(*March 26, 2007*)

Cryptographic Primitives

Paul Garrett garrett@math.umn.edu http://www.math.umn.edu/˜garrett/

This is a version of an article of the same name to appear in [GL].

A **symmetric** or **private-key** cipher is one in which knowledge of the *encryption* key is explicitly or implicitly equivalent to knowing the *decryption* key. An **asymmetric** or **public-key** cipher is one in which the *encryption* key is effectively public knowledge, without giving any useful information about the *decryption* key. Until 30 years ago all ciphers were private-key. The very possibility of public-key crypto did not exist until the secret work Ellis-Cocks-Williamson at the UK's CESG-at-GCHQ in the 1960's, and public-domain work of Merkle, Diffie-Hellman, and Rivest-Shamir-Adleman in the 1970's. Even more significant than the *secrecy* achievable by public-key ciphers is the variety of effects achievable that were (and continue to be) simply impossible with even the best symmetric-key ciphers. *Key exchange* and *signatures* (authentication) are the most notable among well-established uses. Further examples are given in section 6.

Other articles in address specific aspects of public-key cryptography at greater length. D. Lieman's article [Lie] concerns refinements of protocols appropriate to genuine practical implementations. N. Howgrave-Graham [HG] treats proofs of security. J. Silverman's [Sil3] discusses elliptic curves. W. Whyte's [Wh] and W.D. Banks' [Ba] articles consider the problem of designing faster cryptosystems of various sorts. And I. Shparlinski [Shp2] discusses design and attacks upon systems based upon various hidden-number problems. Given these, we will emphasize algorithms related mostly to RSA, primality testing, and factoring attacks, as opposed to discrete logs and/or elliptic curves, and give only simple naive forms of protocol-ideas rather than refined forms.

By now there are many introductory texts on cryptography. Many of them are reviewed in [L3].

1 Prototypes

Several points which occur either literally or figuratively throughout discussions of public-key ciphers are already nicely illustrated in the case of the Rivest-Shamir-Adleman cipher, RSA, from [RSA], together with the Diffie-Hellman key exchange [DH]. Subordinate algorithmic issues will be made more precise in the sequel. A relevant notion is that of **trapdoor**: a computation which runs much more easily in one direction than back. A prototypical example is that *multiplication* of large integers is easy, while *factoring* of large integers is difficult (to the best of our knowledge). And then there is the question of how to make such an irreversible process into a useful one. Another apparently comparably difficult task, relevant to both RSA and Diffie-Hellman, is computation of discrete logarithms modulo primes, while exponentiation (by the square-and-multiply algorithm (4.3)) is easy.

A key point is that while *factoring* large numbers into primes appears to be difficult, merely *testing* large numbers for primality is definitely easy (e.g., by Miller-Rabin (4.13)).

A critical but often tacit issue is the generation of sufficiently many high-quality random bits incidental to generation of random primes.

And here might be the time to introduce a common computer-language notation

$$x \,\%\, n = x \textbf{ reduced} \text{ modulo } n$$

This notation allows us to easily make a distinction between reduction and equivalence classes or equalities modulo n.

1.1 RSA cipher

For Alice to set things up so that Bob (or other people without prior contact with her) can send her messages that cannot be decrypted by an eavesdropper Eve, Alice proceeds as follows. *(But it must be disclaimed at once that this is a naive and very incomplete implementation of RSA!)*

First, there are one-time set-up computations. Alice generates two large random primes p and q (at least $> 2^{512}$, and probably $> 2^{1024}$), and computes her **RSA modulus** $n = pq$. Alice will keep the primes p and q secret, while she will make n public. Alice chooses an **encryption exponent** $e > 2$. Often the Fermat prime $e = 2^{16} + 1 = 65537$ is used, for reasons that will become clearer subsequently. The encryption exponent e is made public. Alice computes the **decryption exponent** d with the property that

$$ed = 1 \bmod (p-1)(q-1)$$

Alice keeps the decryption exponent d secret. This ends Alice's set-up. The encryption key e is the **public key** and the decryption key d is the **private key**. Integers n of the form $n = pq$ with p and q distinct primes are called **RSA moduli**, or sometimes **Blum integers** if both primes are 3 modulo 4.

For Bob to send a message to Alice that only Alice can read, he proceeds as follows. Note that Bob's procedure requires no prior contact or arrangment with Alice, in particular, no pre-existing secret shared by the two of them. For simplicity, we assume that the plaintext x is an integer in the range $1 < x < n$. The **encoding** step $x \longrightarrow E_{n,e}(x)$ is

$$E_{n,e}(x) = x^e \,\%\, n$$

This produces a **ciphertext** $y = x^e \,\%\, n$ in the range $0 < y < n$. Bob sends this over a (probably insecure) channel to Alice.

Alice decrypts Bob's message as follows. The **decryption** step $y \longrightarrow D_{n,d}(y)$ is

$$D_{n,d}(y) = y^d \,\%\, n$$

This blunt description shows the simplicity of set-up and operation, but several things need clarification:

Why is it feasible for Alice to find two large random primes?
Why is it feasible for Bob to compute x^e reduced mod n?
Why is it feasible for Alice to compute $d = e^{-1} \bmod (p-1)(q-1)$?
Why is it feasible for Alice to compute y^d reduced mod n?
Why does the decryption step decrypt, that is, why is $y^d \bmod n = x$?
Why is it *not* feasible for Eve (the eavesdropper) to compute d from n and e?
Why is it *not* feasible for Eve to compute x from $x^e \bmod n$?
How do we get a good supply of *random numbers*?

The decryption step decrypts: At first glance, possibly the least obvious fact is that the decryption step really does recover the plaintext. For the decryption step to genuinely decrypt, recovering the plaintext, the two keys e, d must have the property that

$$(x^e)^d = x \bmod n$$

for integers x. **Euler's theorem** (4.5) asserts in general that if $\gcd(x, n) = 1$, then

$$x^{\varphi(n)} = 1 \bmod n$$

where φ is the Euler totient function

$$\varphi(n) = \text{number of integers } 1 \leq \ell \leq n \text{ with } \gcd(\ell, n) = 1$$

Thus, e and d should be mutual **multiplicative inverses modulo** $\varphi(n)$

$$d \cdot e = 1 \bmod \varphi(n)$$

For integers n of the form $n = p \cdot q$ with distinct primes p, q it is easy to see that

$$\varphi(p \cdot q) = (p-1)(q-1)$$

Then one can easily verify that the encryption and decryption really work for $\gcd(x, n) = 1$: let $ed = 1 + \ell(p-1)(q-1)$. Then

$$D_{n,d}(E_{n,e}(x)) = (x^e)^d = x^{ed} \bmod n$$
$$= x^{1+\ell\cdot(p-1)(q-1)} = x \cdot (x^{\varphi(n)})^\ell = x \bmod n$$

by invoking Euler's theorem.

Euler's theorem requires that x be prime to n. The probability that a random x in the range $1 < x < n$ would be divisible by p or q is

$$\frac{1}{p} + \frac{1}{q} - \frac{1}{pq}$$

which is sufficiently tiny that we ignore it. In any case, testing for this is easy, via the Euclidean algorithm. And we note that a systematic ability to choose such messages would be (therefore) tantamount to being able to systematically factor large integers. And the encryption exponent e must be prime to $\varphi(n) = (p-1)(q-1)$ so that it will have a multiplicative inverse modulo $\varphi(n)$. Failure in this regard would be detected in set-up.

The security of RSA depends upon the difficulty of *factoring* integers into primes, or at least factoring integers $n = pq$ which are the product of two primes: if Eve can factor n into p and q she knows as much as Alice about the set-up. Eve can compute $\varphi(n) = (p-1)(q-1)$ and compute the decryption exponent d just as Alice (probably by the extended Euclidean algorithm (4.2)). Factoring is not *provably* difficult, though no polynomial-time public-domain algorithm exists.

Given $n = pq$ the product of two big primes p, q (with the primes secret), it seems hard to compute $\varphi(n)$ when only n is given. Once the factorization $n = p \cdot q$ is known, it is easy to compute $\varphi(n)$ as

$$\varphi(n) = \varphi(pq) = (p-1)(q-1)$$

In fact, for numbers $n = pq$ of this special form, knowing both n and $\varphi(n)$ *gives* the factorization $n = p \cdot q$ with little computation. Indeed, p, q are the roots of the equation

$$x^2 - (p+q)x + pq = 0$$

Already $pq = n$, so if $p + q$ is expressed in terms of n and $\varphi(n)$, we have the coefficients of this equation expressed in terms of n and $\varphi(n)$, giving an easy route to p and q separately. Since

$$\varphi(n) = (p-1)(q-1) = pq - (p+q) + 1 = n - (p+q) + 1$$

rearrange to get

$$p + q = n - \varphi(n) + 1$$

Therefore, p and q are the roots of the equation

$$x^2 - (n - \varphi(n) + 1)x + n = 0$$

Computing roots: Also, even more obviously, the security of RSA also depends upon the difficulty of computing e^{th} *roots* modulo n: if Eve can compute e^{th} roots modulo n, then she can compute the plaintext as the e^{th} root of the ciphertext $y = x^e \,\%\, n$. While computing roots modulo *primes* is easy (4.17), computing roots modulo composite numbers appears to be hard (without factoring).

Common modulus: The *users* of a system with modulus n (the product of two secret primes p, q), public key e, and private key d *do not need to know the primes* p, q. Therefore, one might envisage the scenario in which a **central agency** uses the same modulus $n = pq$ for several different users, only with different encryption/decryption exponents e_i, d_i. However, compromise of one key pair e_1, d_1 fatally compromises the others: anyone knowing $N = e_1d_1 - 1$ need not determine $(p-1)(q-1)$ exactly, since an inverse $D = e_2^{-1} \bmod N$ for N any multiple of $(p-1)(q-1)$ also works as a decryption exponent for encryption exponent e_2. That is, a multiplicative inverse modulo a *multiple* $N = e_1d_1 - 1$ of $(p-1)(q-1)$ is a multiplicative inverse modulo $(p-1)(q-1)$, even if it is needlessly larger. It is also true, though much less important, that the search space for n is greatly reduced by knowledge of $e_1d_1 - 1$, since

$$\varphi(n) \mid (e_1d_1 - 1)$$

Fast exponentiation: Naively, raising large numbers to large powers is slow, so from a naive viewpoint it may be unclear why the algorithms in authorized execution of RSA are any faster than a hostile attack. In fact, the required exponentiation can be arranged to be much faster than prime factorizations for numbers in the relevant range. That is, to compute x^e we do *not* compute all of $x^1, x^2, x^3, \ldots, x^{e-1}, x^e$. The *square-and-multiply* exponentiation algorithm (4.3) is a simple example of a fast exponentiation algorithm, consisting of a systematization of the idea of using a binary expansion of the exponent to reduce the total number of operations. For example,

$$x^{17} = x^1 \cdot ((((x^2)^2)^2)^2)$$

Key generation: To set up $n = pq$ from secret primes p, q, and to determine a key pair e, d with $ed = 1 \bmod \varphi(n)$, requires two large primes p, q. Since the security of RSA is based upon the intractability of factoring, it is fortunate that primality testing is much easier than factorization. That is, we *are* able to obtain many 'large' primes $p, q >> 2^{512}$ *cheaply*, despite the fact that we cannot generally factor 'large' numbers $n = pq >> 2^{1024}$ into primes. The Fermat pseudoprime test (4.13) combined with the Miller-Rabin strong pseudoprime test (4.13) provide one efficient test for primality.

Relative difficulty of factoring: The various modern factorization tests, such as Pollard's rho (8.2), or even the better subexponential sieve methods (8.4), while vastly better than antique trial division (4.7), are still too slow to compete, for sufficiently large modulus n. Again, there is no proof that factoring is more difficult than primality testing.

Padding: From a mathematical viewpoint, it is obvious that if an attacker knows the encryptions y_1, y_2 of two different plaintexts x_1, x_2 (with the same public key), then the attacker knows the encryption of $x_1x_2 \,\%\, n$, where n is the RSA modulus. This sort of **leakage** of information may seem obscure, but can be systematically exploited. Thus, in reality, with most public-key ciphers, plaintexts must be *padded* with various amounts of random material.

It seems that attacks on RSA will succeed only when RSA is **improperly implemented**, for example, with too-small modulus. The key size (the size of the RSA modulus $n = pq$) must not be too small, or a brute-force attempt to factor n may succeed in a time smaller than one would want. E.g., in 1999 Adi Shamir designed a specialized computer 'Twinkle' which speeded up execution of certain factorization attacks by a factor of 100 or 1000. To break the RSA *function* means to invert the function

$$x \longrightarrow x^e \,\%\, n$$

without being given the decryption exponent d in advance. To break the RSA *cipher* means something with more possibilities than breaking the RSA function: it means to recover all or part of a plaintext without being given the decryption exponent. Unbreakability of the cipher, rather than of the function, is **semantic security**.

Both $p-1$ and $q-1$ should have at least one very large prime factor, since there are factorization attacks against $n = pq$ that are possible if $p-1$ or $q-1$ have only smallish prime factors (Pollard's $p-1$ attack (8.3)). The primes p and q should not be 'close' to each other, since there are factorization attacks on n that succeed in this case (Fermat, etc.). The ratio p/q should not be 'close' to a rational number with smallish numerator and denominator, or else D. H. Lehmer's Continued Fraction factorization attack on $n = pq$ will succeed. If quantum computers ever become a reality, 1993 work of Peter Shor [Sho1], [Sho2] shows that there is a fast *quantum* algorithm to factor large numbers.

Forward search attack. If the collection of all possible messages is known, and is relatively small, then the attacker need only encrypt all the messages until a match is found. For example, if the message is known to be either 'yes' or 'no', then only one encryption need be computed to know which is the plaintext. To defend against this, messages should be padded by adding random bits at front and/or back.

Small decryption exponent attack: To save some computation time, one might try to arrange so that the decryption exponent is small. But if $d < \frac{1}{3}N^{1/4}$, then there is an efficient algorithm to recover this decryption exponent! See [Wie]. In [Bo] it is speculated that using a decryption modulus less than $\sqrt{N}$ may be vulnerable to such attacks.

Small public exponent attacks: A traditional choice of encryption exponent $e = 3$ is insecure. Suppose that Alice wishes to send the same secret message to several different people. If she does the obvious thing, encrypting the message using each of their public keys and sending, then eavesdropper Eve collects all the encrypted messages. *If the number of messages is greater than or equal the encryption exponent e, then Eve can recover the message.* For example, if $e = 3$ then just 3 different encryptions are sufficient to recover the message. More sophisticated attacks of this sort are **Hastad's broadcast attack** and **Coppersmith's short pad attack** [Co2]. If the larger $e = 2^{16}+1 = 65537$ is used, these vulnerabilities seem to be eliminated.

Partial disclosure: When the (public) encryption exponent is small, **partial disclosure** of the decryption key breaks RSA: [Bo]. From [BDF], the $n/4$ least significant bits of the (private) decryption key d, the entire decryption key can be recovered in time linear in $e \ln_2 e$. This is related to a factorization result: given either the $n/4$ highest or $n/4$ lowest bits of p, one can efficiently factor $n = pq$ [Co2].

Timing, power attacks. Especially in situations such as smart cards where an adversary may have virtually unlimited opportunities to do known-plaintext attacks and measurements, naive forms of RSA are vulnerable to **timing attacks**, where the adversary *times* the computation, thereby finding the number of exponentiations occuring in a decryption, thereby getting information on the decryption exponent [Ko]. Similarly, power attacks measure power consumption to acquire information about sizes of keys.

1.2 Diffie-Hellman key exchange

The discovery [DH] of the possibility of a **key exchange** procedure was a great surprise, even more so than the notion of public-key *cipher*. The goal is that Alice and Bob, who have not met previously, establish a *shared secret* using only an insecure channel to communicate. The eavesdropper Eve has greater computing power than Alice and Bob, and hears everything that passes between them. We do suppose that Eve cannot actively disrupt the communications, but, rather, is a passive eavesdropper.

Typically the shared secret would be used to create a *session key*, that is, a key for a faster symmetric cipher, to be used just during the current communication session.

Again, a crucial mathematical point is that computing *discrete logarithms* modulo primes is relatively difficult, while exponentiating integers modulo large primes is relatively easy. And, again, it is relatively easy to find large primes in the first place. And there is the often-tacit assumption that we can generate sufficiently many high-quality random numbers.

Alice and Bob agree on a large random prime p (at the very least $\sim 2^{1024}$) and a random *base* g in the range $1 < g < p$. These are public knowledge. Alice secretly chooses a random a in the range $1 < a < p$ and computes $A = g^a \% p$. Similarly, Bob secretly chooses a random b in the range $1 < b < p$ and computes $B = g^b \% p$. Alice sends A over the channel, and Bob sends B over the channel. So Alice knows p, g, a, A, B, Bob knows p, g, A, b, B, and Eve knows p, g, A, B.

Alice computes $K_A = B^a \% p$
Bob computes $K_B = A^b \% p$

They have computed the same thing, since

$$K_A = (B^a) = (g^b)^a = (g^a)^b = A^b = K_B \bmod p$$

Alice and Bob have a shared secret which it is apparently *infeasible* for Eve to obtain.

The feasibility for Alice and Bob follows from the feasibility of exponentiating via the square-and-multiply algorithm, and the feasibility of probabilistic methods (e.g., testing for primality via Miller-Rabin (4.13)) for finding large primes. The infeasibility for the eavesdropper Eve depends (at least) upon the (apparent) infeasibility of computing discrete logarithms.

2 Complexity

It is apparently unreasonable to demand *absolute* unbreakability of ciphers. It is far more reasonable, and sufficient for real use, to require that it be *infeasible* for an attacker to break a system. As one would anticipate, this is a context-dependent qualification.

2.1 Polynomial-time algorithms

A common description of the **complexity** of a computation is as *runtime complexity*, the number of **bit operations** it takes. A **bit** is a 0 or a 1. A bit operation is the application of one of the 2^4 functions that takes two bits as input and produces a single bit as output, or one of the 2^2 functions that takes a single bit as input and produces a single bit output. The abstract model of a bit operation ignores the cost and/or difficulty of reading and writing, and of moving bits among registers.

An algorithm is **polynomial-time** if the run-time is at most polynomial in the number of bits in the input (with a polynomial independent of the input). This means that *even in the worst-case scenario* (for possibly tricky or nasty inputs) the estimate must hold. It is entirely possible that most often even less time is used, but the worst-case scenario is accounted for in this traditional terminology. Questions with polynomial-time algorithms to answer them make up class **P**.

If the *correctness of a guess* at an answer to a question can be proven or disproven in polynomial time, then the question is in class **NP** (*Non-deterministic Polynomial time*). This class of questions certainly contains the class **P** (*Polynomial time*). It is widely believed that the class **P** is strictly smaller than the class **NP**, but this is the main open question in complexity theory. Problems in **P** are considered **easy**, and those in **NP** but not in **P** are considered **hard**. Ironically, since we don't have a proof that **P** is strictly smaller than **NP**, for now we have no proof that there are *any* hard problems in this sense.

A question A is *reducible* to another question B (in polynomial time) if there is a polynomial-time algorithm to answer question A (for a given input) from an answer to question B (with related input). In this context, a possibly unknown and unspecified process to answer B is an **oracle** for B. Surprisingly, it has been shown that there are problems C in **NP** so that *any* problem in **NP** can be reduced to to C in polynomial time.

Such problems are **NP-hard** or **NP-complete**. (Finer distinctions can be made which make these two phrases mean different things.)

In effect, Turing, Church, Post, Kleene, and others verified that *computability* is machine-independent (given a sufficiently complicated computing machine). Similar considerations indicate that the notion of polynomial time is machine-independent. Nevertheless, for practical purposes, the degree of the polynomial and the constants appearing can have a great practical impact.

To avoid misunderstanding, it might be mentioned that discussion of manipulation of integers typically tacitly assumes that they are re-written in binary. This re-writing, and conversion back to decimal, are both polynomial-time, so this conversion does not affect qualitative conclusions about run-time.

It is easy to verify that the usual arithmetic algorithms involving Hindu-Arabic numerals are polynomial-time. Especially for large-ish integers, there are several algorithms for multiplication which are faster than the method taught in grade school. See [Kn]. However, the speed-ups don't occur until the integers are considerably larger than those used by schoolchildren. Indeed, direct experimentation seems to indicate that the overhead for clever algorithms makes these inferior to optimized machine-code implementations of classical algorithms, until the integers become larger than current cryptographic uses demand.

It is more common to worry about *time* (i.e., runtime) rather than *space* (i.e., memory used), but this concern may be simply an artifact of the apparent fact that in practice most bottlenecks are run-time rather than memory.

If an algorithm does not (in its worst case) run in polynomial time, then (by default) we say it runs in **exponential time**. Note that this is measured in terms of input size.

Thus, for example, recall that the naive trial-division test for primality of a number N uses roughly $\sqrt{N}$ steps to prove that N is prime (if it is). The *size* n of N as input is $n = \ln_2 N$, and

$$\sqrt{N} = 2^{\ln_2 N/2} = 2^{n/2}$$

This grows faster than any polynomial in n. That is, trial division is an exponential algorithm.

2.2 Probabilistic algorithms

The notion that an algorithm could have random elements is surprising. Even more surprising is that for some important problems there are *much* faster probabilistic algorithms than deterministic ones. There is some charming and colorful terminology for probabilistic algorithms. A **Monte Carlo** algorithm *always* yields an answer (in polynomial time), but this answer has only a *probability* of being correct. It is **yes-biased** if a 'yes' answer is true with some probability, but a 'no' answer is always correct. (Similarly for **no-biased** Monte Carlo algorithms.) A Monte Carlo algorithm is either yes-biased or no-biased, so 'half' its answers are *certain.*

The pseudo-primality tests mentioned in 4.13 are yes-biased Monte Carlo algorithms, since a 'yes' answer (asserting primality) may be wrong, but a 'no' answer (asserting compositeness) is always correct. This holds for the simplest test, the Fermat test, and also for the Miller-Rabin test for strong pseudoprimality.

A **Las Vegas** algorithm has *expected* run time which is polynomial, so may occasionally fail to give an answer (in polynomial time). But if it *does* give an answer for a particular input (after whatever time) then the answer is *correct.*

A less standard usage is that an **Atlantic City** algorithm gives a correct answer at least 3/4 of the time, and runs in polynomial time. (The number 3/4 can be replaced with any probability above 1/2.) Atlantic City algorithms are *two-sided* versions of Monte Carlo algorithms.

