf{Z}$ and denoted by $n \bmod m$. It follows that for $u, v \in \mathbf{Z}/m\mathbf{Z}$ the sum $u + v \in \mathbf{Z}/m\mathbf{Z}$ can be computed at the cost of an integer addition and an integer subtraction:

$$\text{compute } w = u + v \in \mathbf{Z} \text{ and } x = w - m,$$

then the sum equals $x \in \mathbf{Z}/m\mathbf{Z}$ if $x \geq 0$ or $w \in \mathbf{Z}/m\mathbf{Z}$ if $x < 0$. The intermediate result $w = u + v$ is at most $2m - 2$, i.e., still of the same order of magnitude as m. Similarly, $u - v \in \mathbf{Z}/m\mathbf{Z}$ can be computed at the cost of an integer subtraction and an integer addition. The computation of the product $uv = (uv) \bmod m \in \mathbf{Z}/m\mathbf{Z}$ is more involved. It requires the computation of a product of integers $< m$, resulting in an intermediate result of twice the order of magnitude of m, followed by the computation of a remainder of the intermediate result modulo m. Because for large m the latter computation is often costly (and cumbersome to implement in hardware), various faster methods have been developed to perform calculations in $\mathbf{Z}/m\mathbf{Z}$. The most popular of these methods is so-called Montgomery arithmetic [78], as described in Section 3.2 below.

3.1.1 Remark on moduli of a special form. Computing the remainder modulo m of the intermediate result uv can be done quickly (and faster than using Montgomery arithmetic) if m is chosen such that it has a special form. For instance, if the most significant block m_{s-1} is chosen to be equal to 1, and $m_i = 0$ for $s/2 < i < s - 1$, then the reduction modulo m can be done at about half the cost of the computation of the product uv. Such moduli can be used in most applications mentioned in Section 2, including RSA [62]. Their usage has so far seen only limited applications in cryptography.

3.2 Montgomery arithmetic

Let $m > 2$ be an integer coprime to B and let $m = \sum_{i=0}^{s-1} m_i B^i$ be its normalized radix B representation. It follows that for common choices of B the integer m must be odd. In cryptographic applications m is usually a large prime or a product of large primes, so requiring m to be odd is not a serious restriction.

3.2.1 Montgomery representation. Let R be the smallest power of B that is larger than m, so $R = B^s$. This R is referred to as the *Montgomery radix.* Its value depends on the value of the ordinary radix B. The Montgomery representation $\tilde{x}$ of $x \in \mathbf{Z}/m\mathbf{Z}$ is defined as

$$\tilde{x} = (xR \bmod m) \in \mathbf{Z}/m\mathbf{Z}.$$

Montgomery arithmetic in $\mathbf{Z}/m\mathbf{Z}$ is arithmetic with the Montgomery representations of elements of $\mathbf{Z}/m\mathbf{Z}$ (but see 3.2.6).

3.2.2 Montgomery addition and subtraction. Let $u, v \in \mathbf{Z}/m\mathbf{Z}$ be represented by their Montgomery representations $\tilde{u}$ and $\tilde{v}$. The Montgomery sum of $\tilde{u}$ and $\tilde{v}$ is the Montgomery representation of the sum $u + v$, i.e., the element $\tilde{z} \in \mathbf{Z}/m\mathbf{Z}$ for which the corresponding z satisfies $z = u + v \in \mathbf{Z}/m\mathbf{Z}$. From

$$\begin{aligned}\tilde{z} = zR \bmod m &= (u + v \bmod m)R \bmod m \\ &\equiv (uR \bmod m + vR \bmod m) \bmod m \\ &= (\tilde{u} + \tilde{v}) \bmod m\end{aligned}$$

it follows that Montgomery addition is the same as ordinary addition in $\mathbf{Z}/m\mathbf{Z}$:

$$\tilde{z} = (\tilde{u} + \tilde{v}) \bmod m.$$

It therefore requires just an integer addition and an integer subtraction. Similarly, Montgomery subtraction is the same as ordinary subtraction in $\mathbf{Z}/m\mathbf{Z}$.

3.2.3 Montgomery multiplication. The Montgomery product of $\tilde{u}$ and $\tilde{v}$ is the element $\tilde{z} \in \mathbf{Z}/m\mathbf{Z}$ for which $z = (uv) \bmod m$:

$$\begin{aligned}\tilde{z} = zR \bmod m &= ((uv) \bmod m)R \bmod m \\ &= (uR \bmod m)v \bmod m \\ &= (uR \bmod m)(vR \bmod m)R^{-1} \bmod m \\ &= \tilde{u}\tilde{v}R^{-1} \bmod m.\end{aligned}$$

Thus, the Montgomery product of $\tilde{u}$ and $\tilde{v}$ is their ordinary integer product divided by the Montgomery radix R modulo m. This can be computed as follows.

Let $w \in \mathbf{Z}$ be the integer product of $\tilde{u}$ and $\tilde{v}$ regarded as elements of $\mathbf{Z}$ (as opposed to $\mathbf{Z}/m\mathbf{Z}$) and let

$$w = \sum_{i=0}^{2s-1} w_i B^i \leq (m-1)^2$$

be w's not necessarily normalized radix B representation. This w must be divided by R modulo m. Division by R modulo m is equivalent to s-fold division by B modulo m, since $R = B^s$. If $w_0 = 0$, division by B can be carried out by shifting out the least significant block $w_0 = 0$ of w:

$$w/B = \sum_{i=0}^{2s-2} w_{i+1} B^i.$$

If $w_i \neq 0$, then a multiple t of m is found such that $(w + t) \bmod B = 0$. The resulting $w + t$ can be divided by B by shifting out its least significant block. But because $w \equiv w + t \bmod m$, dividing $w + t$ by B is equivalent to dividing w by B modulo m. The same process is then applied to the resulting number (w/B or $(w+t)/B$) until the division by B has been carried out s times.

It remains to explain how an appropriate multiple t of m is found with $(w + t) \bmod B = 0$, in the case that $w_0 \neq 0$. Because m and B are coprime, so are $m_0 = m \bmod B$ and B. It follows that there exists an integer $m_0^{-1} \in \{0, 1, \ldots, B-1\}$ such that $m_0 m_0^{-1} \equiv 1 \bmod B$ (see 3.2.7). Consider the integer

$$t = ((B - w_0)m_0^{-1} \bmod B)m = \sum_{i=0}^{s} t_i B^i.$$

From $m_0 m_0^{-1} \equiv 1 \bmod B$ it follows that

$$t_0 = t \bmod B = ((B - w_0)m_0^{-1} \bmod B)m_0 \bmod B = B - w_0.$$

The integer t is a multiple of m and

$$\begin{aligned}(w+t) \bmod B &= (w \bmod B + t \bmod B) \bmod B\\ &= (w_0 + t_0) \bmod B\\ &= (w_0 + B - w_0) \bmod B\\ &= 0.\end{aligned}$$

Thus, $w+t$ is divisible by B.

Although the result of this s-fold division by B modulo m is equivalent to wR^{-1} modulo m, it is not necessarily an element of $\mathbf{Z}/m\mathbf{Z}$. In the course of the above process at most $\sum_{i=0}^{s-1}(B-1)mB^i = m(R-1)$ is added to the original w, after which R is divided out. Therefore, the result is bounded by

$$\frac{w+m(R-1)}{R} \leq \frac{(m-1)^2+m(R-1)}{R} < 2m.$$

It follows that the result can be normalized to $\mathbf{Z}/m\mathbf{Z}$ using at most a single subtraction by m.

The division of w by R modulo m requires code that is very similar to the code for ordinary multiplication of the integers $\tilde{u}$ and $\tilde{v}$. Also, the time required for the division of w by R modulo m is approximately the same as the time required for the computation of w itself (i.e., the multiplication). Thus, the total runtime of Montgomery multiplication modulo m is approximately equal to the time required for two ordinary multiplications of integers of size comparable to m.

The multiplication and division by R modulo m can be merged in such a way that no intermediate result is larger than B^{s+1}. This makes Montgomery multiplication ideally suited for fast and relatively simple hardware implementations.

3.2.4 Montgomery squaring. The Montgomery square of $\tilde{u}$ is the Montgomery product of $\tilde{u}$ and $\tilde{u}$. The square $w = \tilde{u}^2$ (where $\tilde{u}$ is regarded as an element of $\mathbf{Z}$) can be computed in slightly more than half the time of an ordinary product of two integers of about the same size. However, the reduction of the resulting w by R modulo m cannot take advantage of the fact that w is a square. It takes the same time as in the general case. As a result it is reasonable to assume that the time required for a Montgomery squaring is 80% of the time required for a Montgomery product, with the same modulus [25].

3.2.5 Conversion to Montgomery representation. Given $u \in \mathbf{Z}/m\mathbf{Z}$, its Montgomery representation $\tilde{u}$ can be computed as the Montgomery product of u and $R^2 \bmod m$. This follows from the fact that the Montgomery product is the ordinary integer product divided by the Montgomery radix R modulo m:

$$(uR^2 \bmod m)R^{-1} \bmod m = uR \bmod m = \tilde{u}.$$

Similarly, given $\tilde{u}$, the corresponding u can be computed as the Montgomery product of $\tilde{u}$ and 1 (one):

$$\tilde{u}R^{-1} \bmod m = uRR^{-1} \bmod m = u.$$

It follows that conversion to and from the Montgomery representation each require a Montgomery multiplication. It is assumed that the square $R^2 \bmod m$ of the Montgomery radix is computed once and for all per modulus m. In typical applications of Montgomery arithmetic conversions are carried out only before and after a lengthy computation (Section 3.4). The overhead caused by the conversions is therefore minimal. In some applications conversions can be avoided altogether, see 3.5.3.

3.2.6 Subtraction-less Montgomery multiplication. If R is chosen such that $4m < R$ (as opposed to $m < R$), then the subtraction that may be needed at the end of each Montgomery multiplication can be mostly avoided. If Montgomery multiplication with modulus m as in 3.2.3 is applied to two non-negative integers $< 2m$ (as opposed to $< m$), then w as in 3.2.3 is $\leq (2m-1)^2$. The result of the division of w by R modulo m is therefore bounded by

$$\frac{w + m(R-1)}{R} \leq \frac{(2m-1)^2 + m(R-1)}{R} < 2m$$

because $4m < R$. Thus, if $4m < R$ and the Montgomery product with modulus m is computed of two Montgomery representations non-uniquely represented as elements of $\mathbf{Z}/2m\mathbf{Z}$, then the outcome is again an element of $\mathbf{Z}/2m\mathbf{Z}$ if the subtraction is omitted. This implies that, in lengthy operations involving Montgomery products, the subtractions are not needed if all operations are carried out in $\mathbf{Z}/2m\mathbf{Z}$ instead of $\mathbf{Z}/m\mathbf{Z}$. In this case, all intermediate results are non-uniquely represented as elements of $\mathbf{Z}/2m\mathbf{Z}$ instead of $\mathbf{Z}/m\mathbf{Z}$. At the end of the computation each relevant result can then be normalized to $\mathbf{Z}/m\mathbf{Z}$ at the cost of a single subtraction.

3.2.7 Computing the auxiliary inverse for Montgomery arithmetic. The computation of $m_0^{-1} \bmod B$ may be carried out once and for all per modulus m, so it does not have to be done very efficiently. As shown in the C-program fragment in Figure 1, this computation does not require the more involved methods from Section 3.3.

3.3 Euclidean algorithms

The greatest common divisor $\gcd(m, n)$ of two non-negative integers m and n (not both zero) is defined as the largest positive integer that divides both m and n. If $m > 0$ and $n = 0$ then $\gcd(m, n) = m$. If $m, n > 0$, then $\gcd(m, n)$ can be computed by finding the prime factorization of m and n and by determining the product of all common factors. A method that is in general much faster is the following algorithm due to Euclid, from about 300 BC.

Fig. 1. Computation of $m_0^{-1} \bmod B$, for $B = 2^k$ and k the bit-length of an `unsigned long`.

```
unsigned long m0, m0shift, mask, product, result;
<m0 = ‘the least-significant block of the Montgomery modulus m’>
if (!(m0&1)) exit(0);
for (m0shift=(m0<<1), mask = 2, product = m0, result = 1; mask;
     m0shift <<= 1, mask <<= 1)
  if (product & mask) {
    /* invariant: product == m0 * result */
    product += m0shift;
    result += mask;
  }
```

3.3.1 Euclidean algorithm. The Euclidean algorithm is based on the observation that d divides m and n if and only if d divides $m - kn$ and n for arbitrary $k \in \mathbf{Z}$. It follows that if $n > 0$ then

$$\gcd(m, n) = \gcd(n, m \bmod n).$$

Subsequently, if $m \bmod n = 0$, then $\gcd(m, n) = \gcd(n, 0) = n$. However, if $m \bmod n \neq 0$, then

$$\gcd(m, n) = \gcd(n, m \bmod n) = \gcd(m \bmod n, n \bmod (m \bmod n)).$$

Thus, $\gcd(m, n)$ can be computed by replacing (m, n) by $(n, m \bmod n)$ until $n = 0$, at which point m is the greatest common divisor of the original m and n.

The integer n strictly decreases per iteration. This guarantees termination. The most convenient way to prove fast termination is to use least absolute (as opposed to least non-negative) residues, and to use absolute values. Then $[\log_2(n)]$ strictly decreases per iteration, so that $O(\log_2(n))$ iterations suffice. If one incorporates the effect of the decreasing size of the operands, then the overall runtime can be seen to be $O(\log_2(m)\log_2(n))$.

3.3.2 Extended Euclidean algorithm. If the quotients $[m/n]$ are explicitly computed in the Euclidean algorithm, then it can be extended so that it not only computes $d = \gcd(m, n)$, but an integer e such that $ne = d \bmod m$ as well. This is particularly useful when m and n are coprime. In that case $d = 1$ and e is the multiplicative inverse of n modulo m. The ability to quickly compute multiplicative inverses is important in many public key cryptosystems (Section 2 and 3.3.7).

If $ne = d \bmod m$, then $(ne - d)/m$ is an integer, say f, and $ne - fm = \gcd(m, n)$. In the literature this is the identity that is typically solved by the extended Euclidean algorithm.

To solve $ne = d \bmod m$, proceed as follows. Let $s = m$ and $t = n$, then the following two initial equivalences modulo m hold trivially:

$$n \cdot 0 \equiv s \bmod m,$$
$$n \cdot 1 \equiv t \bmod m.$$

This is the initial instance of the following pair of equivalences modulo m:

$$n \cdot u \equiv s \bmod m,$$
$$n \cdot v \equiv t \bmod m.$$

Given such a pair of equivalences, a new pair with 'smaller' right hand sides can be found by applying the ordinary Euclidean algorithm to s and t. If $t \neq 0$ and $s = qt + r$ with $0 \leq r < t$, then $n \cdot (u - vq) \equiv r \bmod m$, so that the new pair is given by

$$n \cdot v \equiv t \bmod m,$$
$$n \cdot (u - vq) \equiv r \bmod m.$$

Thus, per iteration the 4-tuple (s, t, u, v) is replaced by $(t, s \bmod t, v, u - v[s/t])$. This terminates with $t = 0$ and s equal to $d = \gcd(m, n)$, at which point u equals the desired e with $ne \equiv d \bmod m$.

3.3.3 Binary Euclidean algorithm. The efficiency of the Euclidean algorithms from 3.3.1 and 3.3.2 depends on the efficiency of the division operations. On many platforms divisions on small operands are relatively slow compared to additions, shifts (i.e., division or multiplication by a power of 2), and multiplications on small operands. The binary variants, here and in 3.3.4, avoid divisions. They are often faster when the operands are small. For large operands the difference in efficiency between the binary and ordinary versions is much smaller – the ordinary version may even turn out to be faster – because the quotients are on average often quite small. This results in divisions that are often, and for larger block-lengths, fast compared to shifts over several blocks.

The binary variant of the Euclidean algorithm works as follows. Assume that $m > n$ are both positive and odd: common factors 2 are removed and later included in the gcd, any remaining factors 2 can be removed. Because m and n are odd, $m - n > 0$ is even and the odd part $w > 0$ of $m - n$ is at most equal to $(m-n)/2$. Furthermore, $\gcd(m, n) = \gcd(n, w)$, the pair n, w is strictly 'smaller' than the pair m, n, and n and w are again both odd. As soon as $n = w$, then n is the desired gcd. Otherwise replace (m, n) by (n, w) if $n > w$ and by (w, n) if $w > n$.

