certificate, in the public-key database, contains a lot more than his public key. It contains information about Bob—his name, address, and so on—and it is signed by someone Alice trusts: Trent (usually known as a **certification authority**, or CA). By signing both the key and the information about Bob, Trent certifies that the information about Bob is correct and that the public key belongs to Bob. Alice checks Trent's signature and then uses the public key, secure in the knowledge that it is Bob's and no one else's. Certificates play an important role in a number of public-key protocols such as PEM [825] (see Section 24.10) and X.509 [304] (see Section 24.9).

A complicated noncryptographic issue surrounds this type of system. What is the meaning of certification? Or, to put it another way, who is trusted to issue certificates to whom? Anyone may sign anyone else's certificate, but there needs to be some way to filter out questionable certificates: for example, certificates for employees of one company signed by the CA of another company. Normally, a certification chain transfers trust: A single trusted entity certifies trusted agents, trusted agents certify company CAs, and company CAs certify their employees.

Here are some more things to think about:

- What level of trust in someone's identity is implied by his certificate?
- What are the relationships between a person and the CA that certified his public key, and how can those relationships be implied by the certificate?
- Who can be trusted to be the "single trusted entity" at the top of the certification chain?
- How long can a certification chain be?

Ideally, Bob would follow some kind of authentication procedure before the CA signs his certificate. Additionally, some kind of timestamp or an indication of the certificate's validity period is important to guard against compromised keys [461].

Timestamping is not enough. Keys may be invalidated before they have expired, either through compromise or for administrative reasons. Hence, it is important the CA keep a list of invalid certificates, and for users to regularly check that list. This key revocation problem is still a difficult one to solve.

And one public-key/private-key pair is not enough. Certainly any good implementation of public-key cryptography needs separate keys for encryption and digital signatures. This separation allows for different security levels, expiration times, backup procedures, and so on. Someone might sign messages with a 2048-bit key stored on a smart card and good for twenty years, while they might use a 768-bit key stored in the computer and good for six months for encryption.

And a single pair of encryption and signature keys isn't enough, either. A private key authenticates a relationship as well as an identity, and people have more than one relationship. Alice might want to sign one document as Alice the individual, another as Alice, vice-president of Monolith, Inc., and a third as Alice, president of her community organization. Some of these keys are more valuable than others, so they can be better protected. Alice might have to store a backup of her work key

with the company's security officer; she doesn't want the company to have a copy of the key she signed her mortgage with. Just as Alice has multiple physical keys in her pocket, she is going to have multiple cryptographic keys.

Distributed Key Management

In some situations, this sort of centralized key management will not work. Perhaps there is no CA whom Alice and Bob both trust. Perhaps Alice and Bob trust only their friends. Perhaps Alice and Bob trust no one.

Distributed key management, used in PGP (see Section 24.12), solves this problem with **introducers**. Introducers are other users of the system who sign their friends' public keys. For example, when Bob generates his public key, he gives copies to his friends: Carol and Dave. They know Bob, so they each sign Bob's key and give Bob a copy of the signature. Now, when Bob presents his key to a stranger, Alice, he presents it with the signatures of these two introducers. If Alice also knows and trusts Carol, she has reason to believe that Bob's key is valid. If she knows and trusts Carol and Dave a little, she has reason to believe that Bob's key is valid. If she doesn't know either Carol or Dave, she has no reason to trust Bob's key.

Over time, Bob will collect many more introducers. If Alice and Bob travel in similar circles, the odds are good that Alice will know one of Bob's introducers. To prevent against Mallory's substituting one key for another, an introducer must be sure that Bob's key belongs to Bob before he signs it. Perhaps the introducer should require the key be given face-to-face or verified over the telephone.

The benefit of this mechanism is that there is no CA that everyone has to trust. The down side is that when Alice receives Bob's public key, she has no guarantee that she will know any of the introducers and therefore no guarantee that she will trust the validity of the key.

PART III

CRYPTOGRAPHIC ALGORITHMS

CHAPTER 11

Mathematical Background

11.1 INFORMATION THEORY

Modern information theory was first published in 1948 by Claude Elmwood Shannon [1431,1432]. (His papers have been reprinted by the IEEE Press [1433].) For a good mathematical treatment of the topic, consult [593]. In this section, I will just sketch some important ideas.

Entropy and Uncertainty

Information theory defines the **amount of information** in a message as the minimum number of bits needed to encode all possible meanings of that message, assuming all messages are equally likely. For example, the day-of-the-week field in a database contains no more than 3 bits of information, because the information can be encoded with 3 bits:

```
000 = Sunday
001 = Monday
010 = Tuesday
011 = Wednesday
100 = Thursday
101 = Friday
110 = Saturday
111 is unused
```

If this information were represented by corresponding ASCII character strings, it would take up more memory space but would not contain any more information. Similarly, the "sex" field of a database contains only 1 bit of information, even though it might be stored as one of two 6-byte ASCII strings: "MALE" or "FEMALE."

Formally, the amount of information in a message M is measured by the **entropy** of a message, denoted by $H(M)$. The entropy of a message indicating sex is 1 bit; the entropy of a message indicating the day of the week is slightly less than 3 bits. In

general, the entropy of a message measured in bits is $\log_2 n$, in which n is the number of possible meanings. This assumes that each meaning is equally likely.

The entropy of a message also measures its **uncertainty**. This is the number of plaintext bits needed to be recovered when the message is scrambled in ciphertext in order to learn the plaintext. For example, if the ciphertext block "QHP*5M" is either "MALE" or "FEMALE," then the uncertainty of the message is 1. A cryptanalyst has to learn only one well-chosen bit to recover the message.

Rate of a Language

For a given language, the **rate of the language** is

$$r = H(M)/N$$

in which N is the length of the message. The rate of normal English takes various values between 1.0 bits/letter and 1.5 bits/letter, for large values of N. Shannon, in [1434], said that the entropy depends on the length of the text. Specifically he indicated a rate of 2.3 bits/letter for 8-letter chunks, but the rate drops to between 1.3 and 1.5 for 16-letter chunks. Thomas Cover used a gambling estimating technique and found an entropy of 1.3 bits/character [386]. (I'll use 1.3 in this book.) The **absolute rate** of a language is the maximum number of bits that can be coded in each character, assuming each character sequence is equally likely. If there are L characters in a language, the absolute rate is:

$$R = \log_2 L$$

This is the maximum entropy of the individual characters.

For English, with 26 letters, the absolute rate is $\log_2 26$, or about 4.7 bits/letter. It should come as no surprise to anyone that the actual rate of English is much less than the absolute rate; natural language is highly redundant.

The **redundancy** of a language, denoted D, is defined by:

$$D = R - r$$

Given that the rate of English is 1.3, the redundancy is 3.4 bits/letter. This means that each English character carries 3.4 bits of redundant information.

An ASCII message that is nothing more than printed English has 1.3 bits of information per byte of message. This means it has 6.7 bits of redundant information, giving it an overall redundancy of 0.84 bits of information per bit of ASCII text, and an entropy of 0.16 bits of information per bit of ASCII text. The same message in BAUDOT, at 5 bits per character, has a redundancy of 0.74 bits per bit and an entropy of 0.26 bits per bit. Spacing, punctuation, numbers, and formatting modify these results.

Security of a Cryptosystem

Shannon defined a precise mathematical model of what it means for a cryptosystem to be secure. The goal of a cryptanalyst is to determine the key K, the plaintext P, or both. However, he may be satisfied with some probabilistic information about P: whether it is digitized audio, German text, spreadsheet data, or something else.

In most real-world cryptanalysis, the cryptanalyst has some probabilistic information about P before he even starts. He probably knows the language of the plaintext. This language has a certain redundancy associated with it. If it is a message to Bob, it probably begins with "Dear Bob." Certainly "Dear Bob" is more probable than "e8T&g [,m." The purpose of cryptanalysis is to modify the probabilities associated with each possible plaintext. Eventually one plaintext will emerge from the pile of possible plaintexts as certain (or at least, very probable).

There is such a thing as a cryptosystem that achieves **perfect secrecy**: a cryptosystem in which the ciphertext yields no possible information about the plaintext (except possibly its length). Shannon theorized that it is only possible if the number of possible keys is at least as large as the number of possible messages. In other words, the key must be at least as long as the message itself, and no key can be reused. In still other words, the one-time pad (see Section 1.5) is the only cryptosystem that achieves perfect secrecy.

Perfect secrecy aside, the ciphertext unavoidably yields some information about the corresponding plaintext. A good cryptographic algorithm keeps this information to a minimum; a good cryptanalyst exploits this information to determine the plaintext.

Cryptanalysts use the natural redundancy of language to reduce the number of possible plaintexts. The more redundant the language, the easier it is to cryptanalyze. This is the reason that many real-world cryptographic implementations use a compression program to reduce the size of the text before encrypting it. Compression reduces the redundancy of a message as well as the work required to encrypt and decrypt.

The entropy of a cryptosystem is a measure of the size of the keyspace, K. It is approximated by the base two logarithm of the number of keys:

$$H(K) = \log_2 K$$

A cryptosystem with a 64-bit key has an entropy of 64 bits; a cryptosystem with a 56-bit key has an entropy of 56 bits. In general, the greater the entropy, the harder it is to break a cryptosystem.

Unicity Distance

For a message of length n, the number of different keys that will decipher a ciphertext message to some intelligible plaintext in the same language as the original plaintext (such as an English text string) is given by the following formula [712,95]:

$$2^{H(K) - nD} - 1$$

Shannon [1432] defined the **unicity distance**, U, also called the unicity point, as an approximation of the amount of ciphertext such that the sum of the real information (entropy) in the corresponding plaintext plus the entropy of the encryption key equals the number of ciphertext bits used. He then went on to show that ciphertexts longer than this distance are reasonably certain to have only one meaningful decryption. Ciphertexts significantly shorter than this are likely to have multiple, equally valid decryptions and therefore gain security from the opponent's difficulty in choosing the correct one.

For most symmetric cryptosystems, the unicity distance is defined as the entropy of the cryptosystem divided by the redundancy of the language.

$$U = H(K)/D$$

Unicity distance does not make deterministic predictions, but gives probabilistic results. Unicity distance estimates the minimum amount of ciphertext for which it is likely that there is only a single intelligible plaintext decryption when a brute-force attack is attempted. Generally, the longer the unicity distance, the better the cryptosystem. For DES, with a 56-bit key, and an ASCII English message, the unicity distance is about 8.2 ASCII characters or 66 bits. Table 11.1 gives the unicity distances for varying key lengths. The unicity distances for some classical cryptosystems are found in [445].

Unicity distance is not a measure of how much ciphertext is required for cryptanalysis, but how much ciphertext is required for there to be only one reasonable solution for cryptanalysis. A cryptosystem may be computationally infeasible to break even if it is theoretically possible to break it with a small amount of ciphertext. (The largely esoteric theory of relativized cryptography is relevant here [230,231,232,233,234,235].) The unicity distance is inversely proportional to the redundancy. As redundancy approaches zero, even a trivial cipher can be unbreakable with a ciphertext-only attack.

Shannon defined a cryptosystem whose unicity distance is infinite as one that has **ideal secrecy**. Note that an ideal cryptosystem is not necessarily a perfect cryptosystem, although a perfect cryptosystem would necessarily be an ideal cryptosystem. If a cryptosystem has ideal secrecy, even successful cryptanalysis will leave some uncertainty about whether the recovered plaintext is the real plaintext.

Information Theory in Practice

While these concepts have great theoretical value, actual cryptanalysis seldom proceeds along these lines. Unicity distance guarantees insecurity if it's too small but does not guarantee security if it's high. Few practical algorithms are absolutely impervious to analysis; all manner of characteristics might serve as entering wedges

Table 11.1
Unicity Distances of ASCII Text Encrypted with Algorithms with Varying Key Lengths

Key Length (in bits)	Unicity Distance (in characters)
40	5.9
56	8.2
64	9.4
80	11.8
128	18.8
256	37.6

to crack some encrypted messages. However, similar information theory considerations are occasionally useful, for example, to determine a recommended key change interval for a particular algorithm. Cryptanalysts also employ a variety of statistical and information theory tests to help guide the analysis in the most promising directions. Unfortunately, most literature on applying information theory to cryptanalysis remains classified, including the seminal 1940 work of Alan Turing.

Confusion and Diffusion

The two basic techniques for obscuring the redundancies in a plaintext message are, according to Shannon, confusion and diffusion [1432].

Confusion obscures the relationship between the plaintext and the ciphertext. This frustrates attempts to study the ciphertext looking for redundancies and statistical patterns. The easiest way to do this is through substitution. A simple substitution cipher, like the Caesar Cipher, is one in which every identical letter of plaintext is substituted for a single letter of ciphertext. Modern substitution ciphers are more complex: A long block of plaintext is substituted for a different block of ciphertext, and the mechanics of the substitution change with each bit in the plaintext or key. This type of substitution is not necessarily enough; the German Enigma is a complex substitution algorithm that was broken during World War II.

Diffusion dissipates the redundancy of the plaintext by spreading it out over the ciphertext. A cryptanalyst looking for those redundancies will have a harder time finding them. The simplest way to cause diffusion is through transposition (also called **permutation**). A simple transposition cipher, like columnar transposition, simply rearranges the letters of the plaintext. Modern ciphers do this type of permutation, but they also employ other forms of diffusion that can diffuse parts of the message throughout the entire message.

Stream ciphers rely on confusion alone, although some feedback schemes add diffusion. Block algorithms use both confusion and diffusion. As a general rule, diffusion alone is easily cracked (although double transposition ciphers hold up better than many other pencil-and-paper systems).

11.2 Complexity Theory

Complexity theory provides a methodology for analyzing the **computational complexity** of different cryptographic techniques and algorithms. It compares cryptographic algorithms and techniques and determines their security. Information theory tells us that all cryptographic algorithms (except one-time pads) can be broken. Complexity theory tells us whether they can be broken before the heat death of the universe.

Complexity of Algorithms

An algorithm's complexity is determined by the computational power needed to execute it. The computational complexity of an algorithm is often measured by two variables: T (for **time complexity**) and S (for **space complexity**, or memory require-

ment). Both T and S are commonly expressed as functions of n, where n is the size of the input. (There are other measures of complexity: the number of random bits, the communications bandwidth, the amount of data, and so on.)

Generally, the computational complexity of an algorithm is expressed in what is called "big O" notation: the order of magnitude of the computational complexity. It's just the term of the complexity function which grows the fastest as n gets larger; all lower-order terms are ignored. For example, if the time complexity of a given algorithm is $4n^2 + 7n + 12$, then the computational complexity is on the order of n^2, expressed $O(n^2)$.

Measuring time complexity this way is system-independent. You don't have to know the exact timings of various instructions or the number of bits used to represent different variables or even the speed of the processor. One computer might be 50 percent faster than another and a third might have a data path twice as wide, but the order-of-magnitude complexity of an algorithm remains the same. This isn't cheating; when you're dealing with algorithms as complex as the ones presented here, the other stuff is negligible (is a constant factor) compared to the order-of-magnitude complexity.

This notation allows you to see how the input size affects the time and space requirements. For example, if $T = O(n)$, then doubling the input size doubles the running time of the algorithm. If $T = O(2^n)$, then adding one bit to the input size doubles the running time of the algorithm (within a constant factor).

Generally, algorithms are classified according to their time or space complexities. An algorithm is **constant** if its complexity is independent of n: $O(1)$. An algorithm is **linear**, if its time complexity is $O(n)$. Algorithms can also be **quadratic**, **cubic**, and so on. All these algorithms are **polynomial**; their complexity is $O(n^m)$, when m is a constant. The class of algorithms that have a polynomial time complexity are called **polynomial-time** algorithms.

Algorithms whose complexities are $O(t^{f(n)})$, where t is a constant greater than 1 and $f(n)$ is some polynomial function of n, are called **exponential**. The subset of exponential algorithms whose complexities are $O(c^{f(n)})$, where c is a constant and $f(n)$ is more than constant but less than linear, is called **superpolynomial**.

Ideally, a cryptographer would like to be able to say that the best algorithm to break this encryption algorithm is of exponential-time complexity. In practice, the strongest statements that can be made, given the current state of the art of computational complexity theory, are of the form "all known cracking algorithms for this cryptosystem are of superpolynomial-time complexity." That is, the cracking algorithms that we know are of superpolynomial-time complexity, but it is not yet possible to prove that no polynomial-time cracking algorithm could ever be discovered. Advances in computational complexity may some day make it possible to design algorithms for which the existence of polynomial-time cracking algorithms can be ruled out with mathematical certainty.

As n grows, the time complexity of an algorithm can make an enormous difference in whether the algorithm is practical. Table 11.2 shows the running times for different algorithm classes in which n equals one million. The table ignores constants, but also shows why ignoring constants is reasonable.