There is notation for these classes of algorithms. The class **ZPP** (*Zero-error Probabilistic Polynomial time*) consists of questions answerable by a (polynomial-time) Las Vegas algorithm. The class **RP** (*Randomized*

Polynomial time) consists of those questions answerable in polynomial time by a Monte Carlo method. Questions answerable by polynomial-time Atlantic City algorithms are in **BPP** (*Bounded-error Probabilistic Polynomial time*).

2.3 Subexponential algorithms

At this time, the best known algorithms for factoring integers into primes, for computing discrete logarithms, and for other important computations, are *not* polynomial-time, but are nevertheless significantly faster than literal exponential-time. We can quantify this distinction. The class of $L(a, b)$ of algorithms consists of those whose runtime for input of size n is

$$O(\, e^{(b+\epsilon)) \cdot n^a \, (\ln n)^{1-a}}) \qquad \text{(for every } \varepsilon > 0)$$

with $b \geq 0$, $0 \leq a \leq 1$. The union

$$\bigcup_{0<a<1,\ 0<b} L(a, b)$$

for all $0 < a < 1$, $0 < b$ is the collection of **subexponential algorithms**.

E.g., for $b \geq 0$ $L(0, b)$ is the collection of polynomial-time algorithms with runtimes $O(n^b)$. On the other hand, $L(1, b)$ is genuinely exponential-time algorithms with runtimes $O(e^{bn})$.

Currently, the best algorithms currently known for *factoring* and the best algorithms for computing *discrete logarithms* in $\mathbb{Z}/p^e$ (p prime) are (conjecturally) in a class

$$L(\frac{1}{3}, 1.923)$$

See [Co1] for factoring and [A] for discrete logs in $\mathbb{Z}/p^e$. These run-time estimates are *conjectural* since they rely upon plausible but unproven hypotheses on the distributions of various special sorts of integers. It is a striking coincidence that the two conjectural runtimes are the same. For less elementary abstract discrete logarithm problems, such as on *elliptic curves*, no runtime this low is known. This gives a technical advantage to elliptic curve ciphers, and this margin is important in practice.

2.4 Kolmogoroff complexity

Solomonoff, Chaitin, and Kolmogorov independently developed a version of complexity, **program (-length) complexity**, which emphasizes *program length* rather than *runtime*. This notion of complexity affords a useful perspective on *(pseudo) random numbers.*

The idea is that a rather special string of characters such as

110110110110110110110110110110110110110110110110110

can be described much more briefly than by listing the characters, as ‘17 copies of 110’. By contrast, the string

111011110111010111111111110001000110000010011000110

admits no obvious simpler description than to just list all the characters directly.

From a probabilistic viewpoint, one might pretend that every string of (for example) 51 0’s and 1’s is equally likely to be chosen ‘at random’. However, this is naive. For example, of the two strings above, the second is plausibly ‘random’, but the first is not. That is, we would not be suspicious if we were told that the second one was chosen ‘at random’ (whatever this means). However, the first is ‘too structured’ and we would doubt that it was ‘chosen at random’.

We can modify our notion of 'random string of characters of length n' to incorporate the idea that a *truly random object* should not have a description any simpler than the thing itself. This makes more legitimate our objection to the 17 copies of the string 110, while still admitting the other string as random, since no simpler description string is obvious.

An informal definition of the **Kolmogorov complexity** of an object is the *length of the shortest description of the object.* Here the object itself *is* a description of itself. The issue is whether there is a *shorter* description of it than just it itself, where 'description' certainly may include a program (in *some* language, running on *some* machine) which generates the thing.

Of course this pseudo-definition makes sense only in relation to a given descriptive apparatus. Otherwise we are vulnerable to the usual paradoxes concerning such things as 'the putative first integer not describable in fewer than twelve words'.

It can be proven, not surprisingly, after making things more precise, that 'most' strings of characters of a given length n have no description significantly shorter than n, so that the shortest description of a typical one is achieved just by writing it out. A counting argument essentially suffices.

A necessary preliminary to a serious treatment of this subject is proof that program/description length only depends in a rather shallow manner upon the machine or descriptive apparatus. [LV] gives a thorough introduction to many facets of this subject, with historical and conceptual background.

2.5 Linear complexity, LFSRs, LCGs

The family of LFSRs (linear feedback shift registers) and LCGs (linear congruential generators) generate potentially infinite streams

$$b_0,\, b_1,\, b_2,\, \ldots$$

of values b_i from a fixed finite alphabet, specified by very simple rules and short lists of parameters. This sharply circumscribed model of generation of an output stream can be viewed as a much simpler restricted cousin of Kolmogoroff (program) complexity. These LFSRs and LCGs incidentally give examples of pRNGs (pseudo-random number generators) unsuitable for cryptographic uses, though useful in other situations if used carefully.

For example, a **linear feedback shift register** (LFSR) most often uses alphabet $\mathbb{Z}/2$, is specified completely by a **length** n and coefficients $c_1, c_1, \ldots, c_n \in \mathbb{Z}/2$, and the values b_i with $i \geq n$ are produced by a recursion

$$b_i = c_1 b_{i-1} + c_2 b_{i-2} + \ldots c_n b_{i-n}$$

Thus, the *seed* consisting of the first n values

$$\text{seed} = (b_0, b_1, \ldots, b_{n-1})$$

together with the n coefficients c_i completely determine the output stream. In this simple case of alphabet $\mathbb{Z}/2$, the non-zero coefficients are necessarily 1, so it suffices to tell merely the indices i such that $c_i \neq 0$. Engineering terminology of circuit diagrams often refers to indices with non-zero coefficients as *taps*.

The elementary features of LFSRs are easily analyzed by linear algebra: the matrix giving the linear recurrence relation

$$\begin{pmatrix} c_1 & c_2 & c_3 & \ldots & c_{n-1} & c_n \\ 1 & 0 & 0 & \ldots & 0 & 0 \\ 0 & 1 & 0 & \ldots & 0 & 0 \\ 0 & 0 & 1 & \ldots & 0 & 0 \\ \vdots & \vdots & & & \vdots & \vdots \\ 0 & 0 & & & 0 & 0 \\ 0 & 0 & & \ldots & 1 & 0 \end{pmatrix} \begin{pmatrix} b_i \\ b_{i-1} \\ \\ \\ \vdots \\ b_{i-n+2} \\ b_{i-n+1} \end{pmatrix} = \begin{pmatrix} b_{i+1} \\ b_i \\ \\ \\ \vdots \\ b_{i-n+3} \\ b_{i-n+2} \end{pmatrix}$$

has (as usual) characteristic polynomial

$$P(x) = x^n + c_1x^{n-1} + c_2x^{n-2} + \ldots + c_{n-1}x + c_n$$

where we take advantage of the fact that the characteristic is 2 to ignore signs.

Since the internal state

$$(b_{i-n+1}, b_{i-n+2}, \ldots, b_{i-1}, b_i)$$

has $2^n - 1$ possible non-zero values when all b_j are in $\mathbb{F}_2$, this internal state will return to its initial state in at most $2^n - 1$ steps. This maximum is achieved when the degree-n characteristic polynomial $P(x)$ is *primitive*, meaning that

$$x^{2^n-1} = 1 \bmod P(x)$$

but

$$x^\ell \neq 1 \bmod P(x)$$

for $1 \leq \ell < 2^n - 1$. Since the multiplicative group $\mathbb{F}_q^\times$ of a finite field $\mathbb{F}_q$ is cyclic, the primitive polynomials $P(x)$ in $\mathbb{F}_2[x]$ are exactly the irreducibles (over $\mathbb{F}_2$) of generators for $\mathbb{F}_{2^n}^\times$. There are $\varphi(2^n-1)$ such generators (with Euler's totient function φ (4.5)), and thus there are $\varphi(2^n - 1)/n$ such primitive polynomials of degree n. Similar counting arguments apply generally.

The further issue of finding *sparse* primitive polynomials (i.e., with very few non-zero coefficients) is of obvious interest for speed-up of computation. The extreme case of trinomials is already non-trivial. See [Go].

The **linear complexity** of a stream of outputs in $\{0, 1\}$ is the smallest value of n such that the outputs can be produced by a length n LFSR.

Unlike other notions of complexity which are not effectively computable, the linear complexity of a stream of bits is computable in polynomial time (for given length of bit stream). This is the *Massey-Berlekamp* algorithm [Ma], which computes the constants for a LFSR of length n given just $2n$ initial bits. Since presumably any LFSR used in practice would be essentially primitive, the output bitstream would give $2^n - 1$ bits before repeating. Thus, the number (i.e., $2n$) of (initial) bits needed to determine the constants is polynomial in the logarithm of the size of the output (i.e., $2^n - 1$). The procedure of finding a LFSR to generate a given bit stream is also called *shift register synthesis.*

Linear congruential generators (LCGs) generate streams of values

$$b_0, b_1, \ldots$$

in a finite field $\mathbb{F}_q$ given the initial state b_0 and given structure constants A, B in $\mathbb{F}_q$, by

$$b_i = Ab_{i-1} + B$$

Since

$$\begin{pmatrix} A & B \\ 0 & 1 \end{pmatrix} \begin{pmatrix} b_i \\ 1 \end{pmatrix} = \begin{pmatrix} b_{i+1} \\ 1 \end{pmatrix}$$

these also admit an elementary analysis by linear algebra, and, therefore, are inappropriate for cryptographic use.

2.6 Quantum algorithms

One should be aware that *quantum cryptography* and *quantum computing* are very different things. Quantum crypto is already feasible, if not widely used. Quantum computing is not yet publicly known to be feasible.

The practical impact of the feasibility of quantum cryptography is modest, given its complication and given the efficacy of other more familiar cryptographic schemes.

That is, it *does* appear that practical **quantum channels** can be and are being used to communicate with absolute security, in a certain qualified sense. In very rough terms, these are channels designed to be very sensitive to quantum effects, such as changes in state due to observation. The idea is that if anyone eavesdrops on (observes) the channel, this will alter the message for reasons of basic quantum mechanics. Thus, detection of eavesdropping is certain.

A less practical but more dramatic sort of quantum cryptography is **quantum teleportation**, i.e., using the Einstein-Podolsky-Rosen effect [EPR], related to Bell's theorem [Be]. The idea is that if two particles have quantum states which are 'entangled', and then they are separated widely in space, a change of state in one (perhaps caused by an observation being made) *instantly* causes a corresponding change of state in the other. Information has traveled faster than light, and without interaction with any intervening medium. If we could really do this reliably, quantum teleportation would provide absolutely secure communication of a novel type.

There is only very primitive progress (in the public domain, at least) toward building a *quantum computer*. While transistors make use of quantum effects, designers intend for these circuits to behave compatibly with macroscopic objects, *avoiding* the peculiarities of quantum behavior. But by the 1980s *exploitation* rather than *avoidance* of quantum effects had been explicitly considered. For example, R. Feynman noted the infeasibility of using classical computers to efficiently simulate quantum events.

The possible impact of hypothetical quantum computers was unclear until about 1993, at which time Peter Shor's algorithm for factoring large numbers quickly on a quantum computer [Sho1], [Sho2] decisively illustrated the cataclysmic changes that could occur in computation. For example, RSA would be irretrievably broken. Similarly, the discrete log problem in $(\mathbb{Z}/p)^\times$ (p prime) has a polynomial-time quantum algorithm.

Another example of a quantum algorithm which transcends human intuition based on macroscopic processes is Grover's algorithm, which allows a search of an unstructured collection of n things in $\sqrt{n}$ time. Even for ciphers not dependent upon number theoretic issues with polynomial-time quantum algorithms, widespread availability of Grover's algorithm would speed up brute-force attacks, necessitating substantial increases in key lengths in all ciphers.

In the early 1990s enthusiasts were optimistic, but (public-domain) progress toward building a sufficiently large quantum computer has been much slower than hoped. For example, in 2000 [KLMT] reported successful manipulation of 3 qubits, then 5 qubits, and now 7 qubits. In all cases they used nuclear magnetic resonance (NMR).

Even if we imagine that quantum computers will be constructed some time soon, low-level traditional computational techniques must be rebuilt. For example, from [PB], it is impossible to delete a copy of an arbitrary quantum state perfectly. While a classical computer can delete information, and can undo the deletion using a copy, a quantum computer cannot delete quantum information.

A technical issue which has effectively disappeared for classical computers is (internal) *error correction*, and *fault-tolerant* computing. Nowadays classical computers are sufficiently reliable (in hardware) that little internal redundancy is built in. By contrast, the very nature of quantum effects exploited by (hypothetical) quantum computers will require development of quantum error-correction techniques. As a sample, see [CS].

It is unclear whether there will ever be quantum computers accessible to ordinary people. Indeed, one might speculate upon whether ordinary people would be *allowed* to own them.

3 Background on Symmetric Ciphers

Until the 1960's, the only ciphers were **symmetric ciphers**, meaning that knowledge of the encryption key is equivalent to knowledge of the decryption key (modulo feasible computations). It is worthwhile to see why certain old symmetric ciphers fail, as these failures illustrate possible difficulties for the asymmetric (public-key) cipher situation as well. Further, many contemporary applications of public-key ciphers also make substantial use of (modern) symmetric ciphers, so it would be naive to discuss public-key ciphers without some acknowledgment of symmetric ciphers.

3.1 Bad Old Symmetric Ciphers

Cryptograms (substitution ciphers), are broken by analysis of letter frequences and small words in English (or other alphabet-based natural languages). The fact that these occur in many newspapers as puzzles should already suggest that they are completely insecure. Shift ciphers and affine ciphers are special cases of cryptograms. Cryptograms give an immediate example of a cipher in which the key space (i.e., set of all possible keys) is fairly large ($26! \sim 4 \cdot 10^{26}$) but the cipher is nevertheless insecure.

Anagrams (permutation ciphers) can be broken by *multiple anagramming.* This means to consider two (or more) ciphertexts encrypted with the same key, to attempt *simultaneous* decryption, rejecting rearrangements if prohibited pairs or triples of letters occur in *either* putative decryption. In a language such as English where many letter combinations are rare or effectively prohibited, this procedure is surprisingly effective. Again, anagrams are ciphers with unlimited key space but nevertheless insecure.

It is worth noting that an invidious systemic weakness in many classical ciphers is that they admit *incremental* or *gradual* decryption, wherein information about the key or plaintext is found a little bit at a time. A good cipher does not allow this, as it is a fatal weakness. And note that various forms of information leakage in naive implementations of modern ciphers give an opening to incremental decryption. From the viewpoint of computational feasibility, a cipher which permits incremental decryption in effect allows an attacker to replace a search through an unstructured set of (for example) 2^n possible decryptions with a search through a binary tree with 2^n leaves, eliminating *branches* of the tree at each step, rather than single leaves. Thus, rather than the 2^n steps for the unstructured case, an attacker may face only n steps to break the cipher.

Vigenère ciphers are interesting **polyalphabetic** ciphers, meaning that they may encrypt the same plaintext character differently depending upon its position. They do not change the position of characters in the plaintext. A Vigenère cipher encrypts a stream

$$x_0 \, x_1 \, x_2 \, \ldots$$

from an alphabet $\mathbb{Z}/N$ using a key $k_0, \ldots, k_{n-1}$ of length n (from $\mathbb{Z}/N$) to the ciphertext

$$\ldots, \; y_i = (x_i + k_{i \,\%\, n}), \; \ldots$$

That is, a potentially infinite sequence is made from the key material by forcing it to repeat, and then the plaintext sequence and periodic key sequence are added termwise. These ciphers do not admit quite so obvious an attack as cryptograms and anagrams, but, nevertheless, are completely broken. An attack using trigrams due to Kasiski as early as 1880 breaks the cipher, as does the even-stronger Friedman attack from the early 20th century (see [G] or [St], for example), unless the key is random and nearly as long as the plaintext itself. Again, Vigenère ciphers have an indefinitely large key space, but this fails to assure security.

Enigma and **Purple** ciphers were WWII ciphers which were broken due to a combination of key distribution flaws, human error, and modest keyspace size. It is worth noting that the break would have been impossible without Alan Turing's innovations in automated computing. One broader lesson from those episodes is that, even though a hardware or software improvement may not convert an exponential-time computation to a polynomial-time computation, reduction by a large constant factor may be decisive in practical terms.

3.2 One-time pads and their failure modes

The OTP (One-Time Pad) seems to be the only cipher provably secure in absolute terms, i.e., not relative to any unproven assumptions about feasibility of computation. An OTP for a message of length $\leq n$ in alphabet $\mathbb{Z}/2$ uses a *random* key

$$k_0, k_1, \ldots, k_{n-1}$$

of length n, and encrypts a binary plaintext

$$x_0, x_1, \ldots, x_{n-1}$$

to ciphertext

$$..., y_i = x_i + k_i, \ldots \qquad \text{(all in } \mathbb{Z}/2)$$

Each key can be used *only once*. This set-up obviously admits variation. The binary case is sometimes called a **Vernam cipher**.

The proof of security is simply that the *conditional* probability that the plaintext is a particular message *given* the ciphertext is the same as the unconditional probability of the message (i.e., without knowing the ciphertext). In effect, this is the definition of the *randomness* of the key. Intuitively, an attacker learns nothing about the plaintext from seeing the ciphertext.

The operational difficulty is the non-reusability of the key(s), and the requirement that the key(s) be random. Indeed, the **failure modes** of the OTP are catastrophic. For example, if a *periodic* key is used then the OTP becomes a Vigenère cipher, and is broken. If a key is reused, the two messages together effectively create a Vigenère cipher, again creating a break. For that matter, creating securely-random keys is a non-trivial task (9.6).

Thus, **key distribution** and **key generation** are critical bottlenecks for use of OTPs. Nevertheless, because of the security when used properly, OTPs *are* used in critical applications such as nuclear launch security, and highest-level diplomatic communications. The infrastructure costs are too high for more mundane uses.

3.3 DES and AES

The DES (Digital Encryption Standard) from 1976 was a stop-gap measure which accidentally became a *de facto* standard for 30 years, at least in the United States. It is a symmetric cipher whose antecedent *Lucifer* was designed at IBM, and subsequently tweaked at NSA for reasons which were not explained at the time, and perhaps never quite explained fully. The role of the NSA (perhaps unduly) compromised the world's opinion of this modern cipher. It was never publicly broken structurally, but its too-small keyspace made it obsolete by rendering it vulnerable to brute-force attacks [EFF]. [L1] is a more substantial review of the story of DES.

The new standard, the AES (Advanced Encryption Standard), Rijndael, was chosen in late 2000 as a replacement, from among finalists *MARS* from IBM, *RC6* from RSA Laboratories, *Rijndael* from Joan Daemen and Vincent Rijmen [DR], *Serpent* from Ross Anderson, Eli Biham, and Lars Knudsen, *Twofish* from Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson [ST]. The NIST (National Institute of Standards) had stipulated that any patent rights to a candidate cipher would have to be relinquished if the cipher were chosen as the new standard. [L2] discusses AES in a fashion parallel to the discussion of DES in [L1].

DES uses a 64-bit key, of which 8 bits are error-check bits, so it actually has a 56-bit key. The new standard, AES, was mandated by NIST to have 128-bit, 192-bit, and 256-bit keys, and available block sizes of 128, 192, and 256 bits. DES encrypts blocks of 64 bits of plaintext at a time. The AES encrypts blocks of 128, 196, or 256 bits, as mandated.

It is not unreasonable to suppose that the NSA was ahead of public cryptanalytic technique, and in particular was aware of two 'modern' cryptanalytic attacks: **differential cryptanalysis** and **linear cryptanalysis**, beyond brute-force attacks. Very roughly, differential cryptanalysis makes a systematic study of how small changes in plaintext affect the ciphertext. Linear cryptanalysis tries to approximate the effect of encryption by *linear* functions. These and other attacks can be viewed as aiming to effectively demonstrate that the complexity of a cipher is significantly lower than intended. In either case, any detectable pattern allows an attacker increased efficiency. See [St] for a readable account of both these techniques. Differential cryptanalysis was developed by Biham and Shamir [BS1], [BS2], [BS3]. Linear cryptanalysis by Matsui [Mat1], [Mat2]. The analysis of Twofish in [ST] also gives clear examples of these attacks.

We give a rough description of DES for contrast both to AES and to public-key ciphers. A DES encryption consists of 16 **rounds**, meaning repetitions of a simpler process. Each round is **Feistel network**, described just below. Many modern symmetric ciphers are composed of *rounds* each of which is a Feistel network. Rijndael is unusual in that its rounds are *not* Feistel networks.

A **Feistel network** is simple and gives a function *guaranteed to be invertible* and *to be its own inverse.* Fix a positive integer n, which would be 32 in the case of DES. Given a string of $2n$ bits, break them into two parts, the *left* and *right* halves, L and R. View L and R as **vectors** of length n with entries in $\mathbb{F}_2 = \mathbb{Z}/2$. For *any* function f that accepts as inputs n bits and produces an output of n bits, the corresponding Feistel network F_f takes two n-bit pieces L and R as inputs, and produces $2n$ bits of output by

$$F_f(L, R) = (L + f(R), R)$$

where $+$ is vector addition in $\mathbb{F}_2^n$. The key property of a Feistel network is the obvious feature that F_f is invertible and is its own inverse. Each **round** of DES is an enhanced version of the Feistel network we just described, with a specific choice of the function f, depending on the key.

In DES, at the very outset, the 64-bit key has every eighth bit removed, leaving just 56 bits of key material. An important feature of DES (as with most symmetric ciphers) is that each round uses a different 48-bit **subkey** of the 56-bit key. The description of the round subkeys is **key scheduling**, and is a critical feature of any cipher. For each round of DES, the key is broken into two 28-bit halves, each of which is shifted left (with wrap-around) by either 1 or 2 bits, depending upon which round we're in, by an irregular pattern which we'll suppress. Then the *shifted* batch of 56 bits is mapped to 48 bits by a **'compression permutation'** (it's not invertible). The compression permutation is the same for each round, though we won't give it here. So the result of the initial key permutation, with the round-dependent shifts by 1 or 2 bits, followed by the compression permutation, produces a 48-bit **subkey** for each of the 16 rounds.

As indicated above, each round of DES manipulates halves of the text, in a manner that depends on the round subkey. Let L_i be the left half in the i^{th} round, and let R_i be the right half in the i^{th} round. The formula to get from one round to the next is of the form

$$(L_i, R_i) = (R_{i-1}, L_{i-1} + f(R_{i-1}))$$

where f depends on the i^{th} subkey. That is, in addition to the Feistel trick we *interchange* halves of the text, so that both halves will get obscured in the course of several rounds. We must describe the function f and how it depends on the round subkey.