3.3.4 Binary extended Euclidean algorithm. This combines the ideas from 3.3.2 and 3.3.3. Let $m > n$ be positive and odd, as in 3.3.3, and let

$$n \cdot u \equiv s \bmod m,$$
$$n \cdot v \equiv t \bmod m$$

be a pair of equivalences modulo m as in 3.3.2. Initially $u = 0$, $v = 1$, $s = m$, and $t = n$. Let $s - t = 2^k w$ for some odd w and $k > 0$, then

$$n \cdot ((u - v)/2^k \bmod m) \equiv w \bmod m.$$

Because m is odd, $(u - v)/2^k \bmod m$ can be computed quickly by applying the following k times:

$$x/2 \bmod m = \begin{cases} x/2 & \text{if } x \text{ is even} \\ (x+m)/2 & \text{otherwise.} \end{cases}$$

The resulting iteration is particularly fast for $m, n < B$ (i.e., for m and n that can, in C, be represented by a single `unsigned long` each). For larger m and n it may be faster or slower than the method from 3.3.2, depending on the characteristics of the computer one is using. For large m and n, however, the method from 3.3.5 below is often much faster than either the binary or ordinary method.

3.3.5 Lehmer's method. Let again

$$n \cdot u \equiv s \bmod m,$$
$$n \cdot v \equiv t \bmod m$$

be a pair of equivalences modulo m, with initial values $u = 0$, $v = 1$, $s = m$, and $t = n$. The sequence of operations carried out by the extended Euclidean algorithm described in 3.3.2 depends on the quotients $[s/t]$. Each operation requires an update of s, t, u and v that is relatively costly (since it involves large integers). D.H. Lehmer observed that the process can be made faster by postponing the costly updates. This is done by bounding the sequence of quotients in a cheap way both from below and from above. Only if the two bounds are different the expensive update steps are performed.

Assume that s and t have the same block-length. Let $\bar{s}$ and $\bar{t}$ be the most significant blocks of s and t, respectively (Section 3.1). Application of extended Euclidean algorithm 3.3.2 to $\bar{s}$ and $\bar{t} + 1$ leads to a sequence of quotients, and application to $\bar{s} + 1$ and $\bar{t}$ leads to another sequence of quotients. From

$$\frac{\bar{s}}{\bar{t}+1} < \frac{s}{t} < \frac{\bar{s}+1}{\bar{t}}$$

it follows that as long as the two resulting quotient sequences are the same, the same quotients are obtained if the extended Euclidean algorithm would be applied to s and t. After k iterations the initial pair $(\bar{s}, \bar{t}+1)$ is transformed into

$$(a_k\bar{s} + b_k(\bar{t}+1), c_k\bar{s} + d_k(\bar{t}+1)),$$

for $a_k, b_k, c_k, d_k \in \mathbf{Z}$ that depend on the sequence of quotients. If the first k quotients are the same, then the initial pair $(\bar{s}+1, \bar{t})$ is transformed into

$$(a_k(\bar{s}+1) + b_k\bar{t}, c_k(\bar{s}+1) + d_k\bar{t}),$$

the pair (s, t) would be transformed into

$$(a_k s + b_k t, c_k s + d_k t),$$

and therefore (u, v) into

$$(a_k u + b_k v, c_k u + d_k v).$$

Thus, in Lehmer's method the extended Euclidean algorithm is simultaneously applied to two small initial pairs (without keeping track of the corresponding u's and v's) that depend, as described above, on the 'true' large pair (s, t), while keeping track of the small linear transformation (i.e., the a_k, b_k, c_k, and d_k) on one of the small pairs. As soon as the quotients become different the last linear transformation for which the quotients were the same is applied to (s, t) and the corresponding (u, v). If necessary, the process is repeated (by first determining two new small initial pairs depending on the updated smaller pair (s, t)). The only 'expensive' steps are the occasional updates of the 'true' (s, t) and (u, v). All other steps involve only small numbers. The number of small steps before a large update is proportional to the number of bits in the radix B (Section 3.1).

For large m and n Lehmer's method is substantially faster than either the ordinary or binary versions of the extended Euclidean algorithm. A disadvantage is that it requires a much larger amount of code.

3.3.6 Chinese remaindering. Chinese remaindering is not a Euclidean algorithm, but an application of the ability to compute modular inverses that was not explicitly described in Section 2. Let p and q be two positive coprime integers, and let $r(p) \in \mathbf{Z}/p\mathbf{Z}$ and $r(q) \in \mathbf{Z}/q\mathbf{Z}$. According to the Chinese remainder theorem there exists a unique $r \in \mathbf{Z}/pq\mathbf{Z}$ such that $r \bmod p = r(p)$ and $r \bmod q = r(q)$. This r can be computed as follows:

$$r = r(p) + p\left((r(q) - r(p))(p^{-1} \bmod q) \bmod q\right),$$

where $r(p)$ and $r(q)$ are regarded as elements of $\mathbf{Z}$ and the result is regarded as an element of $\mathbf{Z}/pq\mathbf{Z}$. The integer $p^{-1} \bmod q$ can be computed using one of the extended Euclidean algorithms described above, or using the method from 3.3.8.

If $\gcd(p, q) = d > 1$, a very similar approach can be used, under the necessary and sufficient condition that $r(p) \equiv r(q) \bmod d$, to find r such that $r \bmod p = r(p)$ and $r \bmod q = r(q)$. Apply the above to $p' = p/d$, $r(p') = 0$, $q' = q/d$, and $r(q') = (r(q) - r(p))/d$ to find r' that is 0 modulo p' and $(r(q) - r(p))/d$ modulo q'. Then $r = r(p) + dr'$, since $dr' \equiv 0 \bmod p$ and $dr' \equiv r(q) - r(p) \bmod q$. The resulting r is unique modulo pq/d.

3.3.7 RSA with Chinese remaindering. As an application of Chinese remaindering, consider the private RSA operation, described in Section 2.2:

given integers n, d and an element $w \in \mathbf{Z}/n\mathbf{Z}$, compute $r = w^d \in \mathbf{Z}/n\mathbf{Z}$.

This computation takes about $c \log_2(d) \log_2(n)^2$ seconds (Section 3.4), for some constant c depending on the computer one is using. Since d is usually of the same order of magnitude as n, this becomes $c \log_2(n)^3$ seconds. With Chinese remaindering it can be done about 4 times faster, assuming $n = pq$ for primes p, q of about the same order of magnitude:

1. Compute $d(p) = d \bmod (p-1)$ and $d(q) = d \bmod (q-1)$. These values depend on n and d only, so they can be computed once and for all.
2. Compute $w(p) = w \bmod p \in \mathbf{Z}/p\mathbf{Z}$ and $w(q) = w \bmod q \in \mathbf{Z}/q\mathbf{Z}$.
3. Compute $r(p) = w(p)^{d(p)} \in \mathbf{Z}/p\mathbf{Z}$ and $r(q) = w(q)^{d(q)} \in \mathbf{Z}/q\mathbf{Z}$.
4. Use Chinese remaindering to compute $r \in \mathbf{Z}/pq\mathbf{Z} = \mathbf{Z}/n\mathbf{Z}$ such that $r \bmod p = r(p)$ and $r \bmod q = r(q)$. This r equals $w^d \in \mathbf{Z}/n\mathbf{Z}$.

The exponentiations in Step 3 take about

$$c \log_2(d(p)) \log_2(p)^2 + c \log_2(d(q)) \log_2(q)^2 \approx 2c \log_2(p)^3$$

seconds. The other steps are negligible. Since $\log_2(p) \approx \frac{1}{2}\log_2(n)$ the total runtime becomes $\frac{c}{4}\log_2(p)^3$ seconds.

If n is the product of t primes, then a total speed-up of a factor t^2 is obtained. For RSA it may therefore be attractive to use moduli that consist of more than two prime factors. This is known as *RSA multiprime*. Care should be taken to make sure that the resulting moduli are not susceptible to an attack using the elliptic curve factoring method (4.2.3).

3.3.8 Exponentiation based inversion. If m is prime, then multiplicative inverses modulo m can be computed using exponentiations modulo m (Section 3.4), based on Fermat's little theorem. If $x \in \mathbf{Z}/m\mathbf{Z}$ then $x^m \equiv x \bmod m$ due to m's primality, so that, if $x \neq 0$,

$$x^{m-2} \equiv x^{-1} \bmod m.$$

This way of computing inverses takes $O(\log_2(m)^3)$ operations as opposed to just $O(\log_2(m)^2)$ for the methods based on the Euclidean algorithm. Nevertheless, it may be an option in restricted environments where exponentiation must be available and the space for other code is limited. When combined with Chinese remaindering it can also be used to compute inverses modulo composite m with known factorization.

3.4 Exponentiation

As shown in Section 2 exponentiation is of central importance for many public key cryptosystems. In this subsection the most important exponentiation methods are sketched. See [44] for a complete treatment.

Let g be an element of some group G that is written multiplicatively (with the notation as in Section 2.1). Suppose that $g^e \in G$ must be computed, for some positive integer e. Let $e = \sum_{i=0}^{L-1} e_i 2^i$ with $e_i \in \{0, 1\}$, $L \geq 1$, and $e_{L-1} = 1$.

3.4.1 Square and multiply exponentiation. The most common and simplest exponentiation method is the square and multiply method. It comes in two variants depending on the order in which the bits e_i of e are processed: left-to-right if the e_i are processed for $i = L-1$, $L-2$, ... in succession, right-to-left if they are processed the other way around. The left-to-right variant is the simplest of the two. It works as follows.

1. Initialize r as $g \in G$.
2. For $i = L-2, L-3, \ldots, 0$ in succession, do the following:
   (a) Replace r by $r \times r$.
   (b) If $e_i = 1$, then replace r by $g \times r \in G$.
3. The resulting r equals $g^e \in G$.

In some applications it may be hard to access the e_i from $i = L-2$ downwards. In that case, it may be more convenient to use the right-to-left variant:

1. Initialize s as $g \in G$ and r as the unit element in G.
2. For $i = 0, 1, \ldots, L-1$ in succession, do the following:
   (a) If $e_i = 1$, then replace r by $s \times r \in G$.
   (b) If $i < L-1$, then replace s by $s \times s$.
3. The resulting r equals $g^e \in G$.

If G is the group of the integers (under addition, i.e., g^e actually equals the ordinary integer product eg), then the right-to-left method corresponds to the ordinary schoolbook multiplication method applied to the binary representations of g and e.

For randomly selected L-bit exponents either method performs $L-1$ squarings in G and on average about $\frac{L}{2}$ multiplications in G. If $G = \mathbf{Z}/m\mathbf{Z}$, the group law denoted by $\times$ is multiplication modulo m, and Montgomery arithmetic is used, then one may expect that the average runtime of square and multiply exponentiation is about the same as the time required for $1.3\log_2(e)$ multiplications in $\mathbf{Z}/m\mathbf{Z}$ (see 3.2.4). In many practical circumstances this can be improved using so-called windows exponentiation methods.

3.4.2 Window exponentiation. The window exponentiation method is similar to left-to-right exponentiation, except that the bits of e are processed w bits at a time, for some small *window size* $w \in \mathbf{Z}_{>0}$.

1. Let
$$e = \sum_{i=0}^{\bar{L}-1} \bar{e}_i 2^{wi}, \text{ for } \bar{e}_i \in \{0, 1, \ldots, 2^w - 1\} \text{ and } \bar{e}_{\bar{L}-1} \neq 0,$$
   if $\bar{e}_i \neq 0$, let $\bar{e}_i = \bar{d}_i 2^{\ell_i}$ with $\bar{d}_i < 2^w$ odd and $0 \leq \ell_i < w$, and if $\bar{e}_i = 0$ let $\ell_i = w$.
2. For all odd positive $d < 2^w$ compute $g(d) = g^d$.
3. Initialize r as $g(\bar{d}_{\bar{L}-1})$.
4. For $j = 1, 2, \ldots, \ell_{\bar{L}-1}$ in succession replace r by $r \times r$.

5. For $i = \bar{L}-2, \bar{L}-3, \ldots, 0$ in succession, do the following:
   (a) If $\bar{e}_i \neq 0$, then
      – For $j = 1, 2, \ldots, w - \ell_i$ in succession replace r by $r \times r$.
      – Replace r by $g(\bar{d}_i) \times r$.
   (b) For $j = 1, 2, \ldots, \ell_i$ in succession replace r by $r \times r$.
6. The resulting r equals $g^e \in G$.

For L-bit exponents, the number of squarings in G is $L-1$, as in square and multiply exponentiation. The number of multiplications in G is $2^{w-1}-1$ for Step 2 and at most $\frac{L}{w}$ for Step 5a, for a total of about $2^{w-1}+\frac{L}{w}$. It follows that the optimal window size w is proportional to $\log\log(e)$. For $w = 1$ window exponentiation is identical to left-to-right square and multiply exponentiation.

3.4.3 Sliding window exponentiation. In window exponentiation 3.4.2 the exponent e is split into consecutive 'windows' of w consecutive bits, irrespective of the bit pattern encountered. A slight improvement can be obtained by breaking e up into odd windows of at most w consecutive bits, where the windows are not necessarily consecutive and may be separated by zero bits. There are various ways to achieve this. A convenient way is a greedy approach which can easily be seen to be optimal. It determines the 'next' odd window of at most w consecutive bits in the following way:

– Scan the exponent (from left to right) for the first unprocessed one bit (replacing the prospective result r by $r \times r$ for each zero bit encountered).
– As soon as a one bit is found, determine the largest odd window of at most w consecutive bits that starts with the one bit just found. Let d be the odd integer thus determined, of length $k \leq w$.
– Replace r by $g(d) \times r^{2^k}$.

This method and any other variant that determines the w-bit windows in a flexible way is referred to as sliding window exponentiation.

3.4.4 Multi-exponentiation. Multi-exponentiation refers to the computation of $g^e \times h^d$ for some other element $h \in G$ and $d = \sum_{i=0}^{L-1} d_i 2^i \in \mathbf{Z}_{>0}$ with $d_i \in \{0, 1\}$ (Section 2.4). Obviously, this can be done at the cost of two separate exponentiations, followed by a multiplication in G. It is also possible to combine the two exponentiations. This results in the following multi-exponentiation method.

1. Compute $f = g \times h \in G$.
2. Initialize r as the unit element in G.
3. For $i = L-1, L-2, \ldots, 0$ in succession, do the following:
   (a) Replace r by $r \times r$.
   (b) If $e_i = 1$, then
      – If $d_i = 1$ replace r by $f \times r \in G$.
      – Else if $d_i = 0$ replace r by $g \times r \in G$.

(c) Else if $e_i = 0$, then
   – If $d_i = 1$ replace r by $h \times r \in G$.
4. The resulting r equals $g^e \times h^d \in G$.

There are L squarings in G and, on average for random L-bit e and d, about $\frac{3L}{4}$ multiplications in G. If $G = \mathbf{Z}/m\mathbf{Z}$ and Montgomery arithmetic is used, then one may expect that the average runtime of the above multi-exponentiation is about the same as the time required for $1.55 \log_2(e)$ multiplications in $\mathbf{Z}/m\mathbf{Z}$ (see 3.2.4). Thus, multi-exponentiation is only about 20% slower than single square and multiply exponentiation. Improvements can be obtained by using (sliding) windows. All resulting methods can be generalized to compute the product of more than two powers.

3.4.5 Miscellaneous exponentiation tricks. Because exponentiation methods are of such great importance for public key cryptography, the cryptographic literature contains a wealth of tricks and special-purpose methods. Some of the most important are sketched here.

Exponentiation with precomputation. Suppose that g^e must be computed for the same g and many different e's (of bit-length at most L). In this case it may be worthwhile to precompute and store the L values g^{2^i} for $i = 0, 1, \ldots, L-1$. Each application of the right-to-left square and multiply exponentiation then requires only $(\sum_{i=0}^{L-1} e_i) - 1 \approx (\log_2(e))/2$ multiplications in G. However, storing L precomputed values may be prohibitive. A single precomputed value has, relatively speaking, a much higher pay-off. Given $h = g^{2^{L/2}}$ and writing $e = \bar{e} + 2^{L/2}\bar{\bar{e}}$, the value $g^e = g^{\bar{e}} \times h^{\bar{\bar{e}}}$ can be computed using multi-exponentiation 3.4.4. This takes $\frac{L}{2}$ squarings and on average about $\frac{3L}{8}$ multiplications in G. For $G = \mathbf{Z}/m\mathbf{Z}$ and using Montgomery arithmetic this becomes about $0.78 \log_2(e)$ multiplications in G. Because h can be computed at the cost of $\frac{L}{2}$ squarings in G, this method can also be used to reduce the runtime for ordinary square and multiply exponentiation in general (i.e., not for fixed g): it is reduced from L squarings and about $\frac{L}{2}$ multiplications to $\frac{L}{2} + \frac{L}{2} = L$ squarings and about $\frac{3L}{8}$ multiplications in G. See [18] for more involved and efficient precomputation methods.