Table 11.2
Running Times of Different Classes of Algorithms

Class	Complexity	# of Operations for $n = 10^6$	Time at 10^6 O/S
Constant	$O(1)$	1	1 μsec.
Linear	$O(n)$	10^6	1 sec.
Quadratic	$O(n^2)$	10^{12}	11.6 days
Cubic	$O(n^3)$	10^{18}	32,000 yrs.
Exponential	$O(2^n)$	$10^{301,030}$	$10^{301,006}$ times the age of the universe

Assuming that the unit of "time" for our computer is a microsecond, the computer can complete a constant algorithm in a microsecond, a linear algorithm in a second, and a quadratic algorithm in 11.6 days. It would take 32,000 years to complete a cubic algorithm; not terribly practical, but a computer built to withstand the next ice age would deliver a solution eventually. Performing the exponential algorithm is futile, no matter how well you extrapolate computing power, parallel processing, or contact with superintelligent aliens.

Look at the problem of a brute-force attack against an encryption algorithm. The time complexity of this attack is proportional to the number of possible keys, which is an exponential function of the key length. If n is the length of the key, then the complexity of a brute-force attack is $O(2^n)$. Section 12.3 discusses the controversy surrounding a 56-bit key for DES instead of a 112-bit key. The complexity of a brute-force attack against a 56-bit key is 2^{56}; against a 112-bit key the complexity is 2^{112}. The former is possible; the latter isn't.

Complexity of Problems

Complexity theory also classifies the inherent complexity of problems, not just the complexity of particular algorithms used to solve problems. (Excellent introductions to this topic are [600,211,1226]; see also [1096,27,739].) The theory looks at the minimum time and space required to solve the hardest instance of a problem on a theoretical computer known as a **Turing machine**. A Turing machine is a finite-state machine with an infinite read-write memory tape. It turns out that a Turing machine is a realistic model of computation.

Problems that can be solved with polynomial-time algorithms are called **tractable**, because they can usually be solved in a reasonable amount of time for reasonable-sized inputs. (The exact definition of "reasonable" depends on the circumstance.) Problems that cannot be solved in polynomial time are called **intractable**, because calculating their solution quickly becomes infeasible. Intractable problems are sometimes just called **hard**. Problems that can only be solved with algorithms that are superpolynomial are computationally intractable, even for relatively small values of n.

It gets worse. Alan Turing proved that some problems are **undecidable**. It is impossible to devise any algorithm to solve them, regardless of the algorithm's time complexity.

Problems can be divided into complexity classes, which depend on the complexity of their solutions. Figure 11.1 shows the more important complexity classes and their presumed relationships. (Unfortunately, not much about this material has been proved mathematically.)

On the bottom, the class **P** consists of all problems that can be solved in polynomial time. The class **NP** consists of all problems that can be solved in polynomial time only on a nondeterministic Turing machine: a variant of a normal Turing machine that can make guesses. The machine guesses the solution to the problem—either by making "lucky guesses" or by trying all guesses in parallel—and checks its guess in polynomial time.

NP's relevance to cryptography is this: Many symmetric algorithms and all public-key algorithms can be cracked in nondeterministic polynomial time. Given a ciphertext C, the cryptanalyst simply guesses a plaintext, X, and a key, k, and in polynomial time runs the encryption algorithm on inputs X and k and checks whether the result is equal to C. This is important theoretically, because it puts an upper bound on the complexity of cryptanalysis for these algorithms. In practice, of course, it is a deterministic polynomial-time algorithm that the cryptanalyst seeks. Furthermore, this argument is not applicable to all classes of ciphers; in particular, it is not applicable to one-time pads—for any C, there are many X, k pairs that yield C when run through the encryption algorithm, but most of these Xs are nonsense, not legitimate plaintexts.

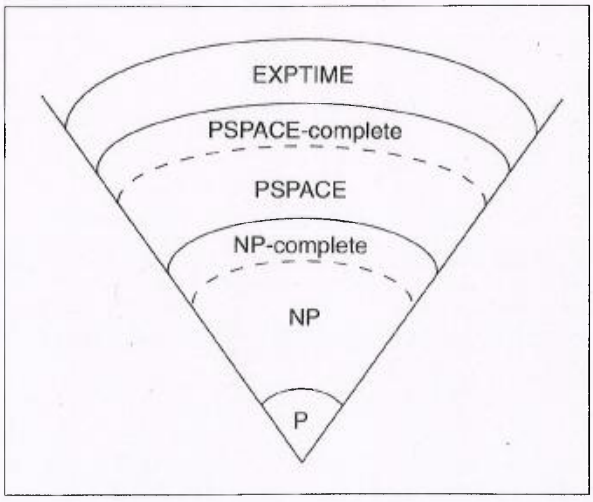

Figure 11.1 Complexity classes.

The class **NP** includes the class **P**, because any problem solvable in polynomial time on a deterministic Turing machine is also solvable in polynomial time on a nondeterministic Turing machine; the guessing stage can simply be omitted.

If all **NP** problems are solvable in polynomial time on a deterministic machine, then **P = NP**. Although it seems obvious that some **NP** problems are much harder than others (a brute-force attack against an encryption algorithm versus encrypting a random block of plaintext), it has never been proven that **P ≠ NP** (or that **P = NP**). However, most people working in complexity theory believe that they are unequal.

Stranger still, specific problems in **NP** can be proven to be as difficult as any problem in the class. Steven Cook [365] proved that the Satisfiability problem (given a propositional Boolean formula, is there a way to assign truth values to the variables that makes the formula true?) is **NP-complete**. This means that, if Satisfiability is solvable in polynomial time, then **P = NP**. Conversely, if any problem in **NP** can be proven not to have a deterministic polynomial-time algorithm, the proof will show that Satisfiability does not have a deterministic polynomial-time algorithm either. No problem is harder than Satisfiability in **NP**.

Since Cook's seminal paper was published, a huge number of problems have been shown to be equivalent to Satisfiability; hundreds are listed in [600], and some examples follow. By equivalent, I mean that these problems are also **NP-complete**; they are in **NP** and also as hard as any problem in **NP**. If their solvability in deterministic polynomial time were resolved, the **P** versus **NP** question would be solved. The question of whether **P = NP** is the central unsolved question of computational complexity theory, and no one expects it to be solved anytime soon. If someone showed that **P = NP**, then most of this book would be irrelevant: As previously explained, many classes of ciphers are trivially breakable in nondeterministic polynomial time. If **P = NP**, they are breakable by feasible, deterministic algorithms.

Further out in the complexity hierarchy is **PSPACE**. Problems in **PSPACE** can be solved in polynomial space, but not necessarily polynomial time. **PSPACE** includes **NP**, but some problems in **PSPACE** are thought to be harder than **NP**. Of course, this isn't proven either. There is a class of problems, the so-called **PSPACE-complete** problems, with the property that, if any one of them is in **NP** then **PSPACE = NP** and if any one of them is in **P** then **PSPACE = P**.

And finally, there is the class of problems called **EXPTIME**. These problems are solvable in exponential time. The **EXPTIME-complete** problems can actually be proven not to be solvable in deterministic polynomial time. It has been shown that **P** does not equal **EXPTIME**.

NP-Complete Problems

Michael Garey and David Johnson compiled a list of over 300 **NP-complete** problems [600]. Here are just a few of them:

— Traveling Salesman Problem. A traveling salesman has to visit n different cities using only one tank of gas (there is a maximum distance he can travel). Is there a route that allows him to visit each city

exactly once on that single tank of gas? (This is a generalization of the Hamiltonian Cycle problem—see Section 5.1.)

— Three-Way Marriage Problem. In a room are n men, n women, and n clergymen (priests, rabbis, whatever). There is also a list of acceptable marriages, which consists of one man, one woman, and one clergyman willing to officiate. Given this list of possible triples, is it possible to arrange n marriages such that everyone is either marrying one person or officiating at one marriage?

— Three-Satisfiability. There is a list of n logical statements, each with three variables. For example: if (x and y) then z, (x and w) or (not z), if ((not u and not x) or (z and (u or not x))) then (not z and u) or x), and so on. Is there a truth assignment for all the variables that satisfies all the statements? (This is a special case of the Satisfiability problem previously mentioned.)

11.3 NUMBER THEORY

This isn't a book on number theory, so I'm just going to sketch a few ideas that apply to cryptography. If you want a detailed mathematical text on number theory, consult one of these books: [1430,72,1171,12,959,681,742,420]. My two favorite books on the mathematics of finite fields are [971,1042]. See also [88,1157, 1158,1060].

Modular Arithmetic

You all learned modular arithmetic in school; it was called "clock arithmetic." Remember these word problems? If Mildred says she'll be home by 10:00, and she's 13 hours late, what time does she get home and for how many years does her father ground her? That's arithmetic modulo 12. Twenty-three modulo 12 equals 11.

$$(10 + 13) \bmod 12 = 23 \bmod 12 = 11 \bmod 12$$

Another way of writing this is to say that 23 and 11 are equivalent, modulo 12:

$$10 + 13 \equiv 11 \pmod{12}$$

Basically, $a \equiv b \pmod{n}$ if $a = b + kn$ for some integer k. If a is non-negative and b is between 0 and n, you can think of b as the remainder of a when divided by n. Sometimes, b is called the **residue** of a, modulo n. Sometimes a is called **congruent** to b, modulo n (the triple equals sign, $\equiv$, denotes congruence). These are just different ways of saying the same thing.

The set of integers from 0 to $n - 1$ form what is called a **complete set of residues** modulo n. This means that, for every integer a, its residue modulo n is some number from 0 to $n - 1$.

The operation $a \bmod n$ denotes the residue of a, such that the residue is some integer from 0 to $n - 1$. This operation is **modular reduction**. For example, $5 \bmod 3 = 2$.

This definition of mod may be different from the definition used in some programming languages. For example, PASCAL's modulo operator sometimes returns a

negative number. It returns a number between $-(n-1)$ and $n-1$. In C, the % operator returns the remainder from the division of the first expression by the second; this can be a negative number if either operand is negative. For all the algorithms in this book, make sure you add n to the result of the modulo operator if it returns a negative number.

Modular arithmetic is just like normal arithmetic: It's commutative, associative, and distributive. Also, reducing each intermediate result modulo n yields the same result as doing the whole calculation and then reducing the end result modulo n.

$$(a+b) \bmod n = ((a \bmod n) + (b \bmod n)) \bmod n$$
$$(a-b) \bmod n = ((a \bmod n) - (b \bmod n)) \bmod n$$
$$(a*b) \bmod n = ((a \bmod n)*(b \bmod n)) \bmod n$$
$$(a*(b+c)) \bmod n = (((a*b) \bmod n) + ((a*c) \bmod n)) \bmod n$$

Cryptography uses computation mod n a lot, because calculating discrete logarithms and square roots mod n can be hard problems. Modular arithmetic is also easier to work with on computers, because it restricts the range of all intermediate values and the result. For a k-bit modulus, n, the intermediate results of any addition, subtraction, or multiplication will not be more than $2k$-bits long. So we can perform exponentiation in modular arithmetic without generating huge intermediate results. Calculating the power of some number modulo some number,

$$a^x \bmod n,$$

is just a series of multiplications and divisions, but there are speedups. One kind of speedup aims to minimize the number of modular multiplications; another kind aims to optimize the individual modular multiplications. Because the operations are distributive, it is faster to do the exponentiation as a stream of successive multiplications, taking the modulus every time. It doesn't make much difference now, but it will when you're working with 200-bit numbers.

For example, if you want to calculate $a^8 \bmod n$, don't use the naïve approach and perform seven multiplications and one huge modular reduction:

$$(a*a*a*a*a*a*a*a) \bmod n$$

Instead, perform three smaller multiplications and three smaller modular reductions:

$$((a^2 \bmod n)^2 \bmod n)^2 \bmod n$$

By the same token,

$$a^{16} \bmod n = (((a^2 \bmod n)^2 \bmod n)^2 \bmod n)^2 \bmod n$$

Computing $a^x \bmod n$, where x is not a power of 2, is only slightly harder. Binary notation expresses x as a sum of powers of 2: 25 is 11001 in binary, so $25 = 2^4 + 2^3 + 2^0$. So

$$a^{25} \bmod n = (a*a^{24}) \bmod n = (a*a^8*a^{16}) \bmod n$$
$$= (a*((a^2)^2)^2*(((a^2)^2)^2)^2) \bmod n = ((((a^2*a)^2)^2)^2*a) \bmod n$$

With judicious storing of intermediate results, you only need six multiplications:

$$(((((((a^2 \bmod n)*a) \bmod n)^2 \bmod n)^2 \bmod n)^2 \bmod n)*a) \bmod n$$

This is called **addition chaining** [863], or the binary square and multiply method. It uses a simple and obvious addition chain based on the binary representation. In C, it looks like:

```
unsigned long qe2(unsigned long x, unsigned long y, unsigned long n) {
        unsigned long s,t,u;
        int i;

        s = 1; t = x; u = y;

        while(u) {
                if(u&1) s = (s*t)%n;
                u>>=1;
                t = (t*t)%n;
        }
        return(s);
}
```

Another, recursive, algorithm is:

```
unsigned long fast_exp(unsigned long x, unsigned long y, unsigned long N) {
        unsigned long tmp;
     if(y==1) return(x % N);
     if (1^(x&1)) {
          tmp = fast_exp(x,y/2,N);
          return ((tmp*tmp)%N);
     }
     else {
          tmp = fast_exp(x,(y-1)/2,N);
          tmp = (tmp*tmp)%N;
          tmp = (tmp*x)%N;
          return (tmp);
     }
}
```

This technique reduces the operation to, on the average, $1.5*k$ operations, if k is the length of the number x in bits. Finding the calculation with the fewest operations is a hard problem (it has been proven that the sequence must contain at least $k - 1$ operations), but it is not too hard to get the number of operations down to $1.1*k$ or better, as k grows.

An efficient way to do modular reductions many times using the same n is **Montgomery's method** [1111]. Another method is called **Barrett's algorithm** [87]. The software performance of these two algorithms and the algorithm previously discussed is in [210]: The algorithm I've discussed is the best choice for singular modular reductions; Barrett's algorithm is the best choice for small arguments; and Montgomery's method is the best choice for general modular exponentiations. (Montgomery's method can also take advantage of small exponents, using something called mixed arithmetic.)

The inverse of exponentiation modulo n is calculating a **discrete logarithm**. I'll discuss this shortly.

Prime Numbers

A **prime** number is an integer greater than 1 whose only factors are 1 and itself: No other number evenly divides it. Two is a prime number. So are 73, 2521, 2365347734339, and $2^{756839} - 1$. There are an infinite number of primes. Cryptography, especially public-key cryptography, uses large primes (512 bits and even larger) often.

Evangelos Kranakis wrote an excellent book on number theory, prime numbers, and their applications to cryptography [896]. Paulo Ribenboim wrote two excellent references on prime numbers in general [1307,1308].

Greatest Common Divisor

Two numbers are **relatively prime** when they share no factors in common other than 1. In other words, if the **greatest common divisor** of a and n is equal to 1. This is written:

$$\gcd(a,n) = 1$$

The numbers 15 and 28 are relatively prime, 15 and 27 are not, and 13 and 500 are. A prime number is relatively prime to all other numbers except its multiples.

One way to compute the greatest common divisor of two numbers is with **Euclid's algorithm**. Euclid described the algorithm in his book, *Elements*, written around 300 B.C. He didn't invent it. Historians believe the algorithm could be 200 years older. It is the oldest nontrivial algorithm that has survived to the present day, and it is still a good one. Knuth describes the algorithm and some modern modifications [863].

In C:

```
/* returns gcd of x and y */

int gcd (int x, int y)
{
     int g;

     if (x < 0)
          x = -x;
     if (y < 0)
          y = -y;
     if (x + y == 0)
          ERROR;
     g = y;
     while (x > 0) {
          g = x;
          x = y % x;
          y = g;
     }
     return g;
}
```

This algorithm can be generalized to return the gcd of an array of m numbers:

```
/* returns the gcd of x1, x2...xm */

int multiple_gcd (int m, int *x)
{
     size_t i;
     int g;

     if (m < 1)
          return 0;
     g = x[0];
     for (i=1; i<m; ++i) {
          g = gcd(g, x[i]);
/* optimization, since for random x[i], g==1 60% of the time: */
          if (g == 1)
               return 1;
     }
     return g;
}
```

Inverses Modulo a Number

Remember inverses? The multiplicative inverse of 4 is 1/4, because $4*1/4 = 1$. In the modulo world, the problem is more complicated:

$$4*x \equiv 1 \pmod 7$$

This equation is equivalent to finding an x and k such that

$$4x = 7k + 1$$

where both x and k are integers.

The general problem is finding an x such that

$$1 = (a*x) \bmod n$$

This is also written as

$$a^{-1} \equiv x \pmod n$$

The modular inverse problem is a lot more difficult to solve. Sometimes it has a solution, sometimes not. For example, the inverse of 5, modulo 14, is 3. On the other hand, 2 has no inverse modulo 14.

In general, $a^{-1} \equiv x \pmod n$ has a unique solution if a and n are relatively prime. If a and n are not relatively prime, then $a^{-1} \equiv x \pmod n$ has no solution. If n is a prime number, then every number from 1 to $n - 1$ is relatively prime to n and has exactly one inverse modulo n in that range.

So far, so good. Now, how do you go about finding the inverse of a modulo n? There are a couple of ways. Euclid's algorithm can also compute the inverse of a number modulo n. Sometimes this is called the **extended Euclidean algorithm**.