For each round, the right half R_{i-1} has applied to it a fixed (rather simple) **'expansion permutation'** or **E-box**, which accepts 32-bit inputs and creates 48-bit outputs, given by a random-looking formula we'll suppress. The pattern is *not* very complicated, but does have the critical effect of allowing one bit of the text to affect more than one bit of the expanded version. This is an **avalanche effect**. At each round, the 48-bit output from the E-box is added to the 48-bit subkey for that round, both treated as vectors in $\mathbb{F}_2^{48}$.

The most serious and mysterious part of DES, critical to security, is the application of the **substitution boxes**, or **S-boxes** to the 48 bits coming from adding the subkey to the output from the E-box. There are 8 S-boxes, each of which takes a 6-bit input and produces a 4-bit output. The 48 bits are broken into 8 pieces of 6 bits each and fed to the S-boxes. The outputs are concatenated to give a 32-bit output. Each of the S-boxes is given by an input-output table. (The explicit listing of these would be unenlightening, and we resist the temptation to prove the point.)

After all 16 rounds, *not* exchanging left and right halves after the last (16th) round, a *final permutation* is applied to the text. The output is the completed ciphertext. Happily, because of the Feistel network property, and by the relatively simple choice of the initial and final permutation, exactly the same process (with the same key) **decrypts**.

The details we've suppressed seem to admit no simple description, which is to say that there appears to be no clarifying underlying pattern offering an explanation. While possibly frustrating from a certain viewpoint which effectively believes in unlimited compressibility of description, from a Kolmogoroff (program) complexity viewpoint the situation is reasonable, or even desirable in terms of resistance to cryptanalytic attacks. That is, it might be argued that the ineffability of the *details* of DES (as opposed to its broad description as 16 rounds of a Feistel network) is essential to its security.

Once more, it appears that DES eventually failed only because its keyspace was too small, *not* because of any (public-knowledge) structural weakness. It is also true that its block size was small enough so that, with vastly increased machine capabilities, one could contemplate dictionary attacks if DES were operated in the most naive mode (which no cipher should be).

AES, Rijndael: Of all the AES candidates, Rijndael is/was the most mathematically structured. Rijndael can accommodate key sizes of 128, 192, or 256 bits (versus 56 for DES). It acts on plaintexts of block sizes 128, 192, or 256 bits. Rijndael is composed of *rounds*, with 10, 12, or 16 rounds depending on the key size. Each round, rather than being a Feistel network, has four parts:

$$\text{State} \longrightarrow \text{ByteSub(State)} = \text{Affine(Inverse(State))}$$

$$\text{State} \longrightarrow \text{ShiftRow(State)}$$

$$\text{State} \longrightarrow \text{MixColumn(State) (except in the final round)}$$

$$\text{State} \longrightarrow \text{AddRoundKey(State)}$$

described in more detail below. (These names are used in the semi-official Java description/implementation of Rijndael.)

First, whatever the block size, the 128, 196, or 256 bits are grouped into bunches of 8 bits as *bytes*, and then read into a 4-by-n matrix, where n is 4, 6, or 8, respectively.

The *ByteSub* operation acts on each byte of the text, and is the analogue of the *S-boxes* in DES and many other ciphers. The *Inverse* operation is inversion in the finite field $\mathbb{F}_{2^8}$ with 2^8 elements, modeled as $\mathbb{F}_2[x]$ modulo the irreducible ('Rijndael') octic

$$x^8 + x^4 + x^3 + x + 1$$

and taking representatives which are polynomials of degree less than 8. The 8 bits are taken as the coefficients in $\mathbb{F}_2$ of such a representative polynomial. The *Affine* map is the second half of Rijndael's *S-box*, consisting of the $\mathbb{F}_2$-affine map

$$x \longrightarrow Ax + b$$

on $\mathbb{F}_2^8$ where

$$A = \begin{pmatrix} 1&0&0&0&1&1&1&1\\ 1&1&0&0&0&1&1&1\\ 1&1&1&0&0&0&1&1\\ 1&1&1&1&0&0&0&1\\ 1&1&1&1&1&0&0&0\\ 0&1&1&1&1&1&0&0\\ 0&0&1&1&1&1&1&0\\ 0&0&0&1&1&1&1&1 \end{pmatrix} \qquad b = \begin{pmatrix} 1\\1\\0\\0\\0\\1\\1\\0 \end{pmatrix}$$

and now a byte is viewed as an element of $\mathbb{F}_2^8$. Note that A is a circulant matrix.

The *ShiftRow* and *MixColumn* operations are more elementary *diffusion* operations. *ShiftRow* acts on the 4-by-n matrix of bytes by leaving the top row alone, shifting the second row to the right by 1 (with wrap-around), shifting the third row to the right by 2 (with wrap-around), and the bottom row by 3 (with wrap-around). The *MixColumn* operation acts on each column of the 4-by-n matrix of bytes, by left multiplication by the (circulant) matrix

$$\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix}$$

where the entries are in hexadecimal (i.e., base 16).

The *AddKey* operation is, as usual, simply an addition of the text with the round subkey (as vectors over $\mathbb{F}_2$). The non-trivial part of the keyed part of each round is *key scheduling.* In the case of Rijndael, key scheduling is the least comprehensible and least obviously structured of the whole, using an irregular recursion which invokes hard-coded constants. See the AES homepage [AES] for these details.

4 Review of Number Theory

The algorithms here are well-known and are discussed in many sources. We emphasize that the fundamental arithmetic operations, namely addition, subtraction, multiplication, and division-with-remainder, all require a number of single-digit operations (or, essentially equivalently, *bit-operations*) at most polynomial in the number of digits of the integers. Optimization of these operations is not our immediate concern, but can have a practical impact (see [Ber], [Kn]).

4.1 Euclidean algorithm

To compute *gcd*'s, and to compute e^{-1} mod m, use the familiar Euclidean algorithm. To compute $\gcd(x, y)$:

```
Initialize $X = x$, $Y = y$, $R = X \bmod Y$
while $R > 0$
    replace $X$ by $Y$
    replace $Y$ by $R$
    replace $R$ by $X \bmod Y$
When $R = 0$, $Y = \gcd(x, y)$
```

To compute $\gcd(x, y)$ takes at most $2 \ln_2 y$ steps, if $x \geq y$, since one can readily show that in any *two* steps the sizes of the variables X, Y decrease by at least a factor of 2.

4.2 Extended Euclidean algorithm

To compute multiplicative inverses $e^{-1} \bmod x$ with $\gcd(e, x) = 1$, minimizing memory use, rewrite each of the steps in the previous as

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & -q \\ 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} \text{new } X \\ \text{new } Y \end{pmatrix}$$

where $R = X - qY$ with $|R| < Y$. Thus, we obtain an integral matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with determinant ± 1 such that

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ e \end{pmatrix} = \begin{pmatrix} \gcd(x, e) \\ 0 \end{pmatrix}$$

When $\gcd(x, e) = 1$, we have

$$ax + be = 1$$

and thus

$$b = e^{-1} \bmod x$$

The two integers a, b with that property $ax + be = 1$ are useful in a computational form of **Sun-Ze**'s theorem (the so-called *Chinese Remainder Theorem*): given x and e relatively prime, and arbitrary s, t, the simultaneous system of congruences

$$\begin{cases} y & = & s \bmod x \\ y & = & t \bmod e \end{cases}$$

has a unique solution y modulo $x \cdot e$, given by

$$y = ax \cdot t + be \cdot s \bmod xe$$

4.3 Square-and-multiply fast exponentiation

To compute $x^e \,\%\, m$, keep track of a triple (X, E, Y) initialized to be $(X, E, Y) = (x, e, 1)$. At each step of the algorithm:

For E odd replace Y by by $(X \times Y) \,\%\, m$ and replace E by $E - 1$
For E even replace X by $(X \times X) \,\%\, m$ and replace E by $E/2$
When $E = 0$ the value of Y is $x^e \,\%\, m$

This algorithm takes at most $2 \ln_2 E$ steps.

4.4 Fermat's Little Theorem

This asserts that for p prime, for an arbitrary integer b

$$b^p = b \bmod p$$

This has an elementary proof by induction on b, using properties of binomial coefficients, but also is a special case of Euler's theorem (4.5).

The converse of Fermat's theorem is false, but not *very* false. The only *non-prime* $n < 5000$ with $2^n = 2 \bmod n$ are

$$341,\ 561,\ 645,\ 1105,\ 1387,\ 1729,\ 1905,\ 2047$$

$$2465,\ 2701,\ 2821,\ 3277,\ 4033,\ 4369\ 4371,\ 4681$$

Requiring also $3^n = 3 \bmod n$ leaves

$$561,\ 1105,\ 1729,\ 2465,\ 2701,\ 2821$$

Requiring also $5^n = 5 \bmod n$ leaves

$$561,\ 1105,\ 1729,\ 2465,\ 2821$$

Compared with 669 primes under 5000, this is a *false positive* failure rate of less than 1%. We say that n is a **Fermat pseudoprime base** b if $b^n = b \bmod n$. There are only 172 non-prime Fermat pseudoprimes base 2 under 500,000 versus 41,538 primes, a false positive rate of less than 0.41% There are only 49 non-prime Fermat pseudoprimes base 2 and 3 under 500,000, a false positive rate of less than 0.118% There are only 32 non-prime Fermat pseudoprimes base 2, 3, 5 under 500,000:

561	1105	1729	2465	2821	6601	8911	10585
15841	29341	41041	46657	52633	62745	63973	75361
101101	115921	126217	162401	172081	188461	252601	278545
294409	314821	334153	340561	399001	410041	449065	488881

Sadly, adding more such requirements does not shrink this list further. n is a **Carmichael number** if it is a *non-prime* Fermat pseudoprime to *every* base b. In 1994 Alford, Granville, and Pomerance [AGP] showed that there are infinitely-many Carmichael numbers. And it appears that among *large* numbers Carmichael numbers become more common. It is relatively elementary to prove that a Carmichael number must be odd, square-free, and divisible by at least three primes.

4.5 Euler's Theorem

This is a generalization of Fermat's Little Theorem to composite moduli. Let $\varphi(n)$ be Euler's *totient function*, which counts the integers ℓ in the range $1 \le \ell \le n$ which are relatively prime to n. For $\gcd(x, m) = 1$

$$x^{\varphi(n)} = 1 \bmod n$$

This is an immediate corollary of Lagrange's theorem, applied to the multplicative group $\mathbb{Z}/n^\times$ of $\mathbb{Z}/n$.

Euler's theorem proves that **RSA decryption works**, using $\varphi(pq) = (p-1)(q-1)$: with $y = x^e \bmod n$, letting $ed = 1 + M\cdot\varphi(n)$, all equalities modulo n,

$$y^d = (x^e)^d = x^{1+M\cdot\varphi(n)} = x\cdot(x^{\varphi(n)})^M = x\cdot 1^M = x \bmod n$$

4.6 Primitive roots, discrete logarithms

A primitive root b modulo m is a generator for the group $(\mathbb{Z}/m)^\times$. Existence of a primitive root modulo m is equivalent to the cyclic-ness of the multiplicative group $(\mathbb{Z}/m)^\times$. For p prime, $(\mathbb{Z}/p)^\times$ is cyclic, because $\mathbb{Z}/p$ is a field. Relatively elementary arguments then show that there are primitive roots modulo p^ℓ and $2p^\ell$ for $p > 2$ prime, and modulo 4. Non-existence of primitive roots for all other moduli is easier. This was understood by Fermat and Euler.

Because of the cyclic-ness of $(\mathbb{Z}/p)^\times$ for $p > 2$ prime, we have **Euler's criterion**: $b \in (\mathbb{Z}/p)^\times$ is a square modulo p if and only if

$$b^{(p-1)/2} = 1 \bmod p$$

An analogous result holds for q^{th} powers when p is a prime with $p = 1 \bmod q$.

Modulo a prime p, for a fixed primitive root b, for $x \in (\mathbb{Z}/p)^\times$ the **discrete logarithm** or **index** of x modulo p base g is the integer ℓ (uniquely determined modulo $p-1$) such that

$$x = b^\ell \bmod p$$

4.7 Futility of trial division

Trial division attempts to divide a given number N by integers from 2 up through $\sqrt{N}$. Either we find a proper factor of N, or N is prime. (If N has a proper factor ℓ larger than $\sqrt{N}$, then $N/\ell \le \sqrt{N}$.) The extreme case takes roughly $\sqrt{N}$ steps. A crude estimate shows the hopelessness of factoring integers by trial division in the range relevant to public-key cryptography. If $N \sim 10^{200}$ is prime, or if it is the product of two primes each $\sim 10^{100}$, then it will take about 10^{100} trial divisions to discover this. (Minor clevernesses do not seriously reduce this number.) If we could do 10^{12} trials per second, and if there were a 10^{12} hosts on the internet, with $< 10^8$ seconds per year, a massively parallel trial division would take 10^{68} years. Even if CPU speeds were to increase by a factor of 10^{10}, this would still leave 10^{58} years.

Concrete examples of trial division runtimes. Someone who had not been paying attention might still think that trial division does succeed. Some very specific examples, done on a 2.44 Gigahertz machine, coded in C++ using GMP:

1002904102901	has factor	1001401	('instantaneous')
100001220001957	has factor	10000019	(3 seconds)
10000013000000861	has factor	100000007	(27 seconds)
1000000110000000721	has factor	1000000007	(4 minutes)

Nowhere near 10^{200}, and the runtimes are ballooning already.

4.8 Quadratic reciprocity

For an odd prime p, the **quadratic symbol** or **Legendre symbol** $(b/p)_2$ is

$$\left(\frac{b}{p}\right)_2 = \begin{cases} 0 & \text{if } \gcd(b,p) > 1 \\ 1 & \text{if } x^2 = b \bmod p \text{ has a solution } x \text{ and } \gcd(b,p) = 1 \\ -1 & \text{if } x^2 = b \bmod p \text{ has no solution } x \text{ and } \gcd(b,p) = 1 \end{cases}$$

For arbitrary integers n factored as

$$n = 2^{e_0}\, p_1^{e_1} \dots p_k^{e_k}$$

with odd primes p_i, the **extended quadratic symbol** or **Jacobi symbol** is

$$\left(\frac{b}{n}\right)_2 = \left(\frac{b}{p_1}\right)_2^{e_1} \dots \left(\frac{b}{p_k}\right)_2^{e_k}$$

While for p prime the quadratic (Legendre) symbol $(b/p)_2$ tells whether or not b is a square modulo p, this is *not* the case for non-prime n. Thus, the Jacobi symbol is of interest mostly as an auxiliary gadget useful in computation of Legendre symbols. Its utility in this auxiliary role is due to the fact that Jacobi symbols *can be computed quickly* using quadratic reciprocity (4.8).

With this notation, Euler's criterion for b to be a square modulo prime p is

$$\left(\frac{b}{p}\right)_2 = b^{(p-1)/2} \bmod p$$

From this follows the basic multiplicative property of quadratic symbols (for prime p)

$$\left(\frac{ab}{p}\right)_2 = \left(\frac{a}{p}\right)_2 \cdot \left(\frac{b}{p}\right)_2$$

The multiplicative property for the Jacobi symbols is analogous: For arbitrary odd integer n, and for a and b relatively prime to n,

$$\left(\frac{ab}{n}\right)_2 = \left(\frac{a}{n}\right)_2 \cdot \left(\frac{b}{n}\right)_2$$

This follows directly from considering a prime factorization of n and invoking the multiplicativity of the Legendre symbol.

The **Law of Quadratic Reciprocity** is the first result of *modern* number theory. Lagrange had conjectured it in the late 18th century, but it was not proven until 1796, by Gauss. From a naive viewpoint, there is no reason why any such thing should be true. But, by now, 200 years later, this result has been well assimilated and is understood as the simplest representative of a whole family of *reciprocity laws*, themselves part of *classfield theory*, itself has been assimilated into the Langlands program. Quadratic reciprocity asserts that, for distinct odd primes p and q,

$$\left(\frac{p}{q}\right)_2 = (-1)^{(p-1)\cdot(q-1)/4} \left(\frac{q}{p}\right)_2$$

Also there are the *supplementary laws*

$$\left(\frac{-1}{p}\right)_2 = (-1)^{(p-1)/2}$$

$$\left(\frac{2}{p}\right)_2 = (-1)^{(p^2-1)/8}$$

Granting this, we have an algorithm for telling whether $x^2 = c \bmod p$ is solvable or not, although this does not *find* the square root if it exists. The difference between *proving existence* of the square root and the harder task of *finding it* is parallel to the difference between proving that a number *fails to be prime* and actually *finding a proper factor.* Indeed, apart from taking out powers of 2, there is no reason to factor into primes the inputs to these quadratic symbols! This is especially relevant for large integers.

As a corollary, we have*(Quadratic Reciprocity for Jacobi symbols)*: For m, n odd positive integers

$$\left(\frac{m}{n}\right)_2 = (-1)^{(m-1)(n-1)/4} \left(\frac{n}{m}\right)_2$$

4.9 The Prime Number Theorem

This theorem asserts that the number $\pi(x)$ of primes below x has the asymptotic property

$$\pi(x) \sim \frac{x}{\ln x}$$

meaning that

$$\lim_{x\longrightarrow\infty} \frac{\pi(x)}{x/\ln x} = 1$$

This was proven independently in 1896 by Hadamard and de la Vallèe Poussin. The argument has been greatly simplified since then, but still makes essential use of the fact that the (analytically continued) Riemann-Euler zeta function

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}$$

is non-zero on the line $\mathrm{Re}(s) = 1$, at the edge of the region of convergence in the complex plane.

As a raw heuristic, from the Prime Number Theorem we imagine that roughly $1/\ln n$ of integers near n are prime. Equivalently, we expect to test about $\frac{1}{2}\ln n$ randomly chosen integers near n before finding a prime. Note that this is laughably unsupported without knowing an error term in the assertion of asymptotics of the prime-counting function $\pi(x)$. Nevertheless, it is useful to think of the primes as having *density* $1/\ln n$ near n.

4.10 The Riemann Hypothesis (RH)

From 1858 or so, this is the conjecture that all the zeros of (the analytically continued) $\zeta(s)$ in the strip $0 < \mathrm{Re}(s) < 1$ actually lie on the middle *critical* line $\mathrm{Re}(s) = 1/2$. If this were true, then we would have a much better error term in the prime number theorem than any presently known, namely

$$\pi(x) = \mathrm{li}(x) + O(\sqrt{x}\,\ln x)$$

where

$$\mathrm{li}(x) = \int_2^x \frac{dt}{\ln t} \sim \frac{x}{\ln x}$$

differs slightly from $x/\ln x$, though is roughly asymptotic to it. (The proof that RH implies this error term took several decades of development in complex analysis.) If this were so, then there would be rigorous justification for the heuristic that primes have density $1/\ln n$ near n.

4.11 Dirichlet's Theorem

This theorem on primes in arithmetic progressions is that for $\gcd(a, N) = 1$ there are infinitely many primes in the arithmetic progression

$$\{a + N\ell : \ell = 1, 2, 3, \ldots\}$$

This can be proven simply for some special values of a, such as $a = 1$, but is best proven using the behavior of *Dirichlet L-functions*

$$L(s, \chi) = \sum_{n \geq 1} \frac{\chi(n)}{n^s}$$

where χ is a *Dirichlet character* modulo some N, meaning that $\chi(n)$ depends only upon n mod N, and that χ has the multiplicative property $\chi(mn) = \chi(m)\chi(n)$. A little more specifically, as Euler proved the infinitude of primes from the blowing-up of $\zeta(s)$ as $s \longrightarrow 1^+$, Dirichlet proved this theorem from the fact that for χ taking values other than 1 the L-function $L(s, \chi)$ has a non-zero *finite* limit as $s \longrightarrow 1^+$.

4.12 The Extended Riemann Hypothesis (ERH)

This is a similar conjecture about the zeros of the (analytically continued) Dirichlet L-functions, that the only zeros of these L-functions in the strip $0 < \mathrm{Re}(s) < 1$ actually lie on the middle *critical* line $\mathrm{Re}(s) = 1/2$. This conjecture would imply a sharp error term for the asymptotics of primes in arithmetic sequences. Also, the ERH arises in many questions other basic questions, such as estimates relevant to a deterministic form of the Miller-Rabin pseudoprimality test (4.13).

4.13 Pseudoprimes, probable primes

We should substantiate the assertion that testing large integers for primality is feasible. First, any practical procedure to hunt for large primes typically uses Fermat's pseudo-primality testing base 2 as a first filter, given the simplicity of this test (and the square-and-multiply algorithm). Recently it has been shown [AKS], [Bor], [Ber] that there is a deterministic polynomial-time algorithm to test primality. However, it seems that the algorithm is slower than the best probabilistic algorithms, so will not replace them in the short run. About **terminology**: we will call an integer that has passed some primality test a *pseudoprime.* This usage is not universal. Sometimes the term *pseudoprime* is meant to refer only to a *non-prime* integer that has, nevertheless, passed a primality test, reserving the term *probable prime* for integers that have passed a test but whose actual primality is unknown.

The first probabilistic (pseudo-) primality test which admitted a useful analysis was that of Solovay and Strassen [So] from 1977. The provable result is that, if n were composite, then at least half the integers b in the range $1 < b < n$ would detect this via the Solovay-Strassen test. Specifically, an integer n passes the **Euler pseudoprime** (Solovay-Strassen) test base b if

$$b^{(n-1)/2} = \left(\frac{b}{n}\right)_2 \bmod n$$

where the right-hand side is an extended (Jacobi) quadratic symbol.

A better test with provable behavior is the **Miller-Rabin** test [Ra], [M] for 'strong' pseudoprimes, from 1978–80. To explain why this test might work, first note that for $n = r \cdot s$ composite (with $\gcd(r, s) = 1$), by Sun-Ze's theorem there are at least 4 solutions to

$$x^2 = 1 \bmod n$$

Namely, there are 4 choices of sign in

$$x = \pm 1 \bmod r \quad x = \pm 1 \bmod s$$

Thus, if we find $b \neq \pm 1 \bmod n$ such that $b^2 = 1 \bmod n$, n is definitely *not* composite. Roughly, the **Miller-Rabin test** (details shortly) looks for such extra square roots of 1 modulo n.

Theorem: (Miller-Rabin) For composite n, at least $3/4$ of b in the range $1 < b < n$ will detect the compositeness (via the Miller-Rabin test).