Exponentiation with signed exponent representation. Let $e = \sum_{i=0}^{L-1} e_i 2^i$ be the binary representation of an L-bit exponent. Any block of $k+1$ consecutive one bits with maximal $k > 1$, i.e., $e_i = e_{i+1} = \ldots = e_{i+k-1} = 1$ and $e_{i+k} = 0$, can be transformed into $e_i = -1$, $e_{i+1} = \ldots = e_{i+k-1} = 0$, and $e_{i+k} = 1$, without affecting the value of e. Applying this transformation from right to left results in a signed-bit representation of e of length $\leq L+1$ where no two consecutive signed-bits are non-zero (and where a signed-bit is in $\{-1, 0, 1\}$). Such a representation is unique, for random e the number of non-zero signed-bits (the *weight*) is $\log_2(e)/3$ on average, and it is of minimal weight among the signed-bit representations of e.

It follows that if g^{-1} is available, g^e can be computed using a variation of square and multiply exponentiation at the cost of L squarings and on average $\log_2(e)/3$ multiplications in G. This may be worthwhile for groups G where

the computation of g^{-1} given $g \in G$ can be performed at virtually no cost. A common example is the group of points of an elliptic curve over a finite field.
Exponentiation with Frobenius. Let $\mathbf{F}_{p^\ell}$ be a finite field of characteristic p. Often elements of $\mathbf{F}_{p^\ell}$ can be represented in such a way that the Frobenius map that maps $x \in \mathbf{F}_{p^\ell}$ to $x^p \in \mathbf{F}_{p^\ell}$ is essentially for free (3.6.3). If that is the case, $G = \mathbf{F}_{p^\ell}^*$ or a subgroup thereof, and $e > p$, the computation of g^e can take advantage of the free Frobenius map. Let $e = \sum_{i=0}^{L'-1} f_i p^i$ be the radix p representation of e, then g^{f_i} for $0 \le i < L'$ can be computed at the cost of $\log_2(p)$ squarings in G and, on average, about $\frac{L'}{2}\log_2(p) \approx \log_2(e)/2$ multiplications in G (using L'-fold right-to-left square and multiply exponentiation). The value g^e then follows with an additional $L'-1$ multiplications in G and $L'-1$ applications of the Frobenius map.

If G is the group of points over $\mathbf{F}_{p^\ell}$ of an elliptic curve defined over $\mathbf{F}_p$, then the Frobenius endomorphism on G may be computable for free. As shown in [55] this may be used to speed-up exponentiation (or rather scalar multiplication, as it should be referred to (Remark 3.7.2)).

3.5 Prime generation

3.5.1 The prime number theorem. The ability to quickly generate primes of almost any size is of great importance in cryptography. That this is possible is in the first place due to the prime number theorem:

> Let $\pi(x)$ be the prime counting function, the number of primes $\le x$. The function $\pi(x)$ behaves as $\frac{x}{\log(x)}$ for $x \to \infty$.

The prime number theorem says that a randomly selected L-bit number has reasonably large probability, namely more than $\frac{1}{L}$, to be prime, for any L of cryptographic interest. It then remains to find out if the candidate is prime or not. That can be done quickly, in all practical circumstances, using a variation of Fermat's little theorem, as shown in 3.5.2 below. Thus, searching for primes can be done quickly.

The prime number theorem also holds in arithmetic progressions:

> Let $\pi(x, m, a)$ for integers m, a with $\gcd(m, a) = 1$ be the number of primes $\le x$ that are congruent to a modulo m. The function $\pi(m, x, a)$ behaves as $\frac{x}{\phi(m)\log(x)}$ for $x \to \infty$, where $\phi(m)$ is the Euler phi function, the number of positive integers $\le m$ which are relatively prime to m.

It follows that primes in certain residue classes, such as $(3 \bmod 4)$ or 1 modulo some other prime, can be found just as quickly by limiting the search to the desired residue class. It also follows that the search can be made more efficient by restricting it to numbers free of small divisors in certain fixed residue classes.

3.5.2 Probabilistic compositeness testing. Fermat's little theorem says that

$$a^p = a \text{ for } a \in \mathbf{Z}/p\mathbf{Z} \text{ and prime } p.$$

It follows that if n and a are two integers for which $\gcd(n, a) = 1$ and

$$a^{n-1} \not\equiv 1 \bmod n,$$

then n cannot be a prime number. Fermat's little theorem can thus be used to prove the compositeness of a composite n without revealing a non-trivial factor of n. Based on the methods from Section 3.4 this can be done efficiently, if a suitable a can be found efficiently. The latter is in general, however, not the case: there are infinitely many composite numbers n, so-called Carmichael numbers, for which $a^{n-1} \equiv 1 \bmod n$ for all a coprime to n. This was shown in [4].

Suitable a's always exist, and can easily be found, for Selfridge's variation of Fermat's little theorem:

> If p is an odd prime, $p - 1 = 2^t \cdot u$ for integers t, u with u odd, and a is an integer not divisible by p, then
>
> $$\text{either } a^u \equiv 1 \bmod p \text{ or } a^{2^i u} \equiv -1 \bmod p \text{ for some } i \text{ with } 0 \le i < t.$$

Let n be an odd composite number and $n = 2^t \cdot u$ for integers t, u with u odd. An integer $a \in \{2, 3, \ldots, n-1\}$ is called a witness to the compositeness of n if

$$a^u \not\equiv 1 \bmod n \text{ and } a^{2^i u} \not\equiv -1 \bmod n \text{ for } i \text{ with } 0 \le i < t.$$

For odd composite n and a randomly selected $a \in \{2, 3, \ldots, n-1\}$ there is a probability of at least 75% that a is witness to the compositeness of n, as shown in [97]. It follows that in practice the compositeness of any composite n can efficiently be proved:

Probabilistic compositeness test

> Let n be an odd positive number, and let k be a positive integer. Randomly and independently select at most k values $a \in \{2, 3, \ldots, n-1\}$ until an a is found that is a witness to n's compositeness, or until all a's have been tried and no witness has been found. In the former case a proof of n's compositeness has been found and n is declared to be composite. In the latter case declare 'failure'.

If n is an odd composite, then the chance that a randomly selected a is not a witness is less than $\frac{1}{4}$. Therefore, the chance that an odd composite n escapes to be declared composite by the probabilistic compositeness test with inputs n and k is less than $(\frac{1}{4})^k$.

3.5.3 Probabilistic compositeness test using Montgomery arithmetic.

If n is subjected to a probabilistic compositeness test, the computations of a^u and $a^{2^i u}$ can be carried out in Montgomery arithmetic, even though a is generated as an ordinary integer in $\{2, 3, \ldots, n-1\}$ and **not** converted to its Montgomery representation. All one has to do is change the '$\not\equiv 1$' and '$\not\equiv -1$' tests to '$\not\equiv R$' and '$\not\equiv -R$', respectively, where R is the Montgomery radix corresponding to n, as in 3.2.1.

3.5.4 Generating industrial primes. Combined with the prime number theorem (3.5.1), the probabilistic compositeness test from 3.5.2 makes it possible to generate L-bit primes quickly. Randomly select an odd L-bit number, and subject it to the probabilistic compositeness test with some appropriately chosen value k. If the number is declared to be composite, one tries again by selecting another L-bit number at random. Otherwise, if the probabilistic compositeness test declares 'failure', the selected number is called a *probable prime*, because if it were composite that would have been proved with overwhelming probability (namely, larger than $1 - (\frac{1}{4})^k$).

A probable prime – or, following Carl Pomerance, an *industrial prime* – is a number for which many attempts to prove its compositeness failed, not a number that was proved to be prime. In all practical circumstances, however, it is a prime number and if k is chosen properly it can be used without reservations in cryptographic applications. If odd L-bit random numbers are selected uniformly and independently, then the resulting (probable) primes will be uniformly distributed over the L-bit primes. See [76] for a further discussion on the probability that a probable prime is composite, and the choice of k.

Based on the prime number theorem, one may expect that on average approximately $\frac{L}{2}\log(2)$ composite n's are selected before a (probable) prime is found. It may also be expected that, for the composites, the first attempted witness is indeed a witness (because the 75% probability is on average much higher for random odd composite n). It follows that the total number of exponentiations modulo n (required for the computation of $a^u \bmod n$) is about $\frac{L}{2}\log(2) + k$: one for each 'wrong' guess, and k for the final 'correct' choice. In the remainder of this subsection it is shown how the $\frac{L}{2}\log(2)$ term can be reduced, at the cost of less costly computations or by giving up the uniformity of the distribution of the resulting primes.

Similar methods and estimates apply to the generation of primes with specific properties, such as primes that are 3 mod 4 or 1 modulo some other (large) prime.

3.5.5 Remark on provable primality. Probable primes can be proved to be prime – if they are indeed prime – using a general purpose primality test. Compared to the probabilistic compositeness test from 3.5.2, these tests are rather involved. They are hardly relevant for cryptology, and are therefore beyond the scope of these notes. See [8, 64] or [3, 14, 23, 24, 42, 82] for more details.

It is also possible to generate primes uniformly in such a way that very simple primality proofs (based on Pocklington's theorem and generalizations thereof [8, 14, 64]) can be applied to them. See [73] for details.

3.5.6 Prime generation with trial division. Most random odd numbers have a small factor. Therefore, most 'wrong' guesses in 3.5.4 can be cast out much faster by finding their smallest prime factor than by attempting to find a witness. This can, for instance, conveniently be done by subjecting each candidate to trial division with the primes up to some bound B (rejecting a candidate as soon as a divisor is found). The ones for which no small factor is found are subjected to the probabilistic compositeness test. The choice for B depends on the desired length L and the relatively speeds of trial division and modular exponentiation. It is best determined experimentally. This approach leads to a substantial speed-up.

3.5.7 Prime generation with a wheel. Let P be the product of a small number of (small) primes. Trial division with the primes dividing P can be avoided by selecting L-bit random numbers of the form $aP+b$ for appropriately sized random integers a and b with b coprime to P. Selection of b can be done by picking random b with $0 < b < P$ until $\gcd(b, P) = 1$. For small P the computation of $\gcd(b, P)$ is very fast, so that this may on average turn out to be faster than trial division with the primes in P. Alternatively, if P is small enough, b can be selected from the precomputed and stored set $C_P = \{c : 0 < c < P, \gcd(c, P) = 1\}$. Given a candidate value $aP + b$ that is coprime to P, it can either right away be subjected to the probabilistic compositeness test, or it may be faster to perform trial division with larger primes first.

If for each choice of a a value b with $0 < b < P$ is randomly selected (from C_P or not), the resulting primes are again uniformly distributed. If for each choice of a all values in C_P are tried in order, then this method is referred to as a *wheel*. Primes resulting from a wheel-based search are no longer uniformly distributed. For RSA based cryptography this may be considered to be a (negligibly small) security risk, since the prime factors are the secrets and should in principle be indistinguishable from random primes. For discrete logarithm based cryptography usage of such primes does in general not pose an additional security risk.

3.5.8 Prime generation with sieving. An even faster way to perform the trial divisions is by using a *sieve*. It also results in even greater non-uniformity, but the security impact (even for RSA) is still negligible. Let $(s(i))_{i=0}^{[cL]-1}$ be an array of $[cL]$ bits, for a small constant c. Initially $s_i = 0$ for all i. Let n be a random L-bit integer and let B' be the sieving bound. For all primes $p \leq B'$, set the bit $s(j)$ to one if $n + j$ is divisible by p. This is done by replacing $s(kp - (n \bmod p))$ by one for all integers k such that $0 < kp - (n \bmod p) < [cL]$. For each p this requires the computation of $n \bmod p$ (once) and about $\frac{cL}{p}$ additions and bit assignments. This process is referred to as *sieving* and the array s is called a *sieve*. After sieving with all $p \leq B'$, the locations j for which $s(j)$ has not been touched, i.e., $s(j) = 0$, correspond to integers $n + j$ that are not divisible by the primes $\leq B'$. Those integers $n + j$ are subjected to the probabilistic compositeness test, until a probable prime is found. Based on 3.5.1 it may be expected that this will happen if c is large enough (say 2 or 3).

In general sieving works much faster than individual trial division such as in 3.5.6. As a result the optimal sieving bound B' is usually substantially larger than the optimal trial division bound B. Given L, the best B' is best determined experimentally.

3.6 Finite field arithmetic

Fast arithmetic in finite fields plays an important role in discrete logarithm based cryptography, both using a traditional multiplicative group of a finite field, and using the group of an elliptic curve over a finite field. For prime fields refer to Sections 3.1, 3.2, and 3.3 for addition, multiplication, and inversion in $\mathbf{Z}/m\mathbf{Z}$ with m prime (so that $\mathbf{Z}/m\mathbf{Z}$ is a finite field). In principle prime fields suffice for most cryptographic applications. Extension fields, however, have certain practical advantages over prime fields because they may allow faster arithmetic without compromising security. This subsection briefly reviews some of the most important notions and methods for arithmetic in finite extensions of prime fields. Let $\mathbf{F}_{p^\ell}$ denote a finite field of p^ℓ elements, where p is prime and $\ell > 1$. So, $\mathbf{F}_p$ is the prime field underlying $\mathbf{F}_{p^\ell}$.

3.6.1 Basic representation. Let $f(X) \in \mathbf{F}_p[X]$ be an irreducible polynomial of degree ℓ and let α be such that $f(\alpha) = 0$. Then $\{\alpha^0, \alpha^1, \alpha^2, \ldots, \alpha^{\ell-1}\}$ forms a basis for $\mathbf{F}_{p^\ell}$ over $\mathbf{F}_p$, i.e., the set

$$S = \{\sum_{i=0}^{\ell-1} a_i\alpha^i \,:\, a_i \in \mathbf{F}_p\}$$

is isomorphic to $\mathbf{F}_{p^\ell}$. This follows from the fact that $\#S = p^\ell$ (otherwise f would not be irreducible) and because an addition and a multiplication can be defined on S that make it into a finite field. For $a = \sum_{i=0}^{\ell-1} a_i\alpha^i, b = \sum_{i=0}^{\ell-1} b_i\alpha^i \in S$ the sum $a + b$ is defined as

$$a + b = \sum_{i=0}^{\ell-1}(a_i + b_i)\alpha^i \in S$$

and $-a = \sum_{i=0}^{\ell-1} -a_i\alpha^i \in S$. The product ab is defined as the remainder modulo $f(\alpha)$ of the polynomial $\left(\sum_{i=0}^{\ell-1} a_i\alpha^i\right) \times \left(\sum_{i=0}^{\ell-1} b_i\alpha^i\right)$ of degree $2\ell - 2$ in α. Thus $ab \in S$. Because f is irreducible, $\gcd(a, f) = 1$ for any $a \in S \setminus \{0\}$. The inverse $a^{-1} \in S$ can therefore be found be generalizing the extended Euclidean algorithm (3.3.2) from $\mathbf{Z}$ to $\mathbf{F}_p[X]$. Note that $S = (\mathbf{F}_p[X])/(f(X))$.

To make the reduction modulo $f(\alpha)$ easier to compute it is customary to assume that f is monic, i.e., that its leading coefficient equals one. Also, it is advantageous for the reduction to select f in such a way that it has many zero coefficients, for the same reason that moduli of a special form are advantageous (3.1.1). A 'Montgomery-like' approach to avoid the reduction modulo f, though easy to realize, is not needed because it would be slower than reduction modulo a well chosen f.

If $g(X) \in \mathbf{F}_p[X]$ is another irreducible polynomial of degree ℓ with, say, $g(\beta) = 0$, then elements of $\mathbf{F}_{p^\ell}$ can be represented as $\sum_{i=0}^{\ell-1} b_i\beta^i$ with $b_i \in \mathbf{F}_p$. An effective isomorphism between the two representations of $\mathbf{F}_{p^\ell}$ can be found by finding a root of $g(X)$ in $(\mathbf{F}_p[X])/(f(X))$. This can be done efficiently using a method to factor polynomials over finite fields [53].

To represent the elements of a finite field $\mathbf{F}_{p^{\ell k}}$ the above construction can be used with an irreducible polynomial in $\mathbf{F}_p[X]$ of degree ℓk. Alternatively $\mathbf{F}_{p^{\ell k}}$ can be constructed as a kth degree extension of $\mathbf{F}_{p^\ell}$ using an irreducible polynomial in $\mathbf{F}_{p^\ell}[X]$ of degree k. It depends on the circumstances and sizes of p, ℓ, and k which of these two methods is preferable in practice.

3.6.2 Finding an irreducible polynomial. A random monic polynomial of degree ℓ in $\mathbf{F}_p[X]$ is irreducible with probability about $\frac{1}{\ell}$. This result is comparable to the prime number theorem (3.5.1 and the definition of norm in 4.2.2). Irreducibility of polynomials in $\mathbf{F}_p[X]$ can be tested in a variety of ways [53]. For instance, $f \in \mathbf{F}_p[X]$ of degree ℓ is irreducible if and only if

$$\gcd(f, X^{p^i} - X) = 1 \text{ for } i = 1, 2, \ldots, [\ell/2].$$

This follows from the facts that $X^{p^i} - X \in \mathbf{F}_p[X]$ is the product of all monic irreducible polynomials in $\mathbf{F}_p[X]$ of degrees dividing i, and that if f is irreducible it has a factor of degree at most $[\ell/2]$. This irreducibility condition can be tested efficiently using generalizations of the methods described in Section 3.3. Irreducible polynomials can thus be found quickly. In practice, however, it is often desirable to impose some additional restrictions on f. Some of these are sketched in 3.6.3 below.