Here's the algorithm in C++:

```
#define isEven(x)   (x & 0x01 == 0)
```

```
#define isOdd(x)    (x & 0x01)
#define swap(x,y)   (x ^= y, y ^= x, x ^= y)

void ExtBinEuclid(int *u, int *v, int *u1, int *u2, int *u3)
{
    // warning: u and v will be swapped if u < v
    int k, t1, t2, t3;

    if ( *u < *v ) swap(*u,*v);
    for (k = 0; isEven(*u) && isEven(*v); ++k) {
        *u >>= 1; *v >>= 1;
    }
    *u1 = 1; *u2 = 0; *u3 = *u; t1 = *v; t2 = *u-1; t3 = *v;
    do {
        do {
            if (isEven(*u3)) {
                if (isOdd(*u1) || isOdd(*u2)) {
                    *u1 += *v; *u2 += *u;
                }
                *u1 >>= 1; *u2 >>= 1; *u3 >>= 1;
            }
            if (isEven(t3) || *u3 < t3) {
                swap(*u1,t1); swap(*u2,t2); swap(*u3,t3);
            }
        } while (isEven(*u3));
        while (*u1 < t1 || *u2 < t2) {
            *u1 += *v; *u2 += *u;
        }
        *u1 -= t1; *u2 -= t2; *u3 -= t3;
    } while (t3 > 0);
    while (*u1 >= *v && *u2 >= *u) {
        *u1 -= *v; *u2 -= *u;
    }
    *u <<= k; *v <<= k; *u3 <<= k;
}

main(int argc, char **argv) {
    int a, b, gcd;

    if ( argc < 3 ) {
        cerr << "Usage: xeuclid u v" << endl;
        return -1;
    }
    int u = atoi(argv[1]);
    int v = atoi(argv[2]);
    if ( u <= 0 || v <= 0 ) {
        cerr << "Arguments must be positive!" << endl;
        return -2;
    }
    // warning: u and v will be swapped if u < v
    ExtBinEuclid(&u, &v, &a, &b, &gcd);
    cout << a << " * " << u << " + (-"
        << b << ") * " << v << " = " << gcd << endl;
    if ( gcd == 1 )
```

```
        cout << "the inverse of " << v << " mod " << u << " is: "
             << u - b << endl;
    return 0;
}
```

I'm not going to prove that it works or give the theory behind it. Details can be found in [863], or in any of the number theory texts previously listed.

The algorithm is iterative and can be slow for large numbers. Knuth showed that the average number of divisions performed by the algorithm is:

$$.843 * \log_2(n) + 1.47$$

Solving for Coefficients

Euclid's algorithm can be used to solve this class of problems: Given an array of m variables $x_1, x_2, \ldots x_m$, find an array of m coefficients, $u_1, u_2 \ldots u_m$, such that

$$u_1 * x_1 + \ldots + u_m * x_m = 1$$

Fermat's Little Theorem

If m is a prime, and a is not a multiple of m, then **Fermat's little theorem** says

$$a^{m-1} \equiv 1 \pmod{m}$$

(Pierre de Fermat, pronounced "Fair-ma," was a French mathematician who lived from 1601 to 1665. This theorem has nothing to do with his last theorem.)

The Euler Totient Function

There is another method for calculating the inverse modulo n, but it's not always possible to use it. The **reduced set of residues** mod n is the subset of the complete set of residues that is relatively prime to n. For example, the reduced set of residues mod 12 is {1,5,7,11}. If n is prime, then the reduced set of residues mod n is the set of all numbers from 1 to $n - 1$. The number 0 is never part of the reduced set of residues for any n not equal to 1.

The **Euler totient function**, also called the Euler phi function and written as $\phi(n)$, is the number of elements in the reduced set of residues modulo n. In other words, $\phi(n)$ is the number of positive integers less than n that are relatively prime to n (for any n greater than 1). (Leonhard Euler, pronounced "Oiler," was a Swiss mathematician who lived from 1707 to 1783.)

If n is prime, then $\phi(n) = n - 1$. If $n = pq$, where p and q are prime, then $\phi(n) = (p - 1)(q - 1)$. These numbers appear in some public-key algorithms; this is why.

According to **Euler's generalization of Fermat's little theorem**, if $\gcd(a,n) = 1$, then

$$a^{\phi(n)} \bmod n = 1$$

Now it is easy to compute $a^{-1} \bmod n$:

$$x = a^{\phi(n)-1} \bmod n$$

For example, what is the inverse of 5, modulo 7? Since 7 is prime, $\phi(7) = 7 - 1 = 6$. So, the inverse of 5, modulo 7, is

$$5^{6-1} \bmod 7 = 5^5 \bmod 7 = 3$$

Both methods for calculating inverses can be extended to solve for x in the general problem (if $\gcd(a,n) = 1$):

$$(a*x) \bmod n = b$$

Using Euler's generalization, solve

$$x = (b*a^{\phi(n)-1}) \bmod n$$

Using Euclid's algorithm, solve

$$x = (b*(a^{-1} \bmod n)) \bmod n$$

In general, Euclid's algorithm is faster than Euclid's generalization for calculating inverses, especially for numbers in the 500-bit range. If $\gcd(a,n) \neq 1$, all is not lost. In this general case, $(a*x) \bmod n = b$, can have multiple solutions or no solution.

Chinese Remainder Theorem

If you know the prime factorization of n, then you can use something called the **Chinese remainder theorem** to solve a whole system of equations. The basic version of this theorem was discovered by the first-century Chinese mathematician, Sun Tse.

In general, if the prime factorization of n is $p_1*p_2*\ldots*p_t$, then the system of equations

$$(x \bmod p_i) = a_i, \text{ where } i = 1, 2, \ldots, t$$

has a unique solution, x, where x is less than n. (Note that some primes can appear more than once. For example, p_1 might be equal to p_2.) In other words, a number (less than the product of some primes) is uniquely identified by its residues mod those primes.

For example, use 3 and 5 as primes, and 14 as the number. 14 mod 3 = 2, and 14 mod 5 = 4. There is only one number less than 3*5 = 15 which has those residues: 14. The two residues uniquely determine the number.

So, for an arbitrary $a < p$ and $b < q$ (where p and q are prime), there exists a unique x, where x is less than pq, such that

$$x \equiv a \pmod{p}, \text{ and } x \equiv b \pmod{q}$$

To find this x, first use Euclid's algorithm to find u, such that

$$u*q \equiv 1 \pmod{p}$$

Then compute:

$$x = (((a - b)*u) \bmod p)*q + b$$

Here is the Chinese remainder theorem in C:

```
/* r is the number of elements in arrays m and u;
m is the array of (pairwise relatively prime) moduli
u is the array of coefficients
```

```
return value is n such than n == u[k]%m[k] (k=0..r-1) and
   n < m[0]*m[1]*...*m[r-1]
*/

/* totient() is left as an exercise to the reader. */

int chinese_remainder (size_t r, int *m, int *u)
{
    size_t i;
    int modulus;
    int n;

    modulus = 1;
    for (i=0; i<r; ++i)
        modulus *= m[i];

    n = 0;
    for (i=0; i<r; ++i) {
        n += u[i] * modexp(modulus / m[i], totient(m[i]),
    m[i]);
        n %= modulus;
    }

    return n;
}
```

The converse of the Chinese remainder theorem can also be used to find the solution to the problem: if p and q are primes, and p is less than q, then there exists a unique x less than pq, such that

$$a \equiv x \pmod{p}, \text{ and } b \equiv x \pmod{q}$$

If $a \geq b \bmod p$, then

$$x = (((a - (b \bmod p)) * u) \bmod p) * q + b$$

If $a < b \bmod p$, then

$$x = (((a + p - (b \bmod p)) * u) \bmod p) * q + b$$

Quadratic Residues

If p is prime, and a is greater than 0 and less than p, then a is a **quadratic residue** mod p if

$$x^2 \equiv a \pmod{p}, \text{ for some } x$$

Not all values of a satisfy this property. For a to be a quadratic residue modulo n, it must be a quadratic residue modulo all the prime factors of n. For example, if $p = 7$, the quadratic residues are 1, 2, and 4:

$$1^2 = 1 \equiv 1 \pmod{7}$$
$$2^2 = 4 \equiv 4 \pmod{7}$$
$$3^2 = 9 \equiv 2 \pmod{7}$$

$4^2 = 16 \equiv 2 \pmod 7$
$5^2 = 25 \equiv 4 \pmod 7$
$6^2 = 36 \equiv 1 \pmod 7$

Note that each quadratic residue appears twice on this list.

There are no values of x which satisfy any of these equations:

$x^2 \equiv 3 \pmod 7$
$x^2 \equiv 5 \pmod 7$
$x^2 \equiv 6 \pmod 7$

The **quadratic nonresidues** modulo 7, the numbers that are not quadratic residues, are 3, 5, and 6.

Although I will not do so here, it is easy to prove that, when p is odd, there are exactly $(p-1)/2$ quadratic residues mod p and the same number of quadratic nonresidues mod p. Also, if a is a quadratic residue mod p, then a has exactly two square roots, one of them between 0 and $(p-1)/2$, and the other between $(p-1)/2$ and $(p-1)$. One of these square roots is also a quadratic residue mod p; this is called the **principal square root**.

If n is the product of two primes, p and q, there are exactly $(p-1)(q-1)/4$ quadratic residues mod n. A quadratic residue mod n is a perfect square modulo n. This is because to be a square mod n, the residue must be a square mod p and a square mod q. For example, there are six quadratic residues mod 35: 1, 4, 9, 11, 16, 29. Each quadratic residue has exactly four square roots.

Legendre Symbol

The **Legendre symbol**, written $L(a,p)$, is defined when a is any integer and p is a prime greater than 2. It is equal to 0, 1, or –1.

$L(a,p) = 0$ if a is divisible by p.
$L(a,p) = 1$ if a is a quadratic residue mod p.
$L(a,p) = -1$ is a is a quadratic nonresidue mod p.

One way to calculate $L(a,p)$ is:

$$L(a,p) = a^{(p-1)/2} \bmod p$$

Or you can use the following algorithm:

1. If $a = 1$, then $L(a,p) = 1$
2. If a is even, then $L(a,p) = L(a/2,p) * (-1)^{(p^2-1)/8}$
3. If a is odd (and $\neq 1$), then $L(a,p) = L(p \bmod a, a) * (-1)^{(a-1)*(p-1)/4}$

Note that this is also an efficient way to determine whether a is a quadratic residue mod p (when p is prime).

Jacobi Symbol

The **Jacobi symbol**, written $J(a,n)$, is a generalization of the Legendre symbol to composite moduli; it is defined for any integer a and any odd integer n. The function shows up in primality testing. The Jacobi symbol is a function on the set of reduced residues of the divisors of n and can be calculated by several formulas [1412]. This is one method:

Definition 1: $J(a,n)$ is only defined if n is odd.

Definition 2: $J(0,n) = 0$.

Definition 3: If n is prime, then the Jacobi symbol $J(a,n) = 0$ if n divides a.

Definition 4: If n is prime, then the Jacobi symbol $J(a,n) = 1$ if a is a quadratic residue modulo n.

Definition 5: If n is prime, then the Jacobi symbol $J(a,n) = -1$ if a is a quadratic nonresidue modulo n.

Definition 6: If n is composite, then the Jacobi symbol $J(a,n) = J(a,p_1) * \ldots * J(a,p_m)$, where $p_1 \ldots p_m$ is the prime factorization of n.

The following algorithm computes the Jacobi symbol recursively:

Rule 1: $J(1,n) = 1$

Rule 2: $J(a*b,n) = J(a,n)*J(b,n)$

Rule 3: $J(2,n) = 1$ if $(n^2 - 1)/8$ is even, and -1 otherwise

Rule 4: $J(a,n) = J((a \bmod n),n)$

Rule 5: $J(a,b_1*b_2) = J(a,b_1)*J(a,b_2)$

Rule 6: If the greatest common divisor of a and $b = 1$, and a and b are odd:

Rule 6a: $J(a,b) = J(b,a)$ if $(a - 1)(b - 1)/4$ is even

Rule 6b: $J(a,b) = -J(b,a)$ if $(a - 1)(b - 1)/4$ is odd

Here is the algorithm in C:

```
/* This algorithm computes the Jacobi symbol recursively */

int jacobi(int a, int b)
{
int g;

    assert(odd(b));

    if (a >= b) a %= b;    /* by Rule 4 */
    if (a == 0) return 0;  /* by Definition 1 */
    if (a == 1) return 1;  /* by Rule 1 */

    if (a < 0)
        if (((b-1)/2 % 2 == 0)
```

```
            return jacobi(-a,b);
        else
            return -jacobi(-a,b);

    if (a % 2 == 0) /* a is even */
        if (((b*b - 1)/8) % 2 == 0)
            return +jacobi(a/2, b)
        else
            return -jacobi(a/2, b) /* by Rule 3 and Rule 2 */
    g = gcd(a,b);

    assert(odd(a)); /* this is guaranteed by the (a % 2 == 0)
test */

    if (g == a) /* a exactly divides b */
        return 0; /* by Rule 5 */
    else if (g != 1)
        return jacobi(g,b) * jacobi(a/g, b); /* by Rule 2 */
    else if (((a-1)*(b-1)/4) % 2 == 0)
        return +jacobi(b,a);      /* by Rule 6a */
    else
        return -jacobi(b,a);      /* by Rule 6b */
}
```

If n is known to be prime beforehand, simply compute $a^{\lfloor (n-1)/2 \rfloor} \bmod n$ instead of running the previous algorithm; in this case $J(a,n)$ is equivalent to the Legendre symbol.

The Jacobi symbol cannot be used to determine whether a is a quadratic residue mod n (unless n is prime, of course). Note that, if $J(a,n) = 1$ and n is composite, it is not necessarily true that a is a quadratic residue modulo n. For example:

$$J(7,143) = J(7,11)*J(7,13) = (-1)(-1) = 1$$

However, there is no integer x such that $x^2 \equiv 7 \pmod{143}$.

Blum Integers

If p and q are two primes, and both are congruent to 3 modulo 4, then $n = pq$ is sometimes called a **Blum integer**. If n is a Blum integer, each quadratic residue has exactly four square roots, one of which is also a square; this is the principal square root. For example, the principal square root of 139 mod 437 is 24. The other three square roots are 185, 252, and 413.

Generators

If p is a prime, and g is less then p, then g is a **generator** mod p if

for each b from 1 to $p - 1$, there exists some a where $g^a \equiv b \pmod{p}$.

Another way of saying this is that g is **primitive** with respect to p.
For example, if $p = 11$, 2 is a generator mod 11:

$$2^{10} = 1024 \equiv 1 \pmod{11}$$
$$2^1 = 2 \equiv 2 \pmod{11}$$

$$2^8 = 256 \equiv 3 \pmod{11}$$
$$2^2 = 4 \equiv 4 \pmod{11}$$
$$2^4 = 16 \equiv 5 \pmod{11}$$
$$2^9 = 512 \equiv 6 \pmod{11}$$
$$2^7 = 128 \equiv 7 \pmod{11}$$
$$2^3 = 8 \equiv 8 \pmod{11}$$
$$2^6 = 64 \equiv 9 \pmod{11}$$
$$2^5 = 32 \equiv 10 \pmod{11}$$

Every number from 1 to 10 can be expressed as $2^a \pmod{p}$.

For $p = 11$, the generators are 2, 6, 7, and 8. The other numbers are not generators. For example, 3 is not a generator because there is no solution to

$$3^a \equiv 2 \pmod{11}$$

In general, testing whether a given number is a generator is not an easy problem. It is easy, however, if you know the factorization of $p - 1$. Let $q_1, q_2, \ldots, q_n$ be the distinct prime factors of $p - 1$. To test whether a number g is a generator mod p, calculate

$$g^{(p-1)/q} \bmod p$$

for all values of $q = q_1, q_2, \ldots, q_n$.

If that number equals 1 for some value of q, then g is not a generator. If that value does not equal 1 for any values of q, then g is a generator.

For example, let $p = 11$. The prime factors of $p - 1 = 10$ are 2 and 5. To test whether 2 is a generator:

$$2^{(11-1)/5} \pmod{11} = 4$$
$$2^{(11-1)/2} \pmod{11} = 10$$

Neither result is 1, so 2 is a generator.

To test whether 3 is a generator:

$$3^{(11-1)/5} \pmod{11} = 9$$
$$3^{(11-1)/2} \pmod{11} = 1$$

Therefore, 3 is not a generator.

If you need to find a generator mod p, simply choose a random number from 1 to $p - 1$ and test whether it is a generator. Enough of them will be, so you'll probably find one fast.

Computing in a Galois Field

Don't be alarmed; that's what we were just doing. If n is prime or the power of a large prime, then we have what mathematicians call a **finite field**. In honor of that fact, we use p instead of n. In fact, this type of finite field is so exciting that mathematicians gave it its own name: a **Galois field**, denoted as GF(p). (Évariste Galois was a French mathematician who lived in the early nineteenth century and did a lot of work in number theory before he was killed at age 20 in a duel.)

In a Galois field, addition, subtraction, multiplication, and division by nonzero elements are all well-defined. There is an additive identity, 0, and a multiplicative identity, 1. Every nonzero number has a unique inverse (this would not be true if p were not prime). The commutative, associative, and distributive laws are true.