Pseudo-corollary: If n passes the Miller-Rabin test with k random bases b, then

$$\text{`probability } n \text{ is prime'} \geq 1 - \left(\frac{1}{4}\right)^k$$

The **Miller-Rabin test base** b is

```
factor $n - 1 = 2^s \cdot m$ with $m$ odd
replace $b$ by $b^m \bmod n$
    if $b = \pm 1 \bmod n$ stop: $n$ is 3/4 prime
    else continue
set $r = 0$
while $r < s$
    replace $b$ by $b^2 \bmod n$
    if $b = -1 \bmod n$ stop: $n$ is 3/4 prime
    elsif $b = +1 \bmod n$ stop: $n$ is composite
    else continue
    replace $r$ by $r + 1$
if we fall out of the loop, $n$ is composite.
```

If n passes this test it is a **strong pseudoprime base** b. The proof is not terribly difficulty, but is not short, and requires attention to detail. See [BS], [Ko1], [G]. In the course of the proof one also verifies that strong pseudoprimes base b are a subset of Euler pseudoprimes base b, which are themselves a subset of the Fermat pseudoprimes base b.

If the *Generalized Riemann Hypothesis* were true, then for composite n there would be $1 < b < 2\ln^2 n$ such that n fails the test base b (see [M]). But even if we knew the truth of the Generalized Riemann Hypothesis, we would not use $2\ln^2 n$ bases b in the Miller-Rabin test, since a mere 20 (or 50, or 100 would provide sufficient surety, and would take much less time.

Failure rate of Miller-Rabin? The fraction of b's which detect compositeness is apparently much greater than 3/4. For $n = 21311$ the detection rate is 0.9976. For 64777 the detection rate is 0.99972. For 1112927 the detection rate is 0.9999973. Under $50,000$ there are only 9 non-prime strong pseudoprimes base 2, namely 2047, 3277, 4033, 4681, 8321, 15841, 29341, 42799, 49141. Under $500,000$ there are only 33 non-prime strong pseudoprimes base 2. Under $500,000$ there are *no* non-prime strong pseudoprimes base 2 and 3. In the range $100,000,000 < n < 101,000,000$ there are 3 strong pseudoprimes base 2 whose compositeness is detected base 3, namely 100463443, 100618933, 100943201.

4.14 Hunting for big primes

A naive but reasonable approach to find a (probable) prime above a lower bound $2n$, assuming no randomness properties are needed, is as follows. The test to which we will subject each candidate N is:

```
Trial division by a few small primes:
    if $N$ passes, continue, else reject $N$ at this point
Fermat pseudoprime test base 2
    if $N$ passes, continue, else reject $N$ at this point
Miller-Rabin pseudoprime test base with bases
    the first 20 primes
```

Then test $N = 2n + 1$. If it passes, we're done. If not, replace N by $N + 2$ and continue. Our heuristic from the Prime Number Theorem is that we expect to test roughly $\frac{1}{2} \cdot \ln n$ candidates near n before finding a prime.

As examples, we hunt using Fermat base 2, and then Miller-Rabin base 2, 3, 5, to find the next prime after some representative starting points. Even the 1000-digit computation took only 90 seconds.

First 'prime' after	10^{21}	is	$10^{21} + 117$
First 'prime' after	10^{50}	is	$10^{50} + 151$
First 'prime' after	10^{100}	is	$10^{100} + 267$
First 'prime' after	10^{200}	is	$10^{200} + 357$
First 'prime' after	10^{300}	is	$10^{300} + 331$
First 'prime' after	10^{1000}	is	$10^{1000} + 453$

4.15 Generating big random primes

When a cipher or protocol calls for a *random* prime above 2^{1024}, one should not pick the first (probable) prime above 2^{1024} each time, to say the least. Rather, for example, one could reasonably use 1024 random bits $b_0, b_1, \ldots, b_{1023}$ and start testing at

$$N = 2^{1024} + \sum_{i=0}^{1023} b_i \cdot 2^i$$

This leaves open the question of generating high-enough quality random bits for this purpose.

4.16 Continued fractions

For real x the recursion

$$\begin{array}{rcll} a_0 & = & \text{floor}(x) & x_1 = 1/(x-a_0) \\ a_1 & = & \text{floor}(x_1) & x_2 = 1/(x-a_1) \\ a_2 & = & \text{floor}(x_2) & x_3 = 1/(x-a_2) \\ & \ldots & & \end{array}$$

gives

$$x = a_0 + \cfrac{1}{a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \ddots + \cfrac{1}{a_n + x_n}}}}$$

The usual notational relief is to write this as

$$x = a_0 + \frac{1}{a_1+} \frac{1}{a_2+} \ldots \frac{1}{a_n + x_n}$$

The **nth convergent** p_n/q_n is obtained by truncating

$$\frac{p_n}{q_n} = a_0 + \frac{1}{a_1+} \frac{1}{a_2+} \ldots \frac{1}{a_n}$$

where the numerator p_n and denominator q_n are relatively prime integers. For example, because

$$x = 1 + \frac{1}{1+x}$$

has positive root $\sqrt{2}$

$$\sqrt{2} = 1 + \frac{1}{2+} \frac{1}{2+} \ldots$$

Similarly,

$$\frac{1+\sqrt{5}}{2} = 1 + \frac{1}{1+} \frac{1}{1+} \ldots$$

The recursion for the convergents in terms of the integers a_i appearing in the expansion is

$$p_i = a_i p_{i-1} + p_{i-2}$$

$$q_i = a_i q_{i-1} + q_{i-2}$$

In a similar vein, one can easily verify that

$$p_{n+1}/q_{n+1} - p_n/q_n = \frac{(-1)^n}{q_n \, q_{n+1}}$$

and x is always between successive convergents

$$p_{2n}/q_{2n} \leq x \leq p_{2n+1}/q_{2n+1}$$

These convergents are the best rational approximations to x for given size of denominator.

4.17 Square roots modulo primes

These can be computed quickly by a probabilistic algorithm. This is a necessary ingredient in the *quadratic sieve* (8.4). As a preamble, recall from above that for a prime p equal to 3 modulo 4, and for a square b mod p we have

$$\text{square root of } b \bmod p = b^{(p+1)/4} \bmod p$$

This (deterministic) formula is a special case of the following (probabilistic) algorithm.

Suppose that b is a square modulo p. This can be tested efficiently by Euler's criterion, or by quadratic symbol computations. Remove all the factors of 2 from $p-1$, giving

$$p-1 = 2^s \cdot (\text{odd})m$$

Our first approximation at a square root of b mod p is

$$c = b^{(m+1)/2} \bmod p$$

Specifically, we claim that

$$c^2/b = 2^{s-1}\text{th root of unity mod } p$$

Indeed, modulo p,

$$(c^2/b)^{2^{s-1}} = ((b^{(m+1)/2})^2 \cdot b^{-1})^{2^{s-1}} = b^{(m+1)\cdot 2^{s-1}} \cdot b^{-2^{s-1}} = b^{(p-1)/2} = \left(\frac{b}{p}\right)_2$$

by Euler. So it remains to modify c by a 2^sth root of unity mod p to get the actual square root of b.

Let g be a *non-square* modulo p. Again, testing whether a randomly chosen g is a square or not is easy, and if one chooses unluckily one simply tries again, knowing that half the integers mod p are non-squares. Claim that $h = g^m \,\%\, p$ is a *primitive* 2^sth root of 1 modulo p. Indeed, by Fermat's Little Theorem it is *a* 2^sth root of unity. If h were a 2^{s-1}th root of unity, then (since primitive roots exist modulo p) h would be a square, which it is not.

Thus, we wish to determine an integer $0 \le j < 2^{s-1}$ such that

$$h^j \cdot c = \text{ square root of } b \text{ mod } p$$

Write j in binary

$$j = j_0 + j_1 \cdot 2^1 + j_2 \cdot 2^2 + \ldots + j_{s-2} \cdot 2^{s-2}$$

with $j_i \in \{0,1\}$. We recursively determine the coefficients j_i as follows. First, compute

$$(c^2/b)^{2^{s-2}} \,\%\, p$$

Above, we saw that the square of this is 1 mod p so it is ± 1. If it is $+1$, take $j_0 = 0$, and if it is -1, take $j_0 = 1$. Thus

$$(h^{j_0}c)^2/a \text{ is a } 2^{s-2}\text{th root of 1 mod } p$$

Inductively, suppose $j_0, \ldots, j_k$ have been chosen, such that

$$(h^{j_0+\ldots+j_k 2^k} \cdot c)^2/b$$

is a 2^{s-k-2}th root of 1 mod p. Compute

$$\Big((h^{j_0+\ldots+j_k 2^k} \cdot c)^2/b\Big)^{2^{s-k-3}} = \pm 1$$

If the value is $+1$, take $j_{k+1} = 0$, and take $j_{k+1} = 0$ if the value is -1. Thus, finally,

$$(h^{j_0+\ldots+j_{s-2}2^{s-2}} \cdot c)^2/b = 1$$

which is to say that

$$h^{j_0+\ldots+j_{s-2}2^{s-2}} \cdot c = \text{ square root of } b \text{ mod } p$$

Example: To find a square root of $b = 2$ modulo $p = 401$, first take out the powers of 2 from $p - 1$

$$p - 1 = 401 - 1 = 2^4 \cdot 25$$

Thus, in the notation above, $m = 25$, and $s = 4$. Let

$$c = b^{(m+1)/2} = 2^{(25+1)/2} = 2^{13} = 172 \bmod 401$$

From above,

$$c^2/b = 172^2/2 = 172^2 \cdot 201 = 356 \bmod 101$$

is not 1, so $c = 172$ is not a square root of 2 mod 401. Test $g = 3$ to see whether or not it is a square mod 401: via the square and multiply algorithm gives

$$3^{(401-1)/2} = -1 \bmod 401$$

so $g = 3$ is a non-square. Thus, a primitive $2^s = 2^4$ root of unity mod p is

$$h = g^m = 3^{25} = 268 \bmod 401$$

Compute

$$(c^2/b)^{2^{s-2}} = (172^2/2)^4 = -1 \bmod 401$$

so take $j_0 = 1$. Next, compute

$$((h^{j_0} \cdot c)^2/b)^{2^{s-3}} = ((268 \cdot 172)^2/2)^2 = -1 \bmod 401$$

so $j_1 = 1$. According to the discussion, it should be that

$$h^{j_0 + j_1 2} \cdot c = 268^{1+2} \cdot 172 = 348 \bmod 401$$

is a square root of 2 modulo 401, as indeed can be checked.

4.18 Hensel's lemma

This is an efficient device to find solutions of polynomial equations modulo p^{e+1} from solutions modulo p^e. It is an ingredient in the quadratic sieve, among many other things. It is an analogue of the Newton-Raphson method of approximating *real* roots of polynomials over $\mathbb{R}$, but works better.

Let $f(x)$ be a polynomial with integer coefficients, let p be a prime, and suppose that we have an integer x_0 such that for an integer exponent $e > 0$

$$f(x_0) = 0 \bmod p^e$$

while

$$f'(x_0) \neq 0 \bmod p$$

where f' is the usual derivative of f. Let $f'(x_0)^{-1}$ be an integer which is a multiplicative inverse to $f'(x_0)$ mod p. Then the simplest form of Hensel's lemma asserts that

$$x_1 = x_0 - f(x_0) \cdot f'(x_0)^{-1}$$

satisfies

$$f(x_1) = 0 \bmod p^{e+1}$$

For example to find a square root of 2 modulo 17^3, take $f(x) = x^2 - 2$, and begin with $x_0 = 6$, since $f(6) = 0 \bmod 17$. Here

$$f'(6) = 2 \cdot 6 = 12$$

for which an inverse modulo 17 is 10 (e.g., by extended Euclid). Thus,

$$x_1 = x_0 - f(x_0) \cdot f'(x_0)^{-1} = 6 - 34 \cdot 10 = 244 \bmod 17^2$$

For a square root x_2 modulo 17^3, we can re-use $f'(x_0)$, and continue to obtain

$$x_2 = x_1 - f(x_1) \cdot 10 = 244 - (244 \cdot 244 - 2) \cdot 10 = 4290 \bmod 17^3$$

which (as can be readily verified) is a square root of 2 modulo 17^3.

Solving equations $x^2 = n \bmod 2^e$ is slightly more complicated, and, at some point one certainly will view Hensel's lemma as a p-adic algorithm in order to fit the whole thing into a more stable world-view. But for our immediate purposes (e.g., the quadratic sieve (8.4)), observe that the only odd square modulo 4 is 1, and the only odd square modulo 8 is still 1. But for $n = 1 \bmod 8$, and with $x^2 = n \bmod 2^e$ with $e \geq 3$, the step becomes

$$\text{replace } x \text{ by } x - \frac{(x^2 - n)/2}{x} \bmod 2^{e+1}$$

Then $x^2 = n \bmod 2^{e+1}$.

5 More public-key ciphers

We briefly describe some other public-key ciphers, to illustrate the variety of trapdoors which have been exploited.

5.1 El Gamal Ciphers

This idea appeared in [ElGamal 1985]. It is slightly more complicated than RSA, but still essentially elementary, and admits abstraction to arbitrary groups in place of $\mathbb{Z}/p^\times$. For example, the elliptic-curve cryptosystems are of this nature. See [Sil3], [Wh] in this volume, and many other sources concerning elliptic curves in cryptography, such as [Ko3], [KMV], [BSS]. **The hard task** here is **computation of discrete logs**, for example in the group $\mathbb{Z}/p^\times$ with p prime. The *easy* task is exponentiation (e.g., by square-and-multiply).

Note that computation of discrete logs is *not* necessarily difficult in every group: in the *additive* group $\mathbb{Z}/n$ computation of the discrete log of m for a base b amounts to solving for x in

$$x \cdot b = m \bmod n$$

That is, the problem becomes that of finding a multiplicative inverse, easily accomplished via the extended Euclidean algorithm.

Description of encryption and decryption. Alice chooses a large random prime $p > 10^{150}$, a primitive root b modulo p, and a random key ℓ, an integer in the range $1 < \ell < p$. Alice computes $c = b^\ell \,\%\, p$ and publishes p, b, and c.

Bob, who has no prior contact with Alice, can send a message that only Alice can decrypt is as follows. We can assume that Bob's plaintext x is an integer in the range $0 < x < p$. Bob chooses an auxiliary random integer r, *a temporary secret known only to Bob*, and encrypts the plaintext x as

$$y = E_{b,c,p,r}(x) = (x \times c^r) \,\%\, p$$

Along with this encrypted message is sent the *header* $b^r \,\%\, p$.

Alice's decryption step requires knowledge of the discrete logarithm ℓ, but *not* the random integer r. First, from the header b^r the Alice computes

$$(b^r)^\ell = b^{r \cdot \ell} = (b^\ell)^r = c^r \bmod p$$

Then the plaintext is recovered by multiplying by the multiplicative inverse $(c^r)^{-1}$ of c^r modulo p:

$$D_{b,c,p,r,\ell}(y) = ((c^r)^{-1} \cdot y) \,\%\, p = (c^r)^{-1} \cdot c^r \cdot x \bmod p = x \bmod p$$

5.2 Knapsack ciphers

There is no single knapsack cipher, but rather a family of ciphers using the same underlying mathematical problem, the *knapsack* or *subset sum problem.* Merkle and Hellman [MH] first made a cipher of this type. Adi Shamir [Sh] found a decisive attack. In fact, the progression of attacks and attempts to defeat them is representative of the iterative processes by which real ciphers come into existence. We describe the basic version here and briefly indicate the nature of Shamir's attack.

The **knapsack problem** is: given a list of positive integers $a_1, \cdots, a_n$, and given another integer b, find a subset $a_{i_1}, \cdots, a_{i_k}$ so that

$$a_{i_1} + \cdots + a_{i_k} = b$$

if this is possible. Each element a_i can be used at most once, although some of the values a_i may be the same. There are variants: one is to tell, for given a_i's and b, whether or not a solution *exists.* Or, granting that a solution exists, *find it.* All of these are provably NP-complete. Visibly, a brute-force search adding up all possible 2^n subsets of the a_i's rapidly becomes infeasible.

Call $a = (a_1, \ldots, a_n)$ the **knapsack vector**, and b the **size** of the knapsack. The a_i's which occur in a solution

$$b = a_{i_1} + \cdots + a_{i_k}$$

are **in the knapsack** of size b. A knapsack vector $a = (a_1, \ldots, a_n)$ of length n can be used to *encrypt* a vector $x = (x_1, \ldots, x_n)$ of bits (0's and 1's) by computing a function

$$b = f_a(x) = x_1\, a_1 + \cdots + x_n\, a_n$$

The knapsack vector a and the size b are transmitted. *Decryption* consists of finding a subset of the knapsack vector entries which add up to the size. Thus, the decryption step solves a knapsack problem. As it stands, authorized decryption would be no easier than unauthorized decryption.

There might be more than one possible decryption, which would be bad, so a knapsack vector with the property that for a given size there is *at most one* solution to the knapsack problem is called **injective**. The function

$$f_a(x) = x_1\, a_1 + \cdots + x_n\, a_n$$

above is injective if and only if the knapsack vector $a = (a_1, \ldots, a_n)$ is injective.

By the NP-completeness of the general knapsack problem, we might be confident of the difficulty of unauthorized decryption. But for the authorized decryptor to decrypt, we must arrange things so that the *authorized* decryption amounts to solving a much easier *subproblem.*

A knapsack vector $a = (a_1, \ldots, a_n)$ is **superincreasing** if each a_i is strictly greater than the sum of the preceding elements in the vector, that is, for every index i

$$a_1 + \cdots + a_{i-1} < a_i$$

If the knapsack vector $a = (a_1, \ldots, a_n)$ is superincreasing, then there is a much easier method for solving the knapsack problem with size b:

If $b < a_n$, then a_n *cannot* be in the knapsack of size b (it just won't fit)
If $b \geq a_n$, then a_n *must* be in the knapsack of size b

(since by the superincreasing property the other a_i's add up to less than a_n, so cannot add up to b.) In the first case, keep the same knapsack size b. In the second case, replace b by $b - a_n$. In either case, replace the knapsack vector by $(a_1, \ldots, a_{n-1})$. This converts the problem into a new knapsack problem with a shorter knapsack vector. The process terminates when we reach a_1. The problem will be solved in the affirmative if $a_1 = b$ (with the value of b at that time), and in the negative if $a_1 \neq b$. If the authorized decryptor can decrypt by solving a superincreasing knapsack problem, the decryption is acceptably easy.

On the other hand, if the knapsack were *visibly* superincreasing, then an unauthorized decryptor could do the same. So try to *disguise* the superincreasing property in a manner known to the authorized decryptor, but unknown to adversaries. Even if the list of knapsack items were wildly rearranged, sorting n things takes only $O(n \ln n)$ steps if done slightly cleverly.

The first idea of hiding is by a secret multiplication and reduction modulo some modulus. Alice chooses a superincreasing knapsack vector $(a_1, \ldots, a_n)$, and chooses an integer (the **modulus**)

$$m > a_1 + \cdots + a_n$$

Choosing m to be larger than the sum of the a_i's is what causes this procedure sometimes to be called **strong modular multiplication**. Alice chooses another number (the **multiplier**) t relatively prime to m. Then Alice computes

$$c_i = (ta_i) \% m$$

This gives a new knapsack vector $c = (c_1, \ldots, c_n)$, which Alice publishes as her *public key.* Alice keeps t and m (and the inverse of t modulo m) secret. For Bob to encrypt an n-bit message x so that (supposedly) only Alice can read it, the encryption step is similar to what was done before, encrypting $x = (x_1, \ldots, x_n)$ as

$$b = f_c(x) = c_1x_1 + \cdots + c_nx_n$$

with altered knapsack vector c. Bob transmits ciphertext b to Alice. Alice's decryption procedure is to compute

$$t^{-1} \cdot b \% m = t^{-1}(c_1x_1 + \cdots + c_nx_n)$$
$$= (t^{-1}c_1)x_1 + \cdots + (t^{-1}c_n)x_n = a_1x_1 + \cdots + a_nx_n \bmod m$$

Since m is larger than the sum of all the a_i's, the equality modulo m is actually an equality of integers,

$$(t^{-1}b) \% m = a_1x_1 + \cdots + a_nx_n$$

Since Alice knows the superincreasing knapsack vector of a_i's, or can compute them from the c_i's by

$$a_i = (t^{-1} \cdot b_i) \% m$$

she can solve the knapsack problem and decrypt.

Big Problem: Although implicitly we might have thought that an adversary must find *the* secret t and m in order to convert the problem into a superincreasing knapsack problem, this turned out not to be so. Certainly if an adversary is lucky enough to find *any* t' and m' so that

$$a' = ({t'}^{-1}c_1 \% m', \ldots, {t'}^{-1}c_n \% m')$$

is superincreasing, this converts the problem to an easy one. Adi Shamir [Sh] found a way to find (t', m') in polynomial time to convert the published vector to a superincreasing one. This breaks this simplest knapsack cipher.

(Note, too, that as it stood this cipher was completely vulnerable to chosen-plaintext attacks.)

5.3 NTRU

The NTRU cipher, invented in 1995 by J. Hoffstein, J. Pipher, and J. Silverman [HPS], is mathematically more complicated than RSA or ElGamal, although it still uses only basic abstract algebra and number theory. W. Banks' article [Ba] in this volume discusses this cipher in more detail. The NTRU cipher is patented.

The hard task. The computational task which is presumed to be difficult, thereby presumed to assure the security of the NTRU cipher, is not as readily described as prime factorization or discrete logarithms: it must be hard to *find the smallest vector in a lattice* $\mathbb{Z}^n \approx \Lambda \subset \mathbb{R}^n$. There is a relatively good algorithm, the LLL (Lenstra-Lenstra-Lovasz) algorithm [LLL], improved in [SE], which *quickly* finds a short vector in a *typical* lattice. But when the smallest vector is near to or longer than the 'expected value' of sizes of shortest vectors, the LLL algorithm does not perform well. Parameter choices for NTRU must be made to play upon this effect.

Description of encryption and decryption. Fix positive integer parameters N, p, q, where p, q need not be prime but must be *relatively* prime, and probably $\gcd(N, pq) = 1$. Let R be the set of polynomials in x with integer coefficients and with degree strictly less than N, with the multiplication $\star$ that is polynomial multiplication followed by reduction modulo $x^N - 1$

$$x^i \star x^j = x^{i+j \,\%\, N}$$

That is,

$$\left(\sum_{0 \le i < N} a_i x^i\right) \star \left(\sum_{0 \le j < N} b_i x^i\right) = \sum_{i,j} a_i b_j \; x^{i+j \,\%\, N} = \sum_{0 \le k < N} \left(\sum_{i+j = k \,\%\, N} a_\ell b_k\right) \; x^k$$

The addition in R is addition of polynomials.