3.6.3 Optimal normal basis. If the elements of $\mathbf{F}_{p^\ell}$ can be represented as $\sum_{i=0}^{\ell-1} a_i \alpha^{p^i}$, with $a_i \in \mathbf{F}_p$ and $f(\alpha) = 0$ for some irreducible $f(X) \in \mathbf{F}_p[X]$ of degree ℓ, then $\{\alpha, \alpha^p, \ldots, \alpha^{p^{\ell-1}}\}$ is called a normal basis for $\mathbf{F}_{p^\ell}$ over $\mathbf{F}_p$. A normal basis representation has the advantage that pth powering consists of a single circular shift of the coefficients:

$$\left(\sum_{i=0}^{\ell-1} a_i \alpha^{p^i}\right)^p = \sum_{i=0}^{\ell-1} a_i^p \alpha^{p^{i+1}} = \sum_{i=0}^{\ell-1} a_{((i-1) \bmod \ell)} \alpha^{p^i},$$

since $a_i^p = a_i$ for $a_i \in \mathbf{F}_p$ and $\alpha^{p^\ell} = \alpha$. As shown in 3.4.5 this may make exponentiation in $\mathbf{F}_{p^\ell}$ cheaper.

As an example, let $\ell + 1$ be prime, and let p be odd and a primitive root modulo $\ell + 1$, i.e., $p \bmod (\ell + 1)$ generates $(\mathbf{Z}/(\ell+1)\mathbf{Z})^*$. The polynomial

$$f(X) = \frac{X^{\ell+1} - 1}{X - 1} = X^\ell + X^{\ell-1} + \ldots + X + 1$$

is irreducible over $\mathbf{F}_p$. If $f(\alpha) = 0$, then $f(\alpha^{p^i}) = 0$ for $i = 0, 1, \ldots, \ell-1$, and $\alpha^i = \alpha^{i \bmod (\ell+1)}$ because $\alpha^{\ell+1} = 1$. With the fact that p is a primitive root modulo $\ell+1$ it follows that $\{\alpha^i : 1 \le i \le \ell\}$ is the same as $\{\alpha^{p^i} : 0 \le i < \ell\}$. Because the latter is a normal basis, pth powering using the basis $\{\alpha^i : 1 \le i \le \ell\}$ is just a permutation of the coefficients. Furthermore, with the basis $\{\alpha^i : 1 \le i \le \ell\}$ the reduction stage of the multiplication in $\mathbf{F}_{p^\ell}$ is easy. It takes just $\ell - 2$ additions and ℓ subtractions in $\mathbf{F}_p$, since $\alpha^i = \alpha^{i \bmod (\ell+1)}$ and $\alpha^0 = -\alpha^\ell - \alpha^{\ell-1} - \ldots - \alpha$. This is an example of an *optimal* normal basis. The above construction fully characterizes optimal normal bases for odd characteristics. For a definition of an optimal basis and characteristic 2 existence results and constructions, see for instance [75].

3.6.4 Inversion using normal bases. As shown in [50], normal bases can be used for the computation of inverses in $\mathbf{F}_{p^\ell}^*$. Let p be odd and $x \in \mathbf{F}_{p^\ell}^*$ then

$$x^{-1} = (x^r)^{-1} x^{r-1}$$

for any integer r. The choice $r = \frac{p^\ell - 1}{p-1}$ makes the computation of x^{-1} particularly fast: first compute x^{r-1} (as described below), compute $x^r = x x^{r-1}$, note that $x^r \in \mathbf{F}_p$ because its $(p-1)$th power equals 1, compute $(x^r)^{-1} \in \mathbf{F}_p$ using a relatively fast inversion in $\mathbf{F}_p$, and finally compute $x^{-1} = (x^r)^{-1} x^{r-1}$ using ℓ multiplications in $\mathbf{F}_p$. Since $r - 1 = p^{\ell-1} + p^{\ell-2} + \ldots + p^2 + p$, computation of x^{r-1} takes about $\log_2(m-1)$ multiplications in $\mathbf{F}_{p^\ell}$ and (free) p^jth powerings for various j, by observing that

$$(x^{p+p^2+\ldots+p^j})(x^{p+p^2+\ldots+p^j})^{p^j} = x^{p+p^2+\ldots+p^{2j}}.$$

Refer to [50] for further details on the computation of x^{r-1} and a similar construction for characteristic 2.

3.6.5 Finding a generator. A primitive normal basis for $\mathbf{F}_{p^\ell}$ over $\mathbf{F}_p$ is a normal basis as in 3.6.3 where α is a generator (or *primitive element*) of $\mathbf{F}_{p^\ell}^*$. In this case the irreducible polynomial $f(X)$ with $f(\alpha) = 0$ is called primitive. Primitive normal bases always exist. In general there is, however, no fast method to find a generator of $\mathbf{F}_{p^\ell}^*$. If the factorization of $p^\ell - 1$ is known, then the best method is to pick random elements $x \in \mathbf{F}_q^*$ until $x^{(p^\ell-1)/q} \neq 1$ for all distinct prime factors q of $p^\ell - 1$.

In cryptographic applications a generator of the full multiplicative group is hardly ever needed. Instead, a generator of an order q subgroup of $\mathbf{F}_{p^\ell}^*$ suffices, for some prime q dividing $p^\ell - 1$. Such a generator can be found by picking random elements $x \in \mathbf{F}_q^*$ until $x^{(p^\ell-1)/q} \neq 1$, in which case $x^{(p^\ell-1)/q}$ is the desired generator. The prime q must be chosen as a factor of the ℓth cyclotomic polynomial. For $q > \ell$ this guarantees that q does not divide $p^d - 1$ for any $d \leq \ell$ (and dividing ℓ) so that $\langle g \rangle$ cannot be embedded in a proper subfield of $\mathbf{F}_{p^\ell}$ (see [61]). The ℓth cyclotomic polynomial $\phi_\ell(X)$ is defined as follows:

$$X^\ell - 1 = \prod_{1 \leq d \leq \ell, d \text{ dividing } \ell} \phi_d(X).$$

3.7 Elliptic curve arithmetic

Elliptic curves were introduced in cryptanalysis with the invention of the elliptic curve factoring method (4.2.3) in 1984 [71]. This triggered applications of elliptic curves in cryptography and primality testing.

Early cryptographic applications concentrated on elliptic curves over fields of characteristic 2. This restriction was inspired by the relatively high computational demands of elliptic curve cryptography, and the possibility to realize competitive hardware implementations in characteristic 2. Nowadays they still enjoy a wide popularity. However, larger characteristic elliptic curves are becoming increasingly common in cryptography. These notes focus on the very basics of elliptic curves over fields of characteristic > 3. For a more general and complete treatment refer to [9, 106].

3.7.1 Elliptic curves and elliptic curve groups. Let $p > 3$ be prime. Any pair $a, b \in \mathbf{F}_{p^\ell}$ such that $4a^3 + 27b^2 \neq 0$ defines an *elliptic curve* $E_{a,b}$ over $\mathbf{F}_{p^\ell}$. Let $E = E_{a,b}$ be an elliptic curve over $\mathbf{F}_{p^\ell}$. The *set of points* $E(\mathbf{F}_{p^\ell})$ over $\mathbf{F}_{p^\ell}$ of E is informally defined as the pairs $x, y \in \mathbf{F}_{p^\ell}$ satisfying the *Weierstrass equation*

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

along with the *point at infinity* $\mathcal{O}$. More precisely, let a projective point $(x : y : z)$ over $\mathbf{F}_{p^\ell}$ be an equivalence class of triples $(x, y, z) \in (\mathbf{F}_{p^\ell})^3$ with $(x, y, z) \neq (0, 0, 0)$. Two triples (x, y, z) and (x', y', z') are equivalent if $cx = x'$, $cy = y'$, and $cz = z'$ for $c \in \mathbf{F}^*_{p^\ell}$. Then

$$E(\mathbf{F}_{p^\ell}) = \{(x : y : z) : \ y^2 z = x^3 + axz^2 + bz^3\}.$$

The unique point with $z = 0$ is the point at infinity and denoted $\mathcal{O} = (0 : 1 : 0)$. The other points correspond to solution $\frac{x}{z}, \frac{y}{z} \in \mathbf{F}_{p^\ell}$ to the Weierstrass equation. They may or may not be normalized to have $z = 1$.

The set $E(\mathbf{F}_{p^\ell})$ is an abelian group, traditionally written additively. The group law is defined as follows. The point at infinity is the zero element, i.e., $P + \mathcal{O} = \mathcal{O} + P = P$ for any $P \in E(\mathbf{F}_{p^\ell})$. Let $P, Q \in E(\mathbf{F}_{p^\ell}) \setminus \{\mathcal{O}\}$ with normalized representations $P = (x_1 : y_1 : 1)$ and $Q = (x_2 : y_2 : 1)$. If $x_1 = x_2$ and $y_1 = -y_2$ then $P + Q = \mathcal{O}$, i.e., the opposite $-(x : y : z)$ is given by $(x : -y : z)$. Otherwise, let

$$\lambda = \begin{cases} \frac{y_1 - y_2}{x_1 - x_2} & \text{if } x_1 \neq x_2 \\ \frac{3x_1^2 + a}{2y_1} & \text{if } x_1 = x_2 \end{cases}$$

and $x = \lambda^2 - x_1 - x_2$, then $P + Q = (x : \lambda(x_1 - x) - y_1 : 1)$. This *elliptic curve addition* allows an easy geometric interpretation. If $P \neq Q$, then $-(P+Q)$ is the third point satisfying the Weierstrass equation on the line joining P and Q. If $P = Q$, then $-(P+Q)$ is the second point satisfying the equation on the tangent to the curve in the point $P = Q$. The existence of this point is a consequence of the condition that a, b defines an elliptic curve over $\mathbf{F}_{p^\ell}$, i.e., $4a^3 + 27b^2 \neq 0$.

Two elliptic curves $a, b \in \mathbf{F}_{p^\ell}$ and $a', b' \in \mathbf{F}_{p^\ell}$ are *isomorphic* if $a' = u^4 a$ and $b' = u^6 b$ for some $u \in \mathbf{F}_{p^\ell}^*$. The corresponding isomorphism between $E_{a,b}(\mathbf{F}_{p^\ell})$ and $E_{a',b'}(\mathbf{F}_{p^\ell})$ sends $(x : y : z)$ to $(u^2 x : u^3 y : z)$.

Computing in the group of points of an elliptic curve over a finite field involves computations in the underlying finite field. These computations can be carried out as set forth in Section 3.6. For extension fields bases satisfying special properties (such as discussed in 3.6.3) may turn out to be useful. In particular optimal normal bases in characteristic 2 (not treated here for elliptic curves) are advantageous, because squaring in the finite fields becomes a free operation.

3.7.2 Remark on additive versus multiplicative notation. In Section 3.4 algorithms are described to compute g^e for g in a group G, where g^e stands for $g \times g \times \ldots \times g$ and $\times$ denotes the group law. In terms of 3.7.1, the group G and the element g correspond to $E(\mathbf{F}_{p^\ell})$ and some $P \in E(\mathbf{F}_{p^\ell})$, respectively, and $\times$ indicates the elliptic curve addition. The latter is indicated by $+$ in 3.7.1 for the simple reason that the group law in $E(\mathbf{F}_{p^\ell})$ is traditionally written additively. Given the additive notation in $E(\mathbf{F}_{p^\ell})$ it is inappropriate to denote $P+P+\ldots+P$ as P^e and to refer to this operation as exponentiation. The usual notation eP is used instead. It is referred to as 'scalar multiplication' or 'scalar product'. In order to compute eP, the methods from Section 3.4 to compute g^e can be applied. Thus, 'square and multiply exponentiation' can be used to compute eP. Given the additive context it may, however, be more appropriate to call it 'double and add scalar multiplication'.

3.7.3 Elliptic curve group representations. There are several ways to represent the elements of $E(\mathbf{F}_{p^\ell})\backslash\{\mathcal{O}\}$ and to perform the elliptic curve addition. For the group law as described in 3.7.1 the elements can be represented using *affine coordinates*, i.e., as $(x, y) \in (\mathbf{F}_{p^\ell})^2$, where (x, y) indicates the point $(x : y : 1)$. If affine coordinates are used the group law requires an inversion in $\mathbf{F}_{p^\ell}$. This may be too costly. If *projective coordinates* are used, i.e., elements of $E(\mathbf{F}_{p^\ell}) \setminus \{\mathcal{O}\}$ are represented as $(x : y : z)$ and not necessarily normalized, then the inversion in $\mathbf{F}_{p^\ell}$ can be avoided. However, this comes at the cost of increasing the number of squarings and multiplications in $\mathbf{F}_{p^\ell}$ per application of the group law.

Another representation that is convenient in some applications is based on the *Montgomery model* of elliptic curves. In the Montgomery model the coordinates of the group elements satisfy an equation that is slightly different from the usual Weierstrass equation. An isomorphic curve in the Weierstrass model can, however, easily be found, and vice versa. As a consequence, group elements can be represented using just two coordinates, as in the affine case. Furthermore, an inversion is not required for the group law, as in the projective case. The group law is more efficient than the group law for the projective case, assuming that the difference $P - Q$ is available whenever the sum $P + Q$ of P and Q must be computed. This condition makes it impossible to use the ordinary square and multiply exponentiation (Remark 3.7.2) to compute scalar products. Refer to [13, 79] for detailed descriptions of the Montgomery model and a suitable algorithm to compute a scalar multiplication in this case.

Refer to [25] for a comparison of various elliptic curve point representations in cryptographic applications.

3.7.4 Elliptic curves modulo a composite. The field $\mathbf{F}_{p^\ell}$ in 3.7.1 and 3.7.3 can be replaced by the ring $\mathbf{Z}/n\mathbf{Z}$ for a composite n (coprime to 6). An elliptic curve over $\mathbf{Z}/n\mathbf{Z}$ is defined as a pair $a, b \in \mathbf{Z}/n\mathbf{Z}$ with $4a^3+27b^2 \in (\mathbf{Z}/n\mathbf{Z})^*$. The set of points, defined in a way similar to 3.7.1, is again an abelian group. The group law so far has limited applications in cryptology. Instead, for cryptanalytic purposes it is more useful to define a *partial addition* on $E(\mathbf{Z}/n\mathbf{Z})$, simply by performing the group law defined in 3.7.1 in $\mathbf{Z}/n\mathbf{Z}$ as opposed to $\mathbf{F}_{p^\ell}$. Due to the existence of zero divisors in $\mathbf{Z}/n\mathbf{Z}$ the resulting operation can break down (which is the reason that it is called partial addition) and produce a non-trivial divisor of n instead of the desired sum $P + Q$. As will be shown in 4.2.3, this is precisely the type of 'accident' one is hoping for in cryptanalytic applications.

3.7.5 Elliptic curve modulo a composite taken modulo a prime. Let p be any prime dividing n. The elliptic curve $a, b \in \mathbf{Z}/n\mathbf{Z}$ as in 3.7.4 and the set of points $E_{a,b}(\mathbf{Z}/n\mathbf{Z})$ can be mapped to an elliptic curve $\bar{a} = a \bmod p, \bar{b} = b \bmod p$ over $\mathbf{F}_p$ and set of points $E_{\bar{a},\bar{b}}(\mathbf{F}_p)$. The latter is done by reducing the coordinates of a point $P \in E_{a,b}(\mathbf{Z}/n\mathbf{Z})$ modulo p resulting in $P_p \in E_{\bar{a},\bar{b}}(\mathbf{F}_p)$. Let $P, Q \in E(\mathbf{Z}/n\mathbf{Z})$. If $P + Q \in E(\mathbf{Z}/n\mathbf{Z})$ is successfully computed using the partial addition, then $P_p + Q_p$ (using the group law in $E(\mathbf{F}_p)$) equals $(P + Q)_p$. Furthermore, $P = \mathcal{O}$ if and only if $P_p = \mathcal{O}_p$.

3.7.6 Generating elliptic curves for cryptographic applications. For cryptographic applications one wants elliptic curves for which the cardinality $\#E(\mathbf{F}_{p^\ell})$ of the group of points is either prime or the product of a small number and a prime. It is known that $\#E(\mathbf{F}_{p^\ell})$ equals $p^\ell + 1 - t$ for some integer t with $|t| \le 2\sqrt{p^\ell}$ (Hasse's theorem [106]). Furthermore, if E is uniformly distributed over the elliptic curves over $\mathbf{F}_{p^\ell}$, then $\#E(\mathbf{F}_{p^\ell})$ is approximately uniformly distributed over the integers close to $p^\ell + 1$. Thus, if sufficiently many elliptic curves are selected at random over some fixed finite field, it may be expected (3.5.1) that a suitable one will be found after a reasonable number of attempts.