Arithmetic in a Galois field is used a great deal in cryptography. All of the number theory works; it keeps numbers a finite size, and division doesn't have any rounding errors. Many cryptosystems are based on GF(p), where p is a large prime.

To make matters even more complicated, cryptographers also use arithmetic modulo **irreducible** polynomials of degree n whose coefficients are integers modulo q, where q is prime. These fields are called GF(q^n). All arithmetic is done modulo $p(x)$, where $p(x)$ is an irreducible polynomial of degree n.

The mathematical theory behind this is far beyond the scope of the book, although I will describe some cryptosystems that use it. If you want to try to work more with this, GF(2^3) has the following elements: 0, 1, x, $x + 1$, x^2, $x^2 + 1$, $x^2 + x$, $x^2 + x + 1$. There is an algorithm for computing inverses in GF(2^n) that is suitable for parallel implementation [421].

When talking about polynomials, the term "prime" is replaced by "irreducible." A polynomial is irreducible if it cannot be expressed as the product of two other polynomials (except for 1 and itself, of course). The polynomial $x^2 + 1$ is irreducible over the integers. The polynomial $x^3 + 2x^2 + x$ is not; it can be expressed as $x(x + 1)(x + 1)$.

A polynomial that is a generator in a given field is called primitive; all its coefficients are relatively prime. We'll see primitive polynomials again when we talk about linear-feedback shift registers (see Section 16.2).

Computation in GF(2^n) can be quickly implemented in hardware with linear-feedback shift registers. For that reason, computation over GF(2^n) is often quicker than computation over GF(p). Just as exponentiation is much more efficient in GF(2^n), so is calculating discrete logarithms [180,181,368,379]. If you want to learn more about this, read [140].

For a Galois field GF(2^n), cryptographers like to use the trinomial $p(x) = x^n + x + 1$ as the modulus, because the long string of zeros between the x^n and x coefficients makes it easy to implement a fast modular multiplication [183]. The trinomial must be primitive, otherwise the math does not work. Values of n less than 1000 [1649,1648] for which $x^n + x + 1$ is primitive are:

1, 3, 4, 6, 9, 15, 22, 28, 30, 46, 60, 63, 127, 153, 172, 303, 471, 532, 865, 900

There exists a hardware implementation of GF(2^{127}) where $p(x) = x^{127} + x + 1$ [1631,1632,1129]. Efficient hardware architectures for implementing exponentiation in GF(2^n) are discussed in [147].

11.4 FACTORING

Factoring a number means finding its prime factors.

$10 = 2*5$

$60 = 2*2*3*5$

$$252601 = 41 * 61 * 101$$

$$2^{113} - 1 = 3391 * 23279 * 65993 * 1868569 * 1066818132868207$$

The factoring problem is one of the oldest in number theory. It's simple to factor a number, but it's time-consuming. This is still true, but there have been some major advances in the state of the art.

Currently, the best factoring algorithm is:

Number field sieve (NFS) [953] (see also [952,16,279]). The **general number field sieve** is the fastest-known factoring algorithm for numbers larger than 110 digits or so [472,635]. It was impractical when originally proposed, but that has changed due to a series of improvements over the last few years [953]. The NFS is still too new to have broken any factoring records, but this will change soon. An early version was used to factor the ninth Fermat number: $2^{512} + 1$ [955,954].

Other factoring algorithms have been supplanted by the NFS:

Quadratic sieve (QS) [1257,1617,1259]. This is the fastest-known algorithm for numbers less than 110 decimal digits long and has been used extensively [440]. A faster version of this algorithm is called the multiple polynomial quadratic sieve [1453,302]. The fastest version of this algorithm is called the double large prime variation of the multiple polynomial quadratic sieve.

Elliptic curve method (ECM) [957,1112,1113]. This method has been used to find 43-digit factors, but nothing larger.

Pollard's Monte Carlo algorithm [1254,248]. (This algorithm also appears in volume 2, page 370 of Knuth [863].)

Continued fraction algorithm. See [1123,1252,863]. This algorithm isn't even in the running.

Trial division. This is the oldest factoring algorithm and consists of testing every prime number less than or equal to the square root of the candidate number.

See [251] for a good introduction to these different factoring algorithms, except for the NFS. The best discussion of the NFS is [953]. Other, older references are [505, 1602,1258]. Information on parallel factoring can be found in [250].

If n is the number being factored, the fastest QS variants have a heuristic asymptotic run time of:

$$e^{(1 + o(1))(\ln(n))^{(1/2)}(\ln(\ln(n)))^{(1/2)}}$$

The NFS is much faster, with a heuristic asymptotic time estimate of:

$$e^{(1.923 + o(1))(\ln(n))^{(1/3)}(\ln(\ln(n)))^{(2/3)}}$$

In 1970, the big news was the factoring of a 41-digit hard number [1123]. (A "hard" number is one that does not have any small factors and is not of a special form that

allows it to be factored more easily.) Ten years later, factoring hard numbers twice that size took a Cray computer just a few hours [440].

In 1988, Carl Pomerance designed a modular factoring machine, using custom VLSI chips [1259]. The size of the number you would be able to factor depends on how large a machine you can afford to build. He never built it.

In 1993, a 120-digit hard number was factored using the quadratic sieve; the calculation took 825 mips-years and was completed in three months real time [463]. Other results are [504].

Today's factoring attempts use computer networks [302,955]. In factoring a 116-digit number, Arjen Lenstra and Mark Manasse used 400 mips-years—the spare time on an array of computers around the world for a few months.

In March 1994, a 129-digit (428-bit) number was factored using the double large prime variation of the multiple polynomial QS [66] by a team of mathematicians led by Lenstra. Volunteers on the Internet carried out the computation: 600 people and 1600 machines over the course of eight months, probably the largest ad hoc multiprocessor ever assembled. The calculation was the equivalent of 4000 to 6000 mips-years. The machines communicated via electronic mail, sending their individual results to a central repository where the final steps of analysis took place. This computation used the QS and five-year-old theory; it would have taken one-tenth the time using the NFS [949]. According to [66]: "We conclude that commonly used 512-bit RSA moduli are vulnerable to any organization prepared to spend a few million dollars and to wait a few months." They estimate that factoring a 512-bit number would be 100 times harder using the same technology, and only 10 times harder using the NFS and current technology [949].

To keep up on the state of the art of factoring, RSA Data Security, Inc. set up the RSA Factoring Challenge in March 1991 [532]. The challenge consists of a list of hard numbers, each the product of two primes of roughly equal size. Each prime was chosen to be congruent to 2 modulo 3. There are 42 numbers in the challenge, one each of length 100 digits through 500 digits in steps of 10 digits (plus one additional number, 129 digits long). At the time of writing, RSA-100, RSA-110, RSA-120, and RSA-129 have been factored, all using the QS. RSA-130 might be next (using the NFS), or the factoring champions might skip directly to RSA-140.

This is a fast-moving field. It is difficult to extrapolate factoring technology because no one can predict advances in mathematical theory. Before the NFS was discovered, many people conjectured that the QS was asymptotically as fast as any factoring method could be. They were wrong.

Near-term advances in the NFS are likely to come in the form of bringing down the constant: 1.923. Some numbers of a special form, like Fermat numbers, have a constant more along the lines of 1.5 [955,954]. If the hard numbers used in public-key cryptography had that kind of constant, 1024-bit numbers could be factored today. One way to lower the constant is to find better ways of representing numbers as polynomials with small coefficients. The problem hasn't been studied very extensively yet, but it is probable that advances are coming [949].

For the most current results from the RSA Factoring Challenge, send e-mail to challenge-info@rsa.com.

Square Roots Modulo n

If n is the product of two primes, then the ability to calculate square roots mod n is computationally equivalent to the ability to factor n [1283,35,36,193]. In other words, someone who knows the prime factors of n can easily compute the square roots of a number mod n, but for everyone else the computation has been proven to be as hard as computing the prime factors of n.

11.5 PRIME NUMBER GENERATION

Public-key algorithms need prime numbers. Any reasonably sized network needs lots of them. Before discussing the mathematics of prime number generation, I will answer a few obvious questions.

1. If everyone needs a different prime number, won't we run out? No. In fact, there are approximately 10^{151} primes 512 bits in length or less. For numbers near n, the probability that a random number is prime is approximately one in ln n. So the total number of primes less than n is $n/(\ln n)$. There are only 10^{77} atoms in the universe. If every atom in the universe needed a billion new primes every microsecond from the beginning of time until now, you would only need 10^{109} primes; there would still be approximately 10^{151} 512-bit primes left.
2. What if two people accidentally pick the same prime number? It won't happen. With over 10^{151} prime numbers to choose from, the odds of that happening are significantly less than the odds of your computer spontaneously combusting at the exact moment you win the lottery.
3. If someone creates a database of all primes, won't he be able to use that database to break public-key algorithms? Yes, but he can't do it. If you could store one gigabyte of information on a drive weighing one gram, then a list of just the 512-bit primes would weigh so much that it would exceed the Chandrasar limit and collapse into a black hole . . . so you couldn't retrieve the data anyway.

But if factoring numbers is so hard, how can generating prime numbers be easy? The trick is that the yes/no question, "Is n prime?" is a much easier question to answer than the more complicated question, "What are the factors of n?"

The wrong way to find primes is to generate random numbers and then try to factor them. The right way is to generate random numbers and test if they are prime. There are several probabilistic primality tests; tests that determine whether a number is prime with a given degree of confidence. Assuming this "degree of confidence" is large enough, these sorts of tests are good enough. I've heard primes generated in this manner called "industrial-grade primes": These are numbers that are probably prime with a controllably small chance of error.

Assume a test is set to fail once in 2^{50} tries. This means that there is a 1 in 10^{15} chance that the test falsely indicates that a composite number is prime. (The test

will never falsely indicate that a prime number is composite.) If for some reason you need more confidence that the number is prime, you can set the failure level even lower. On the other hand, if you consider that the odds of the number being composite are 300 million times less than the odds of winning top prize in a state lottery, you might not worry about it so much.

Overviews of recent developments in the field can be found in [1256,206]. Other important papers are [1490,384,11,19,626,651,911].

Solovay-Strassen

Robert Solovay and Volker Strassen developed a probabilistic primality testing algorithm [1490]. Their algorithm uses the Jacobi symbol to test if p is prime:

(1) Choose a random number, a, less than p.
(2) If the $\gcd(a,p) \neq 1$, then p fails the test and is composite.
(3) Calculate $j = a^{(p-1)/2} \bmod p$.
(4) Calculate the Jacobi symbol $J(a,p)$.
(5) If $j \neq J(a,p)$, then p is definitely not prime.
(6) If $j = J(a,p)$, then the likelihood that p is not prime is no more than 50 percent.

A number a that does not indicate that p is definitely not prime is called a **witness**. If p is composite, the odds of a random a being a witness is no less than 50 percent. Repeat this test t times, with t different random values for a. The odds of a composite number passing all t tests is no more than one in 2^t.

Lehmann

Another, simpler, test was developed independently by Lehmann [903]. Here it tests if p is prime:

(1) Choose a random number a less than p.
(2) Calculate $a^{(p-1)/2} \bmod p$.
(3) If $a^{(p-1)/2} \not\equiv 1$ or $-1 \pmod p$, then p is definitely not prime.
(4) If $a^{(p-1)/2} \equiv 1$ or $-1 \pmod p$, then the likelihood that p is not prime is no more than 50 percent.

Again, the odds of a random a being a witness to p's compositeness is no less than 50 percent. Repeat this test t times. If the calculation equals 1 or -1, but does not always equal 1, then p is probably prime with an error rate of 1 in 2^t.

Rabin-Miller

The algorithm everyone uses—it's easy—was developed by Michael Rabin, based in part on Gary Miller's ideas [1093,1284]. Actually, this is a simplified version of the algorithm recommended in the DSS proposal [1149,1154].

Choose a random number, p, to test. Calculate b, where b is the number of times 2 divides $p-1$ (i.e., 2^b is the largest power of 2 that divides $p-1$). Then calculate m, such that $p = 1 + 2^b * m$.

(1) Choose a random number, a, such that a is less than p.
(2) Set $j = 0$ and set $z = a^m \bmod p$.
(3) If $z = 1$, or if $z = p-1$, then p passes the test and may be prime.
(4) If $j > 0$ and $z = 1$, then p is not prime.
(5) Set $j = j + 1$. If $j < b$ and $z \neq p-1$, set $z = z^2 \bmod p$ and go back to step (4). If $z = p-1$, then p passes the test and may be prime.
(6) If $j = b$ and $z \neq p-1$, then p is not prime.

The odds of a composite passing decreases faster with this test than with previous ones. Three-quarters of the possible values of a are guaranteed to be witnesses. This means that a composite number will slip through t tests no more than $\frac{1}{4}^t$ of the time, where t is the number of iterations. Actually, these numbers are very pessimistic. For most random numbers, something like 99.9 percent of the possible a values are witnesses [96].

There are even better estimations [417]. For n-bit candidate primes (where n is more than 100), the odds of error in one test are less than 1 in $4n2^{\lfloor k/2 \rfloor \lfloor 1/2 \rfloor}$. And for a 256-bit n, the odds of error in six tests are less than 1 in 2^{51}. More theory is in [418].

Practical Considerations

In real-world implementations, prime generation goes quickly.

(1) Generate a random n-bit number, p.
(2) Set the high-order and low-order bit to 1. (The high-order bit ensures that the prime is of the required length and the low-order bit ensures that it is odd.)
(3) Check to make sure p is not divisible by any small primes: 3, 5, 7, 11, and so on. Many implementations test p for divisibility by all primes less than 256. The most efficient is to test for divisibility by all primes less than 2000 [949]. You can do this efficiently using a wheel [863].
(4) Perform the Rabin-Miller test for some random a. If p passes, generate another random a and go through the test again. Choose a small value of a to make the calculations go quicker. Do five tests [651]. (One might seem like enough, but do five.) If p fails one of the tests, generate another p and try again.

Another option is not to generate a random p each time, but to incrementally search through numbers starting at a random point until you find a prime.

Step (3) is optional, but it is a good idea. Testing a random odd p to make sure it is not divisible by 3, 5, and 7 eliminates 54 percent of the odd numbers before you get

to step (4). Testing against all primes less than 100 eliminates 76 percent of the odd numbers; testing against all primes less than 256 eliminates 80 percent. In general, the fraction of odd candidates that is not a multiple of any prime less than n is $1.12/\ln n$. The larger the n you test up to, the more precomputation is required before you get to the Rabin-Miller test.

One implementation of this method on a Sparc II was able to find 256-bit primes in an average of 2.8 seconds, 512-bit primes in an average of 24.0 seconds, 768-bit primes in an average of 2.0 minutes, and 1024-bit primes in an average of 5.1 minutes [918].

Strong Primes

If n is the product of two primes, p and q, it may be desirable to use **strong primes** for p and q. These are prime numbers with certain properties that make the product n difficult to factor by specific factoring methods. Among the properties suggested have been [1328,651]:

The greatest common divisor of $p - 1$ and $q - 1$ should be small.

Both $p - 1$ and $q - 1$ should have large prime factors, respectively p' and q'.

Both $p' - 1$ and $q' - 1$ should have large prime factors.

Both $p + 1$ and $q + 1$ should have large prime factors.

Both $(p - 1)/2$ and $(q - 1)/2$ should be prime [182]. (Note that if this condition is true, then so are the first two.)

Whether strong primes are necessary is a subject of debate. These properties were designed to thwart some older factoring algorithms. However, the fastest factoring algorithms have as good a chance of factoring numbers that meet these criteria as they do of factoring numbers that do not [831].

I recommend against specifically generating strong primes. The length of the primes is much more important than the structure. Moreover, structure may be damaging because it is less random.

This may change. New factoring techniques may be developed that work better on numbers with certain properties than on numbers without them. If so, strong primes may be required once again. Check current theoretical mathematics journals for updates.

11.6 Discrete Logarithms in a Finite Field

Modular exponentiation is another one-way function used frequently in cryptography. Evaluating this expression is easy:

$$a^x \bmod n$$

The inverse problem of modular exponentiation is that of finding the discrete logarithm of a number. This is a hard problem:

Find x where $a^x \equiv b \pmod{n}$.

For example:

If $3^x \equiv 15 \bmod 17$, then $x = 6$

Not all discrete logarithms have solutions (remember, the only valid solutions are integers). It's easy to see that there is no solution, x, to the equation

$$3^x \equiv 7 \pmod{13}$$

It's far more difficult to solve these problems using 1024-bit numbers.

Calculating Discrete Logarithms in a Finite Group

There are three main groups whose discrete logarithms are of interest to cryptographers:

- The multiplicative group of prime fields: GF(p)
- The multiplicative group of finite fields of characteristic 2: GF(2^n)
- Elliptic curve groups over finite fields F: EC(F)

The security of many public-key algorithms is based on the problem of finding discrete logarithms, so the problem has been extensively studied. A good comprehensive overview of the problem, and the best solutions at the time, can be found in [1189,1039]. The best current article on the topic is [934].