On R we also have reduction mod p and reduction mod q of the coefficients of polynomials. Write

$$f \,\%\, p = \text{polynomial } f \text{ with coefficients reduced mod } p$$

$$f \,\%\, q = \text{polynomial } f \text{ with coefficients reduced mod } q$$

Say that a polynomial f has an **inverse** F **mod** p if the polynomial F satisfies

$$(f \star F) \,\%\, p = 1$$

To create an NTRU key, Alice chooses two polynomials f and g of degrees $N - 1$, making sure that f has an inverse F_p mod p and an inverse F_q mod q. Alice's public key is

$$h = (F_q \star g) \,\%\, q$$

Alice's private key is f.

A *message* is a polynomial of degree $N - 1$ with coefficients reduced mod p (in the sense that they lie in the range $(-p/2, p/2]$). For Bob to encrypt a message m for Alice, he randomly chooses a polynomial φ of degree $N - 1$ and computes

$$y = (p\varphi \star h + m) \,\%\, q$$

and transmits this to Alice.

To decrypt, Alice computes

$$a = (f \star y) \,\%\, q$$

and then

$$m = (F_p \star a) \,\%\, p$$

It is not clear that the alleged decryption decrypts. Indeed, the parameters must be chosen properly in order to arrange that correct decryption occurs with high probability.

Why does decryption (mostly) work? The polynomial that Alice computes in the decryption step is

$$\begin{array}{rcll} a & = & f \star y & \\ & = & f \star (p\varphi \star h + m) \,\%\, q & \text{(by definition of encryption)} \\ & = & f \star (p\varphi \star F_q \star g + m) \,\%\, q & \text{(by construction of } h\text{)} \\ & = & (f \star p\varphi \star F_q \star g + f \star m) \,\%\, q & \text{(by distributivity)} \\ & = & (f \star F_q \star p\varphi \star g + f \star m) \,\%\, q & \text{(by commutativity)} \\ & = & (1 \star p\varphi \star g + f \star m) \,\%\, q & \text{(by inverse property of } F_q \text{ mod } q\text{)} \\ & = & (p\varphi \star g + f \star m) \,\%\, q & \end{array}$$

Alice shifts the coefficients by subtracting q if necessary to put them into the interval $(-q/2, q/2]$. By careful choices of parameters, it can be arranged that all the coefficients lie in the range $(-q/2, q/2]$ even before reduction modulo q, so that reduction mod q does nothing. So Alice is really computing

$$\begin{array}{rcll} (a \,\%\, p) & = & (p\varphi \star g + f \star m) \,\%\, q \,\%\, p & \\ & = & (p\varphi \star g + f \star m) \,\%\, p & \\ & = & 0 \star g + f \star m \,\%\, p & \text{(since } p\varphi \,\%\, p = 0\text{)} \\ & = & f \star m \,\%\, p & \end{array}$$

Then the star-multiplication by F_p recovers the plaintext:

$$\begin{array}{rcl} (f \star m) \star F_p \,\%\, p & = & (f \star F_p) \star m \,\%\, p \\ & = & 1 \star m \\ & = & m \end{array}$$

The most serious type of attack on the NTRU cipher seems to be what are called *lattice attacks*, which view the key f as more or less the shortest vector in a special collection ('lattice') of vectors. The LLL algorithm will quickly find a short vector, unless it is frustrated by having the length of the shortest vector be close to or even larger than the expected value of such length (for suitably 'random' lattice). In order to break NTRU, however, the LLL algorithm would have to find one of the shortest vectors in a lattice where all the 'short' vectors have *moderate* length. Suitable parameter settings in NTRU seem to achieve the effect of frustrating LLL.

NTRU sends roughly $\frac{\ln q}{\ln p} > 1$ bits of ciphertext for each bit of plaintext, unlike many older ciphers, which send one bit of ciphertext for each bit of plaintext. For discussion of relative speeds of encryption/decryption, parameter setting, apparent security, and more details, see NTRU's home page at

http://www.ntru.com/

5.4 Arithmetica Key Exchange

The Arithmetica key exchange of Anshel, Anshel, and Goldfeld, [AAG], is a new *key exchange mechanism* (and cipher). By extreme constrast with Diffie–Hellman, it plays upon sophisticated mathematics to make a plausible claim of security. Some general ideas about using word problems in groups to make ciphers appeared earlier in [WM]. A significant new idea in [AAG] is use of the *braid group*, or more generally Artin groups, making use of a new algorithm discovered in [BKL]. The Arithmetica key exchange is patented.

Let G be a group. For a fixed set of elements $S = \{s_1, \ldots, s_n\}$ in G, a **word** in S is any expression

$$s_{i_1}^{k_1} s_{i_2}^{k_2} \ldots s_{i_N}^{k_N}$$

where the exponents k_j are positive or negative integers. The set S *generates* G if every element of G is expressible as a word in the elements of S and their inverses.

The **word problem** in a group G with respect to a subset $S = \{s_1, \ldots, s_n\}$ is the question of determining whether or not two *words* in S are equal, as elements of G. The **conjugacy problem** in a group G with respect to S is the question of determining whether or not two words x, y in S are *conjugate* (meaning, as usual that there is $g \in G$ so that $gxg^{-1} = y$). It is known that in general the word problem is *undecidable*, and the conjugacy problem has a similar status. But note that, even for specific groups where there *is* an algorithm, the algorithm may be 'bad' in that it runs in exponential time in the length of the inputs.

So in general the word problem and conjugacy problem are hard. But in [BKL] a class of groups, the *braid groups* (see below), was distinguished in which the word problem has a polynomial-time solution, and seemingly the conjugacy problem does not. This suggests making a cipher in which authorized decryption requires solving such a word problem, and unauthorized decryption requires solving the conjugacy problem.

A **braid group** with n generators is a group G generated by a set $S = \{s_1, \ldots, s_n\}$ with relations

$$s_i s_{i+1} s_i = s_{i+1} s_i s_{i+1}$$

and

$$s_i s_j = s_j s_i \quad \text{for } |i-j| > 1$$

The definition of **Artin group** includes braid groups as a special case where all parameters m_i are 3 (see just below). An Artin group is a group G generated by a set $S = \{s_1, \ldots, s_n\}$ with relations

$$\underbrace{s_i s_{i+1} s_i \ldots s_i s_{i+1} s_i}_{m_i \text{ factors}} = \underbrace{s_{i+1} s_i s_{i+1} \ldots s_{i+1} s_i s_{i+1}}_{m_i \text{ factors}}$$

and

$$s_i s_j = s_j s_i \quad \text{for } |i-j| > 1$$

Coxeter groups are quotients of Artin groups, by additional relations

$$s_i^2 = e \quad \text{for all indices } i$$

The key exchange. Here Alice and Bob proceed as follows. The public information is a group G and two lists $S_A = \{a_1, \ldots, a_m\}$, $S_B = \{b_1, \ldots, b_n\}$ of elements of G. Alice chooses a secret word a in S_A, and Bob chooses a secret word b in S_B. Alice transmits to Bob the list

$$ab_1a^{-1},\ a_2^b a^{-1},\ \ldots,\ ab_na^{-1}$$

and Bob transmits to Alice the list

$$ba_1b^{-1},\ ba_2b^{-1},\ \ldots,\ ba_mb^{-1}$$

(These must be disguised, *rewritten*, to make this secure.) Then the common key for Alice and Bob will be the expression (the *commutator* of a and b)

$$\text{common key } = aba^{-1}b^{-1}$$

which they can both compute, as follows.

Let

$$a^{-1} = a_{i_1}^{e_{i_1}} \ldots\ a_{i_N}^{k_{i_N}}$$

be an expression of the inverse a^{-1} of Alice's secret a in terms of Alice's a_i's. Then

$$\begin{aligned} ba^{-1}b^{-1} &= b(a_{i_N}^{e_{i_1}} \ldots\ a_{i_N}^{k_{i_N}})b^{-1} \\ &= (ba_{i_1}b^{-1})^{e_{i_1}} \ldots (ba_{i_N}b^{-1})^{k_{i_N}} \end{aligned}$$

These ba_jb^{-1} are exactly what Bob had sent to Alice, so she knows what they are. That is, Alice can compute $ba^{-1}b^{-1}$. And then, since she knows her own secret a, she can compute

$$a \cdot (ba^{-1}b^{-1}) = aba^{-1}b^{-1}$$

Symmetrically, Bob can compute aba^{-1}, and since he knows his own secret b (and its inverse b^{-1}), he can compute

$$(aba^{-1}) \cdot b^{-1} = aba^{-1}b^{-1}$$

Thus, Alice and Bob have a shared secret.

The above description did not depend upon specifics concerning the group G. But for the shared secret $aba^{-1}b^{-1}$ to be unambiguous, it must be convertible to some 'canonical' form. For braid groups this is in effect accomplished in [BKL].

Certainly the most obvious attack on such a cipher would solve the conjugacy problem. At this time the conjugacy problem is not known to have a fast algorithm for braid groups, for example.

6 Protocol Sketches

Beyond key exchanges, trapdoor mechanisms allow other important effects to be achieved. We give simple illustrations below, with the warning that these are (mostly) very naive versions and require considerable further refinement to achieve their stated goals. Typical weaknesses involve set-up, cheating, and issues about whether cheating can be *prevented*, or merely *detected*.

6.1 Signatures

Signatures are as important an application as secrecy. For example, the RSA mechanism can be re-used, effectively run backward, to allow Bob to convince Alice that the message he sent her really came from him, as follows. We assume that Bob has published an RSA modulus n and encryption exponent e, with secret decryption exponent d. To convince Alice that the message x purporting to be from Bob really is from him, Bob computes

$$y = x^d \,\%\, n \text{ (yes, using the \textit{decryption} exponent)}$$

and sends *this* to Alice, together with a little header in plain text explaining to Alice where she can look up Bob's public key e (and modulus n) to use to compute

$$x = y^e \,\%\, n$$

If this computation yields non-gibberish, since d is secret, Alice will believe that Bob sent the message (assuming, as usual, the difficulty of factoring, etc.). This does require that the collection of possible messages x be sufficiently structured, e.g., be in a natural language such as English. However, any eavesdropper can also decrypt. Thus, while this procedure makes a good attempt at **authentication**, it does not in itself provide *secrecy.*

There is also the **ElGamal signature** scheme [E], which has been adopted as the **Digital Signature Standard** (DSA) by the National Institute of Standards and Technology in 1994. To set this up, Bob chooses a large prime p and primitive root g mod p. Bob chooses a secret random private key k, and computes $h = g^k \,\%\, p$. The data p, g, h are the public key.

Given a message x which he wants to sign, Bob chooses an auxiliary random $n \in \mathbb{Z}/(p-1)^\times$, and computes

$$a = g^n \,\%\, p$$

$$b = (x - ka) \cdot n^{-1} \,\%\, p - 1$$

Bob sends x, a, and b to Alice (in addition to the public p, g, h). Alice computes

$$h^a \cdot a^b \,\%\, p$$

and

$$g^x \,\%\, p$$

If these are the same, Alice believes that Bob sent the message x. We can see directly that if a, b are computed propertly then these values will be the same. Indeed, in that case

$$h^a \cdot a^b = g^{ak}\, g^{nb} = g^{ak+nb} = g^{ak+n(x-ka)n^{-1}} = g^x \bmod p$$

Why can Eve not forge a signature in the El Gamal scheme? Clearly several approaches that Eve might take to compute a, b without knowing the key k involve computing a discrete log. There are, however, further possible attacks, such as the *existential* attack discussed in Stinson [St] chapter 7, wherein Eve manages to sign a random message.

And, as one might anticipate, the *nonce* n should not be used to sign two different messages, or Eve can do a feasible computation to break the scheme.

It must be emphasized that for real use, such as in long-lasting documents, considerable refinement of these basic ideas is necessary. Again, see Stinson's [St] chapter 7 for further discussion of signatures.

6.2 Thresh-hold schemes

Also called **secret-sharing schemes**, these are arrangements by which a group of entities restrict access to a secret, so that the secret is inaccessible unless a sufficient number (but not necessarily all) of the entities cooperate.

Say that a secret is **k-shared** among t people if any k or more of them can learn the secret, but a group of fewer than k cannot learn the secret. The problem is: given a **secret** x to be **k-shared** among t people $A_1, A_2, \ldots, A_t$, give A_i a piece of information a_i so that A_i knows a_i (but not a_j for $j \neq i$), *no part* of the secret x can be recovered from any $k-1$ of the a_i, and the secret x can be computed (feasibly) from any k of the a_i's. For given t and x, a list of $a_1, \ldots, a_t$ accomplishing this is a **(k, t)-thresh-hold scheme**. A simple example uses Sun-Ze's theorem, as follows.

Let $m_1, \ldots, m_t$ be mutually relatively prime integers greater than 1. Let $a_1, \ldots, a_t$ be integers. Let $M = m_1 \ldots m_t$, $M_i = M/m_i$, and let $n_i = M_i^{-1} \bmod m_i$. Since m_i is relatively prime to m_j for $j \neq i$, m_i is also relatively prime to M_i, by unique factorization. Thus, the multiplicative inverse N_i exists and is computable, via Euclid. The family

$$x = a_i \bmod m_i \quad \text{for all } i$$

is equivalent to the single congruence

$$x = \sum_{i=1}^{t} a_i\, M_i\, n_i \bmod M$$

Fix k with $1 < k \le t$. Let H_k be the *smallest* product of k different m_i's, and let h_{k-1} be the *largest* product of $k-1$ different m_i's. We must assume that

$$H_k \ge (N+1) \cdot h_{k-1}$$

for some large positive integer N. For any **secret** number x in the range

$$h_{k-1} < x < H_k$$

let $a_i = x \,\%\, m_i$. Then the set $\{a_1, \ldots, a_t\}$ is a (k,t) thresh-hold scheme for x.

This is not hard to see. Suppose that $a_1, \ldots, a_k$ are known. Let $M' = a_1 \ldots a_k$, and $M'_i = M'/m_i$ for $1 \le i \le k$. Let $n'_i = M'^{-1}_i$ mod m_i for $1 \le i \le k$. Let

$$x' = \sum_{i=1}^{k} a_i\, M'_i\, n_i \mod M'$$

Then

$$x' = x \text{ mod } M'$$

Since $M' \ge H_k > x$, the secret x is already reduced modulo M', so can be computed by

$$x = x' \,\%\, M'$$

On the other hand, suppose only $a_1, \ldots, a_{k-1}$ are known. Let $M' = a_1 \ldots a_{k-1}$, and $M'_i = M'/m_i$ for $1 \le i \le k-1$. Let $n'_i = M'^{-1}_i$ mod m_i for $1 \le i \le k-1$. Let

$$x' = \sum_{i=1}^{k-1} a_i\, M'_i\, n_i \mod M'$$

Since $M' = m_1 \ldots m_{k-1}$,

$$x' = x \text{ mod } m_1 m_2 \ldots m_{k-1}$$

Also $m_1 m_2 \ldots m_{k-1} \le h_{k-1}$. Since $h_{k-1} < x < H_k$ and

$$(H_k - h_{k-1})/h_{k-1} > N$$

there are at least N possibilities for y mod M, so

$$x' = y \text{ mod } M'$$

Thus, knowledge of only $k-1$ of the a_i's is insufficient to discover the secret x.

6.3 Zero-knowledge proofs

As a simple case, we look at a protocol for Peter (the prover) to prove to Vera (the verifier) that he knows the factorization of a large integer n which is the product of two large primes p, q (both equal to 3 modulo 4), without imparting to Vera the factorization itself. Our first attempt will fail, since it will allow Vera to cheat. However, this too-naive attempt is enlightening.

First, for prime $p = 3$ mod 4, and for $x \ne 0$ mod p, if x is a square modulo p then there is a distinguished square root b of x, the so-called **principal square root** of x, given by the formula

$$b = x^{(p+1)/4} \text{ mod } p$$

This principal square root b is also characterized as the square root of x which is itself a square. And, indeed, *if* $x = a^2 \bmod p$, then a direct computation (using Fermat's little theorem) verifies that the formula works, and yields a value which is itself a square.

The (faulty) protocol is as follows. Vera chooses random integer x and sends $x^4 \,\%\, n$ to Peter. Peter computes the principal square root $y_1 = (x^4)^{(p+1)/4}$ of $x^4 \bmod p$ and principal square root $y_2 = (x^4)^{(q+1)/2}$ of $x^4 \bmod q$, and uses Sun-Ze's theorem (via the Euclidean algorithm) to compute y so that $y = y_1 \bmod p$ and $y = y_2 \bmod q$. Peter sends this value back to Vera.

Since the principal square root of $x^4 \bmod p$ is certainly x^2 (modulo p), and similarly modulo q, Peter must have computed

$$y = x^2 \bmod pq$$

Since Vera already can compute x^2, Peter has imparted no new information to Vera. (But Vera can cheat! See below.)

Why should Vera be convinced that Peter can factor n into p and q? Because, in any case, being able to take square roots modulo n (a product of two secret primes p and q) by whatever means gives a probabilistic algorithm for factoring n, as follows. Given an **oracle** (an otherwise unexplained mechanism) which computes square roots mod n, repeatedly do the following: choose random x, compute $x^2 \,\%\, n$, and give the result to the oracle, which returns a square root y of x^2 modulo n. Since there are exactly two square roots of any nonzero square modulo a prime, by Sun-Ze there are exactly 4 square roots of any square modulo $n = pq$, and $\pm x$ are just 2 of them. Let the other two be $\pm x'$. Assuming that the original x really was 'random', the probability is $1/2$ that the oracle will return $\pm x'$ as y. If so, then n does not divide either of $x \pm y$ (since $y \neq \pm x \bmod n$), but nevertheless n divides $x^2 - y^2$ (since $x^2 = y^2 \bmod n$). So p divides one of $x \pm y$ and q divides the other of the two. Therefore, $\gcd(x - y, n)$ (computed via Euclid) is either p or q. The oracle can be called repeatedly, with probability $1/2$ that a factorization will be obtained at each step. So the probability that we fail to obtain a factorization after ℓ invocations of the oracle is $(1/2)^\ell$, which goes to 0 quickly. Thus, even if Peter did not know the factorization initially, his ability to take square roots modulo n would allow him to factor n.

But the fact that a square root oracle for n can be used to factor n also allows Vera to cheat, as follows. Instead of giving Peter fourth powers $x^4 \bmod n$ she chooses random x and gives Peter just the square $x^2 \bmod p$. If she can do this repeatedly then she is using Peter as a square-root oracle for n, and thus Vera can factor n, defeating the purpose of the protocol.

The following much-enhanced version of the above idea is the **Fiege-Fiat-Shamir** scheme. Again, Peter knows the factorization $n = p \cdot q$ with secret distinct primes p, q. He wishes to prove to Vera that he knows p and q, without divulging p and q. (Thus, by the way, such a proof would be *reproducible*.)

Peter chooses random (secret) v, and computes and publishes $s = v^2 \,\%\, n$. We already know that if one can compute square roots modulo n then one can (probabilistically) factor n. Thus, whoever can prove that they know v may reasonably claim that they are Peter.

To prove to Vera that he's Peter, Peter chooses $r_1, \ldots, r_k$ random, and sends to Vera the set of squares $s_1 = r_1^2 \,\%\, n, \ldots, s_k = r_k^2 \,\%\, n$. (Again, if anyone can find all the square roots mod n, then they can probably factor n.)

Vera chooses a partition of the set of indices $1, 2, \ldots, k$ into two sets S_1 and S_2, and sends these sets of indices back to Peter. For $i \in S_1$, Peters further sends $t_i = v \cdot r_i$ to Vera, and for $i \in S_2$ Peter sends r_i to Vera. Vera checks whether or not $t_i^2 = v \cdot s_i \bmod n$ and whether $r_i^2 = s_i \bmod n$. If so, she believes that Peter is himself, that is, that he knows the factorization.

6.4 Electronic money

A drawback to conventional credit cards, as opposed to paper money, is that the credit card's value exists only insofar as it is connected to the identity of the cardholder. And, then, the cardholder's transactions are traceable. By contrast, traditional paper money has value *in itself*, and in particular has no connection to one's identity, so does not yield information about other transactions. Current electronic banking and commercial transactions have the same problem as credit cards, in that *someone* knows quite a lot about your on-line financial transactions, where you are when they are made, and your general spending patterns.

Ideally, *e-money* would have a value independent of the possessor's identity, would be *divisible*, would be *transferable*, would *not* be *reusable* (meaning that you could not spend the same dollar twice), would *not* depend upon a *central authority*, and transactions could take place *offline*. These requirements were laid out and met in [OO]. Systems meeting varying subsets of these desiderata appear in [Ch] and [CFN]. See also [Br], which is given a careful exposition in [TW].

A technical but important issue in electronic transactions is *failure mode*: what happens if an electronic transaction is interrupted before completion? Is money withdrawn from your bank, or not? Perhaps it is withdrawn from your account, but doesn't make it to your hands? Protocols, software and hardware ensuring *atomicity* of transactions are important.

6.5 More...

realized via trapdoor mechanisms. **Bit commitment schemes** effectively allow Alice the electronic equivalent of putting either a 'yes' or a 'no' into a sealed envelope and giving the envelope to Bob to be opened later. **Coin flipping over the phone** allows two parties who don't trust each other to be confident that the other reports honestly the outcomes of certain events. Many questions about **electronic voting** remain open.

7 Certifiable Large Primes

For those who must have the traditional version of certainty, there *are* some reasonable algorithms for producing very large numbers together with additional data which makes possible a feasible computation whereby to prove primality. The additional data (and explanation of its role) is a **primality certificate**. At the very least, observe that for an integer $p \sim 2^{1024}$ it is ridiculously infeasible to pretend to list the failed trial divisions that would prove p prime. Yet to make the claim that one has done the trial divisions is not necessarily persuasive at all.

The oldest and very simple way to feasibly certify the primality of a large number is the **Lucas–Pocklington–Lehmer** criterion, originating in work of Edouard Lucas in 1876 and 1891. This is a technique especially useful to test primality of numbers N where the factorization of $N-1$ is known, most often applied to numbers of special forms, such as Fermat numbers $2^{2^n}+1$. A more sophisticated variant of it gives the Lucas-Lehmer test for primality of Mersenne numbers 2^n-1, among which are the largest explicitly known primes. We give a sufficiently simple version so that we can give the quick proof.

Theorem: Let $N-1 = p \cdot U$, where p is prime, $p > U$, and suppose that there is b such that $b^{N-1} = 1 \bmod N$ but $\gcd(b^U - 1, N) = 1$. Then N is prime.