It follows that for cryptographic applications of elliptic curves it is desirable to have an efficient and, if at all possible, simple way to determine $\#E(\mathbf{F}_{p^\ell})$, in order to check if $\#E(\mathbf{F}_{p^\ell})$ satisfies the condition of being prime or almost prime. The first polynomial-time algorithm solving this so-called elliptic curve point counting problem was due to Schoof [102]. It has been improved by many different authors (see [9]). Although enormous progress has been made, the point counting problem is a problem one tries to avoid in cryptographic applications. This may be done by using specific curves with known properties (and group cardinalities), or by using curves from a prescribed list prepared by some trusted party. However, if one insists on a randomly generated elliptic curve (over, say, a randomly generated finite field), one will have to deal with the point counting. As a result, parameter selection in its full generality for elliptic curve cryptography must still considered to be a nuisance. This is certainly the case when compared to the ease of parameter selection in RSA or systems relying on the discrete logarithm problem in ordinary multiplicative groups of finite fields. Refer to the proceedings of recent cryptographic conferences for progress on the subject of point counting and elliptic curve parameter initialization.

Let E be a suitable elliptic curve, i.e., $\#E(\mathbf{F}_{p^\ell}) = sq$ for a small integer s and prime q. A generator of the order q subgroup of $E(\mathbf{F}_{p^\ell})$ can be found in a way similar to 3.6.5. Just pick random points $P \in E(\mathbf{F}_{p^\ell})$ until $sP \neq \mathcal{O}$ (Remark 3.7.2). The desired generator is then given by sP.

Given an elliptic curve E over $\mathbf{F}_{p^\ell}$, the set of points $E(\mathbf{F}_{p^{\ell k}})$ can be considered over an extension field $\mathbf{F}_{p^{\ell k}}$ of $\mathbf{F}_{p^\ell}$. In particular the case $\ell = 1$ is popular in cryptographic applications. This approach facilitates the computation of $\#E(\mathbf{F}_{p^{\ell k}})$ (based on Weil's theorem [74]). However, it is frowned upon by some because of potential security risks. Also, elliptic curves E over $\mathbf{F}_{p^\ell}$ and their corresponding group $E(\mathbf{F}_{p^\ell})$ with ℓ composite are not considered to be a good choice. See [108] and the references given there.

4 Factoring and Computing Discrete Logarithms

Let n be an odd positive composite integer, let $g \in G$ be an element of known order of a group of known cardinality $\#G$, and let $h \in \langle g \rangle$. If required, the probabilistic compositeness test from 3.5.2 can be used to ascertain that n is composite. Checking that $h \in \langle g \rangle$ can often be done by verifying that $h \in G$ and that $h^{\text{order}(g)} = 1$. It is assumed that the elements of G are uniquely represented. The group law in G is referred to as 'multiplication'. If $G = E(\mathbf{F}_p)$ then this multiplication is actually elliptic curve addition (Remark 3.7.2).

This section reviews the most important methods to solve the following two problems:

Factoring

Find a not necessarily prime factor p of n with $1 < p < n$.

Computing discrete logarithms

Find $\log_g(h)$, i.e., the integer $t \in \{0, 1, \ldots, \text{order}(g) - 1\}$ such that $g^t = h$.

For generic G and prime order(g) it has been proved that finding $\log_g(h)$ requires, in general, $c\sqrt{\text{order}(g)}$ group operations [84, 104], for a constant $c > 0$. However, this generic model does not apply to any practical situation. For integer factoring no lower bound has been published.

4.1 Exponential-time methods

In this subsection the basic exponential-time methods are sketched:

Factoring: methods that may take n^c operations on integers of size comparable to n,

Computing discrete logarithms: methods that may take order$(g)^c$ multiplications in G,

where c is a positive constant. These runtimes are called exponential-time because $n^c = e^{c \log(n)}$, so that the runtime is an exponential function of the input length $\log(n)$.

4.1.1 Exhaustive search. Factoring n by exhaustive search is referred to as *trial division*: for all primes in succession check if they divide n, until a proper divisor is found. Because n has a prime divisor $\leq \sqrt{n}$, the number of division attempts is bounded by $\pi(\sqrt{n}) \approx \frac{\sqrt{n}}{\log(\sqrt{n})}$ (see 3.5.1). More than 91% of all positive integers have a factor < 1000. For randomly selected composite n trial division is therefore very effective. It is useless for composites as used in RSA

Similarly, $\log_g(h)$ can be found by comparing g^t to h for $t = 0, 1, 2, \ldots$ in succession, until $g^t = h$. This takes at most order(g) multiplications in G. There are no realistic practical applications of this method, unless order(g) is very small.

4.1.2 Pollard's $p-1$ method [88]. According to Fermat's little theorem (see 3.5.2), $a^{p-1} \equiv 1 \bmod p$ for prime p and any integer a not divisible by p. It follows that $a^k \equiv 1 \bmod p$ if k is an integer multiple of $p-1$. Furthermore, if p divides n, then p divides $\gcd(a^k-1, n)$. This may make it possible to find a prime factor p of n by computing $\gcd(a^k-1, n)$ for an arbitrary integer a with $1 < a < n$ and a coprime to n (assuming one is not so lucky to pick an a with $\gcd(a, n) \neq 1$). This works if one is able to find a multiple k of $p-1$.

The latter may be possible if n happens to have a prime factor p for which $p-1$ consists of the product of some small primes. If that is the case then k can be chosen as a product of many small prime powers. Obviously, only some of the primes dividing k actually occur in $p-1$. But $p-1$ is not known beforehand, so one simply includes as many small prime powers in k as feasible, and hopes for the best, i.e., that the resulting k is a multiple of $p-1$. The resulting k may be huge, but the number a^k-1 has to be computed only modulo n. Refer to [79] for implementation details.

As a result a prime factor p of n can be found in time proportional to the largest prime factor in $p-1$. This implies that the method is practical only if one is lucky and the largest prime factor in $p-1$ happens to be sufficiently small. For reasonably sized p, such as used as factors of RSA moduli, the probability that p can be found using Pollard's $p-1$ method is negligible. Nevertheless, the existence of Pollard's $p-1$ method is the reason that many cryptographic standards require RSA moduli consisting of primes p for which $p-1$ has a large prime factor. Because a variation using $p+1$ (the 'next' cyclotomic polynomial) follows in a straightforward fashion [115], methods have even been designed to generate primes p that can withstand both a $p-1$ and a $p+1$ attack. This conveniently overlooks the fact that there is an attack for each cyclotomic polynomial [7]. However, in view of the elliptic curve factoring method (4.2.3) and as argued in [99] all these precautions do not make sense.

4.1.3 The Silver-Pohlig-Hellman method [87]. Just as p dividing n can be found easily if $p-1$ is the product of small primes, discrete logarithms in $\langle g \rangle$ can be computed easily if $\text{order}(g)$ has just small factors. Assume that $\text{order}(g)$ is composite and that q is a prime dividing $\text{order}(g)$. From $g^t = h$ it follows that

$$\left(g^{\frac{\text{order}(g)}{q}}\right)^{t \bmod q} = h^{\frac{\text{order}(g)}{q}}.$$

Thus, t modulo each of the primes q dividing $\text{order}(g)$ can be found by solving the discrete logarithm problems for the $\frac{\text{order}(g)}{q}$th powers of g and h. If $\text{order}(g)$ is a product of distinct primes, then t follows using the Chinese remainder theorem (3.3.6). If $\text{order}(g)$ is not squarefree, then use the extension as described in [64].

The difficulty of a discrete logarithm problem therefore depends mostly on the size of the largest prime factor in $\text{order}(g)$. Because the factorization of $\#G$ is generally assumed to be known (and in general believed to be easier to find than discrete logarithms in G), the order of g is typically assumed to be a sufficiently large prime in cryptographic applications.

4.1.4 Shanks' baby-step-giant-step [53, Exercise 5.17].

Let $s = [\sqrt{\mathrm{order}(g)}] + 1$. Then for any $t \in \{0, 1, \ldots, \mathrm{order}(g) - 1\}$ there are non-negative integers $t_0, t_1 \leq s$ such that $t = t_0 + t_1 s$. From $g^t = h$ it follows that $hg^{-t_0} = g^{t_1 s}$. The values g^{is} for $i \in \{0, 1, \ldots, s\}$ are computed (the 'giant steps') at the cost of about s multiplications in G, and put in a hash table. Then, for $j = 0, 1, \ldots, s$ in succession hg^{-j} is computed (the 'baby steps') and looked up in the hash table, until a match is found. The value t can be derived from j and the location of the match. This (deterministic) method requires at most about $2\sqrt{\mathrm{order}(g)}$ multiplications in G. This closely matches the lower bound mentioned above. The greatest disadvantage of this method is that it requires storage for $\sqrt{\mathrm{order}(g)}$ elements of G. Pollard's rho method, described in 4.1.5 below, requires hardly any storage and achieves essentially the same speed.

4.1.5 Pollard's rho method [89]. The probability that among a group of 23 randomly selected people at least two people have the same birthday is more than 50%. This probability is much higher than most people would expect. It is therefore referred to as the birthday paradox. It lies at the heart of the most effective general purpose discrete logarithm algorithm, Pollard's rho method. If elements are drawn at random from $\langle g \rangle$ (with replacement) then the expected number of draws before an element is drawn twice (a so-called 'collision') is $\sqrt{\mathrm{order}(g)\pi/2}$. Other important cryptanalytic applications are the use of 'large primes' in subexponential-time factoring and discrete logarithm methods (Section 4.2) and collision search for hash functions (not treated here).

Application to computing discrete logarithms. A collision of randomly drawn elements from $\langle g \rangle$ it in itself not useful to solve the discrete logarithm problem. The way this idea is made to work to compute discrete logarithms is as follows. Define a (hopefully) random walk on $\langle g \rangle$ consisting of elements of the form $g^e h^d$ for known e and d, wait for a collision, i.e., e, d and e', d' such that $g^e h^d = g^{e'} h^{d'}$, and compute $\log_g(h) = \frac{e-e'}{d'-d} \bmod \mathrm{order}(g)$. A 'random' walk $(w_i)_{i=1}^{\infty}$ on $\langle g \rangle$ can, according to [89], be achieved as follows. Partition G into three subsets G_1, G_2, and G_3 of approximately equal cardinality. This can usually be done fairly accurately based on the representation of the elements of G. One may expect that this results in three sets $G_j \cap \langle g \rangle$, for $j = 1, 2, 3$, of about the same size. Take $w_1 = g$ (so $e = 1$, $d = 0$), and define w_{i+1} as a function of w_i:

$$w_{i+1} = \begin{cases} hw_i & \text{if } w_i \in G_1 \quad ((e,d) \to (e, d+1)) \\ w_i^2 & \text{if } w_i \in G_2 \quad ((e,d) \to (2e, 2d)) \\ gw_i & \text{if } w_i \in G_3 \quad ((e,d) \to (e+1, d)). \end{cases}$$

This can be replaced by any other function that allows easy computation of the exponents e and d and that looks sufficiently random. It is not necessary to compare each new w_i to all previous ones (which would make the method as slow as exhaustive search). According to *Floyd's cycle-finding algorithm* it suffices to compare w_i to w_{2i} for $i = 1, 2, \ldots$ (see [53]). A pictorial description of the sequence $(w_i)_{i=0}^{\infty}$ is given by a ρ (the Greek character rho): starting at the tail of the ρ it iterates until it bites in its own tail, and cycles from there on.

As shown in [111] partitioning G into only three sets does in general not lead to a truly random walk. In practice that means that the collision occurs somewhat later than it should. Unfortunately a truly random walk is hard to achieve. However, as also shown in [111], if G is partitioned into substantially more than 3 sets, say about 15 sets, then the collision occurs on average almost as fast as it would for a truly random walk. An improvement of Floyd's cycle-finding algorithm is described in [15].

Parallelization. If m processors run the above method independently in parallel, each starting at its own point $w_1 = g^e h^d$ for random e, d, a speed-up of a factor $\sqrt{m}$ can be expected. A parallelization that achieves a speed-up of a factor m when run on m processors is described in [112]. Define *distinguished points* as elements of $\langle g \rangle$ that occur with relatively low probability θ and that have easily recognizable characteristics. Let each processor start at its own randomly selected point $w_1 = g^e h^d$. As soon as a processor hits a distinguished point the processor reports the distinguished point to a central location and starts afresh. In this way m processors generate $\geq m$ independent 'trails' of average length $1/\theta$. Based on the birthday paradox, one may expect that $\sqrt{\mathrm{order}(g)\pi/2}$ trail points have to be generated before a collision occurs among the trails, at an average cost of $(\sqrt{\mathrm{order}(g)\pi/2})/m$ steps per processor. However, this collision itself goes undetected. It is only detected, at the central location, at the first distinguished point after the collision. Each of the two contributing processors therefore has to do an additional expected $1/\theta$ steps to reach that distinguished point. For more details, also on the choice of θ, see [112].

Pollard's parallelized rho is currently the method of choice to attack the discrete logarithm problem in groups of elliptic curves. Groups of well over 2^{100} elements can successfully be attacked. For the most recent results, consult [22].

Pollard's lambda method for catching kangaroos. There is also a non-parallelized discrete logarithm method that is based on just two trails that collide, thus resembling a λ (the Greek character lambda). It finds t in about $2\sqrt{w}$ applications of the group law if t is known to lie in a specified interval of width w. See [89] for a description of this method in terms of tame and wild kangaroos and [90] for a speed-up based on the methods from [112].

Application to factoring. The collision idea can also be applied in the context of factoring. If elements are drawn at random from $\mathbf{Z}/n\mathbf{Z}$ (with replacement) then the expected number of draws before an element is drawn that is identical modulo p to some element drawn earlier is $\sqrt{p\pi/2}$. Exponents do not have to be carried along, a random walk in $\mathbf{Z}/n\mathbf{Z}$ suffices. According to [89] this can be achieved by picking $w_1 \in \mathbf{Z}/n\mathbf{Z}$ at random and by defining

$$w_{i+1} = (w_i^2 + 1) \bmod n.$$

With Floyd's cycle-finding algorithm one may expect that $\gcd(w_i - w_{2i}, n) > 1$ after about $\sqrt{p}$ iterations. In practice products of $|w_i - w_{2i}|$ for several consecutive i's are accumulated (modulo n) before a gcd is computed.

One of the earliest successes of Pollard's rho method was the factorization of the eighth Fermat number $F_8 = 2^{2^8} + 1$. It was found because the factor happened to be unexpectedly small [17]. See also 4.2.6.

4.2 Subexponential-time methods

4.2.1 The L function. For $t, \gamma \in \mathbf{R}$ with $0 \le t \le 1$ the notation $L_x[t, \gamma]$ introduced in [64] is used for any function of x that equals

$$e^{(\gamma+o(1))(\log x)^t (\log\log x)^{1-t}}, \text{ for } x \to \infty.$$

For $t = 0$ this equals $(\log x)^\gamma$ and for $t = 1$ it equals x^γ (up to the $o(1)$ in the exponent). It follows that for $0 \le t \le 1$ the function $L_x[t, \gamma]$ interpolates between polynomial-time ($t = 0$) and exponential-time ($t = 1$). Runtimes equal to $L_x[t, \gamma]$ with $0 < t < 1$ are called subexponential-time in x because they are asymptotically less than $e^{c \log(x)}$ for any constant c.

In this subsection the basic subexponential-time methods for factoring and computing discrete logarithms in $\mathbf{F}_{p^\ell}^*$ are sketched:

Factoring: methods for which the number of operations on integers of size comparable to n is expected to be $L_n[t, \gamma]$ for $n \to \infty$,

Computing discrete logarithms in $\mathbf{F}_{p^\ell}^*$: methods for which the number of multiplications in $\mathbf{F}_{p^\ell}$ is expected to be $L_{p^\ell}[t, \gamma]$ for $p \to \infty$ and fixed ℓ, or fixed p and $\ell \to \infty$,

where $\gamma > 0$ and t are constants with $0 < t < 1$. For most methods presented below these (probabilistic) runtimes cannot rigorously be proved. Instead, they are based on heuristic arguments.

The discrete logarithm methods presented below work only for $G = \mathbf{F}_{p^\ell}^*$, because explicit assumptions have to be made about properties of the representation of group elements. Thus, unlike Section 4.1 it is explicitly assumed that $G = \mathbf{F}_{p^\ell}^*$, and that g generates $\mathbf{F}_{p^\ell}^*$.