If p is the modulus and is prime, then the complexity of finding discrete logarithms in GF(p) is essentially the same as factoring an integer n of about the same size, when n is the product of two approximately equal-length primes [1378,934]. This is:

$$e^{(1 + o(1))(\ln (p))^{(1/2)}(\ln (\ln (p)))^{(1/2)}}$$

The number field sieve is faster, with an heuristic asymptotic time estimate of

$$e^{(1.923 + o(1))(\ln (p))^{(1/3)}(\ln (\ln (p)))^{(2/3)}}$$

Stephen Pohlig and Martin Hellman found a fast way of computing discrete logarithms in GF(p) if $p - 1$ has only small prime factors [1253]. For this reason, only fields where $p - 1$ has at least one large factor are used in cryptography. Another algorithm [14] computes discrete logarithms at a speed comparable to factoring; it has been expanded to fields of the form GF(p^n) [716]. This algorithm was criticized [727] for having some theoretical problems. Other articles [1588] show how difficult the problem really is.

Computing discrete logarithms is closely related to factoring. If you can solve the discrete logarithm problem, then you can factor. (The converse has never been proven to be true.) Currently, there are three methods for calculating discrete logarithms in a prime field [370,934,648]: the linear sieve, the Gaussian integer scheme, and the number field sieve.

The preliminary, extensive computing has to be done only once per field. Afterward, individual logarithms can be quickly calculated. This can be a security disad-

vantage for systems based on these fields. It is important that different applications use different prime fields. Multiple users in the same application can use a common field, though.

In the world of extension fields, $GF(2^n)$ hasn't been ignored by researchers. An algorithm was proposed in [727]. Coppersmith's algorithm makes finding discrete logarithms in fields such as $GF(2^{127})$ reasonable and finding them in fields around $GF(2^{400})$ possible [368]. This was based on work in [180]. The precomputation stage of this algorithm is enormous, but otherwise it is nice and efficient. A practical implementation of a less efficient version of the same algorithm, after a seven-hour precomputation period, found discrete logs in $GF(2^{127})$ in several seconds each [1130,180]. (This particular field, once used in some cryptosystems [142,1631,1632], is insecure.) For surveys of some of these results, consult [1189,1039].

More recently, the precomputations for $GF(2^{227})$, $GF(2^{313})$, and $GF(2^{401})$ are done, and significant progress has been made towards $GF(2^{503})$. These calculations are being executed on an nCube-2 massively parallel computer with 1024 processors [649,650]. Computing discrete logarithms in $GF(2^{593})$ is still barely out of reach.

Like discrete logarithms in a prime field, the precomputation required to calculate discrete logarithms in a polynomial field has to be done only once. Taher ElGamal [520] gives an algorithm for calculating discrete logs in the field $GF(p^2)$.

CHAPTER 17

Other Stream Ciphers and Real Random-Sequence Generators

17.1 RC4

RC4 is a variable-key-size stream cipher developed in 1987 by Ron Rivest for RSA Data Security, Inc. For seven years it was proprietary, and details of the algorithm were only available after signing a nondisclosure agreement.

In September, 1994 someone posted source code to the Cypherpunks mailing list—anonymously. It quickly spread to the Usenet newsgroup sci.crypt, and via the Internet to ftp sites around the world. Readers with legal copies of RC4 confirmed compatibility. RSA Data Security, Inc. tried to put the genie back into the bottle, claiming that it was still a trade secret even though it was public; it was too late. It has since been discussed and dissected on Usenet, distributed at conferences, and taught in cryptography courses.

RC4 is simple to describe. The algorithm works in OFB: The keystream is independent of the plaintext. It has a 8 * 8 S-box: $S_0, S_1, \ldots, S_{255}$. The entries are a permutation of the numbers 0 through 255, and the permutation is a function of the variable-length key. It has two counters, i and j, initialized to zero.

To generate a random byte, do the following:

$$i = (i + 1) \bmod 256$$
$$j = (j + S_i) \bmod 256$$
swap S_i and S_j
$$t = (S_i + S_j) \bmod 256$$
$$K = S_t$$

The byte K is XORed with the plaintext to produce ciphertext or XORed with the ciphertext to produce plaintext. Encryption is fast—about 10 times faster than DES.

Initializing the S-box is also easy. First, fill it linearly: $S_0 = 0, S_1 = 1, \ldots, S_{255} = 255$. Then fill another 256-byte array with the key, repeating the key as necessary to fill the entire array: $K_0, K_1, \ldots, K_{255}$. Set the index j to zero. Then:

for $i = 0$ to 255:
  $j = (j + S_i + K_i) \bmod 256$
  swap S_i and S_j

And that's it. RSADSI claims that the algorithm is immune to differential and linear cryptanalysis, doesn't seem to have any small cycles, and is highly nonlinear. (There are no public cryptanalytic results. RC4 can be in about 2^{1700} ($256! \times 256^2$) possible states: an enormous number.) The S-box slowly evolves with use: i ensures that every element changes and j ensures that the elements change randomly. The algorithm is simple enough that most programmers can quickly code it from memory.

It should be possible to generalize this idea to larger S-boxes and word sizes. The previous version is 8-bit RC4. There's no reason why you can't define 16-bit RC4 with a 16 * 16 S-box (100K of memory) and a 16-bit word. You'd have to iterate the initial setup a lot more times—65,536 to keep with the stated design—but the resulting algorithm should be faster.

RC4 has special export status if its key length is 40 bits or under (see Section 13.8). This special export status has nothing to do with the secrecy of the algorithm, although RSA Data Security, Inc. has hinted for years that it does. The name is trademarked, so anyone who writes his own code has to call it something else. Various internal documents by RSA Data Security, Inc. have not yet been made public [1320,1337].

So, what's the deal with RC4? It's no longer a trade secret, so presumably anyone can use it. However, RSA Data Security, Inc. will almost certainly sue anyone who uses unlicensed RC4 in a commercial product. They probably won't win, but they will certainly make it cheaper for a company to license than fight.

RC4 is in dozens of commercial cryptography products, including Lotus Notes, Apple Computer's AOCE, and Oracle Secure SQL. It is part of the Cellular Digital Packet Data specification [37].

17.2 SEAL

SEAL is a software-efficient stream cipher designed at IBM by Phil Rogaway and Don Coppersmith [1340]. The algorithm was optimized for 32-bit processors: To run well it needs eight 32-bit registers and a cache of a few kiloytes. Using a relatively slow operation, SEAL preprocesses the key operation into a set of tables. These tables are then used to speed up encryption and decryption.

Pseudo-random Function Family

One novel feature of SEAL is that is isn't really a traditional stream cipher: it is a **pseudo-random function family**. Given a 160-bit key k, and a 32-bit n, SEAL stretches n into an L-bit string $k(n)$. L can take any value less than 64 kilobytes. SEAL is supposed to enjoy the property that if k is selected at random, then $k(n)$ should be computationally indistinguishable from a random L-bit function of n.

The practical effect of SEAL being a pseudo-random function family is that it is useful in applications where traditional stream ciphers are not. With most stream ciphers you generate a sequence of bits in one direction: Knowing the key and a position i, the only way to determine the ith bit generated is to generate all the bits up until the ith one. But a pseudo-random function family is different: You get easy access at any desired position in the key stream. This is very useful.

Imagine you need to secure a hard drive. You want to encrypt each and every 512-byte sector. With a pseudo-random function family like SEAL, you can encrypt the contents of sector n by XORing it with $k(n)$. It is as though the entire disk is XORed with a long pseudo-random string, where any piece of that long string can be computed without any trouble.

A pseudo-random function family also simplifies the synchronization problem encountered with standard stream ciphers. Suppose you send encrypted messages over a channel that sometimes drops messages. With a pseudo-random function family, you can encrypt under k the nth message you transmit, x_n, as n together with the XOR of x_n and $k(n)$. The receiver doesn't need to store any state to recover x_n, nor does he need to worry about lost messages affecting the message decryption process.

Description of SEAL

The inner loop of SEAL is shown by Figure 17.1. Three key-derived tables, called R, S, and T, drive the algorithm. The preprocessing step maps the key k, to these tables using a procedure based on SHA (see Section 18.7). The 2-kilobyte table, T, is a $9 * 32$ bit S-box.

SEAL also uses four 32-bit registers, A, B, C, and D, whose initial values are determined by n and the k-derived tables R and T. These registers get modified over several iterations, each one involving 8 rounds. In each round 9 bits of a first register (either A, B, C, or D) are used to index into table T. The value retrieved from T is then added to or XORed with the contents of a second register: again one of A, B, C, or D. The first register is then circularly shifted by nine positions. In some rounds the second register is further modified by adding or XORing it with the (now shifted) first register. After 8 rounds of this, A, B, C, and D are added to the keystream, each masked first by adding or XORing it with a certain word from S. The iteration is completed by adding to A and C additional values dependent on n, n_1, n_2, n_3, n_4; exactly which one depends on the parity of the iteration number.

The important ideas in this design seem to be:

1. Use a large, secret, key-derived S-box (T).
2. Alternate arithmetic operations which don't commute (addition and XOR).
3. Use an internal state maintained by the cipher which is not directly manifest in the data stream (the n_i values which modify A and C at the end of each iteration).
4. Vary the round function according to the round number, and vary the iteration function according to the iteration number.

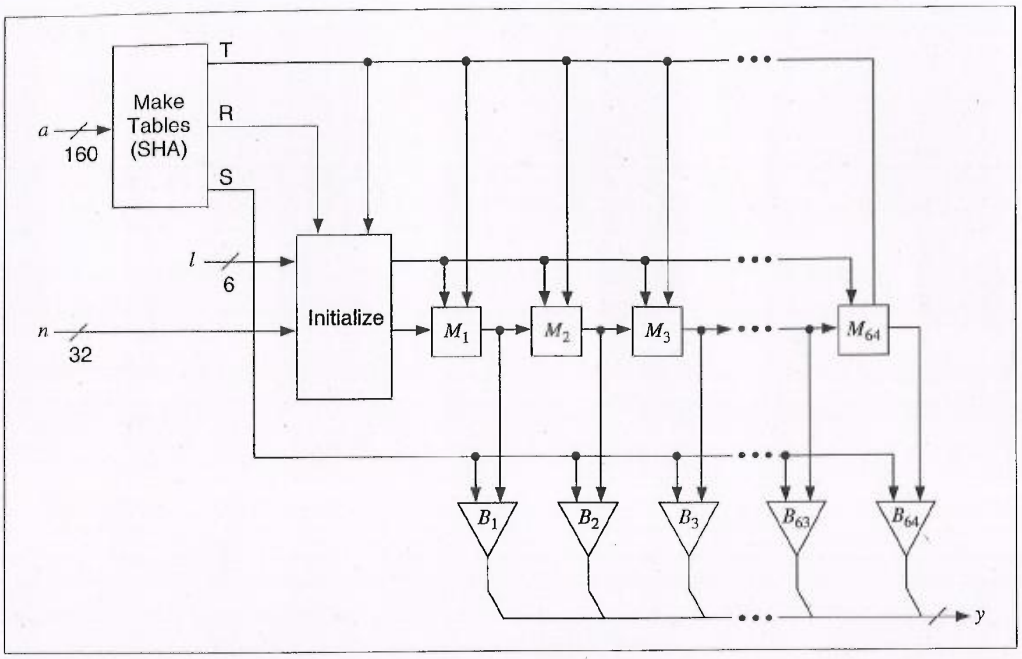

Figure 17.1 The inner loop of SEAL.

SEAL requires about five elementary machine operations to encrypt each byte of text. It runs at 58 megabits per second on a 50 megahertz 486 machine. This is probably the fastest software algorithm in the book.

On the other hand, SEAL must preprocess its key into internal tables. These tables total roughly 3 kilobytes in size, and their calculation takes about 200 SHA computations. Thus, SEAL is not appropriate to use in situations where you don't have the time to perform the key setup or you don't have the memory to store the tables.

Security of SEAL

SEAL is a new algorithm and has yet to be subjected to any published cryptanalysis. This suggests caution. However, SEAL seems to be well thought through. Its peculiarities do, in the end, make a good deal of sense. And Don Coppersmith is generally regarded as the world's cleverest cryptanalyst.

Patents and Licenses

SEAL is being patented [380]. Anyone wishing to license SEAL should contact the Director of Licenses, IBM Corporation, 500 Columbus Ave., Thurnwood, NY, 10594.

17.3 WAKE

WAKE is the Word Auto Key Encryption algorithm, invented by David Wheeler [1589]. It produces a stream of 32-bit words which can be XORed with a plaintext

stream to produce ciphertext, or XORed with a ciphertext stream to produce plaintext. And it's fast.

WAKE works in CFB; the previous ciphertext word is used to generate the next key word. It also uses an S-box of 256 32-bit values. This S-box has a special property: The high-order byte of all the entries is a permutation of all possible bytes, and the low-order 3 bytes are random.

First, generate the S-box entries, S_i, from the key. Then initialize four registers with the key (or with another key): a_0, b_0, c_0, and d_0. To generate a 32-bit keystream word, K_i:

$$K_i = d_i$$

The ciphertext word C_i, is the plaintext word, P_i XORed with K_i.

Then, update the four registers:

$$a_{i+1} = M(a_i, d_i)$$
$$b_{i+1} = M(b_i, a_{i+1})$$
$$c_{i+1} = M(c_i, b_{i+1})$$
$$d_{i+1} = M(d_i, c_{i+1})$$

Function M is

$$M(x,y) = (x + y) >> 8 \oplus S_{(x+y) \wedge 255}$$

This is shown in Figure 17.2. The operation >> is a right shift, not a rotation. The low-order 8 bits of $x + y$ are the input into the S-box. Wheeler gives a procedure for

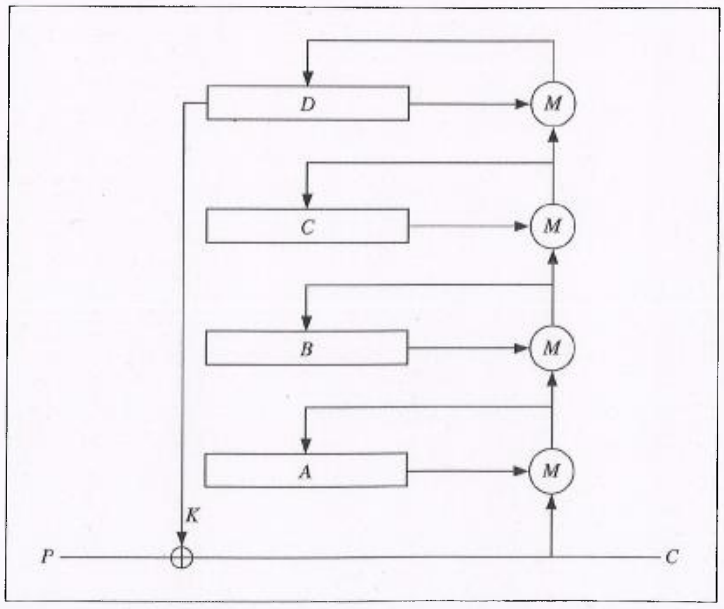

Figure 17.2 Wake.

generating the S-box, but it isn't really complete. Any algorithm to generate random bytes and a random permutation will work.

WAKE's biggest asset is that it is fast. However, it's insecure against a chosen-plaintext or chosen-ciphertext attack. It is being used in the current version of Dr. Solomon's Anti-Virus program.

17.4 FEEDBACK WITH CARRY SHIFT REGISTERS

A feedback with carry shift register, or FCSR, is similar to a LFSR. Both have a shift register and a feedback function; the difference is that a FCSR also has a carry register (see Figure 17.3). Instead of XORing all the bits in the tap sequence, add the bits together and add in the contents of the carry register. The result mod 2 becomes the new bit. The result divided by 2 becomes the new content of the carry register.

Figure 17.4 is an example of a 3-bit FCSR tapped at the first and second bit. Its initial value is 001, and the initial contents of the carry register is 0. The output bit is the right-most bit of the shift register.

Shift Register	Carry Register
0 0 1	0
1 0 0	0
0 1 0	0
1 0 1	0
1 1 0	0
1 1 1	0
0 1 1	1

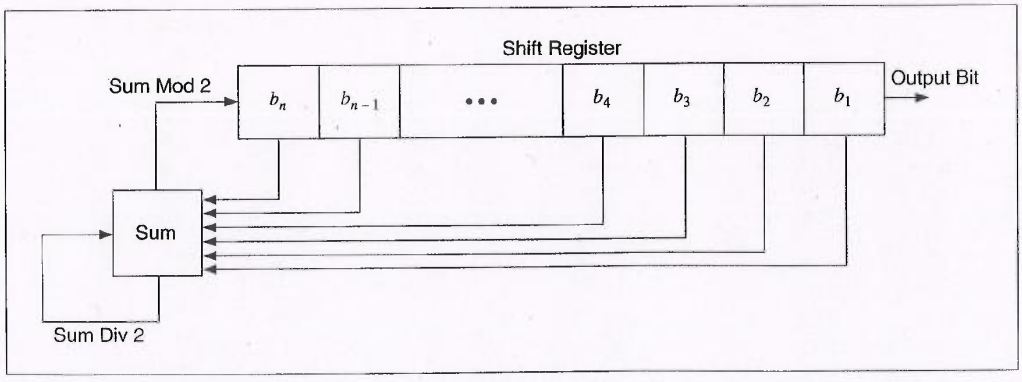

Figure 17.3 Feedback with carry shift register.