To see this, we recall a fact observed by Fermat and Euler. Namely, for an integer N and integer b with $b^{N-1} = 1 \bmod N$ but $\gcd(b^{(N-1)/p} - 1, N) = 1$, we claim that any prime divisor q of N satisfies $q = 1 \bmod p$. To see this, as $b \cdot b^{N-2} = 1 \bmod N$ it must be that b is prime to N, so b is prime to q. Let t be the order of

b in $\mathbb{Z}/q^\times$. By Fermat's Little Theorem $b^{q-1} = 1 \bmod q$, so $t|q-1$. But the *gcd* condition implies that

$$b^{(N-1)/p} \neq 1 \bmod q$$

Thus, t does not divide $(N-1)/p$. Yet, $t|N-1$. Thus, $p|t$. From $t|q-1$ and $p|t$ we get $p|q-1$, or $q = 1 \bmod p$. This proves the claim. Thus, if the conditions of the theorem are met, then all divisors of N are 1 modulo p. If N were not prime, it would have a prime divisor q in the range $1 < q \leq \sqrt{N}$. But $q = 1 \bmod p$ and $p > \sqrt{N}$ make this impossible. Thus, N is prime.

Let's construct a chain of every-larger certified primes. To get started, by trial division, $p = 1000003$ is prime. Testing by Miller–Rabin, the first strong pseudoprime above $1000 \cdot p$ of the form $p \cdot U + 1$ is $N = 1032003097 = 1032 \cdot p + 1$. By luck, with $b = 2$ $2^{N-1} = 1 \bmod N$ while (by Euclid) $\gcd(2^{(N-1)/p} - 1, N) = \gcd(2^{1032} - 1, N) = 1$ Therefore, N is *certified* prime.

Now let p be the certified prime 1032003097. The first strong pseudoprime above $10^9 \cdot p$ of the form $p \cdot U + 1$ is

$$N = 1032003247672452163 = p \cdot (10^9 + 146) + 1$$

By luck, with $b = 2$, $2^{N-1} = 1 \bmod N$, while $\gcd(2^{(N-1)/p} - 1, N) = 1$ Therefore, N is *certified* prime.

Now let p be the certified prime 1032003247672452163. The first strong pseudoprime N above $10^{17} \cdot p$ of the form $p \cdot U + 1$ is

$$N = p \cdot (10^{17} + 24) + 1 = 103200324767245241068077944138851913$$

By luck, with $b = 2$, $2^{N-1} = 1 \bmod N$, while $\gcd(2^{(N-1)/p} - 1, N) = 1$. Therefore, N is *certified* prime.

Now let p be the prime just certified. The first strong pseudoprime N above $10^{34} \cdot p$ of the form $p \cdot U + 1$ is

$$p \cdot (10^{34} + 224) + 1$$

$$= 1032003247672452410680779441388542246872747862933999249459487102828513$$

Again, luckily, $b = 2$ provides a certificate that N is N is prime.

Let p be prime just certified. The first strong pseudoprime N above $10^{60} \cdot p$ of the form $p \cdot U + 1$ is (computing for about 5 seconds)

$$\begin{aligned} & p \cdot (10^{60} + 1362) + 1 \\ = \quad & 1032003247672452410680779441388542246872747862933999249460892691 2 \\ & 518428801833472215991711945402406825893161069777638214340524347 07 \end{aligned}$$

By luck, $b = 2$ works again and N is *certified* prime.

Let p be the prime just certified. The first strong pseudoprime N above $10^{120} \cdot p$ of the form $p \cdot U + 1$ is (computing a few seconds)

$$p \cdot (10^{120} + 796) + 1 =$$

$$\begin{aligned} & 103200324767245241068077944138854224687274786293399924946089269125 18 \\ & 42880183347221599171194540240682589316106977763822255527019854272118 \\ & 90190043534527962851070729889546340257087058223646693262594438839294 \\ & 0270854031583341095621154300001861505738026773 \end{aligned}$$

$b = 2$ works again and N is *certified* prime.

Listing the primes p and the values of b in the last few small paragraphs gives a *certification* that the last N is prime, since anyone who cares to do so can (in easy polynomial time, of course with the help of a machine) reproduce the (quick) computations reported above. That is, that relatively small amount of auxiliary information reduces the proof/testing of primality of N to an easy (machine-assisted) computation.

It is only mildly ironic that we use a probabilistic algorithm (Miller-Rabin) to make a good first guess at candidates for (deterministic) certification.

8 Factorization Algorithms

The claim that factorization is harder than primality testing (or certification of primality) cannot be rigorously substantiated currently. Indeed, there are few natural algorithms which can be *proven* strictly more difficult than others. And it appears to be difficult to give interesting lower bounds on the computational complexity (in the sense of run-time per input size) of factoring, despite beliefs concerning its difficulty. [Shp] illustrates some recent work in the direction of lower bounds.

As some sort of backward evidence that factoring is hard, we give a brief introduction to factoring methods.

In extreme contrast to trial division, these methods have various peculiar failure modes, as well as demanding discretion in setting parameters. Thus, if there were any doubt: *one should not use these factorization methods on an integer n without having first used Fermat or Miller-Rabin to be sure that n is definitely composite.*

8.1 Euler-Fermat trick

This observation applies only to numbers of special forms. For example, for the special form $N = b^n - 1$, a prime p divisor of N *either* divides $b^d - 1$ for some divisor $d < n$ of n, *or* $p = 1 \bmod n$. This follows immediately from the fact that if p divides N then b is an n^{th} root of unity in the finite field $\mathbb{Z}/p$. This reduces the amount of required trial division by a significant constant factor, making feasible or palatable certain historic hand calculations. Similarly, if the Fermat number $N = 2^{2^q} + 1$ were divisible by a prime p, then 2 would be a 2^q-th root of -1 modulo p, from which it would follow that $p = 1 \bmod 2^{q+1}$. This observation allowed Euler to find the factorization

$$2^{2^5} + 1 = 641 \cdot 6700417$$

with only $(641 - 1)/2^{5+1} = 10$ trial divisions rather than $(641 - 1)/2 = 320$, disproving the primality of this number as conjectured by Fermat. Indeed, this constant-factor speed-up finds a six-digit factor of $2^{2^7} + 1$ in 150 trial divisions rather than 300000. (Still, the Lucas-Lehmer tests for primality of Mersenne numbers $2^p - 1$ and Fermat numbers $2^{2^p} + 1$ are much more efficient, though they test primality without attempting to find a factor.)

8.2 Pollard's rho method

This method [Po1] quickly finds relatively small factors p of composite numbers in perhaps $\sqrt{p}$ steps, and uses very little memory. (An algorithm [Po2] similar in spirit computes discrete logarithms.) It is very simple to implement, and illustrates the surprising power of algorithms which are irremediably probabilistic. Finally, though an obvious heuristic suggests the reasonableness of Pollard's rho, it is difficult to *prove* that it works as well as it does. See [B] for a rare result in this direction.

The description (hence, implementation) is indeed simple: Given an integer N, define a function f by $f(x) = x^2 + 2 \,\%\, N$, and initialize by setting $x = 2$, $y = f(x)$.

```
    Compute $g = \gcd(x - y, N)$
    If $1 < g < N$, stop: $g$ is a proper factor of $N$
    If $g = 1$, replace $x$ by $f(x)$ and $y$ by $f(f(y))$ and repeat.
```

(If $g = N$, we have *failure*, and the algorithm needs to be reinitialized.)

Why does this work? Indeed, the additional point that it hardly matters what function f is used may be disconcerting. The most compelling heuristic explanation is not rigorous, and attempts to be more rigorous do not easily succeed. The probabilistic idea involved is the so-called *birthday paradox*, that with n draws (with replacement) from a set of N things, if $n >> \sqrt{N}$ then the probability is greater than 1/2 that there will be a duplication. Perhaps less well-known is the other critical component, the Floyd cycle-detection method, used to make exploitation of the birthday-paradox idea practical.

The Birthday Paradox computation is easy: the probability that n things drawn (with replacement) from among N will be distinct is

$$P = \left(1-\frac{1}{N}\right)\left(1-\frac{2}{N}\right)\ldots\left(1-\frac{n-1}{N}\right)$$

Taking logarithms and using $\ln(1-x) \le -x$,

$$\ln P \le \sum_{1\le i\le n-1} -\frac{i}{N} = \frac{-n(n-1)}{2N} \sim \frac{-n^2}{2N}$$

Thus, $P \le e^{-n^2/2N}$, and $e^{-n^2/2N} \le 1/2$ for

$$n \ge \sqrt{2N\ln 2} \sim \frac{6}{5}\sqrt{N} \sim \sqrt{N}$$

The heuristic for Pollard's rho is as follows. Suppose that N has a proper divisor p, much smaller than N. From the birthday paradox, if we have more than $\sqrt{p}$ integers $x_1, x_2, \ldots, x_t$, then the probability is greater than 1/2 that two of these will be the same modulo p. The idea of Pollard's rho method is that $\sqrt{p}$ is much smaller than $\sqrt{N}$, so we should expect that if we choose a 'random sequence' of integers $x_1, x_2, \ldots$, there will be two the same modulo p long before there are two the same modulo N. Thus, supposing that $x_i = x_k \bmod p$, compute $g = \gcd(x_i - x_j, N)$. This *gcd* will be a multiple of p, and will be a divisor of N *probably* strictly smaller than N itself.

It is too naive to compute the greatest common divisors $\gcd(x_i - x_j, N)$ as we go along, for *all* $i < j$, since this might take about $\sqrt{p}\cdot\sqrt{p} \sim \sqrt{N}$ comparisons, as bad as trial division.

We need a clever way to exploit the birthday paradox. Let's pretend that the function $f(x) = x^2 + 2\,\%\,n$ used in the algorithm is a 'random map' from $\mathbb{Z}/N$ to itself, and we make the random x_i's by

$$x_0 = 2 \qquad\qquad x_i = f(x_{i-1}) \quad (\text{for } i > 0)$$

Note that if ever $x_j = x_i \bmod p$ with $i < j$, then equality persists: $x_{j+\ell} = x_{i+\ell} \bmod p$ for all $\ell \ge 0$. Thus,

$$x_t = x_{t-(j-i)} = x_{t-2(j-i)} = x_{t-3(j-i)} = \ldots = x_{t-\ell(j-i)} \bmod p \qquad\qquad (\text{for } \ell \text{ not too large})$$

(Here *not too large* means that we do not go back to a point before the cycle was entered.)

Floyd's cycle-detection method is the following efficient way of looking for matches. Do not keep in memory the whole list of x_i's. Rather, just remember the last one computed, and separately compute $y_i = x_{2i}$. The efficient way to compute the sequence of y_i's is as

$$y_{i+1} = f(f(y_i))$$

At each step we only remember the last x_t and y_t computed, and consider $\gcd(x_t - y_t, N)$. Let j be the index where we repeat a value, namely, let j be the first index such that $x_j = x_i \bmod p$ for some $i < j$. Then

$$x_t = x_{t-\ell(j-i)} \bmod p$$

whenever $t - \ell(j-i) \geq i$. Thus, we have a cycle beginning at i and closing at j, but Floyd's cycle-detection trick looks for a cycle beginning at some index s and ending at $2s$. We can see that this is feasible: taking $t = 2s$,

$$y_s = x_{2s} = x_{2s-\ell(j-i)} \bmod p$$

for all s with $2s - \ell(j-i) \geq i$. So when $s = \ell(j-i)$ with $\ell(j-i) \geq i$

$$y_s = x_{2s} = x_{2s-\ell(j-i)} = x_{2s-s} = x_s \bmod p$$

for $s = 2s - \ell(j-i) \geq i$. This proves that we find x_i and x_j which are the same mod p.

The point that once we enter a *cycle* the data repeats in a simple fashion explains the name *rho method*: we could imagine travelling along a literal lower-case *rho* character, up the tail to the circle forming the body of the character, thus entering the cycle.

Especially if one feels that Pollard's rho method is simply too wacky to work, some small examples are useful. Compare to the trial division examples above. In less than 10 seconds *total* (in C++ on a 2.4 G machine)

factor	10000103	of	100001220001957	(2661 steps)
factor	100000007	of	10000013000000861	(14073 steps)
factor	1000000103	of	1000000110000000721	(9630 steps)
factor	10000000019	of	100000001220000001957	(129665 steps)
factor	100000000103	of	10000000010600000000309	(162944 steps)

Pollard's rho can easily factor integers which are well out of the range of trial division.

8.3 Pollard's $p-1$ method

This factorization method is specialized, finding prime factors p of given n with the property that $p-1$ is divisible only by 'small' factors. It is easy to misunderstand this, so it bears reflection. Certainly trial division cannot make use of such a property. Integers n divisible only by primes q below a bound B are **(B-) smooth**. Primes p such that $p-1$ is relatively smooth are called *weak*, while primes p such that $p-1$ has at least one large-ish prime factor p' are *strong*. Sometimes it is required that this prime factor p' of $p-1$ itself have the property that $p'-1$ has a large-ish prime factor p", etc. (It should be said that such considerations seem to have been swamped by other issues.)

Fix an integer B. Given an integer n, Pollard's $p-1$ algorithm finds a prime factor p of n such that $p-1$ is B-smooth, using $O(B \ln n / \ln B)$ multiplications modulo n. Keeping B small is obviously desirable. On the other hand, a too-small value of B will cause the algorithm to fail to find factors. In practice, because the value of B must be kept too small to find all possible factors, the algorithm is used 'just a little' hoping for luck or negligence on the part of an adversary. The serious question of how large we should expect the prime factors of a randomly chosen number to be is not trivial and does not have a simple answer.

As usual, let floor(x) be the largest integer $\leq x$, and ceil(x) the smallest integer $\geq x$. Given an integer n known to be composite, but **not a prime power**, and given a smoothness bound B, choose a random integer b with $2 \leq b \leq n-1$. Compute $g = \gcd(b, n)$. If $g \geq 2$, then g is a proper factor, and stop. Otherwise, let $p_1, p_2, \ldots, p_t$ be the primes less than or equal B. For $i = 1, 2, 3, \ldots, t$: let $q = p_i$, and

```
    Compute $\ell = \text{ceil}(\ln n / \ln q)$.
    Replace $b$ by $b^{q^\ell}$.
    Compute $g = \gcd(b-1, n)$.
    If $1 < g < n$: stop, $g$ is a proper factor.
    Else if $g = 1$: continue.
    Else if $g = n$: stop, failure.
```

Why does this work? Let

$$p = 1 + p_1^{e_1} \ldots p_t^{e_t}$$

be a prime factor of n such that $p-1$ is B-smooth, with some integer exponents e_i. For any integer b prime to p by Fermat's Little Theorem $b^{p-1} = 1 \bmod p$, so

$$b^{p_1^{e_1} \ldots p_t^{e_t}} = 1 \bmod p$$

The quantity

$$\ell_i = \mathrm{ceil}\,(\ln n / \ln p_i)$$

is larger than or equal e_i. Let $T = p_1^{\ell_1} \ldots p_t^{\ell_t}$. Then $p_1^{e_1} \ldots p_t^{e_t}$ divides T, so certainly $b^T = 1 \bmod p$ for any integer b prime to p. That is,

$$p | \gcd(b^T - 1, n)$$

The actual algorithm given above computes *gcd*'s more often than indicated in the last paragraph, providing some opportunities to avoid $\gcd(b^T - 1, n) = n$.

Example: Factor 54541557732143. Initialize $b = 3$. The exponent for 2 is 46, the exponent for 3 is 29, and the exponent for 5 is 20. Replace b by

$$b^{2^{47}} \,\%\, 54541557732143 = 7359375584408$$

Since $\gcd(b-1, n) = 1$, continue. Replace b by

$$b^{3^{29}} \,\%\, 54541557732143 = 8632659376632$$

Since $\gcd(b-1, n) = 1$, continue. Replace b by

$$b^{5^{20}} \,\%\, 54541557732143 = 28295865457806$$

This time,

$$\gcd(n, b-1) = \gcd(54541557732143, 2268486536233 - 1) = 54001$$

Thus, we find the proper factor 54001 of 54541557732143. The prime 54001 is $\{2, 3, 5\}$-weak, meaning that $54001 - 1$ is $\{2, 3, 5\}$-smooth.

Incidentally, the format of the example shows how to use an indefinitely large bound B for the smoothness of $p-1$ for factors p of n. Then the issue becomes deciding when to quit in the case that one has found no weak prime factors.

There is an analogous $p+1$ method due to Williams [Wil]. In fact, for any cyclotomic polynomial

$$\varphi_n(x) = \frac{x^n - 1}{\prod_{d|n,\ d<n} \varphi_d(x)}$$

there is a $\varphi_n(p)$-method, with Pollard's being the case $n = 1$, and Williams' the case $n = 2$. However, for larger n these seem not to have immediate utility for factoring.

8.4 Toward the quadratic sieve

We look first at a *generic random squares* algorithm, then the *continued fractions* refinement of this, and then a different refinement, the quadratic sieve. A *random squares* or *Fermat* factoring method earns the latter name since such factorization algorithms can be broadly viwed as a natural outgrowth of a more special method used by Fermat in case an integer n is a product $n = ab$ of two integers relatively close to $\sqrt{n}$. Such algorithms use much more memory than Pollard's algorithms, and, indeed, this is a chief bottleneck in scaling upward.

A **factor base** is a chosen initial set of primes

$$B = \{p_1, \ldots, p_t\}$$

(with $p_1 = 2$, $p_2 = 3$, etc.) Choose integers a_1, a_2, ..., and let $b_i = a_i^2 \,\%\, n$. We hope/require that every b_i be **smooth** with respect to the factor basis B (that is, all the prime factors of b_i are in B). Then find a subset of the b_is whose product is a perfect square in $\mathbb{Z}$, as follows. Write the factorizations

$$b_i = \prod_{j=1}^{t} p_j^{e_{ij}}$$

Suppose that we have $t+1$ of these a_i's with $b_i = a_i^2 \,\%\, n$ being B-smooth. Let v_i be the vector (in $\mathbb{F}_2^{t+1}$) of the exponents mod 2:

$$v_i = (e_{i1} \,\%\, 2, e_{i2} \,\%\, 2, e_{i3} \,\%\, 2, \ldots, e_{it} \,\%\, 2)$$

Since there are $t+1$ of these vectors in a t-dimensional vector space, they are linearly dependent over $\mathbb{F}_2$. That is, there are $c_1, \ldots, c_{t+1}$ in $\mathbb{F}_2$ such that

$$c_1 v_1 + \cdots + c_{t+1} v_{t+1} = (0, \ldots, 0) \in \mathbb{F}_2^t$$

Use *Gaussian elimination* to find such a relation. Then

$$\prod_{i=1}^{t+1} b_i^{c_i} = \prod_{i=1}^{t+1} (\prod_{j=1}^{t} p_j^{e_{ij}})^{c_i} = \prod_{j=1}^{t} \prod_{i=1}^{t+1} p_j^{c_i e_{ij}} = \prod_{j=1}^{t} p_j^{\Sigma_i c_i e_{ij}}$$

has *even* exponents $\sum_i c_i e_{ij}$, so is the square of an integer.

Take

$$x = \prod_{i=1}^{t+1} a_i^{c_i} \,\%\, n \qquad y = \prod_{j=1}^{t} p_j^{(\Sigma_i c_i e_{ij})/2} \,\%\, n$$

Then

$$x^2 = y^2 \bmod n$$

and a chance that

$$1 < \gcd(x-y, n) < n$$

thereby possibly obtaining a proper factor of n. If by mischance $x = \pm y \bmod n$, then we won't get a proper factor. In that case, compute one or more new values $b_i = a_i^2 \,\%\, n$ and repeat the Gaussian elimination step just above.

The merits of such a random squares algorithm hinge upon two things: adroit determination of the factor base $B = \{p_1, \ldots, p_t\}$, and choice of the a_i's so that $b_i = a_i^2 \,\%\, n$ factors into primes lying in the factor base. One approach, the basic **random squares** approach, also called **Dixon's algorithm**, chooses the a_i 'randomly', and uses **trial division** by primes in B to see whether or not $b_i = a_i^2 \,\%\, n$ is B-smooth. This requires $O(t)$ trial divisions for each b_i. If b_i is not B-smooth, reject a_i and choose another. In practice, 'random' selection often means testing of successive a_i's in a specified interval.

Example: very small for the sake of human accessibility to details. Let $n = 10019$. We will try to select a_i's just above the square root of n, which has integer part $m = 101$. With any factor base smaller than $B = \{2, 3, 5, 7, 11, 13\}$ we do not find enough smooth $b_i = a_i^2 \,\%\, n$ below $2m$. This gives some idea of suitable choice of factor base, since needing to test more than a small fraction of the candidate a_i's between m and $2m$ amounts to a failure. We do find 6 B-smooth b_i's, whose vector of exponents for the primes $2, 3, 5, 7, 11, 13$ we display

$101^2 \,\%\, 10019 =$	182	with exponents	1	0	0	1	0	1
$102^2 \,\%\, 10019 =$	385	with exponents	0	0	1	1	1	0
$107^2 \,\%\, 10019 =$	1430	with exponents	1	0	1	0	1	1
$113^2 \,\%\, 10019 =$	2750	with exponents	1	0	3	0	1	0
$137^2 \,\%\, 10019 =$	8750	with exponents	1	0	4	1	0	0
$142^2 \,\%\, 10019 =$	126	with exponents	1	2	0	1	0	0

The first 3 rows sum to 0 mod 2, so we take

$$x = a_1 \cdot a_2 \cdot a_3 = 101 \cdot 102 \cdot 107 = 1102314$$

and

$$\begin{aligned} y &= 2^{(1+1+0)/2} \cdot 3^{(0+0+0)/2} \cdot 5^{(0+1+1)/2} \cdot 7^{(1+1+0)/2} \cdot 11^{(0+1+1)/2} \cdot 13^{(1+0+1)/2} \\ &= 10010 \end{aligned}$$

Then we find a proper factor of 10019 via Euclid

$$\gcd(10019, x - y) = 233$$

Improvement: If instead of reducing $b_i = a_i^2$ modulo n into the range $0 \le b_i < n$ we reduce into the range $\frac{-n}{2} < b_i < \frac{n}{2}$ (and add -1 to the factor base), then we obviously improve the chances that b_i is B-smooth. We will not use this for the moment, but will have an additional motivation to incorporate this improvement in the continued fractions algorithm (below).