4.2.2 Smoothness.

Integers. A positive integer is B-smooth (or simply smooth if B is clear from the context) if all its prime factors are $\le B$. Let $\alpha, \beta, r, s \in \mathbf{R}_{>0}$ with $s < r \le 1$. It follows from [19, 35] that a random positive integer $\le L_x[r, \alpha]$ is $L_x[s, \beta]$-smooth with probability

$$L_x[r - s, -\alpha(r - s)/\beta], \text{ for } x \to \infty.$$

Polynomials over $\mathbf{F}_p$. Assume that, as in 3.6.1, elements of $\mathbf{F}_{p^\ell}^*$ with p prime and $\ell > 1$ are represented as non-zero polynomials in $\mathbf{F}_p[X]$ of degree $< \ell$. The *norm* of $h \in \mathbf{F}_{p^\ell}^*$ in this representation is defined as $p^{\text{degree}(h)}$.

A polynomial in $\mathbf{F}_p[X]$ is B-smooth if it factors as a product of irreducible polynomials in $\mathbf{F}_p[X]$ of norm $\le B$. Let $\alpha, \beta, r, s \in \mathbf{R}_{>0}$ with $r \le 1$ and $\frac{r}{100} < s < \frac{99r}{100}$. It follows from [86] that a random polynomial in $\mathbf{F}_p[X]$ of norm $\le L_x[r, \alpha]$ is $L_x[s, \beta]$-smooth with probability

$$L_x[r - s, -\alpha(r - s)/\beta], \text{ for } x \to \infty.$$

Note the similarity with integer smoothness probability.

4.2.3 Elliptic Curve Method [71]. Rephrasing Pollard's $p-1$ method 4.1.2 in terms of 4.2.2, it attempts to find p dividing n for which $p-1$ is B-smooth by computing $\gcd(a^k-1, n)$ for an integer k that consists of the primes $\leq B$ and some of their powers. Most often the largest prime in $p-1$ is too large to make this practical. The elliptic curve method is similar to Pollard's $p-1$ method in the sense that it tries to take advantage of the smoothness of a group order ($\#\mathbf{Z}/p\mathbf{Z}^*$ in Pollard's $p-1$ method): if the group order is smooth a randomly generated unit in the group (a^k) may lead to a factorization. In Pollard's $p-1$ method the groups are fixed as the groups $\mathbf{Z}/p\mathbf{Z}^*$ for the primes p dividing the number one tries to factor. The elliptic curve method randomizes the choice of the groups (and their orders). Eventually, if one tries often enough, a group of smooth order will be encountered and a factorization found.

Let $a, b \in \mathbf{Z}/n\mathbf{Z}$ be randomly selected so that they define an elliptic curve E over $\mathbf{Z}/n\mathbf{Z}$ (see 3.7.4). According to 3.7.5 an elliptic curve $E_p = E_{a \bmod p, b \bmod p}$ over $\mathbf{F}_p$ is defined for each prime p dividing n, and $\#E_p(\mathbf{F}_p)$ behaves as a random integer close to $p+1$ (see 3.7.6). Based on 4.2.2 with $r=1$, $\alpha=1$, $s=1/2$, and $\beta=\sqrt{1/2}$ it is not unreasonable to assume that $\#E_p(\mathbf{F}_p) = L_p[1,1]$ is $L_p[1/2, \sqrt{1/2}]$-smooth with probability $L_p[1/2, -\sqrt{1/2}]$. Thus, for a fixed p, once every $L_p[1/2, \sqrt{1/2}]$ random elliptic curves over $\mathbf{Z}/n\mathbf{Z}$ one expects to find a curve for which the group order $\#E_b(\mathbf{F}_p)$ is $L_p[1/2, \sqrt{1/2}]$-smooth.

Assume that $\#E_p(\mathbf{F}_p)$ is $L_p[1/2, \sqrt{1/2}]$-smooth, let k be the product of the primes $\leq L_p[1/2, \sqrt{1/2}]$ and some of their powers, and let P be a random element of $E(\mathbf{Z}/n\mathbf{Z})$. If one attempts to compute kP in $E(\mathbf{Z}/n\mathbf{Z})$ using the partial addition defined in 3.7.4 and the computation does not break down, then the result is some point $R \in E(\mathbf{Z}/n\mathbf{Z})$. According to 3.7.5 the point $R_p \in E_p(\mathbf{F}_p)$ would have been obtained by computing the elliptic curve scalar product of k and the point $P_p \in E_p(\mathbf{F}_p)$ as defined in 3.7.5. If enough prime powers are included in k, then the order of $P_p \in E_p(\mathbf{F}_p)$ divides k, so that $R_p = \mathcal{O}_p \in E_p(\mathbf{F}_p)$, where $\mathcal{O}_p$ is the zero element in $E_p(\mathbf{F}_p)$. But, according to 3.7.5, $R_p = \mathcal{O}_p$ implies $R = \mathcal{O}$. The latter implies that $R_q = \mathcal{O}_q$ for any prime q dividing n. It follows that, if R has been computed successfully, k must be a multiple of the order of P when taken modulo any prime dividing n. Given how much luck is already involved in picking E such that $\#E_p(\mathbf{F}_p)$ is smooth for one particular p dividing n, it is unlikely that this would happen for all prime factors of n simultaneously. Thus if E was randomly selected in such a way that $\#E_p(\mathbf{F}_p)$ is $L_p[1/2, \sqrt{1/2}]$-smooth, it is much more likely that R has not been computed to begin with, but that the partial addition broke down, i.e., produced a non-trivial factor of n. From $\mathcal{O}_p = (0:1:0)$ it follows that p divides that factor.

Since one in every $L_p[1/2, \sqrt{1/2}]$ elliptic curves over $\mathbf{Z}/n\mathbf{Z}$ can be expected to be lucky, the total expected runtime is $L_p[1/2, \sqrt{1/2}]$ times the time required to compute kP, where k is the product of powers of primes $\leq L_p[1/2, \sqrt{1/2}]$. The latter computation requires $L_p[1/2, \sqrt{1/2}]$ partial additions, i.e., cost proportional to $\log(n)^2 L_p[1/2, \sqrt{1/2}]$. The total cost is proportional to

$$L_p[1/2, \sqrt{1/2}] \cdot \log(n)^2 L_p[1/2, \sqrt{1/2}] = \log(n)^2 L_p[1/2, \sqrt{2}].$$

It follows that using the elliptic curve method small factors can be found faster than large factors. For $p \approx \sqrt{n}$, the worst case, the expected runtime becomes $L_n[1/2, 1]$. For RSA moduli it is known that the worst case applies. For composites without known properties and, in particular, a smallest factor of unknown size, one generally starts off with a relatively small k aimed at finding small factors. This k is gradually increased for each new attempt, until a factor is found or until the factoring attempt is aborted. See [13, 79] for implementation details of the elliptic curve method. The method is ideally parallelizable: any number of attempts can be run independently on any number of processors in parallel, until one of them is lucky.

The reason that the runtime argument is heuristic is that $\#E_p(\mathbf{F}_p)$ is contained in an interval of short length (namely, about $4\sqrt{p}$) around $p + 1$. Even though $\#E_p(\mathbf{F}_p)$ cannot be distinguished from a truly random integer in that interval, intervals of short length around $p + 1$ cannot be proved (yet) to be smooth with approximately the same probability as random integers $\leq p$. The heuristics are, however, confirmed by experiments and the elliptic curve method so far behaves as heuristically predicted.

Remarkable successes of the elliptic curve method were the factorizations of the tenth and eleventh Fermat numbers, $F_{10} = 2^{2^{10}} + 1$ and $F_{11} = 2^{2^{11}} + 1$; see [16]. Factors of over 50 decimal digits have occasionally been found using the elliptic curve method. The method is not considered to be a threat against RSA, but its existence implies that care should be taken when using RSA moduli consisting of more than two prime factors.

A variant of the elliptic curve method suitable for the computation of discrete logarithms has never been published.

4.2.4 The Morrison-Brillhart approach. The expected runtimes of all factoring methods presented so far depend strongly on properties of the factor to be found, and only polynomially on the size of the number to be factored. For that reason they are referred to as *special purpose* factoring methods. All factoring methods described in the sequel are *general purpose* methods: their expected runtimes depend just on the size of the number n to be factored. They are all based, in some way or another, on Fermat's factoring method of solving a congruence of squares modulo n: try to find integers x and y such that their squares are equal modulo n, i.e., $x^2 \equiv y^2 \bmod n$. Such x and y are useful for factoring purposes because it follows from $x^2 - y^2 \equiv 0 \bmod n$ that n divides $x^2 - y^2 = (x - y)(x + y)$, so that

$$n = \gcd(n, x - y)\gcd(n, x + y).$$

This may yield a non-trivial factorization of n. There is a probability of at least 50% that this produces a non-trivial factor of n if n is composite and not a prime power and x and y are random solutions to $x^2 \equiv y^2 \bmod n$.

Fermat's method to find x and y consists of trying $x = [\sqrt{n}] + 1, [\sqrt{n}] + 2, \ldots$ in succession, until $x^2 - n$ is a perfect square. In general this cannot be expected to be competitive with any of the methods described above, not even trial division. Morrison and Brillhart [83] proposed a faster way to find x and y. Their general approach is not to solve the identity $x^2 \equiv y^2 \bmod n$ right away, but to construct x and y based on a number of other identities modulo n which are, supposedly, easier to solve. Thus, the Morrison-Brillhart approach consists of two stages: a first stage where a certain type of identities modulo n are found, and a second stage where those identities are used to construct a solution to $x^2 \equiv y^2 \bmod n$. This approach applies to all factoring and discrete logarithm algorithms described below.

Let B be a smoothness bound (4.2.2) and let P be the *factor base*: the set of primes $\leq B$ of cardinality $\#P = \pi(B)$ (see 3.5.1). In the first stage, one collects $> \#P$ integers v such that $v^2 \bmod n$ is B-smooth:

$$v^2 \equiv \left(\prod_{p \in P} p^{e_{v,p}} \right) \bmod n.$$

These identities are often referred to as *relations* modulo n, and the first stage is referred to as the *relation collection stage*. Morrison and Brillhart determined relations using continued fractions (4.2.5). Dixon [37] proposed a simpler, but slower, method to find relations: pick $v \in \mathbf{Z}/n\mathbf{Z}$ at random and keep the ones for which $v^2 \in \mathbf{Z}/n\mathbf{Z}$ is B-smooth. Let V be the resulting set of cardinality $\#V > \#P$ consisting of the 'good' v's, i.e., the relations.

Each $v \in V$ gives rise to a $\#P$-dimensional vector $(e_{v,p})_{p \in P}$. Because $\#V > \#P$, the vectors $\{(e_{v,p})_{p \in P} : v \in V\}$ are linearly dependent. This implies that there exist at least $\#V - \#P$ linearly independent subsets S of V for which

$$\sum_{v \in S} e_{v,p} = 2(s_p)_{p \in P} \text{ with } s_p \in \mathbf{Z} \text{ for } p \in P.$$

These can, in principle, be found using Gaussian elimination modulo 2 on the matrix having the vectors $(e_{v,p})_{p \in P}$ as rows. The second stage is therefore referred to as the *matrix step*.

Given such a subset S of V and corresponding integer vector $(s_p)_{p \in P}$, the integers

$$x = \left(\prod_{v \in S} v \right) \bmod n, \ y = \left(\prod_{p \in P} p^{s_p} \right) \bmod n$$

solve the congruence $x^2 \equiv y^2 \bmod n$. Each of the $\#V - \#P$ independent subsets leads to an independent chance of at least 50% to factor n.

There are various ways to analyse the expected runtime of Dixon's method, depending on the way the candidate v's are tested for smoothness. If trial division is used, then the best choice for B turns out to be $B = L_n[1/2, 1/2]$. For each candidate v, the number $v^2 \bmod n$ is assumed to behave as a random number $\leq n = L_n[1, 1]$, and therefore B-smooth with probability $L_n[1/2, -1]$ (see 4.2.2). Testing each candidate takes time $\#P = \pi(B) = L_n[1/2, 1/2]$ (all $\log(n)$ factors 'disappear' in the $o(1)$), so collecting somewhat more than $\#P$ relations can be expected to take time

$$\overbrace{L_n[1/2, 1/2]}^{\substack{\text{number}\\\text{of relations}\\\text{to be collected}}} \cdot \overbrace{L_n[1/2, 1/2]}^{\substack{\text{trial}\\\text{division}}} \cdot \overbrace{(L_n[1/2, -1])^{-1}}^{\substack{\text{inverse of}\\\text{smoothness}\\\text{probability}}} = L_n[1/2, 2].$$

Gaussian elimination on the $\#V \times \#P$ matrix takes time

$$L_n[1/2, 1/2]^3 = L_n[1/2, 1.5].$$

It follows that the total time required for Dixon's method with trial division is $L_n[1/2, 2]$. The runtime is dominated by the relation collection stage.

If trial division is replaced by the elliptic curve method (4.2.3), the time to test each candidate for B-smoothness is reduced to $L_n[1/2, 0]$: the entire cost disappears in the $o(1)$. As a result the two stages can be seen to require time $L_n[1/2, 1.5]$ each. This can be further reduced as follows. In the first place, redefine B as $L_n[1/2, \sqrt{1/2}]$ so that the entire relation collection stage takes time

$$L_n[1/2, \sqrt{1/2}] \cdot L_n[1/2, 0] \cdot (L_n[1/2, -\sqrt{1/2}])^{-1} = L_n[1/2, \sqrt{2}].$$

Secondly, observe that at most $\log_2(n) = L_n[1/2, 0]$ entries are non-zero for each vector $(e_{v,p})_{p \in P}$, so that the total number of non-zero entries of the matrix (the 'weight') is $\#V \times L_n[1/2, 0] = L_n[1/2, \sqrt{1/2}]$. The number of operations required by sparse matrix techniques is proportional to the product of the weight and the number of columns, so the matrix step can be done in time $L_n[1/2, \sqrt{1/2}]^2 = L_n[1/2, \sqrt{2}]$. Thus, using the elliptic curve method to test for smoothness and using sparse matrix techniques for the second stage, the overall runtime of Dixon's method becomes

$$L_n[1/2, \sqrt{2}] + L_n[1/2, \sqrt{2}] = L_n[1/2, \sqrt{2}].$$

Asymptotically the relation collection stage and the matrix step take the same amount of time. Sparse matrix methods are not further treated here; see [27, 58, 80, 96, 114] for various methods that can be applied.

Dixon's method has the advantage that its expected runtime can be rigorously analysed and does not depend on unproved hypotheses or heuristics. In practice, however, it is inferior to the original Morrison-Brillhart continued fraction approach and to the methods described below. The Morrison-Brillhart method was used to factor the seventh Fermat number $F_7 = 2^{2^7} + 1$ (see [83]).

4.2.5 Quadratic Sieve. The most obvious way to speed-up Dixon's method is by generating the integers v in such a way that $v^2 \bmod n$ is substantially smaller than n, thereby improving the smoothness probability. In CFRAC, Morrison and Brillhart's continued fraction method, the residues to be tested for smoothness are of the form $a_i^2 - nb_i^2$, where a_i/b_i is the ith continued fraction convergent to $\sqrt{n}$. Thus, $v = a_i$ for $i = 1, 2, \ldots$ and the residues $v^2 \bmod n = a_i^2 - nb_i^2$ to be tested for smoothness are only about $2\sqrt{n}$. However, each residue has to be processed separately (using trial division or the elliptic curve method).

A simpler way to find residues that are almost as small but that can, in practice, be tested much faster was proposed by Pomerance [91, 92]. Let $v(i) = i + [\sqrt{n}]$ for small i, then

$$v(i)^2 \bmod n = (i + [\sqrt{n}])^2 - n \approx 2i\sqrt{n}.$$

Compared to Dixon's method this approximately halves the size of the residues to be tested for B-smoothness. The important advantage – in practice, not in terms of L_n – compared to the continued fraction method is that a sieve can be used to test the values $(v(i)^2 \bmod n)$ for smoothness for many consecutive i's simultaneously, in a manner similar to 3.5.8. This is based on the observation that if p divides $(v(r)^2 \bmod n)$, i.e., r is a root modulo p of $f(X) = (X + [\sqrt{n}])^2 - n$, then p divides $(v(r+kp)^2 \bmod n)$ for any integer k, i.e., $r + kp$ is a root modulo p of $f(X)$ for any integer k.

This leads to the following sieving step. Let $(s(i))_{i=0}^{L-1}$ be a sequence of L locations, corresponding to $(v(i)^2 \bmod n)$ for $0 \le i < L$, with initial values equal to 0. For all $p \le B$ find all roots r modulo p of $(X + [\sqrt{n}])^2 - n$. For all resulting pairs (p, r) replace $s(r + kp)$ by $s(k + rp) + \log_2(p)$ for all integers k such that $0 \le r + kp < L$. As a result, values $s(i)$ that are close to the 'report bound' $\log_2((i + [\sqrt{n}])^2 - n)$ are likely to be B-smooth. Each such value is tested separately for smoothness. In practice the sieve values $s(i)$ and (rounded) $\log_2(p)$ values are represented by bytes.