1 0 1	1
0 1 0	1
0 0 1	1
0 0 0	1
1 0 0	0

Note that the final internal state (including the contents of the carry register) is the same as the second internal state. The sequence cycles at this point, and has a period of 10.

There are a few things to note here. First, the carry register is not a single bit; it is a number. The size of the carry register must be at least $\log_2 t$, where t is the number of taps. There are only two taps in the previous example, so the carry register only has to be 1 bit wide. If there were four taps, the carry register would have to be 2 bits wide, and could be either 0, 1, 2, or 3.

Second, there is an initial transient before the FCSR settles down into its repeating period. In the previous example, only one state never repeated. For larger and more complicated FCSRs, there may be more.

Third, the maximum period of a FCSR is not $2^n - 1$, where n is the length of the shift register. The maximum period is $q - 1$, where q is the **connection integer**. This number gives the taps and is defined by:

$$q = 2q_1 + 2^2q_2 + 2^4q_4 + \ldots + 2^nq_n - 1$$

(Yes, the q_is are numbered from left to right.) And even worse, q has to be a prime for which 2 is a primitive root. The rest of this discussion assumes q is of this form.

In this example, $q = 2*0 + 4*1 + 8*1 - 1 = 11$. And 11 is a prime with 2 as a primitive root. So the maximum period is 10.

Not all initial states give you the maximum period. For example, look at the FCSR when the initial value is 101 and the carry register is set to 4.

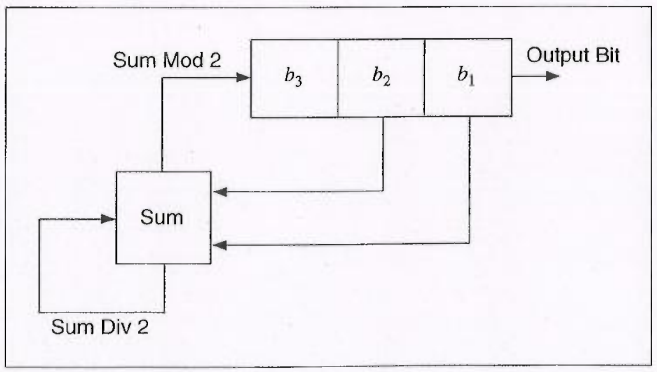

Figure 17.4 3-bit FCSR.

Shift Register	Carry Register
1 0 1	4
1 1 0	2
1 1 1	1
1 1 1	1

At this point the register spits out a neverending stream of 1s.

Any initial state will result in one of four things. First, it is part of the maximum period. Second, it will fall into the maximum period after an initial transient. Third, it will fall into a sequence of all zeros after an initial transient. Fourth, it will fall into a sequence of all ones after an initial transient.

There is a mathematical formula for determining what will happen to a given initial state, but it's much easier to just test it. Run the FCSR for a while. (If m is the initial memory, and t is the number of taps, then $\log_2(t) + \log_2(m) + 1$ steps are enough.) If it degenerates into a neverending stream of 0s or 1s within n bits, where n is the length of the FCSR, don't use it. If it doesn't, then use it. Since the initial state of a FCSR corresponds to the key of the stream cipher, this means that a FCSR-based generator will have a set of weak keys.

Table 17.1 lists all connection integers less than 10,000 for which 2 is a primitive root. These all have maximum period $q - 1$. To turn one of these numbers into a tap sequence, calculate the binary expansion of $q + 1$. For example, 9949 would translate to taps on bits 1, 2, 3, 4, 6, 7, 9, 10, and 13, because

$$9950 = 2^{13} + 2^{10} + 2^9 + 2^7 + 2^6 + 2^4 + 2^3 + 2^2 + 2^1$$

Table 17.2 lists *all* the 4-tap tap sequences that result in a maximal-length FCSR for shift register lengths of 32 bits, 64 bits, and 128 bits. Each of the four values, a, b, c, and d, combine to generate q, a prime for which 2 is primitive.

$$q = 2^a + 2^b + 2^c + 2^d - 1$$

Any of these tap sequences can be used to create a FCSR with period $q - 1$.

The idea of using FCSRs for cryptography is still very new; it is being pioneered by Andy Klapper and Mark Goresky [844,845,654,843,846]. Just as the analysis of LFSRs is based on the addition of primitive polynomials mod 2, analysis of FCSRs is based on addition over something called the **2-adic** numbers. The theory is well beyond the scope of this book, but there seems to be a 2-adic analog for everything. Just as you can define linear complexity, you can define 2-adic complexity. There is even a 2-adic analog to the Berlekamp-Massey algorithm. What this means is that the list of potential stream ciphers has just doubled—at least. Anything you can do with a LFSR you can do with a FCSR.

There are further enhancements to this sort of idea, ones that involve multiple carry registers. The analysis of these sequence generators is based on addition over the ramified extensions of the 2-adic numbers [845,846].

17.5 Stream Ciphers Using FCSRs

There aren't any FCSR stream ciphers in the literature; the theory is still too new. In the interests of getting the ball rolling, I propose some here. I am taking two different tacks: I am proposing FCSR stream ciphers that are identical to previously proposed LFSR generators, and I am proposing stream ciphers that use both FCSRs and LFSRs. The security of the former can probably be analyzed using 2-adic numbers; the latter cannot be analyzed using algebraic techniques—they can probably only be analyzed indirectly. In any case, it is important to choose LFSRs and FCSRs whose periods are relatively prime.

All this will come later. Right now I know of no implementation or analysis of any of these ideas. Wait some years and scan the literature before you trust any of them.

Cascade Generators

There are two ways to use FCSRs in a cascade generator:

- FCSR Cascade. The Gollmann cascade with FCSRs instead of LFSRs.
- LFSR/FCSR Cascade. The Gollmann cascade with the generators alternating between LFSRs and FCSRs.

FCSR Combining Generators

These generators use a variable number of LFSRs and/or FCSRs, and a variety of functions to combine them. The XOR operation destroys the algebraic properties of FCSRs, so it makes sense to use it to combine them. The generator, shown in Figure 17.5, uses a variable number of FCSRs. Its output is the XOR of the outputs of the individual FCSRs.

Other generators along similar lines are:

- FCSR Parity Generator. All registers are FCSRs and the combining function is XOR.
- LFSR/FCSR Parity Generator. Registers are a mix of LFSRs and FCSRs and the combining function is XOR.
- FCSR Threshold Generator. All registers are FCSRs and the combining function is the majority function.
- LFSR/FCSR Threshold Generator. Registers are a mix of LFSRs and FCSRs and the combining function is the majority function.
- FCSR Summation Generator. All registers are FCSRs and the combining function is addition with carry.
- LFSR/FCSR Summation Generator. Registers are a mix of LFSRs and FCSRs and the combining function is addition with carry.

Table 17.1
Connection Integers for Maximal-period FCSRs

2	653	1549	2477	3539
5	659	1571	2531	3547
11	661	1619	2539	3557
13	677	1621	2549	3571
19	701	1637	2557	3581
29	709	1667	2579	3613
37	757	1669	2621	3637
53	773	1693	2659	3643
59	787	1733	2677	3659
61	797	1741	2683	3677
67	821	1747	2693	3691
83	827	1787	2699	3701
101	829	1861	2707	3709
107	853	1867	2741	3733
131	859	1877	2789	3779
139	877	1901	2797	3797
149	883	1907	2803	3803
163	907	1931	2819	3851
173	941	1949	2837	3853
179	947	1973	2843	3877
181	1019	1979	2851	3907
197	1061	1987	2861	3917
211	1091	1997	2909	3923
227	1109	2027	2939	3931
269	1117	2029	2957	3947
293	1123	2053	2963	3989
317	1171	2069	3011	4003
347	1187	2083	3019	4013
349	1213	2099	3037	4019
373	1229	2131	3067	4021
379	1237	2141	3083	4091
389	1259	2213	3187	4093
419	1277	2221	3203	4099
421	1283	2237	3253	4133
443	1291	2243	3299	4139
461	1301	2267	3307	4157
467	1307	2269	3323	4219
491	1373	2293	3347	4229
509	1381	2309	3371	4243
523	1427	2333	3413	4253
541	1451	2339	3461	4259
547	1453	2357	3467	4261
557	1483	2371	3469	4283
563	1493	2389	3491	4349
587	1499	2437	3499	4357
613	1523	2459	3517	4363
619	1531	2467	3533	4373

Table 17.1 (*Cont.*)
Connection Integers for Maximal-period FCSRs

4397	5693	6781	7717	8861
4451	5701	6803	7757	8867
4483	5717	6827	7789	8923
4493	5741	6829	7829	8933
4507	5749	6869	7853	8963
4517	5779	6883	7877	8971
4547	5813	6899	7883	9011
4603	5827	6907	7901	9029
4621	5843	6917	7907	9059
4637	5851	6947	7933	9173
4691	5869	6949	7949	9181
4723	5923	6971	8053	9203
4787	5939	7013	8069	9221
4789	5987	7019	8093	9227
4813	6011	7027	8117	9283
4877	6029	7043	8123	9293
4933	6053	7069	8147	9323
4957	6067	7109	8171	9341
4973	6101	7187	8179	9349
4987	6131	7211	8219	9371
5003	6173	7219	8221	9397
5011	6197	7229	8237	9419
5051	6203	7237	8243	9421
5059	6211	7243	8269	9437
5077	6229	7253	8291	9467
5099	6269	7283	8293	9491
5107	6277	7307	8363	9533
5147	6299	7331	8387	9539
5171	6317	7349	8429	9547
5179	6323	7411	8443	9587
5189	6373	7451	8467	9613
5227	6379	7459	8539	9619
5261	6389	7477	8563	9629
5309	6397	7499	8573	9643
5333	6469	7507	8597	9661
5387	6491	7517	8627	9677
5443	6547	7523	8669	9733
5477	6619	7541	8677	9749
5483	6637	7547	8693	9803
5501	6653	7549	8699	9851
5507	6659	7573	8731	9859
5557	6691	7589	8741	9883
5563	6701	7603	8747	9901
5573	6709	7621	8803	9907
5651	6733	7643	8819	9923
5659	6763	7669	8821	9941
5683	6779	7691	8837	9949

Table 17.2
Tap Sequences for Maximal-length FCSRs

(32, 6, 3, 2)	(64, 24, 19, 2)	(64, 59, 28, 2)	(96, 55, 53, 2)
(32, 7, 5, 2)	(64, 25, 3, 2)	(64, 59, 38, 2)	(96, 56, 9, 2)
(32, 8, 3, 2)	(64, 25, 4, 2)	(64, 59, 44, 2)	(96, 56, 51, 2)
(32, 13, 8, 2)	(64, 25, 11, 2)	(64, 60, 49, 2)	(96, 57, 3, 2)
(32, 13, 12, 2)	(64, 25, 19, 2)	(64, 61, 51, 2)	(96, 57, 17, 2)
(32, 15, 6, 2)	(64, 27, 5, 2)	(64, 63, 8, 2)	(96, 57, 47, 2)
(32, 16, 2, 1)	(64, 27, 16, 2)	(64, 63, 13, 2)	(96, 58, 35, 2)
(32, 16, 3, 2)	(64, 27, 22, 2)	(64, 63, 61, 2)	(96, 59, 46, 2)
(32, 16, 5, 2)	(64, 28, 19, 2)		(96, 60, 29, 2)
(32, 17, 5, 2)	(64, 28, 25, 2)	(96, 15, 5, 2)	(96, 60, 41, 2)
(32, 19, 2, 1)	(64, 29, 16, 2)	(96, 21, 17, 2)	(96, 60, 45, 2)
(32, 19, 5, 2)	(64, 29, 28, 2)	(96, 25, 19, 2)	(96, 61, 17, 2)
(32, 19, 9, 2)	(64, 31, 12, 2)	(96, 25, 20, 2)	(96, 63, 20, 2)
(32, 19, 12, 2)	(64, 32, 21, 2)	(96, 29, 15, 2)	(96, 65, 12, 2)
(32, 19, 17, 2)	(64, 35, 29, 2)	(96, 29, 17, 2)	(96, 65, 39, 2)
(32, 20, 17, 2)	(64, 36, 7, 2)	(96, 30, 3, 2)	(96, 65, 51, 2)
(32, 21, 9, 2)	(64, 37, 2, 1)	(96, 32, 21, 2)	(96, 67, 5, 2)
(32, 21, 15, 2)	(64, 37, 11, 2)	(96, 32, 27, 2)	(96, 67, 25, 2)
(32, 23, 8, 2)	(64, 39, 4, 2)	(96, 33, 5, 2)	(96, 67, 34, 2)
(32, 23, 21, 2)	(64, 39, 25, 2)	(96, 35, 17, 2)	(96, 68, 5, 2)
(32, 25, 5, 2)	(64, 41, 5, 2)	(96, 35, 33, 2)	(96, 68, 19, 2)
(32, 25, 12, 2)	(64, 41, 11, 2)	(96, 39, 21, 2)	(96, 69, 17, 2)
(32, 27, 25, 2)	(64, 41, 27, 2)	(96, 40, 25, 2)	(96, 69, 36, 2)
(32, 29, 19, 2)	(64, 43, 21, 2)	(96, 41, 12, 2)	(96, 70, 23, 2)
(32, 29, 20, 2)	(64, 43, 28, 2)	(96, 41, 27, 2)	(96, 71, 6, 2)
(32, 30, 3, 2)	(64, 45, 28, 2)	(96, 41, 35, 2)	(96, 71, 40, 2)
(32, 30, 7, 2)	(64, 45, 41, 2)	(96, 42, 35, 2)	(96, 72, 53, 2)
(32, 31, 5, 2)	(64, 47, 5, 2)	(96, 43, 14, 2)	(96, 73, 32, 2)
(32, 31, 9, 2)	(64, 47, 21, 2)	(96, 44, 23, 2)	(96, 77, 27, 2)
(32, 31, 30, 2)	(64, 47, 30, 2)	(96, 45, 41, 2)	(96, 77, 31, 2)
	(64, 49, 19, 2)	(96, 47, 36, 2)	(96, 77, 32, 2)
(64, 3, 2, 1)	(64, 49, 20, 2)	(96, 49, 31, 2)	(96, 77, 33, 2)
(64, 14, 3, 2)	(64, 52, 29, 2)	(96, 51, 30, 2)	(96, 77, 71, 2)
(64, 15, 8, 2)	(64, 53, 8, 2)	(96, 53, 17, 2)	(96, 78, 39, 2)
(64, 17, 2, 1)	(64, 53, 43, 2)	(96, 53, 19, 2)	(96, 79, 4, 2)
(64, 17, 9, 2)	(64, 56, 39, 2)	(96, 53, 32, 2)	(96, 81, 80, 2)
(64, 17, 16, 2)	(64, 56, 45, 2)	(96, 53, 48, 2)	(96, 83, 14, 2)
(64, 19, 2, 1)	(64, 59, 5, 2)	(96, 54, 15, 2)	(96, 83, 26, 2)
(64, 19, 18, 2)	(64, 59, 8, 2)	(96, 55, 44, 2)	(96, 83, 54, 2)