Improvement: If we only choose the a_i's in the range $m < a_i < \sqrt{2}m$ then

$$b_i = a_i^2 \,\%\, n = a_i^2 - n$$

Then for a prime p dividing b_i, we have $n = a_i^2 \bmod p$, so n is a square modulo p. The point is that then we can drop primes p with quadratic symbol $(n/p)_2 = -1$ from our factor base. By quadratic reciprocity (and Dirichlet's theorem on primes in an arithmetic progression), this holds for roughly half the primes. That is, given n, after doing an easy preliminary computation of quadratic symbols, for the same computational load we have twice as large a factor basis. Of course, one would repeatedly discover that primes p for which $(n/p)_2 = -1$ do not appear *at all* in factorizations of $b_i = a_i^2 - n$.

Example: The integer $n = 1000001$ is a non-square modulo primes 3, 11, 13, 17, 19, 23. Thus, not surprisingly, only 2 a_i's in the range $m < a_i < \sqrt{2}m$ give $b_i = a_i^2 - n$ which are smooth with respect to the large-ish factor base $\{2, 3, 5, 7, 11, 13, 17, 19, 23\}$. That is, the most naive random square factorization seems to malfunction. But, in the improved form, taking into account the quadratic symbol condition, we realize that the first 4 primes we should include in a factor base are $\{2, 5, 7, 29\}$. And then, with *this* factor base rather than the full list of primes up to that point,

$$\begin{array}{rrlcccc} 1001^2 \,\%\, 1000001 = & 2000 & \text{with exponents} & 4 & 3 & 0 & 0 \\ 1082^2 \,\%\, 1000001 = & 170723 & \text{with exponents} & 0 & 0 & 1 & 3 \\ 1151^2 \,\%\, 1000001 = & 324800 & \text{with exponents} & 6 & 2 & 1 & 1 \\ 1249^2 \,\%\, 1000001 = & 560000 & \text{with exponents} & 7 & 4 & 1 & 0 \end{array}$$

The second and third vectors sum to 0 modulo 2, so we take

$$x = a_2 \cdot a_3 = 1082 \cdot 1151 = 1245382$$

and

$$y = 2^{(0+6)/2} \cdot 5^{(0+2)/2} \cdot 7^{(1+1)/2} \cdot 29^{(3+1)/2} = 235480$$

and via Euclid we find the proper factor

$$\gcd(1000001, x - y) = 9901$$

The **quadratic sieve** of Pomerance [Pom] is the best current method to choose the a_i's in a classical random squares factoring algorithm, and for a while was the best factoring algorithm of any sort. It is apparently subexponential. (The number field sieve uses a more sophisticated mechanism than the random squares factoring discussed here, and is faster for sufficiently large inputs.) The quadratic sieve is the most clever known method to choose the a_i's in a random squares algorithm, and effectively entirely avoids *factoring*, but is still memory-intensive. This makes the operation of the algorithm even less human-accessible than the continued fraction algorithm.

Much as in our version of the basic random squares algorithm, given n to be factored, we will choose the a_i's in a particular range so that we have a fixed polynomial expression for the absolute reduction of a_i^2 modulo n. For example, let $m = \text{ceil}\,(\sqrt{n})$, and choose a_i's only in the range

$$m < a_i < m + \ell$$

with $L << (\sqrt{2} - 1)m$, so that

$$b_i = a_i^2 \,\%\, n = a_i^2 - n$$

Thus, for a prime p dividing b_i, n is a square modulo p. As already exploited in the random squares case, this allows us to drop any prime p from the factor base if $(n/p)_2 = -1$, effectively doubling the size of the factor base for the same computational load.

Before the sieving step, for given n one precomputes some square roots modulo prime powers p^e for the primes p in the factor base B, as follows. Using the square root algorithm modulo primes (4.17), together with Hensel's lemma (4.18), one computes the two square roots $\pm r_p$ of n modulo $p^{e(p)}$ for the largest exponent $e(p)$ (depending upon p) such that at least one of $a = \pm r_p$ falls in the range $m < a < m + \ell$.

Now the **sieving:** first observe the elementary fact that if $a = r_p \bmod p^f$ for some integer $f \le e(p)$ then

$$a^2 - n = 0 \bmod p^f$$

For a in the chosen range $m < a < m + \ell$, initialize a list $\{L(a) : a\}$ of values by

$$L(a) = \ln(a^2 - n) \qquad (\text{for } m < a < m + \ell)$$

Here and in the following all logarithms should be computed in double or triple precision floating-point. The sieving is the following.

```
For each $p$ in the factor base $B$:
    For $i = e(p), e(p) - 1, \ldots, 1$:
        For $m < a < m + \ell$ with $a = \pm r_p \bmod p^i$,
            subtract $\ln p$ from $L(a)$
```

After this take as candidates a_i's the elements $m < a < m + \ell$ with the (modified) value $L(a)$ suitably close to 0 or negative as the most likely to give B-smooth $b_i = a_i^2 - n$. Only *now* do we factor these b_i's and set up a matrix of exponents for the primes in B as before.

Remark: The point of this approach is to reduce the amount of factorization done, since for larger integers factorization is onerous, far more so than the algorithms to compute square roots, and to sieve, in that range. This sieving, much as Eratosthenes' sieve, is even more memory-intensive than the continued fractions algorithm, and manipulation of the large matrix of exponents is a bottleneck in scaling upward. On the other hand, in a certain range it appears that the trade-off of not needing to factor while keeping track of more partial results is advantageous.

Remark: In factoring genuinely large numbers n, it appears that the largest prime in a factor base should be on the order of

$$e^{\sqrt{\ln n \;\ln\ln n}}$$

and the length ℓ of the interval of candidates above m should be in the range

$$e^{\sqrt{\ln n \;\ln\ln n}} < \ell < e^{2\sqrt{\ln n \;\ln\ln n}}$$

We use an example to illustrate further details. Hoping to achieve a degree of human accessibility, not surprisingly we are forced to use an unnaturally small n.

Example: To attempt to factor $n = 21311$, initially take a factor base $\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29\}$. Dropping odd primes p with $(n/p)_2 = -1$ leaves us with $B = \{2, 5, 11, 13, 23, 29\}$. Compute

$$m = \text{floor}(\sqrt{n}) = 145$$

To be sure to have $a^2 \,\% = a^2 - n$, we only consider candidates a below $\sqrt{2}m \sim 205$, for example, a in the range

$$145 \ < \ a \ < \ 201$$

Now we generate the data used to sieve. Since $n = 3 \bmod 4$, the only power of 2 that can ever divide $a^2 - n$ is 2^1, for odd a. Powers of 5 are more interesting. Suppressing the computation of square roots, we find that $a = 169$ is the only integer in the indicated range satisfying

$$a^2 - n = 0 \bmod 5^3$$

Going up 5^2 from this, we see that 194 is a square root mod 5^2. The only other two square roots of n mod only 5^2 fitting into the range are 156 and 181, differing by multiples of 25 from the square root 81 of n modulo 5^2. The remaining square roots mod only 5^1 fall into two families mod 5

149	154	159	164	174	179	184	189	199	204
146	151	161	166	171	176	186	191	196	201

The square roots of n modulo 11^3 are 933 and 398, neither of which can be moved into the indicated range modulo 11^3. Modulo 11^2 the only square root is 156 in that range. Modulo only 11 we have two batches (distinguished mod 11), $167, 178, 189, 200$, and $152, 163, 174, 185, 196$. The only square roots modulo 13^2 are 37 and 132, neither of which can be adjusted by multiples of 169 to be in the indicated range, so we only have candidates which are square roots of n modulo 13 and no higher power of 13, namely $154, 158, 167, 171, 180, 184, 193, 197$. Modulo 23^2 there are again no square roots of n in the indicated range, but $155, 167, 178, 190, 201$ are the square roots modulo 23 in that range. Modulo 29 there are $150, 169, 179, 198$ and none modulo 29^2.

Since there are 6 primes in the factor base, we want at least 7 linear relations mod 2 among the exponents in factorizations. If any primes arise that are not already in the factor base, we'll need still more relations.

Of the candidates in the range $146 \le a \le 200$, we sieve out (to keep) those a such that

$$(a^2 - n)/(\text{powers of 2, 5, 11, 13, 23, 29}) < 50$$

setting the thresh-hold of 50 somewhat at random. (For readability, we don't take logarithms.) The following table lists the candidates $a = 146, \ldots, 200$ with prime powers from our factor base dividing $a^2 - n$, boxing the leftover if that leftover is under 50.

146	147	148	149	150	151	152	153	154	155	156
–	2	–	2	–	2	–	2	–	2	–
5	–	–	5	–	5	–	–	5	–	5^2
–	–	–	–	–	–	11	–	–	–	11^2
–	–	–	–	–	–	–	–	13	–	–
–	–	–	–	–	–	–	–	–	23	–
–	–	–	–	29	–	–	–	–	–	–
$\boxed{1}$	149	593	89	$\boxed{41}$	149	163	1049	$\boxed{37}$	59	$\boxed{1}$

157	158	159	160	161	162	163	164	165	166	167
2	–	2	–	2	–	2	–	2	–	2
–	–	5	–	5	–	–	5	–	5	–
–	–	–	–	–	–	11	–	–	–	11
–	13	–	–	–	–	–	–	–	–	13
–	–	–	–	–	–	–	–	–	–	23
–	–	–	–	–	–	–	–	–	–	–
1669	281	397	4289	461	4933	239	1117	2957	1249	$\boxed{1}$

168	169	170	171	172	173	174	175	176	177	178
–	2	–	2	–	2	–	2	–	2	–
–	5^3	–	5	–	–	5	–	5	–	–
–	–	–	–	–	–	11	–	–	–	11
–	–	–	13	–	–	–	–	–	–	–
–	–	–	–	–	–	–	–	–	–	23
–	29	–	–	–	–	–	–	–	–	–
6913	$\boxed{1}$	7589	61	8273	4309	163	4657	1933	5009	$\boxed{41}$

179	180	181	182	183	184	185	186	187	188	189
2	–	2	–	2	–	2	–	2	–	2
5	–	5^2	–	–	5	–	5	–	–	5
–	–	–	–	–	–	11	–	–	–	11
–	13	–	–	–	13	–	–	–	–	–
–	–	–	–	–	–	–	–	–	–	–
29	–	–	–	–	–	–	–	–	–	–
$\boxed{37}$	853	229	11813	6089	193	587	2657	6829	14033	131

190	191	192	193	194	195	196	197	198	199	200
–	2	–	2	–	2	–	2	–	2	–
–	5	–	–	5^2	–	5	–	–	5	–
–	–	–	–	–	–	11	–	–	–	11
–	–	–	13	–	–	–	13	–	–	–
23	–	–	–	–	–	–	–	–	–	–
–	–	–	–	–	–	–	–	29	–	–
643	1517	15553	613	653	8357	311	673	617	1829	1699

We have 8 outcomes below 50, but new primes 37 and 41 have appeared. Of some interest is the fact that the prime 31 did not appear at all. We can enlarge the factor base to include 37 and then discover that the candidate $a = 191$ gives a further relation $191^2 - n = 2 \cdot 5 \cdot 37 \cdot 41$. Thus, with 8 primes appearing, we have the requisite 9 relations

$$\begin{array}{rcl} 146^2 - n &=& 5 \\ 150^2 - n &=& 29 \cdot 41 \\ 154^2 - n &=& 5 \cdot 13 \cdot 37 \\ 156^2 - n &=& 5^2 \cdot 11^2 \\ 167^2 - n &=& 2 \cdot 11 \cdot 13 \cdot 23 \\ 169^2 - n &=& 2 \cdot 5^3 \cdot 29 \\ 178^2 - n &=& 11 \cdot 23 \cdot 41 \\ 179^2 - n &=& 2 \cdot 5 \cdot 29 \cdot 37 \\ 191^2 - n &=& 2 \cdot 5 \cdot 37 \cdot 41 \end{array}$$

Actually, by luck $156^2 - n$ is a square, and

$$\gcd(156 - 5 \cdot 11, n) = 101$$

so we have a proper factor without carrying out the Gaussian elimination. From this point, the algorithm proceeds exactly as the generic random squares and the continued fractions algorithms.

Elliptic curve algorithms and the **number field sieve** are more sophisticated, although implementation is possible without discussion of the underlying ideas. The elliptic curve algorithm is roughly a competitor for the quadratic sieve, while the number field sieve is faster at the top end of the range of integers factorable by any (generic) factoring method. The number field sieve has its origins in a *deterministic* (superpolynomial) algorithm of Adleman, Pomerance, and Rumely [APR] from 1983.

9 Randomness, Information, Entropy

While it is possible to formalize various statistical notions of 'randomness' of a string of 0's and 1's, the cryptographic sense is much stronger, specifically demanding *unpredictability* of the next bit given all the previous bits, by an adversary with substantial computing power. Statistical randomness merely means that the aggregate has properties within statistically plausible bounds of what one would obtain if the 0's and 1's had come from a genuinely random source. In effect, an allegedly random sequence might be *rejected* on statistical grounds due to presence of a simple pattern, but *absence* of a *simple* pattern does not usefully address predictability. Instead, the usual proof of unpredictability asserts something akin to 'if the adversary can *consistently* predict the next bit, then the adversary can factor large integers'. The Blum-Blum-Shub [BBS] and Naor-Reingold [NR] pseudo-random number generators both are of this sort.

Keys to both symmetric and asymmetric ciphers must be *random*, otherwise (in effect) the keyspace is greatly reduced in size, making brute force searches for the key feasible. Trying common words and variations as sources of keys is a **dictionary attack**. Reasonably good password checking programs carry out such attacks to test the quality of users' proposed passwords before accepting them. Several cautionary tales exist wherein the inadequacy of using process id's (PIDs) and current time as seeds for pseudo-random number generators (pRNGs). The SSL (Secure Sockets Layer) implementation in the Netscape browser at one point suffered from a fatally inadequate random number generator.

A naive notion of randomness, e.g, of a 1024-bit integer is that it be the result of choosing from among the first 2^{1024} integers with equal probabilities attached. However, this attempted answer begs the question, in practical terms, since it does not indicate how the 'choice' is to be made. Further, there is an intuitive inadequacy in this naive conception, since (as illustrated already in the discussion of Kolmogorov complexity) bit-strings

$$10101010101010101010$$

$$11100100011011111001$$

which are equally likely if we are choosing random 0's and 1's are not equally *random* to our eyes: the first is highly patterned, and the second seems not so. Again, see [LV] for an engaging dialectic developing the notion of *random*.

9.1 Entropy

The notion of **entropy** in this context was introduced decisively in Shannon's fundamental papers [Sh1], [Sh2] on *information theory*. These papers showed that, in analogy with the thermodynamical use of the term, the term makes sense in application to communication channels and abstractions thereof. (Hartley had used a similar idea earlier, but had been less persuasive in demonstrating its innate relevance to questions of communications.) Shannon's theorems bear upon data compression, information rate of channels and devices, and secrecy issues. The entropy $H(\Omega)$ of a finite probability space Ω is

$$H(\Omega) = \sum_{\omega\in\Omega} -P(\omega)\cdot \ln_2\, P(\omega)$$

where $P()$ is probability. Similarly, the entropy $H(X)$ of a random variable X with finitely-many values is

$$H(X) = \sum_{\text{values } x \text{ of } X} -P(X=x)\cdot \ln_2\, P(X=x)$$

The base-2 normalization of logarithms means that the entropy of a fair coin is 1, justifying taking *bits* as the unit of entropy. We are to think of *entropy* as a synonym for *uncertainty*. One may readily verify properties such as that $H(p_1,\ldots,p_n)$ is maximum when $p_1 = \ldots = p_n = \frac{1}{n}$: the most uncertainty is when all possibilities are equally likely. Also readily verifiable is

$$H(\underbrace{\frac{1}{n},\ldots,\frac{1}{n}}_{n}) \le H(\underbrace{\frac{1}{n+1},\ldots,\frac{1}{n+1}}_{n+1})$$

That is, a larger ensemble of equally likely possibilities is more uncertain than a smaller ensemble.

In this context, a **source** is a probability space whose elements are the (source) **alphabet**. The **entropy of a source** is the entropy of that probability space. For example, English may be naively modeled as the probability space

$$\Omega = \{e, t, o, a, n, i, r, s, \ldots\}$$

where $P(e) \approx 0.11$. $P(t) \approx .9$, etc., where the supposed probabilities are the approximate frequencies with which characters appear.

A naive but useful model of English makes the (naive) assumption that the next letter in a sequence is independent of the previous letter (and the probabilities do not change). This sort of hypothesis on the output of a source is the **i.i.d.** (independent, identically distributed) hypothesis. By contrast, a **Markov** source is one in which the next output of the source depends (in a uniform manner) on the previous output. A **hidden Markov** source is one in which the next output of the source depends (in a uniform manner) on not only the previous output, but on a *state* which may not be visible to an observer.

9.2 Shannon's Noiseless Coding

This is the basic result about **compression**. In a simple form, if the possible single outputs of an i.i.d. *source* are the elements of a finite probability space Ω, to be encoded as strings of 0's and 1's so as to minimize the expected length of the binary codewords,

$$H(\Omega) \le \text{(expected word length with optimal encoding)} \le H(\Omega) + 1$$

where $H(\Omega)$ is entropy, as above. (Analogues of this result hold for certain more complicated sources, as well.) This illustrates the aptness of the notion of entropy, since it is the answer to a fundamental question. Further, given a *fixed* source alphabet, construction of an optimal encoding is straightforward (Huffman encoding). Compression is also called *source coding.* See [G2] or [Ro], for example.

A far subtler aspect of serious compression schemes is, in effect, exactly the choice of an advantageous source alphabet, given a *real* source. E.g., see [Sn].

In this vein, we have a reasonable heuristic test of 'randomness' of a string of characters: if any one of the standard compression utilities (*gzip*, *zip*, *bzip2*, etc.) can compress the string, then the string is assuredly *not* random. Indeed, in successfully choosing a source alphabet with skewed probabilities, the algorithm has detected a pattern. Of course, the repertoire of patterns found in such a manner is limited.

Another basic communications notion is that of a **channel**, meant to mirror aspects of real-world communications channels, and also to abstract critical notions. The simplest interesting channel model is the **binary symmetric channel**, which accepts as inputs sequences of 0's and 1's, flipping each one to the other with a fixed (*bit error*) probability p, independent of previous flips-or-not. To successfully transmit information through such a channel the information must incorporate some sort of **redundancy**. The simplest version of redundancy is a *repetition code*: for example, to communicate a bit, one might send that bit through the channel 3 times, and have recipient take a *vote* of the 3 received bits: if a majority of the received bits are 1's, that is how the message is interpreted, while if a majority are 0's, it is inferred that the message was 0. It is a straightforward and elementary exercise to compute that the probability of *incorrectly* decoding a single bit is

$$\text{3-fold repetition code probability of incorrectible error} = 3p^2(1-p) + p^3$$

For small p, the probability of incorrectible error with the threefold repetition code is about $3p^2$, which is much smaller than the probability p of a single bit being transmitted correctly. Thus, to have probability of no error at least $1 - \epsilon$ without any redundancy scheme allows transmission of not more than

$$\frac{\ln(1-\varepsilon)}{\ln 1 - p} \sim \frac{\varepsilon}{p} \text{ bits}$$

while with three-fold repetition we can transmit as many as

$$\frac{\ln(1-\varepsilon)}{\ln 1 - (3p^2(1-p) + p^3)} \sim \frac{\varepsilon}{3p^2} \text{ bits}$$

Thus, for example, for p small, use of three-fold redundancy increases transmission length before failure by a factor of about $1/3p$.

But this three-fold repetition code uses 3 times the space and time of the original message itself. That is, the **information rate** of the three-fold repetition code is 1/3. It is not hard to imagine that repetition codes are not the most efficient choices available. Encodings with the intent to be robust against channel-induced errors are *error-correcting codes*, or *forward error-correction* schemes (as opposed to error *detection* schemes which must ask for retransmission upon detection of an error). See [Rn], [G2], for example, or the updated encyclopedic classic [MS]. Construction of error-correcting codes is much harder than construction of compression algorithms.

9.3 Shannon's Noisy Coding Theorem

This provides our second illustration of the aptness of the notion of entropy. This result is a surprisingly optimistic assertion about the creation of good error-correcting codes. (Roughly) define the **channel capacity** C of a binary symmetric channel (as above) with bit error rate p to be the upper bound information rates achievable by clever error correction encoding while simultaneously demanding that the probability of uncorrectible errors can be made as nearly 0 as one wishes. Surprisingly, Shannon's **Noisy Coding** theorem proves that

$$\begin{aligned}\text{capacity of channel with bit error rate } p &= 1 - H(\{p, 1-p\}) \\ &= 1 - \left(p \ln_2 \frac{1}{p} + (1-p) \ln_2 \frac{1}{(1-p)} \right)\end{aligned}$$

where $H(\{p, 1-p\})$ is the entropy of the two-element probability space with probabilities p and $1-p$. (The orthodox treatment of this would be to *define* the channel capacity to be the entropy, and then prove that this definition is 'correct'.) Shannon's proof in effect showed that the *average* code achieves that level of performance. Thus, one would tend to say that *random* codes are good. Paradoxically, it appears to be difficult to systematically choose these good random codes.

The fact that channel capacity is not zero while assuring nearly error-free transmission of information is striking. At the same time, the well-definedness of channel capacity as a definite limit on the transmission of correct information is almost equally striking. And, again, more than merely being a heuristic imitation of thermodynamical ideas, Shannon's two theorems show that entropy has a provable significance.

Information theory consists of issues surrounding compression and error correction. The Shannon theorems and successive developments are often implicitly understood as justification for thinking about cryptography in information-theoretic terms, at least as a heuristic, even though it appears to be difficult to give substance to the intuitively appealing notion that a cipher *increases entropy.* More plausible is the notion that limits on hiding information can be objectively measured.

9.4 Statistical randomness

This is quantifiable, although seems sharply insufficient to assure adequate cryptographic randomness. A statistically random sequence of 0's and 1's will pass various *low-level* tests. We will briefly survey *empirical* tests, meaning that one ignores the means of producing the pseudo-random bits, but simply measures various features of samples from the sequence.

First, Pearson's χ^2 test (e.g., see [Kn] section 3.3) gives potential quantitative sense to the intuitively plausible fact that sampling of the outcome of a genuinely random variable is skewed to various degrees. That is, in 100 flips of a fair coin one ought not rely upon an outcome of 50 heads and 50 tails, but, rather, an outcome merely *close* to that. The sense of *close* is quantified by the χ^2 test, though setting of thresh-holds still is a matter of subjective judgement. Again, see [Kn] for discussion of proper use of the χ^2 test in this context. Roughly, the χ^2 test reports the probability that a random variable with alleged probabilities would produce a given sample. Thus, apart from setting thresh-holds, all the following tests can be made quantitative via the χ^2 test, that is, the use of the term *roughly* below can be converted to something numerical, though we will not do so.