With $B = L_n[1/2, 1/2]$ and assuming that $(v(i)^2 \bmod n)$ behaves as a random number close to $\sqrt{n} = L_n[1, 1/2]$ it is B-smooth with probability $L_n[1/2, -1/2]$. This assumption is obviously incorrect: if an odd prime p divides $(v(i)^2 \bmod n)$ (and does not divide n) then $(i + [\sqrt{n}])^2 \equiv n \bmod p$, so that n is a quadratic residue modulo p. As a consequence, one may expect that half the primes $\le B$ cannot occur in $(v(i)^2 \bmod n)$, so that $\#P \approx \pi(B)/2$. On the other hand, for each prime p that may occur, one may expect two roots of $f(X)$ modulo p. In practice the smoothness probabilities are very close to what is naïvely predicted based on 4.2.2.

To find $> \#P = L_n[1/2, 1/2]$ relations,

$$L_n[1/2, 1/2] \cdot (L_n[1/2, -1/2])^{-1} = L_n[1/2, 1]$$

different i's have to be considered, so that the sieve length L equals $L_n[1/2, 1]$. This justifies the implicit assumption made above that i is small. The sieving time consists of the time to find the roots (i.e., $\#P = L_n[1/2, 1/2]$ times an effort that is polynomial-time in n and p) plus the actual sieving time. The latter can be expressed as

$$\sum_{\{(p,r):\ p \in P\}} \frac{L}{p} \approx 2L \sum_{p \in P} \frac{1}{p} = L_n[1/2, 1]$$

because $\sum_{p\in P}\frac{1}{p}$ is proportional to $\log\log(L_n[1/2,1/2])$ (see [46]) and disappears in the $o(1)$. The matrix is sparse again. It can be processed in time $L_n[1/2,1/2]^2 = L_n[1/2,1]$. The total (heuristic) expected runtime of Pomerance's quadratic sieve factoring method becomes

$$L_n[1/2,1] + L_n[1/2,1] = L_n[1/2,1].$$

The relation collection and matrix steps take, when expressed in L_n, an equal amount of time. For all numbers factored so far with the quadratic sieve, however, the actual runtime spent on the matrix step is only a small fraction of the total runtime.

The runtime of the quadratic sieve is the same as the worst case runtime of the elliptic curve method applied to n. The sizes of the various polynomial-time factors involved in the runtimes (which all disappear in the $o(1)$'s) make the quadratic sieve much better for numbers that split into two primes of about equal size. In the presence of small factors the elliptic curve method can be expected to outperform the quadratic sieve.

Because of its practicality many enhancements of the basic quadratic sieve as described above have been proposed and implemented. The most important ones are listed below. The runtime of quadratic sieve when expressed in terms of L_n is not affected by any of these improvements. In practice, though, they make quite a difference.

Multiple polynomials Despite the fact that 'only' $L_n[1/2,1]$ different i's have to be considered, in practice the effect of the rather large i's is quite noticeable: the larger i gets, the smaller the 'yield' becomes. Davis and Holdridge were the first to propose the use of more polynomials [34]. A somewhat more practical but similar solution was independently suggested by Montgomery [92, 107]. As a result a virtually unlimited amount of equally useful polynomials can be generated, each playing the role of $f(X)$ in the description above. As soon as one would be sieving too far away from the origin ($i = 0$), sieving continues with a newly selected polynomial. See [64, 92, 107] for details.

Self-initializing For each polynomial all roots modulo all primes $\leq B$ have to be computed. In practice this is a time consuming task. In [95] it is shown how large sets of polynomials can be generated in such a way that the most time consuming part of the root computation has to be carried out only once per set.

Large primes In the above description sieve values $s(i)$ are discarded if they are not sufficiently close to the report bound. By relaxing the report bound somewhat, relations of the form

$$v^2 = q \cdot \left(\prod_{p\in P} p^{e_{v,p}}\right) \bmod n$$

can be collected at hardly any extra cost. Here q is a prime larger than B, the *large prime*. Relations involving large primes are referred to as *partial* *relations.* Two partial relations with the same large prime can be multiplied (or divided) to yield a relation that is, for factoring purposes, just as useful as any ordinary relation. The latter are, in view of the partial relations, often referred to as *full relations.* However, combined partial relations make the matrix somewhat less sparse. It is a consequence of the birthday paradox (4.1.5) and of the fact that smaller large primes occur more frequently than larger ones, that matches between large primes occur often enough to make this approach worthwhile. Actually, it more than halves the sieving time. The obvious extension is to allow more than a single large prime. Using two large primes again more than halves the sieving time [68]. Three large primes have, yet again, the same effect, according to an as yet unpublished experiment. Large primes can be seen as a cheap way to extend the size of the factor base P – cheap because the large primes are not sieved with.

Parallelization The multiple polynomial variation of the quadratic sieve (or its self-initializing variant) allows straightforward parallelization of the relation collection stage on any number of independent processors [20]. Communication is needed only to send the initial data, and to report the resulting relations back to the central location (where progress is measured and the matrix step is carried out). This can be done on any loosely coupled network of processors, such as the Internet [67].

So far the largest number factored using the quadratic sieve is the factorization of the RSA challenge number, a number of 129 decimal digits [40], reported in [6]. Since that time the number field sieve (4.2.7) has overtaken the quadratic sieve, and the method is no longer used to pursue record factorizations.

4.2.6 Historical note. In the late 1970s Schroeppel invented the *Linear sieve* factoring algorithm, based on the Morrison-Brillhart approach. He proposed to look for pairs of small integers i, j such that

$$(i + [\sqrt{n}])(j + [\sqrt{n}]) - n \approx (i + j)\sqrt{n}$$

is smooth. Compared to Morrison and Brillhart's continued fraction method this had the advantage that smoothness could be tested using a sieve. This led to a much faster relation collection stage, despite the fact that $(i + j)\sqrt{n}$ is larger than $2\sqrt{n}$, the order of magnitude of the numbers generated by Morrison and Brillhart. A disadvantage was, however, that $(i + [\sqrt{n}])(j + [\sqrt{n}])$ is not a square. This implied that all values $i + [\sqrt{n}]$ occurring in a relation had to be carried along in the matrix as well, to combine them into squares. This led to an unusually large matrix, for that time at least.

The runtime of the linear sieve was fully analysed by Schroeppel in terms equivalent to the L function. It was the first factoring method for which a (heuristic) subexponential expected runtime was shown. The runtime of Morrison and Brillhart's continued fraction method, though also subexponential, had up to that time never been analysed.

Schroeppel implemented his method and managed to collect relations for the factorization of the eighth Fermat number $F_8 = 2^{2^8} + 1$. Before he could embark on the matrix step, however, F_8 was factored by a stroke of luck using Pollard's rho method (4.1.5 and [17]). As a result of this fortunate – or rather, unfortunate – factorization the linear sieve itself and its runtime analysis never got the attention it deserved. Fortunately, however, it led to the quadratic sieve when Pomerance, attending a lecture by Schroeppel, realized that it may be a good idea to take $i = j$ in the linear sieve.

4.2.7 Number Field Sieve [65]. At this point the number field sieve is the fastest general purpose factoring algorithm that has been published. It is based on an idea by Pollard in 1988 to factor numbers of the form $x^3 + k$ for small k (see his first article in [65]). This method was quickly generalized to a factoring method for numbers of the form $x^d + k$, a method that is currently referred to as the special number field sieve. It proved to be practical by factoring the ninth Fermat number $F_9 = 2^{2^9} + 1$. This happened in 1990, long before F_9 was expected to 'fall' [66]. The heuristic expected runtime of the special number field sieve is $L_n[1/3, 1.526]$, where $1.526 \approx (32/9)^{1/3}$. It was the first factoring algorithm with runtime substantially below $L_n[1/2, c]$ (for constant c), and as such an enormous breakthrough. It was also an unpleasant surprise for factoring based cryptography, despite the fact that the method was believed to have very limited applicability.

This hope was, however, destroyed by the development of the general number field sieve, as it was initially referred to. Currently it is referred to as the number field sieve. In theory the number field sieve should be able to factor any number in heuristic expected time $L_n[1/3, 1.923]$, with $1.923 \approx (64/9)^{1/3}$. It took a few years, and the dogged determination of a handful of true believers, to show that this algorithm is actually practical. Even for numbers having fewer than 100 decimal digits it already beats the quadratic sieve. The crossover point is much lower than expected and reported in the literature [94].

The number field sieve follows the traditional Morrison-Brillhart approach of collecting relations, based on some concept of smoothness, followed by a matrix step. The reason that it is so much faster than previous smoothness based approaches, is that the numbers that are tested for smoothness are of order $n^{o(1)}$, for $n \to \infty$, as opposed to n^c for a (small) constant c for the older methods. One of the reasons of its practicality is that it allows relatively easy use of more than two large primes, so that relatively small factor bases can be used during sieving. The relation collection stage can be distributed over almost any number of loosely coupled processors, similar to quadratic sieve.

For an introductory description of the number field sieve, refer to [63, 66, 93]. For complete details see [65] and the references given there. The latest developments are described in [21, 81].

The largest 'special' number factored using the special number field sieve is $2^{773} + 1$ (see [33]). This was done by the same group that achieved the current general number field sieve record by factoring a 512-bit RSA modulus [21]. Neither of these records can be expected to stand for a long time. Consult [33] for the most recent information. Such 'public domain' factoring records should not be confused with factorizations that could, in principle or in practice, be obtained by well funded agencies or other large organizations. Also, it should be understood that the computational effort involved in a 512-bit RSA modulus factorization is dwarfed by DES cracking projects. See [69] for a further discussion on these and related issues.

4.2.8 Index calculus method. As was first shown in [1], a variation of the Morrison-Brillhart approach can be used to compute discrete logarithms in $\mathbf{F}_{p^\ell}$. First use the familiar two stage approach to compute the discrete logarithms of many 'small' elements of $\mathbf{F}^*_{p^\ell}$. Next use this information to compute the discrete logarithm of arbitrary elements of $\mathbf{F}^*_{p^\ell}$. The outline below applies to any finite field $\mathbf{F}_{p^\ell}$. The expected runtimes are for $p \to \infty$ and ℓ fixed or p fixed and $\ell \to \infty$, as in 4.2.2. See also [2]. If $\ell = 1$ the 'norm' of a field element is simply the integer representing the field element, and an element is 'prime' if that integer is prime. If $\ell > 1$ the 'norm' is as in 4.2.2, and an element is 'prime' if its representation is an irreducible polynomial over $\mathbf{F}_p$ (see 3.6.1).

Let B be a smoothness bound, and let the factor base P be the subset of $\mathbf{F}^*_{p^\ell}$ of primes of norm $\leq B$. Relations are defined as identities of the form

$$g^v = \prod_{p \in P} p^{e_{v,p}},$$

with $v \in \{0, 1, \ldots, p^\ell - 2\}$ and where g generates $\mathbf{F}^*_{p^\ell}$. This implies that

$$v \equiv \left(\sum_{p \in P} e_{v,p} \log_g(p) \right) \bmod (p^\ell - 1).$$

It follows that with more than $\#P$ distinct relations, the values of $\log_g(p)$ for $p \in P$ can be found by solving the system of linear relations defined by the vectors $(e_{v,p})_{p \in P}$.

Given an arbitrary $h \in \mathbf{F}^*_{p^\ell}$ and $\log_g(p)$ for $p \in P$, the value of $\log_g(h)$ can be found by selecting an integer u such that hg^u is B-smooth, i.e.,

$$hg^u = \prod_{p \in P} p^{e_{u,p}},$$

because it leads to

$$\log_g(h) = \left(\left(\sum_{p \in P} e_{u,p} \log_g(p) \right) - u \right) \bmod (p^\ell - 1).$$

With $B = L_{p^\ell - 1}[1/2, \sqrt{1/2}]$ and Dixon's approach to find relations (i.e., pick v at random and test g^v for B-smoothness) the relation collection stage takes expected time $L_{p^\ell}[1/2, \sqrt{2}]$. This follows from the smoothness probabilities in 4.2.2, the runtime of the elliptic curve method (4.2.3) if $\ell = 1$, and, if $\ell > 1$, the fact that polynomials over $\mathbf{F}_p$ of degree k can be factored in expected time polynomial in k and $\log(p)$ (see [53]). Solving the system of linear equations takes the same expected runtime $L_{p^\ell}[1/2, \sqrt{2}]$ because the relations are sparse, as in 4.2.4. Finally, an appropriate u can be found using the same Dixon approach. This results in an expected runtime $L_{p^\ell}[1/2, \sqrt{1/2}]$ per discrete logarithm to be computed (given the $\log_g(p)$ for $p \in P$).

Variations. Various asymptotically faster variants of the same basic idea have been proposed that reduce the heuristic expected runtime to $L_{p^\ell}[1/2, 1]$ for the preparatory stage and $L_{p^\ell}[1/2, 1/2]$ per individual discrete logarithm; see [31, 59, 86] for details. One of these methods, the *Gaussian integers method* for the computation of discrete logarithms in $\mathbf{F}_p$ is of particular interest. It not only gave Pollard the inspiration for the (special) number field sieve integer factoring method, but it is also still of practical interest despite asymptotically faster methods that have in the mean time been found (see below).

Coppersmith's method. Another important variant of the index calculus method is Coppersmith's method. It applies only to finite fields of small fixed characteristic, such as $\mathbf{F}_{2^\ell}$. It was the first cryptanalytic method to break through the $L_x[1/2, c]$ barrier, with an expected runtime $L_{2^\ell}[1/3, 1.588]$. This is similar to the runtime of the number field sieve, but the method was found much earlier. Refer to [26] for details or to [64] for a simplified description.

Coppersmith's method has proved to be very practical. Also, the constant 1.588 is substantially smaller than the constant 1.923 in the runtime of the number field sieve integer factoring method. For those reasons RSA moduli of a given size s are believed to offer more security than the multiplicative group of a finite field $\mathbf{F}_{p^\ell}$ with small constant p and $p^\ell \approx s$. If such fields are used, then p^ℓ must be chosen considerably larger than s to achieve the same level of security.

In [45] an (incomplete) application of Coppersmith's method to the finite field $\mathbf{F}_{2^{503}}$ is described. A description of a completed discrete logarithm computation in $\mathbf{F}_{2^{607}}$ is forthcoming (no reference available yet).

Discrete logarithm variant of the number field sieve. For $p \to \infty$ and ℓ fixed discrete logarithms in $\mathbf{F}_{p^\ell}$ can be computed in heuristic expected runtime $L_{p^\ell}[1/3, 1.923]$. For small fixed p this is somewhat slower than Coppersmith's method. However, the method applies to finite fields of arbitrary characteristic. The method is based on the number field sieve. For details refer to [43, 100].

The current record for finite field discrete logarithm computation is a 399-bit (120 decimal digit) prime field, reported in [51].

Acknowledgment. These notes were written on the occasion of the tutorial on mathematical foundations of coding theory and cryptology, held from July 23 to 26, 2001, at the Institute for Mathematical Sciences (IMS) of the National University of Singapore, as part of the program on coding theory and data integrity. The author wants to thank the IMS and in particular Professor Harald Niederreiter for their invitation to participate in the program.

References

1. L.M. Adleman, *A subexponential-time algorithm for the discrete logarithm problem with applications*, Proceedings 20th Ann. IEEE symp. on foundations of computer science (1979) 55-60.
2. L.M. Adleman, J. DeMarrais, *A subexponential algorithm for discrete logarithms over all finite fields*, Proceedings Crypto'93, LNCS 773, Springer-Verlag 1994, 147-158.
3. L.M. Adleman, M.A. Huang, *Primality testing and abelian varieties over finite fields*, Lecture Notes in Math. **1512**, Springer-Verlag 1992.
4. W.R. Alford, A. Granville, C. Pomerance, *There are infinitely many Carmichael numbers*, Ann. of Math. **140** (1994) 703-722.
5. I. Anshel, M. Anshel, D. Goldfeld, *An algebraic method for public-key cryptography*, Mathematical Research Letters **6** (1999) 287-291.
6. D. Atkins, M. Graff, A.K. Lenstra, P.C. Leyland, *THE MAGIC WORDS ARE SQUEAMISH OSSIFRAGE*, Proceedings Asiacrypt'94, LNCS 917, Springer-Verlag 1995, 265-277.
7. E. Bach, J. Shallit, *Cyclotomic polynomials and factoring*, Math. Comp. **52** (1989) 201-219.
8. E. Bach, J. Shallit, *Algorithmic number theory, Volume 1, Efficient Algorithms*, The MIT press, 1996.
9. I. Blake, G. Seroussi, N. Smart, *Elliptic curves in cryptography*, Cambridge University Press, 1999.
10. D. Boneh, R.A. DeMillo, R.J. Lipton, *On the importance of checking cryptographic protocols for faults*, Proceedings Eurocrypt'97, LNCS 1233, Springer-Verlag 1997, 37-51.
11. D. Boneh, G. Durfee, *Cryptanalysis of RSA with private key d less than $N^{0.292}$*, Proceedings Eurocrypt'99, LNCS 1592, Springer-Verlag 1999, 1-11.
12. D. Boneh, R.J. Lipton, *Algorithms for black-box fields and their application to cryptography*, Proceedings Crypto'96, LNCS 1109, Springer-Verlag 1996, 283-297.
13. W. Bosma, A.K. Lenstra, *An implementation of the elliptic curve integer factorization method*, chapter 9 in *Computational algebra and number theory* (W. Bosma, A. van der Poorten, eds.), Kluwer Academic Press (1995).
14. W. Bosma, M.P.M van der Hulst, *Primality proving with cyclotomy*, PhD. thesis, Universiteit van Amsterdam (1990).
15. R.P. Brent, *An improved Monte Carlo factorization algorithm*, BIT **20** (1980) 176-184.
16. R.P. Brent, *Factorization of the tenth and eleventh Fermat numbers*, manuscript, 1996.
17. R.P. Brent, J.M. Pollard, *Factorization of the eighth Fermat number*, Math. Comp. **36** (1980) 627-630.
18. E.F. Brickell, D.M. Gordon, K.S. McCurley, D.B. Wilson, *Fast exponentiation with precomputation*, Proceedings Eurocrypt'92, LNCS 658, Springer-Verlag, 1993, 200-207.
19. E.R. Canfield, P. Erdös, C. Pomerance, *On a problem of Oppenheim concerning "Factorizatio Numerorum"*, J. Number Theory **17** (1983) 1-28.
20. T.R. Caron, R.D. Silverman, *Parallel implementation of the quadratic sieve*, J. Supercomput. **1** (1988) 273-290.