Table 17.2 (*Cont.*)
Tap Sequences for Maximal-length FCSRs

(96, 83, 60, 2)	(128, 31, 25, 2)	(128, 81, 55, 2)	(128, 105, 11, 2)
(96, 83, 65, 2)	(128, 33, 21, 2)	(128, 82, 67, 2)	(128, 105, 31, 2)
(96, 83, 78, 2)	(128, 35, 22, 2)	(128, 83, 60, 2)	(128, 105, 48, 2)
(96, 84, 65, 2)	(128, 37, 8, 2)	(128, 83, 61, 2)	(128, 107, 40, 2)
(96, 85, 17, 2)	(128, 41, 12, 2)	(128, 83, 77, 2)	(128, 107, 62, 2)
(96, 85, 31, 2)	(128, 42, 35, 2)	(128, 84, 15, 2)	(128, 107, 102, 2)
(96, 85, 76, 2)	(128, 43, 25, 2)	(128, 84, 43, 2)	(128, 108, 35, 2)
(96, 85, 79, 2)	(128, 43, 42, 2)	(128, 85, 63, 2)	(128, 108, 73, 2)
(96, 86, 39, 2)	(128, 45, 17, 2)	(128, 87, 57, 2)	(128, 108, 75, 2)
(96, 86, 71, 2)	(128, 45, 27, 2)	(128, 87, 81, 2)	(128, 108, 89, 2)
(96, 87, 9, 2)	(128, 49, 9, 2)	(128, 89, 81, 2)	(128, 109, 11, 2)
(96, 87, 44, 2)	(128, 51, 9, 2)	(128, 90, 43, 2)	(128, 109, 108, 2)
(96, 87, 45, 2)	(128, 54, 51, 2)	(128, 91, 9, 2)	(128, 110, 23, 2)
(96, 88, 19, 2)	(128, 55, 45, 2)	(128, 91, 13, 2)	(128, 111, 61, 2)
(96, 88, 35, 2)	(128, 56, 15, 2)	(128, 91, 44, 2)	(128, 113, 59, 2)
(96, 88, 43, 2)	(128, 56, 19, 2)	(128, 92, 35, 2)	(128, 114, 83, 2)
(96, 88, 79, 2)	(128, 56, 55, 2)	(128, 95, 94, 2)	(128, 115, 73, 2)
(96, 89, 35, 2)	(128, 57, 21, 2)	(128, 96, 23, 2)	(128, 117, 105, 2)
(96, 89, 51, 2)	(128, 57, 37, 2)	(128, 96, 61, 2)	(128, 119, 30, 2)
(96, 89, 69, 2)	(128, 59, 29, 2)	(128, 97, 25, 2)	(128, 119, 101, 2)
(96, 89, 87, 2)	(128, 59, 49, 2)	(128, 97, 68, 2)	(128, 120, 9, 2)
(96, 92, 51, 2)	(128, 60, 57, 2)	(128, 97, 72, 2)	(128, 120, 27, 2)
(96, 92, 71, 2)	(128, 61, 9, 2)	(128, 97, 75, 2)	(128, 120, 37, 2)
(96, 93, 32, 2)	(128, 61, 23, 2)	(128, 99, 13, 2)	(128, 120, 41, 2)
(96, 93, 39, 2)	(128, 61, 52, 2)	(128, 99, 14, 2)	(128, 120, 79, 2)
(96, 94, 35, 2)	(128, 63, 40, 2)	(128, 99, 26, 2)	(128, 120, 81, 2)
(96, 95, 4, 2)	(128, 63, 62, 2)	(128, 99, 54, 2)	(128, 121, 5, 2)
(96, 95, 16, 2)	(128, 67, 41, 2)	(128, 99, 56, 2)	(128, 121, 67, 2)
(96, 95, 32, 2)	(128, 69, 33, 2)	(128, 99, 78, 2)	(128, 121, 95, 2)
(96, 95, 44, 2)	(128, 71, 53, 2)	(128, 100, 13, 2)	(128, 121, 96, 2)
(96, 95, 45, 2)	(128, 72, 15, 2)	(128, 100, 39, 2)	(128, 123, 40, 2)
	(128, 72, 41, 2)	(128, 101, 44, 2)	(128, 123, 78, 2)
(128, 5, 4, 2)	(128, 73, 5, 2)	(128, 101, 97, 2)	(128, 124, 41, 2)
(128, 15, 4, 2)	(128, 73, 65, 2)	(128, 103, 46, 2)	(128, 124, 69, 2)
(128, 21, 19, 2)	(128, 73, 67, 2)	(128, 104, 13, 2)	(128, 124, 81, 2)
(128, 25, 5, 2)	(128, 75, 13, 2)	(128, 104, 19, 2)	(128, 125, 33, 2)
(128, 26, 11, 2)	(128, 80, 39, 2)	(128, 104, 35, 2)	(128, 125, 43, 2)
(128, 27, 25, 2)	(128, 80, 53, 2)	(128, 105, 7, 2)	(128, 127, 121, 2)

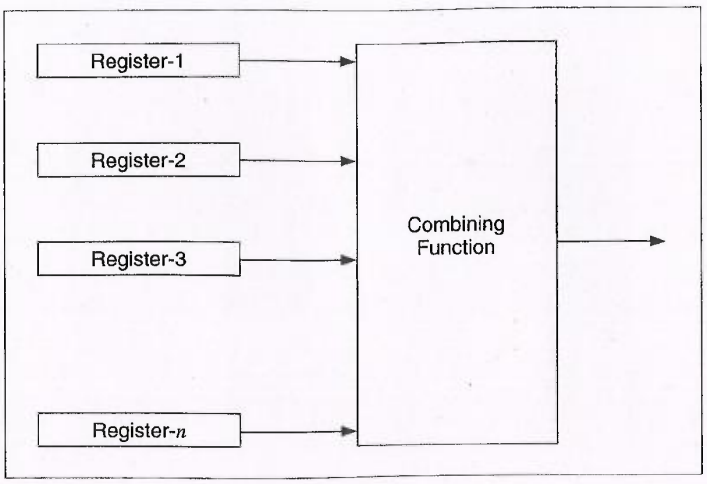

Figure 17.5 Combining Generators.

LFSR/FCSR Summation/Parity Cascade

The theory is that addition with carry destroys the algebraic properties of LFSRs, and that XOR destroys the algebraic properties of FCSRs. This generator combines those ideas, as used in the LFSR/FCSR Summation Generator and the LFSR/FCSR Parity Generator just listed, with the Gollmann cascade.

The generator is a series of arrays of registers, with the clock of each array controlled by the output of the previous array. Figure 17.6 is one stage of this generator. The first array of LFSRs is clocked and the results are combined using addition with carry. If the output of this combining function is 1, then the next array (of FCSRs) is clocked and the output of those FCSRs is combined with the output of the previous combining function using XOR. If the output of the first combining function is 0, then the array of FCSRs is not clocked and the output is simply added to the carry from the previous round. If the output of this second combining function is 1, then the third array of LFSRs is clocked, and so on.

This generator uses a lot of registers: $n*m$, where n is the number of stages and m is the number of registers per stage. I recommend $n = 10$ and $m = 5$.

Alternating Stop-and-Go Generators

These generators are stop-and-go generators with FCSRs instead of some LFSRs. Additionally, the XOR operation can be replaced with an addition with carry (see Figure 17.7).

- FCSR Stop-and-Go Generator. Register-1, Register-2, and Register-3 are FCSRs. The combining operation is XOR.
- FCSR/LFSR Stop-and-Go Generator. Register-1 is a FCSR, and Registers-2 and -3 are LFSRs. The combining operation is addition with carry.

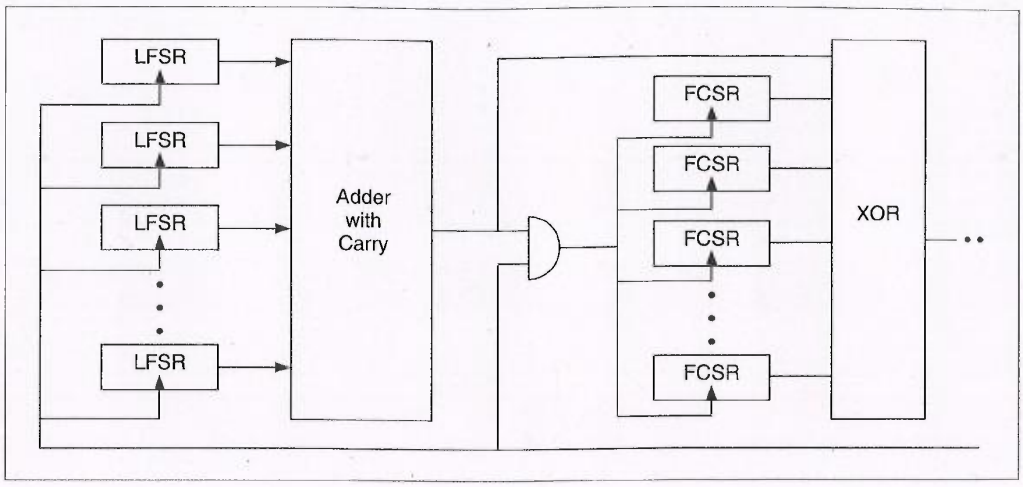

Figure 17.6 Concoction Generator.

— LFSR/FCSR Stop-and-Go Generator. Register-1 is a LFSR, and Registers-2 and -3 are FCSRs. The combining operation is XOR.

Shrinking Generators

There are four basic generator types using FCSRs:

— FCSR Shrinking Generator. A shrinking generator with FCSRs instead of LFSRs.
— FCSR/LFSR Shrinking Generator. A shrinking generator with a LFSR shrinking a FCSR.
— LFSR/FCSR Shrinking Generator: A shrinking generator with a FCSR shrinking a LFSR.

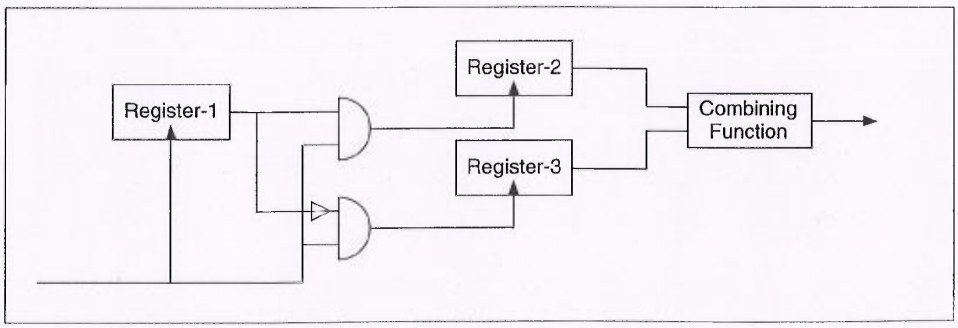

Figure 17.7 Alternating stop-and-go generators.

— FCSR Self-Shrinking Generator. A self-shrinking generator with a FCSR instead of a LFSR.

17.6 NONLINEAR-FEEDBACK SHIFT REGISTERS

It is easy to imagine a more complicated feedback sequence than the ones used in LFSRs or FCSRs. The problem is that there isn't any mathematical theory that can analyze them. You'll get something, but who knows what it is? In particular, here are some problems with nonlinear-feedback shift register sequences.

— There may be biases, such as more ones than zeros or fewer runs than expected, in the output sequence.
— The maximum period of the sequence may be much lower than expected.
— The period of the sequence might be different for different starting values.
— The sequence may appear random for a while, but then "dead end" into a single value. (This can easily be solved by XORing the nonlinear function with the rightmost bit.)

On the plus side, if there is no theory to analyze nonlinear-feedback shift registers for security, there are few tools to cryptanalyze stream ciphers based on them. We can use nonlinear-feedback shift registers in stream-cipher design, but we have to be careful.

In a nonlinear-feedback shift register, the feedback function can be anything you want (see Figure 17.8).

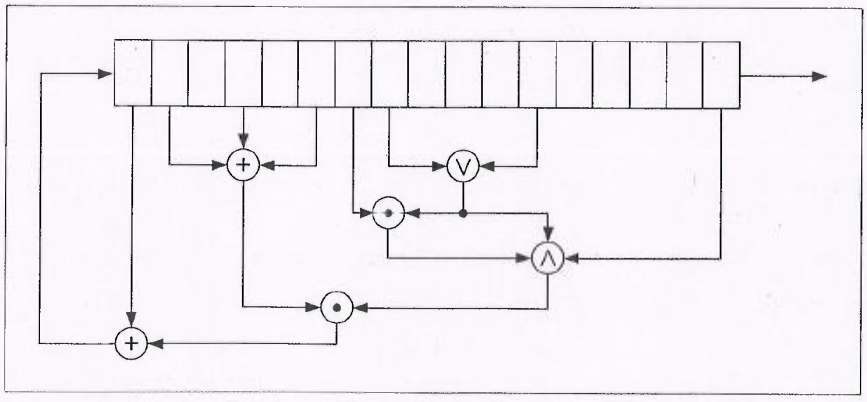

Figure 17.8 A nonlinear-feedback shift register (probably insecure).

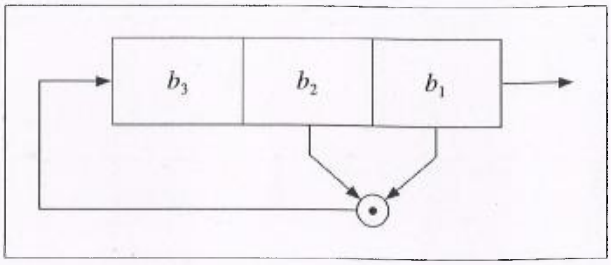

Figure 17.9 3-bit nonlinear feedback shift register.

Figure 17.9 is a 3-bit shift register with the following feedback function: The new bit is the first bit times the second bit. If it is initialized with the value 110, it produces the following sequence of internal states:

1 1 0
0 1 1
1 0 1
0 1 0
0 0 1
0 0 0
0 0 0

And so on forever.

The output sequence is the string of least significant bits:

0 1 1 0 1 0 0 0 0 0 0 0 . . . .

This isn't terribly useful.

It gets even worse. If the initial value is 100, it produces 010, 001, then repeats forever at 000. If the initial value is 111, it repeats itself forever right from the start.

Some work has been done on computing the linear complexity of the product of two LFSRs [1650,726,1364,630,658,659]. A construction that involved computing LFSRs over a field of odd characteristic [310] is insecure [842].

17.7 Other Stream Ciphers

Many other stream ciphers have appeared in the literature here and there. Here are some of them.

Pless Generator

This generator is designed around the capabilities of the J-K flip-flop [1250]. Eight LFSRs drive four J-K flip-flops; each flip-flop acts as a nonlinear combiner for two of

the LFSRs. To avoid the problem that knowledge of an output of the flip-flop identifies both the source and value of the next output bit, clock the four flip-flops and then interleave the outputs to yield the final keystream.

This algorithm has been cryptanalyzed by attacking each of the four flip-flops independently [1356]. Additionally, combining J-K flip-flops is cryptographically weak; generators of this type succumb to correlation attacks [1451].

Cellular Automaton Generator

In [1608,1609], Steve Wolfram proposed using a one-dimensional cellular automaton as a pseudo-random-number generator. Cellular automata is not the subject of this book, but Wolfram's generator consisted of a one-dimensional array of bits, a_1, a_2, a_3, . . . , a_k, . . . , a_n, and an update function:

$$a'_k = a_{k-1} \oplus (a_k \vee a_{k+1})$$

The bit is extracted from one of the a_k values; which one really doesn't matter.

The generator's behavior appears to be quite random. However, there is a known-plaintext attack against these generators [1052]. This attack works on a PC with values of n up to 500 bits. Additionally, Paul Bardell proved that the output of a cellular automaton can also be generated by a linear-feedback shift register of equal length and is therefore no more secure [83].

1/p Generator

This generator was proposed, and then cryptanalyzed, in [193]. If the internal state of the generator at time t is x_t, then

$$x_{t+1} = bx_t \bmod p$$

The output of the generator is the least significant bit of x_t div p, where div is the truncated integer division. For maximum period, the constants b and p should be chosen so that p is prime and b is a primitive root mod p. Unfortunately, this generator isn't secure. (Note that for $b = 2$, an FCSR with a connection integer p outputs the reverse of this sequence.)

crypt(1)

The original UNIX encryption algorithm, crypt(1), is a stream cipher based on the same ideas as the Enigma. This is a 256-element, single-rotor substitution cipher with a reflector. Both the rotor and the reflector are generated from the key. This algorithm is far simpler than the World War II German Enigma and, for a skilled cryptanalyst, very easy to break [1576,1299]. A public-domain UNIX program, called Crypt Breakers Workbench (CBW), can be used to break files encrypted with crypt(1).

Other Schemes

Another generator is based on the knapsack problem (see Section 19.2) [1363]. CRYPTO-LEGGO is insecure [301]. Joan Daemen has developed SubStream, Jam, and StepRightUp [402]; they are all too new to comment on. Many other algorithms

are described in the literature, and even more are kept secret and incorporated into equipment.

17.8 System-Theoretic Approach to Stream-Cipher Design

In practice, stream-cipher design is a lot like block-cipher design. It involves more mathematical theory, but in the end a cryptographer proposes a design and then tries to analyze it.

According to Rainer Rueppel, there are four different approaches to the construction of stream ciphers [1360,1362]:

- System-theoretic approach. Try to make sure that each design creates a difficult and unknown problem for the cryptanalyst, using a set of fundamental design principles and criteria.
- Information-theoretic approach. Try to keep the cryptanalyst in the dark about the plaintext. No matter how much work the cryptanalyst invests, he will never get a unique solution.
- Complexity-theoretic approach. Try to base the cryptosystem on, or make it equivalent to, some known and difficult problem such as factoring or taking discrete logarithms.
- Randomized approach. Try to generate an unmanageably large problem by forcing the cryptanalyst to examine lots of useless data in his attempts at cryptanalysis.

The approaches differ in their assumptions about the capabilities and opportunities of the cryptanalyst, the definition of cryptographic success, and the notion of security. Most of the research in this field is theoretical, but there are some good stream ciphers among the impractical ones.

The system-theoretic approach was used in all the stream ciphers previously listed; it produces most of the stream ciphers that are practical enough to be used in the real world. A cryptographer designs keystream generators that have testable security properties—period, distribution of bit patterns, linear complexity, and so on—and not ciphers based on mathematical theory. The cryptographer also studies various cryptanalytic techniques against these generators and makes sure the generators are immune to these attacks.

Over the years, the approach has resulted in a set of design criteria for stream ciphers [1432,99,1357,1249]. These were discussed by Rueppel in [1362], in which he details the theory behind them.

- Long period, no repetitions.
- Linear complexity criteria—large linear complexity, linear complexity profile, local linear complexity, and so forth.

- Statistical criteria such as ideal k-tuple distributions.
- Confusion—every keystream bit must be a complex transformation of all or most of the key bits.
- Diffusion—redundancies in substructures must be dissipated into long-range statistics.
- Nonlinearity criteria for Boolean functions like mth-order correlation immunity, distance to linear functions, avalanche criterion, and so on.