Samples of necessary but not sufficient statistical requirements on sequences of bits are: **Equidistribution:** The number of 0's should be roughly the number of 1's in a sample (but of course not *exactly*). **Serial test:** The four possible pairs of successive bits, 00, 01, 10, and 11, should be roughly equidistributed. **Poker test:** The various possibilities of (e.g.) 5 consecutive bits, namely, all the same, one 0 and four 1's, etc., with their respective probabilities, should appear roughly as expected. **Coupon collector's test:** This test chooses one or more unusual patterns (such as 11111) and looks at the length of subsequences necessary to see such a pattern. Failure of any of these tests is a fairly sharp denial of randomness.

In the same vein, there are heuristic methods to eliminate statistical biases from a given sequence of bits. Such procedures, or at least their intended effects, are known as **distillation** or **purification** of entropy or randomness. There is an obvious hazard in the very notion. For example, given a sequence of bits, to 'distill' a subsequence with roughly the same number of 0's and 1's, one might explicitly drop any 11's or 00's. But, of course, this produces the not-random $\ldots 101010 \ldots$. As this trivial example illustrates, generally there is a tendency for too-naive distillation attempts to repair one bias at the expense of another. It seems fair to say that the problem of distillation of randomness or entropy is not simple, especially if the goal is the cryptographic one of unpredictability, since the distillation algorithm itself must be deterministic.

Again, statistical randomness is not cryptographic randomness, so the above samples of conditions are *necessary*, not *sufficient* conditions on a pRNG.

9.5 Deterministic pRNGs

As noted earlier, the existence of the Massey-Berlekamp decisively shows that LFSRs and LCGs are not useful as cryptographic pRNGs, even though such a pRNG can be arranged so it their state does not repeat for a very long time. A long period is a necessary, but not sufficient, condition for a good pRNG.

The **Blum-Blum-Shub** pRNG has provable properties, and is easy to describe, but is a bit inefficient, and still needs high quality random numbers to generate the data to construct the generator in the first place. The set-up is choice of large random primes p and q, and random seed s_o. Successive secret internal states are computed by

$$s_{n+1} = s_n^2 \,\%\, pq$$

and public output bits are produced by

$$b_n = s_n \,\%\, 2$$

One can prove (e.g., see also [St]) that being able to predict successive output bits consistently better than half the time allows one to factor pq. But note that perhaps 1000 genuinely random bits are needed to *set up* a secure BBS pRNG, so it would be silly to imagine using BBS to generate any smaller number of bits. Again, with insufficiently random set-up data, an attacker does not try to break the pRNG itself, but, instead, tries to guess the initial data (a dictionary attack). For example, with $p = 103$, $q = 211$, $s_0 = 13$, the first 120 bits generated are

$$\begin{array}{cccccccccccccccccccc}
1&1&0&1&1&1&1&1&1&0&0&1&1&1&1&0&0&1&1&0\\
0&1&0&0&1&1&0&1&1&1&1&1&1&0&0&1&1&1&1&0\\
0&1&1&0&0&1&0&0&1&1&0&1&1&1&1&1&1&0&0&1\\
1&1&1&0&0&1&1&0&0&1&0&0&1&1&0&1&1&1&1&1\\
1&0&0&1&1&1&1&0&0&1&1&0&0&1&0&0&1&1&0&1\\
1&1&1&1&1&0&0&1&1&1&1&0&0&1&1&0&0&1&0&0
\end{array}$$

As dubious as our powers of assessment of randomness may be, there seems to be no obvious elementary pattern. And, again, one should note the fact that the data p, q, and s_0 needed to be chosen.

The Naor-Reingold generator [NR] is of a similar nature, refining these ideas. As with the Blum–Blum–Shub algorithm, it is provably secure *assuming* the infeasibility of factoring integers $N = p \cdot q$, where p and q are large primes. Its set-up is a little more complicated. Fix a size n, and choose two random n-bit primes p and q, both equal to 3 mod 4, and put $N = pq$. Choose a random integer g which is a square mod N. Let

$$a = (a_{1,0}, a_{1,1}, a_{2,0}, a_{2,1}, \ldots, a_{n,0}, a_{n,1})$$

be a random sequence of $2n$ values in $\{1, 2, \ldots, N\}$. Let $r = (r_1, \ldots, r_n)$ be a random sequence of 0's and 1's. For an integer t of at most $2n$ bits, let $b_{2n}(t)$ be the vector of binary expansion coefficient of t, padded as necessary on the left to have length $2n$. For two such binary vectors

$$v = (v_1, \ldots, v_n), \qquad w = (w_1, \ldots, w_n)$$

with v_i's and w_i's all 0's or 1's, define

$$v \cdot w = (v_1, \ldots, v_n) \cdot (w_1, \ldots, w_n) = \Big(\sum_{1 \le i \le n} v_i w_i \Big) \,\%\, 2$$

Then for any n-tuple $x = (x_1, \ldots, x_n)$ of 0's and 1's, define a $\{0,1\}$-valued function

$$f(x) = f_{N,g,a,r}(x) = r \cdot \mathfrak{b}(g^{a_{1,x_1}+a_{2,x_2}+a_{3,x_3}+\cdots+a_{n,x_n}} \,\%\, N)$$

Then, assuming that it is infeasible to factor N, the output of the function $f = f_{N,g,a,r}$ is indistinguishable from random bits.

9.6 Genuine randomness

Repeatedly flipping a (fair) coin to generate successive 0's and 1's yields (so far as we know) 'genuinely random' bits. The problem is that the process is slow. The decay of radioactive isotopes is genuinely random, so far as we know, but perhaps inconvenient, though see [Fo]. Atmospheric noise [RNG] and lava lamps are plausible generators.

Several commercial and open-source products exist which in conjunction with a desktop or laptop computer produce 'random' numbers by timing keyboard or mouse or other activities presumably influenced by irregular external events. For example, on most recent unix/linux systems the devices '/dev/random' and '/dev/urandom' (created by T. Ts'o) collect events, the former producing output whenever sufficient 'entropy' has been collected, while the latter produces output with whatever is available. The manpage for *urandom* contains platform-specific information about these parts of the kernel. For example, in Perl on a typical Linux platform,

```
#!/usr/bin/perl
open R, "/dev/random";
read R, $random, 1;
close R;
print unpack("H*", $random);
```

prints a plausibly random byte in hexadecimal. Typically, commercial products try to remove as many statistical biases as possible from bit streams produced by the environment, intending to leave us with 'genuinely random' bits.

Notes

Kahn [Ka] and Singh [Si] are historical discussions of cryptography. Grittier practical aspects with an exhaustive bibliography are in Schneier [Sch1]. To understand practical realities, one must also read Schneier's [Sch2]. A high-level engineering-oriented formal text paying attention to specifics of NIST standards is Stinson [St]. More mathematical is the Handbook [MOV] by Menezes, van Oorschot, and Vanstone, giving many algorithms in readable pseudo-code, with bibliographic commentaries, but not including more sophisticated items such as elliptic curve techniques. The *Handbook* provides no proofs, neither of underlying mathematical facts (some nonstandard), nor of correctness of algorithms, nor of runtime estimates. [Ga] is a representative of an older time. The discussion [DR] of the Advanced Encryption Standard, Rijndael, by its authors, Daemen and Rijmen, is informative about design issues. Despite being somewhat faster and more flexible than the other AES candidates, it was a surprise to this author that Rijndael, the most mathematically structured of the candidates, should have won. In this context, one should not overlook the new attack [FM] on Rijndael and similar ciphers. Koblitz' [Ko2] goes beyond elliptic curve methods to show how to use jacobians of certain hyperelliptic curves.

Bibliography

[A] L.M. Adleman, *The function field sieve*, Algorithmic Number Theory, Lecture Notes in Computer Science **877** (1994), 108–121.

[APR] L.M. Adleman, C. Pomerance, R. Rumely, *On distinguishing prime numbers from composite numbers*, Annals of Mathematics **117** (1983), 173-206.

[AES] *NIST AES homepage*, http://csrc.nist.gov/CryptoToolkit/aes/

[AGP] W. Alford, A. Granville, C. Pomerance, *There are infinitely-many Carmichael numbers*, Annals of Mathematics **140** (1994), 703–722.

[AKS] M. Agrawal, N. Kayal, N. Saxena, *PRIMES is in P*, (preprint 2002)
http://www.cse.iitk.ac.in/new/primality.html

[AAG] I. Anshel, M. Anshel, D. Goldfeld, *An algebraic method for public-key cryptography*, Mathematics Research Letters **6** (1999), 1–5.

[B] E. Bach, *Toward a theory of Pollard's rho method*, Information and Computation **90** (1991), 139–155.

[BS] E. Bach, J. Shallit, *Algorithmic Number Theory*, MIT Press (1996),

[Ba] W.D. Banks, *Towards faster cryptosystems, II, (this volume)*

[Ba] F.L. Bauer, *Decrypted Secrets: Methods and Max-ims of Cryptology*, Springer-Verlag (2000),

[Be] J. Bell, *Speakable and Unspeakable in Quantum Mechanics*, Cambridge Univ. Press (1993),

[Ber] D. Bernstein, *http://www.cse.iitk.ac.in/new/primality.html*,

[BS1] E. Biham, A. Shamir, *Differential cryptanalysis of DES-like cryptosystems*, Journal of Cryptology **4** (1991), 3–72.

[BS2] E. Biham, A. Shamir, *Differential cryptanalysis of the Data Encryption Standard*, Springer-Verlag (1993),

[BS3] E. Biham, A. Shamir, *Differential cryptanalysis of the full 16-round DES*, Lecture Notes in Computer Science **740** (1993), 494–502.

[BKL] J. Birman, K. Ko, S. Lee, *A new approach to the word and conjugacy problems in the braid groups*, Advances in Mathematics **139** (1998), 322–353.

[BSS] I.F. Blake, G. Seroussi, N.P. Smart , *Elliptic curves in cryptography*, London Math. Soc. Lecture Note Series **265** (2000),

[BBS] L. Blum, M. Blum, M. Shub, *A simple unpredictable random number generator*, SIAM Journal on Computing **15** (1986), 364–383.

[Bo] D. Boneh, *Twenty years of attacks on the RSA cryptosystem*, Notices of the AMS **46** (1999), 203–213.

[BDF] D. Boneh, G. Durfee, Y. Frankel, *An attack on RSA given a fraction of the private key bits*, Advances in Cryptology, AsiaCrypt '98, Lecture Notes in Computer Science **1514** (1998), 25–34.

[Bor] F. Bornemann, *PRIMES is in P: a breakthrough for 'Everyman'*, Notices of the A.M.S. **50** (2003), 545–552.

[Br] S. Brands, *Untraceable off-line cash in wallets with observers.*, Advances in Cryptology, Crypto '93, Lecture Notes in Computer Science **773** (1994), 302–318.

[BLP] J. Buhler, H.W. Lenstra, C. Pomerance, *Factoring integers with the number fields sieve*, Development of the Number Field Sieve (ed. A.K. Lenstra and H.W. Lenstra), Lecture Notes in Mathematics **1554** (1993),

[CS] A.R. Calderbank, P. Shor, *Good quantum error-correcting codes exist*, Phys. Rev. A **54** (1996), 1098–1105.

[Ch] D. Chaum, *Security without identification: transactions systems to make Big Brother obsolete*, Communications of the ACM **28** (1985), 1030–1044.

[CFN] D. Chaum, A. Fiat, M. Naor, *Untraceable electronic cash*, Advances in Cryptology, Crypto '88, Springer-Verlag (1990), 319–327.

[C] H. Cohen, *A Course in Computational Number Theory*, Springer-Verlag (1993),

[Co1] D. Coppersmith, *Modifications to the number field sieve*, Journal of Cryptology **6** (1993), 169–180.

[Co2] D. Coppersmith, *Small solutions to polynomial equations, and low exponent RSA*, vulnerabilities J. Cryptology **10** (1997), 233–594.

[CLR] T.H. Cormen, C.E. Leiserson, R.L. Rivest, *Introduction to Algorithms*, MIT Press (1990),

[CP] R. Crandall, C. Pomerance, *Prime Numbers, a Computational Perspective*, Springer-Verlag (2001),

[DR] J. Daemen, V. Rijmen, *The design of Rijndael,. AES - The Advanced Encryption Standard*, Springer-Verlag (2002),

[DH] W. Diffie, M.E. Hellman, *New directions in cryptography*, IEEE Transactions on Information Theory **22** (1976), 644–654.

[EPR] A. Einstein, B. Podolsky, N. Rosen, *Can quantum-mechanical description of physical reality be*, considred complete? Phys. Rev. **47** (1935), 469-473.

[EFF] Electronic Frontier Foundation, *Cracking DES*, O'Reilly and Associates (1998),

[E] T. ElGamal, *A public key cryptosystem and signature scheme based on discrete logarithms*, IEEE Transactions on Information Theory **IT-31** (1985), 469–473.

[Fo] J. Walker, *http://www.fourmilab.ch/hotbits/*,

[FM] J. Fuller, William Millan, *On Linear Redundancy in the AES S-Box*, (preprint, 2002)

[Ga] H.F. Gaines, *Cryptanalysis, A Study of Ciphers and Their Solution*, Dover (1939, 1956),

[G] P.B. Garrett, *Making and Breaking Codes: An Introduction to Cryptography*, Prentice-Hall (2001),

[G2] P.B. Garrett, *The Mathematics of Coding Theory: Information, Compression*, Error-Correction, and Finite Fields Prentice-Hall (2004),

[GL] P.B. Garrett, D. Lieman (editors), *Public-Key Cryptography (Baltimore 2003)*, Proc. Symp. Applied Math. vol. 62, AMS, 2005.

[Gr1] O. Goldreich, *Modern Cryptography, Probabilistic Proofs, and Pseudorandomness*, Springer-Verlag (1999),

[Gr2] O. Goldreich, *Foundations of Cryptography: Basic Tools*, Cambridge University Press (2001),

[Go] S.W. Golomb, *Shift Register Sequences*, Holden-Day, San Francisco (1967),

[HPS] J. Hoffstein, J. Pipher, J. Silverman, *NTRU: A new high-speed public-key cryptosystem*, Crypto '96 rump session

[HG] Nick Howgrave-Graham, *Public-key cryptology and proofs of security, (this volume)*

[HT] J. Hughes, A. Tannenbaum, *Length-based attacks for certain group-based encryption*, rewriting systems (preprint) (2000),

[Ka] D. Kahn, *The Codebreakers*, Scribner (1996),

[KLMT] E. Knill, R. Laflamme, R. Martinez, C.-H. Tseng, *An algorithmic benchmark for quantum information processing*, Nature **404** (2000), 368–370.

[Kn] D. Knuth, *The Art of Computer Programming*, Addison-Wesley **2** (1989),

[Ko1] N. Koblitz, *A Course in Number Theory and Cryptography*, Springer-Verlag (1994),

[Ko2] N. Koblitz, *Algebraic Aspects of Cryptography*, Springer-Verlag (1998),

[Ko3] N. Koblitz, *Elliptic curve cryptosystems*, Mathematics of Computation **48** (1987), 203–209.

[KMV] N. Koblitz, A. Menezes, S. Vanstone, *The state of elliptic curve cryptography*, Designs, Codes, and Cryptography **19** (2000), 173–193.

[Ko] P.C. Kocher, *Timing attacks on implementations of Diffie-Hellman, RSA, DSS*, and other systems Advances in Cryptology, Crytpo '96, Lecture Notes in Computer Science **1109** (1996), 104–113.

[L1] S. Landau, *Standing the test of time: the data encryption standard*, Notices of the AMS **47** (2000), 341–349.

[L2] S. Landau, *Communication security for the twenty-first century: the advanced encryption standard*, Notices of the AMS **47** (2000), 450–459.

[L3] S. Landau, *[review of ten books on crypto]*, Bulletin of the AMS **41** (2004), 357–367.

[LL] A.K. Lenstra, H.W. Lenstra, Jr. (eds.), *The Development of the Number Field Sieve*, Lecture Notes in Mathematics **1554** (1993),

[LLL] A.K. Lenstra, H.W. Lenstra, L. Lovasz , *Factoring polynomials with polynomial coefficients*, Math. Annalen **261** (1982), 515–534.

[Lie] D. Lieman, *Cryptography in the Real World Today, (this volume)*

[LV] M. Li, P. Vitányi, *An Introduction to Kolmogorov Complexity and Its Applications*, Springer-Verlag (1997),

[MS] F.J. MacWilliams, N.J.A. Sloane, *The Theory of Error-Correcting Codes*, North-Holland (1998),

[Ma] J.L. Massey, *Shift-Register Synthesis and BCH Decoding*, IEEE Trans. on Information Theory **IT-155** (1969), 122–127.

[Mat1] M. Matsui, *Linear cryptanalysis method for DES cipher*, Lecture Notes in Computer Science **765** (1994), 386–397.

[Mat2] M. Matsui, *The first experimental cryptanalysis of the data encryption standard*, Lecture Notes in Computer Science **839** (1994), 1–11.

[MOV] A.J. Menezes, P.C. van Oorschot, S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press (1997),

[MH] R. Merkle, M. Hellman, *Hiding information and signatures in trapdoor knapsacks*, IEEE Trans. on Information Theory **IT-24** (1978), 525–530.

[M] G.L. Miller, *Riemann's hypothesis and tests for primality*, Journal of Computer and Systems Science **13**

(1976), 300–317.

[MB] M.A. Morrison , J. Brillhart, *A method of factoring and the factorization of F_7*, Math. Comp. **29** (1975), 183-205.

[NR] M. Naor, O. Reingold, *Synthesizers and their application to the parallel construction of pseudo-random functions*, Proceedings 36th IEEE Symposium on Foundations of COmputer Science (1995), 170–181.

[O] A.M. Odlyzko, *Discrete logarithms: the past and the future*, Designs, Codes, and Cryptography **19** (2000), 129–145.

[OO] T. Okamoto, K. Ohta, *Universal electronic cash*, Springer Lecture Notes in Computer Science **435** (1992), 324–337.

[PB] A. Pati, S. Braunstein, *Impossibility of deleting and unknown quantum state*, Nature **404** (2000), 164–165.

[P1] J.M. Pollard, *A Monte Carlo method for factorization*, BIT **15** (1975), 331–334.

[P2] J.M. Pollard, *Monte Carlo methods for index computation (mod p)*, Mathematics of Computation **32** (1978), 918–924.

[Pom] C. Pomerance, *Analysis and comparison of some integer factoring algorithms*, Computational Methods in Number Theory, ed. H.W. Lenstra, Jr., R. Tijdeman, Math. Centrum, Amsterdam (1982), 89–139.

[Ra] M.O. Rabin, *Probabilistic algorithms for testing primality*, Journal of Number Theory **12** (1980), 128–138.

[RNG] *http://www.random.org/*,

[RSA] R.L. Rivest, A. Shamir, L. Adleman, *A method for obtaining digital signatures and public key cryptosystems*, Communications of the ACM **21** (1978), 120–126.

[Ro] S. Roman, *Coding and information theory*, Springer-Verlag **GTM 134** (1992),

[Sa] A. Salomaa, *Public-Key Cryptography*, Springer-Verlag (1996),

[Sn] D. Salomon, *Data Compression: the Complete Reference*, Springer-Verlag (1997),

[Sch1] B. Schneier, *Applied Cryptography, Protocols, Algorithms, and Source Code in C*, John Wiley and Sons (1995),

[Sch2] B. Schneier, *Secrets and Lies*, John Wiley and Sons (2000),

[ST] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, N. Ferguson, *The Twofish Encryption Algorithm*, John Wiley and Sons (1999),

[SE] C. Schnorr, M. Euchner, *Lattic basis reduction: improved practical algorithms and solving subset sum problems*, Mathematical Programming **66** (1994), 181–194.

[Sh] A. Shamir, *A polynomial-time algorithm for breaking the Merkle-Hellman cryptosystem*, Proc. 23d FOCS Symposium (1982), 145–152.

[Sh1] C.E. Shannon, *A mathematical theory of communication*, Bell Systems Technical Journal **27** (1948), 379–423, 623–656.

[Sh2] C.E. Shannon, *Communication theory of secrecy systems*, Bell Systems Technical Journal **28** (1949), 656–715.

[Sho1] P. Shor, *Algorithms for quantum computation: discrete logarithms and factoring*, Proceedings of the Thirty-Fifth Annual Symposium on the Foundations of Computer Science (1994), 124–134.

[Sho2] P. Shor, *Polynomial-time algorithms for prime factorization and discrete*, logarithms on a quantum computer SIAM Review **41** (1999), 303–332.

[Shou] V. Shoup, *Lower bounds for discrete logarithms and related problems*, Lecture Notes in Computer Science **1233** (1997), 256–266.

[Shp] I. Shparlinski, *Number Theoretic Methods in Cryptography, Complexity Lower Bounds*, Birkhauser-Verlag (1999),

[Shp2] I. Shparlinski, *Playing 'Hide-and-seek' with numbers: the hidden number*, problems, lattices, and exponential sums Birkhauser-Verlag (1999),

[Sil1] J.H. Silverman, *The arithmetic of elliptic curves*, Graduate Texts in Math., Springer-Verlag **106** (1986),

[Sil2] J.H. Silverman, *Advanced topics in the arithmetic of elliptic curves*, Graduate Texts in Math., Springer-Verlag **151** (1994),

[Sil3] J.H. Silverman, *Elliptic curves and cryptography*, *(this volume)*

[Si] S. Singh, *The Code Book*, Doubleday (1999),

[So] R. Solovay, V. Strassen, *A fast Monte Carlo test for primality*, SIAM Journal on Computing **6** (1977), 84–85.

[St] D. Stinson, *Cryptography: Theory and Practice*, CRC Press (2002),

[TW] W. Trappe, L. Washington, *Introduction to Cryptography with Coding Theory*, Prentice-Hall (2002),

[WM] N. Wagner, M. Magyarik, *A public key cryptosystem based on the word problem*, Lecture Notes in Computer Science **196** (1985), 19–36.

[Wag] S.S. Wagstaff, *Cryptanalysis of Number Theoretic Ciphers*, Chapman-Hall/CRC (2003),

[Wash] L. Washington, *Elliptic Curves: Number Theory and Cryptography*, Chapman-Hall/CRC (2003),

[Wh] W. Whyte, *Towards faster cryptosystems, I*, *(this volume)*

[Wie] M.J. Wiener, *Cryptanalysis of short RSA exponenets*, IEEE Trans. on Information Theory **36** (1990), 553–558.

[Wil] H.C. Williams, *A $p+1$ method of factoring*, Math. Comp. **39** (1982), 225-234.