21. S. Cavallar, B. Dodson, A.K. Lenstra, W. Lioen, P.L. Montgomery, B. Murphy, H. te Riele, et al., *Factorization of a 512-bit RSA modulus*, Proceedings Eurocrypt 2000, LNCS 1807, Springer-Verlag 2000, 1-18.
22. www.certicom.com.
23. H. Cohen, A.K. Lenstra, *Implementation of a new primality test*, Math. Comp. **48** (1986) 187-237.
24. H. Cohen, H.W. Lenstra, Jr., *Primality testing and Jacobi sums*, Math. Comp. **42** (1984) 297-330.
25. H. Cohen, A. Miyaji, T. Ono, *Efficient elliptic curve exponentiation using mixed coordinates*, Proceedings Asiacrypt'98, LNCS 1514, Springer-Verlag 1998, 51-65.
26. D. Coppersmith, *Fast evaluation of logarithms in fields of characteristic two*, IEEE Trans. Inform. Theory 30 (1984) 587-594.
27. D. Coppersmith, *Solving linear equations over GF(2) using block Wiedemann algorithm*, Math. Comp. **62** (1994) 333-350.
28. D. Coppersmith, *Finding a small root of a univariate modular equation*, Proceedings Eurocrypt'96, LNCS 1070, Springer-Verlag 1996, 155-165.
29. D. Coppersmith, *Small solutions to polynomial equations, and low exponent RSA vulnerabilities*, Journal of Cryptology **10** (1997) 233-260.
30. D. Coppersmith, M. Franklin, J. Patarin, M. Reiter, *Low-exponent RSA with related messages*, Proceedings Eurocrypt'96, LNCS 1070, Springer-Verlag 1996, 1-9.
31. D. Coppersmith, A.M. Odlyzko, R. Schroeppel, *Discrete logarithms in GF(p)*, Algorithmica **1** (1986) 1-15.
32. D. Coppersmith, A. Shamir, Lattice attacks on NTRU, Proceedings Eurocrypt'97, LNCS 1233, Springer-Verlag 1997, 52-61.
33. www.cwi.nl.
34. J.A. Davis, D.B. Holdridge, *Factorization using the quadratic sieve algorithm*, Tech. Report SAND 83-1346, Sandia National Laboratories, Albuquerque, NM, 1983.
35. N.G. De Bruijn, *On the number of positive integers* $\leq x$ *and free of prime factors* $> y$*, II*, Indag. Math. **38** (1966) 239-247.
36. W. Diffie, M.E. Hellman, *New directions in cryptography*, IEEE Trans. Inform. Theory 22 (1976) 644-654.
37. J.D. Dixon, *Asymptotically fast factorization of integers*, Math. Comp. **36** (1981) 255-260.
38. *Quantum dreams*, The Economist, March 10, 2001, 81-82.
39. T. ElGamal, *A public key cryptosystem and a signature scheme based on discrete logarithms*, IEEE Transactions on Information Theory 31(4) (1985) 469-472.
40. M. Gardner, *Mathematical games, a new kind of cipher that would take millions of years to break*, Scientific American, August 1977, 120-124.
41. C. Gentry, J. Johnson, J. Stern, M, Szydlo, *Cryptanalysis of the NSS signature scheme* and *Exploiting several flaws in the NSS signature scheme*, presentations at the Eurocrypt 2001 rump session, May 2001.
42. S. Goldwasser, J. Kilian, *Almost all primes can be quickly verified*, Proc. 18th Ann. ACM symp. on theory of computing (1986) 316-329.
43. D. Gordon, *Discrete logarithms in GF(p) using the number field sieve*, SIAM J. Discrete Math. **6** (1993) 312-323.
44. D.M. Gordon, *A survey on fast exponentiation methods*, Journal of Algorithms **27** (1998) 129-146.

45. D.M. Gordon, K.S. McCurley, *Massively parallel computation of discrete logarithms*, Proceedings Crypto'92, LNCS 740, Springer-Verlag 1993, 312-323.
46. G.H. Hardy, E.M. Wright, *An introduction to the theory of numbers*, Oxford Univ. Press, Oxford, 5th ed., 1979.
47. J. Hastad, *Solving simultaneous modular equations of low degree*, SIAM J. Comput. **17** (1988) 336-341.
48. J. Hoffstein, J. Pipher, J.H. Silverman, *NTRU: a ring-based public key cryptosystem*, Proceedings ANTS III, LNCS 1423, Springer-Verlag 1998, 267-288.
49. J. Hoffstein, J. Pipher, J.H. Silverman, *NSS: an NTRU lattice-based signature scheme*, Proceedings Eurocrypt 2001, LNCS 2045, Springer-Verlag 2001, 211-228.
50. T. Itoh, S, Tsujii, *A fast algorithm for computing multiplicative inverses in* $GF(2^k)$ *using normal bases*, Information and computation **78** (1988) 171-177.
51. A. Joux, R. Lercier, *Discrete logarithms in GF(p) (120 decimal digits)*, available from http://listserv.nodak.edu/archives/nmbrthry.html, April 2001.
52. A. Joux, K. Nguyen, *Separating decision Diffie-Hellman from Diffie-Hellman in cryptographic groups*, available from eprint.iacr.org, 2001.
53. D.E. Knuth, *The art of computer programming, Volume 2, Seminumerical Algorithms*, third edition, Addison-Wesley, 1998.
54. K.H. Ko, S.J. Lee, J.H. Cheon, J.W. Han, J. Kang, C. Park, *New public-key cryptosystem using braid groups*, Proceedings Crypto 2000, LNCS 1880, Springer-Verlag 2000, 166-183.
55. T. Kobayashi, H. Morita, K. Kobayashi, F. Hoshino, *Fast elliptic curve algorithm combining Frobenius map and table reference to adapt to higher characteristic*, Proceedings Eurocrypt'99, LNCS 1592, Springer-Verlag (1999) 176-189.
56. N. Koblitz, *Elliptic curve cryptosystems*, Math. Comp. **48** (1987) 203-209.
57. P.C. Kocher, *Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems*, Proceedings Crypto'96, LNCS 1109, Springer-Verlag 1996, 104-113.
58. B.A. LaMacchia, A.M. Odlyzko, *Solving large sparse linear systems over finite fields*, Proceedings Crypto'90, LNCS 537, Springer-Verlag 1990, 109-133.
59. B.A. LaMacchia, A.M. Odlyzko, *Computation of discrete logarithms in prime fields*, Design, Codes and Cryptography **1** (1991) 47-62.
60. A.K. Lenstra, *The long integer package FREELIP*, available from www.ecstr.com or directly from the author.
61. A.K. Lenstra, *Using cyclotomic polynomials to construct efficient discrete logarithm cryptosystems over finite fields*, Proceedings ACISP97, LNCS 1270, Springer-Verlag 1997, 127-138.
62. A.K. Lenstra, *Generating RSA moduli with a predetermined portion*, Proceedings Asiacrypt'98, LNCS 1514, Springer-Verlag 1998, 1-10.
63. A.K. Lenstra, *Integer factoring*, Designs, codes and cryptography **19** (2000) 101-128.
64. A.K. Lenstra, H.W. Lenstra, Jr., *Algorithms in number theory*, chapter 12 in *Handbook of theoretical computer science, Volume A, algorithms and complexity* (J. van Leeuwen, ed.), Elsevier, Amsterdam (1990).
65. A.K. Lenstra, H.W. Lenstra, Jr., (eds.), *The development of the number field sieve*, Lecture Notes in Math. **1554**, Springer-Verlag 1993.
66. A.K. Lenstra, H.W. Lenstra, Jr., M.S. Manasse, J.M. Pollard, *The factorization of the ninth Fermat number*, Math. Comp. **61** (1993) 319-349.
67. A.K. Lenstra, M.S. Manasse, *Factoring by electronic mail*, Proceedings Eurocrypt'89, LNCS 434, Springer-Verlag 1990, 355-371.

68. A.K. Lenstra, M.S. Manasse, *Factoring with two large primes*, Proceedings Eurocrypt'90, LNCS 473, Springer-Verlag 1990, 72-82; Math. Comp. **63** (1994) 785-798.
69. A.K. Lenstra, E.R. Verheul, *Selecting cryptographic key sizes*, to appear in the Journal of Cryptology; available from www.cryptosavvy.com.
70. A.K. Lenstra, E.R. Verheul, *The XTR public key system*, Proceedings Crypto 2000, LNCS 1880, Springer-Verlag, 2000, 1-19; available from www.ecstr.com.
71. H.W. Lenstra, Jr., *Factoring integers with elliptic curves*, Ann. of Math. **126** (1987) 649-673.
72. R. Lidl, H, Niederreiter, *Introduction to finite fields and their applications*, Cambridge University Press, 1994.
73. U. Maurer, *Fast generation of prime numbers and secure public-key cryptographic parameters*, Journal of Cryptology **8** (1995) 123-155.
74. A.J. Menezes (ed.), *Elliptic curve public key cryptosystems*, Kluwer academic publishers, 1993.
75. A.J. Menezes (ed.), *Applications of finite fields*, Kluwer academic publishers, 1993.
76. A.J. Menezes, P.C. van Oorschot, S.A. Vanstone, *Handbook of applied cryptography*, CRC Press, 1997.
77. V. Miller *Use of elliptic curves in cryptography*, Proceedings Crypto'85, LNCS 218, Springer-Verlag 1986, 417-426.
78. P.L. Montgomery, *Modular multiplication without trial division*, Math. Comp. **44** (1985) 519-521.
79. P.L. Montgomery, *Speeding the Pollard and elliptic curve methods of factorization*, Math. Comp. **48** (1987) 243-264.
80. P.L. Montgomery, *A block Lanczos algorithm for finding dependencies over GF(2)*, Proceedings Eurocrypt'95, LNCS 925, Springer-Verlag 1995, 106-120.
81. P.L. Montgomery, B. Murphy, *Improved polynomial selection for the number field sieve*, extended abstract for the conference on the mathematics of public-key cryptography, June 13-17, 1999, the Fields institute, Toronto, Ontario, Canada.
82. F. Morain, *Implementation of the Goldwasser-Kilian-Atkin primality testing algorithm*, INRIA Report 911, INRIA-Rocquencourt, 1988.
83. M.A. Morrison, J. Brillhart, *A method of factorization and the factorization of* F_7, Math. Comp. **29** (1975) 183-205.
84. V.I. Nechaev, *Complexity of a determinate algorithm for the discrete logarithm*, Mathematical Notes, 55(2) (1994) 155-172. Translated from Matematicheskie Zametki, 55(2) (1994) 91-101. This result dates from 1968.
85. P.Q. Nguyen, J. Stern, *Lattice reduction in cryptology: an update*, Proceedings ANTS IV, LNCS 1838, Springer-Verlag 2000, 85-112.
86. A.M. Odlyzko, *Discrete logarithms and their cryptographic significance*, Proceedings Eurocrypt'84, LNCS 209, Springer-Verlag 1985, 224-314.
87. S.C. Pohlig, M.E. Hellman, *An improved algorithm for computing logarithms over* $GF(p)$ *and its cryptographic significance*, IEEE Trans. on Inform Theory 24 (1978) 106-110.
88. J.M. Pollard, *Theorems on factorization and primality testing*, Proceedings of the Cambridge philosophical society, **76** (1974) 521-528.
89. J.M. Pollard, *Monte Carlo methods for index computation (mod p)*, Math. Comp., 32 (1978) 918-924.
90. J.M. Pollard, *Kangaroos, monopoly and discrete logarithms*, Journal of Cryptology **13** (2000) 437-447.

91. C. Pomerance, *Analysis and comparison of some integer factoring algorithms*, in *Computational methods in number theory* (H.W. Lenstra, Jr., R. Tijdeman, eds.), Math. Centre Tracts **154**, **155**, Mathematisch Centrum, Amsterdam (1983) 89-139.
92. C. Pomerance, *The quadratic sieve factoring algorithm*, Proceedings Eurocrypt'84, LNCS 209, Springer-Verlag 1985, 169-182.
93. C. Pomerance, *The number field sieve*, Proc. Symp. Appl. Math. **48** (1994) 465-480.
94. C. Pomerance, *A tale of two sieves*, Notices of the AMS (1996) 1473-1485.
95. C. Pomerance, J.W. Smith, R. Tuler *A pipeline architecture for factoring large integers with the quadratic sieve algorithm*, SIAM J. Comput. **17** (1988) 387-403.
96. C. Pomerance, J.W. Smith, *Reduction of huge, sparse matrices over finite fields via created catastrophes*, Experimental Math. **1** (1992) 89-94.
97. M.O. Rabin, *Probabilistic algorithms for primality testing*, J. Number Theory, **12** (1980) 128-138.
98. R.L. Rivest, A. Shamir, L.M. Adleman, *A method for obtaining digital signatures and public key cryptosystems*, Comm. of the ACM, **21** (1978) 120-126.
99. R.L. Rivest, R.D. Silverman, *Are 'strong' primes needed for RSA?*, manuscript, April 1997, available from www.iacr.org.
100. O. Schirokauer, *Discrete logarithms and local units*, Phil. Trans. R. Soc. Lond. A 345 (1993) 409-423.
101. C.P. Schnorr, *Efficient signature generation by smart cards*, Journal of Cryptology **4** (1991) 161-174.
102. R.J. Schoof, *Elliptic curves over finite fields and the computation of square roots mod p*, Math. Comp. **44** (1985) 483-494.
103. P.W. Shor, *Algorithms for quantum computing: discrete logarithms and factoring*, Proceedings of the IEEE 35th Annual Symposium on Foundations of Computer Science (1994) 124-134.
104. V. Shoup, *Lower bounds for discrete logarithms and related problems*, Proceedings Eurocrypt'97, LNCS 1233, 256-266, Springer 1997.
105. V. Shoup, *NTL*, available from www.shoup.net/ntl.
106. J.H. Silverman, *The arithmetic of elliptic curves*, Springer-Verlag, 1986.
107. R.D. Silverman, *The multiple polynomial quadratic sieve*, Math. Comp. **46** (1987) 327-339.
108. N.P. Smart, *How secure are elliptic curves over composite extension fields?*, Proceedings Eurocrypt 2001, LNCS 2045, Springer-Verlag 2001, 30-39.
109. P. Smith, C. Skinner, *A public-key cryptosystem and a digital signature system based on the Lucas function analogue to discrete logarithms*, Proceedings Asiacrypt'94, LNCS 917, Springer-Verlag 1995, 357-364.
110. J. Stern, *Cryptanalysis of the NTRU signature scheme (NSS)*, manuscript, April 2001.
111. E. Teske, *Speeding up Pollard's rho methods for computing discrete logarithms*, Proceedings ANTS III, LNCS 1423, Springer-Verlag 1998, 541-554.
112. P.C. van Oorschot, M.J. Wiener, *Parallel collision search with cryptanalytic applications*, Journal of Cryptology **12** (1999) 1-28.
113. M. Wiener, *Cryptanalysis of short RSA secret exponents*, IEEE Trans. Inform. Theory 36 (1990) 553-558.
114. D.H. Wiedemann, *Solving sparse linear equations over finite fields*, IEEE Trans. Inform. Theory 32 (1986) 54-62.
115. H.C. Williams, *A $p+1$ method of factoring*, Math. Comp. **39** (1982) 225-234.