This list of design criteria is not unique for stream ciphers designed by the system-theoretic approach; it is true for all stream ciphers. It is even true for all block ciphers. The unique point about the system-theoretic approach is that stream ciphers are designed to satisfy these goals directly.

The major problem with these cryptosystems is that nothing can be proven about their security; the design criteria have never been proved to be either necessary or sufficient for security. A keystream generator may satisfy all the design principles, but could still turn out to be insecure. Another could turn out to be secure. There is still some magic to the process.

On the other hand, breaking each of these keystream generators is a different problem for a cryptanalyst. If enough different generators are out there, it may not be worth the cryptanalyst's time to try to break each one. He may better achieve fame and glory by figuring out better ways to factor large numbers or calculating discrete logarithms.

17.9 COMPLEXITY-THEORETIC APPROACH TO STREAM-CIPHER DESIGN

Rueppel also delineated a complexity-theoretic approach to stream-cipher design. Here, a cryptographer attempts to use complexity theory to prove that his generators are secure. Consequently, the generators tend to be more complicated, based on the same sorts of hard problems as public-key cryptography. And like public-key algorithms, they tend to be slow and cumbersome.

Shamir's Pseudo-Random-Number Generator

Adi Shamir used the RSA algorithm as a pseudo-random-number generator [1417]. While Shamir showed that predicting the output of the pseudo-random-number generator is equivalent to breaking RSA, potential biases in the output were demonstrated in [1401,200].

Blum-Micali Generator

This generator gets its security from the difficulty of computing discrete logarithms [200]. Let g be a prime and p be an odd prime. A key x_0 starts off the process:

$$x_{i+1} = g^{x_i} \bmod p$$

The output of the generator is 1 if $x_i < (p-1)/2$, and 0 otherwise.

If p is large enough so that computing discrete logarithms mod p is infeasible, then this generator is secure. Additional theoretical results can be found in [1627, 986,985,1237,896,799].

RSA

This RSA generator [35,36] is a modification of [200]. The initial parameters are a modulus N which is the product of two large primes p and q, an integer e which is relatively prime to $(p-1)(q-1)$, and a random seed x_0, where x_0 is less than N.

$$x_{i+1} = x_i^e \bmod N$$

The output of the generator is the least significant bit of x_i. The security of this generator is based on the difficulty of breaking RSA. If N is large enough, then the generator is secure. Additional theory can be found in [1569,1570,1571,30,354].

Blum, Blum, and Shub

The simplest and most efficient complexity-theoretic generator is called the Blum, Blum, and Shub generator, after its inventors. Mercifully, we shall abbreviate it to BBS, although it is sometimes called the quadratic residue generator [193].

The theory behind the BBS generator has to do with quadratic residues modulo n (see Section 11.3). Here's how it works.

First find two large prime numbers, p and q, which are congruent to 3 modulo 4. The product of those numbers, n, is a Blum integer. Choose another random integer, x, which is relatively prime to n. Compute

$$x_0 = x^2 \bmod n$$

That's the seed for the generator.

Now you can start computing bits. The ith pseudo-random bit is the least significant bit of x_i, where

$$x_i = x_{i-1}^2 \bmod n$$

The most intriguing property of this generator is that you don't have to iterate through all $i-1$ bits to get the ith bit. If you know p and q, you can compute the ith bit directly.

b_i is the least significant bit of x_i, where $x_i = x_0^{(2^i) \bmod ((p-1)(q-1))}$

This property means you can use this cryptographically strong pseudo-random-bit generator as a stream cryptosystem for a random-access file.

The security of this scheme rests on the difficulty of factoring n. You can make n public, so anyone can generate bits using the generator. However, unless a cryptanalyst can factor n, he can never predict the output of the generator—not even with a statement like: "The next bit has a 51 percent chance of being a 1."

More strongly, the BBS generator is **unpredictable to the left** and **unpredictable to the right**. This means that given a sequence generated by the generator, a cryptanalyst cannot predict the next bit in the sequence nor the previous bit in the sequence. This is not security based on some complicated bit generator that no one understands, but the mathematics behind factoring n.

This algorithm is slow, but there are speedups. As it turns out, you can use more than the least significant bit of each x_i as a pseudo-random bit. According to [1569, 1570,1571,35,36], if n is the length of x_i, the least significant $\log_2 n$ bits of x_i can be used. The BBS generator is comparatively slow and isn't useful for stream ciphers. However, for high-security applications, such as key generation, this generator is the best of the lot.

17.10 OTHER APPROACHES TO STREAM-CIPHER DESIGN

In an information-theoretic approach to stream ciphers, the cryptanalyst is assumed to have unlimited time and computing power. The only practical stream cipher that is secure against an adversary like this is a one-time pad (see Section 1.5). Since bits would be impractical on a pad, this is sometimes called a **one-time tape**. Two magnetic tapes, one at the encryption end and the other at the decryption end, would have the same random keystream on them. To encrypt, simply XOR the plaintext with the bits on the tape. To decrypt, XOR the ciphertext with the bits on the other, identical, tape. You never use the same keystream bits twice. Since the keystream bits are truly random, no one can predict the keystream. If you burn the tapes when you are through with them, you've got perfect secrecy (assuming no one else has copies of the tape).

Another information-theoretic stream cipher, developed by Claus Schnorr, assumes that the cryptanalyst only has access to a limited number of ciphertext bits [1395]. The results are highly theoretical and have no practical value, at least not yet. For more details, consult [1361,1643,1193].

In a randomized stream cipher, the cryptographer tries to ensure that the cryptanalyst has an infeasibly large problem to solve. The objective is to increase the number of bits the cryptanalyst has to work with, while keeping the secret key small. This can be done by making use of a large public random string for encryption and decryption. The key would specify which parts of the large random string are to be used for encryption and decryption. The cryptanalyst, not knowing the key, is forced to pursue a brute-force search through the random string. The security of this sort of cipher can be expressed by the average number of bits a cryptanalyst must examine before the chances of determining the key improve over pure guessing.

Rip van Winkle Cipher

James Massey and Ingemar Ingemarsson proposed the Rip van Winkle cipher [1011], so named because the receiver has to receive 2^n bits of ciphertext before attempting decryption. The algorithm, illustrated in Figure 17.10, is simple to implement, provably secure, and completely impractical. Simply XOR the plaintext with the keystream, and delay the keystream by 0 to 20 years—the exact delay is part of the key. In Massey's words: "One can easily guarantee that the enemy cryptanalyst will need thousands of years to break the cipher, if one is willing to wait millions of years to read the plaintext." Further work on this idea can be found in [1577,755].

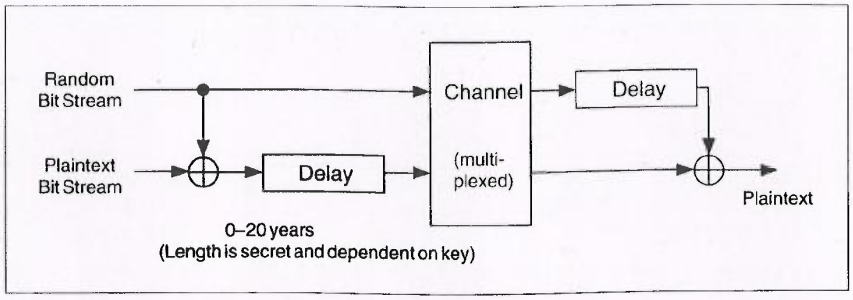

Figure 17.10 Rip van Winkle cipher.

Diffie's Randomized Stream Cipher

This scheme was first proposed by Whitfield Diffie [1362]. The data are 2^n random sequences. The key is k, a random n-bit string. To encrypt a message, Alice uses the kth random string as a one-time pad. She then sends the ciphertext plus the 2^n random strings over $2^n + 1$ different communications channels.

Bob knows k, so he can easily choose which one-time pad to decrypt the message with. Eve has no choice but to examine the random sequences one at a time until she finds the correct one-time pad. Any attack must examine an expected number of bits which is in $O(2^n)$. Rueppel points out that if you send n random strings instead of 2^n, and if the key is used to specify a linear combination of those random strings, the security is the same.

Maurer's Randomized Stream Cipher

Ueli Maurer described a scheme based on XORing the plaintext with several large public random-bit sequences [1034,1029,1030]. The key is the set of starting positions within each sequence. This turns out to be provably almost secure, with a calculable probability of being broken based on how much memory the attacker has at his disposal, without regard to the amount of computing power he has. Maurer claims that this scheme would be practical with about 100 different sequences of 10^{20} random bits each. Digitizing the face of the moon might be one way to get this many bits.

17.11 Cascading Multiple Stream Ciphers

If performance is no issue, there's no reason not to choose multiple stream ciphers and cascade them. Simply XOR the output of each generator with the plaintext to get the ciphertext. Ueli Maurer's result (see Section 15.7) says that if the generators have independent keys, then the security of the cascade is at least as secure as the strongest algorithm in the cascade. It is probably much more secure than that.

Stream ciphers can be combined in all the same ways as block ciphers (see Chapter 15). Stream ciphers can be cascaded (see Section 15.7) with other stream ciphers, or together with block ciphers.

A clever trick is to use one algorithm, either a block or stream algorithm, to frequently rekey a fast stream algorithm (which could even be a block algorithm in OFB mode). The fast algorithm could be weak, since a cryptanalyst would never see very much plaintext encrypted with any one key.

There's a trade-off between the size of the fast algorithm's internal state (which may impact security) and how often you can afford to rekey. The rekey needs to be relatively fast; algorithms that have a long key setup routine aren't suitable for this kind of application. And the rekeying should be independent of the internal state of the fast algorithm.

17.12 CHOOSING A STREAM CIPHER

If the study of stream ciphers offers any lessons, it's that new types of attacks are invented with alarming regularity. Classically, stream ciphers have been based on considerable mathematical theory. This theory can be used to prove good properties about the cipher, but can also be used to find new attacks against the cipher. I worry about any stream cipher based solely on LFSRs for this reason.

I prefer stream ciphers that are designed more along the lines of block ciphers: nonlinear transformations, large S-boxes, and so on. RC4 is my favorite, and SEAL is a close second. I would be very interested in seeing cryptanalytic results against my generators that combine LFSRs and FCSRs; this seems to be a very fruitful area of stream-cipher research to mine for actual designs. Or, you can use a block cipher in OFB or CFB to get a stream cipher.

Table 17.3 gives some timing measurements for some algorithms. These are meant for comparison purposes only.

17.13 GENERATING MULTIPLE STREAMS FROM A SINGLE PSEUDO-RANDOM-SEQUENCE GENERATOR

If you need to encrypt multiple channels of communications in a single box—a multiplexer, for example—the easy solution is to use a different pseudo-random-sequence generator for each stream. This has two problems: It requires more hardware, and all the different generators have to be synchronized. It would be simpler to use a single generator.

Table 17.3
Encryption Speeds of Some Stream Ciphers on a 33MHz 486SX

Algorithm	Encryption Speed (Megabytes/Second)
A5	5
PIKE	62
RC4	164
SEAL	381

One solution is to clock the generator multiple times. If you want three independent streams, clock the generator three times and send 1 bit into each stream. This technique works, but you may have trouble clocking the generator as fast as you would like. For example, if you can only clock the generator three times as fast as the data stream, you can only create three streams. Another way is to use the same sequence for each channel—perhaps with a variable time delay. This is insecure.

A really clever idea [1489], patented by the NSA, is shown in Figure 17.11. Dump the output of your favorite generator into an m-bit simple shift register. At each clock pulse, shift the register one to the right. Then, for each output stream, AND the register with a different m-bit control vector viewed as a unique identifier for the desired output stream, then XOR all the bits together to get the output bit for that stream. If you want several output streams in parallel, you need a separate control vector and an XOR/AND logic array for each output stream.

There are some things to watch out for. If any of the streams are linear combinations of other streams, then the system can be broken. But if you are clever, this is an easy and secure way to solve the problem.

17.14 Real Random-Sequence Generators

Sometimes cryptographically secure pseudo-random numbers are not good enough. Many times in cryptography, you want real random numbers. Key generation is a prime example. It's fine to generate random cryptographic keys based on a pseudo-

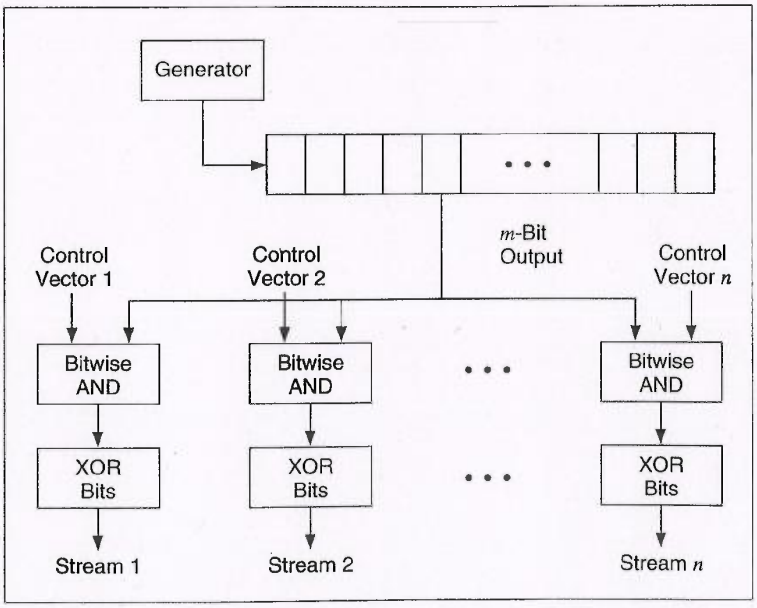

Figure 17.11 Multiple-bit generator.

random sequence generator, but if an adversary gets a copy of that generator and the master key, the adversary can create the same keys and break your cryptosystem, no matter how secure your algorithms are. A random-sequence generator's sequences cannot be reproduced. No one, not even you, can reproduce the bit sequence out of those generators.

There is a large philosophical debate over whether any of these techniques actually produces real random bits. I am not going to address that debate. The point here is to produce bits that have the same statistical properties as random bits and are not reproducible.

The important thing about any real random-sequence generator is that it be tested. There is a wealth of literature on this topic. Tests of randomness can be found in [863,99]. Maurer showed that all these tests can be derived from trying to compress the sequence [1031,1032]. If you can compress a random sequence, then it is not truly random.

Anyhow, what we have here is a whole lot of black magic. The primary point is to generate a sequence of bits that your adversary is unlikely to guess. It doesn't sound like much, but it's harder than you think. I can't prove that any of these techniques generates random bits. These techniques produce a sequence of bits that cannot be easily reproduced. For some details, see [1375,1376,511].

RAND Tables

Back in 1955, when computers were still new, the Rand Corporation published a book that contained a million random digits [1289]. Their method is described in the book:

> The random digits in the book were produced by rerandomization of a basic table generated by an electronic roulette wheel. Briefly, a random frequency pulse source, providing on the average about 100,000 pulses per second, was gated about once per second by a constant frequency pulse. Pulse standardization circuits passed the pulses through a 5-place binary counter. In principle the machine was a 32-place roulette wheel which made, on the average, about 3000 revolutions per trial and produced one number per second. A binary-to-decimal converter was used which converted 20 of the 32 numbers (the other twelve were discarded) and retained only the final digit of two-digit numbers; this final digit was fed into an IBM punch to produce finally a punched card table of random digits.

The book goes on to discuss the results of various randomness tests on the data. It also suggests how to use the book to find a random number:

> The lines of the digit table are numbered from 00000 to 19999. In any use of the table, one should first find a random starting position. A common procedure for doing this is to open the book to an unselected page of the digit table and blindly choose a five-digit number; this number with the first digit reduced modulo 2 determines the starting line; the two digits to the right of the initially selected five-digit number are reduced modulo 50 to determine the starting column in the starting line. To guard against the tendency of books to open repeatedly at the same page and the natural tendency of a person to choose a number toward the center of the

page: every five-digit number used to determine a starting position should be marked and not used a second time for this purpose.

The meat of the book is the "Table of Random Digits." It lists them in 5-digit groups—"10097 32533 76520 13586 . . ."—50 on a line and 50 lines on a page. The table goes on for 400 pages and, except for a particularly racy section on page 283 which reads "69696," makes for a boring read. The book also includes a table of 100,000 normal deviates.

The interesting thing about the RAND book is not its million random digits, but that they were created before the computer revolution. Many cryptographic algorithms use arbitrary constants—so-called "magic numbers." Choosing magic numbers from the RAND tables ensures that they haven't been specially chosen for some nefarious reason. Khafre does this, for example.

Using Random Noise

The best way to collect a large number of random bits is to tap the natural randomness of the real world. Often this method requires specialized hardware, but you can play tricks with computers.

Find an event that happens regularly but randomly: atmospheric noise peaking at a certain threshold, a toddler falling while learning to walk, or some such. Measure the time interval between one event and the next event. Record it. Measure the time interval between the second event and the third event. Record it as well. If the first time interval is greater than the second, output 1 as the bit. If the second time interval is greater than the first, output 0 as the event. Do it again for the next event.

Throw a dart at the New York Stock Exchange closing prices in your local newspaper. Compare the closing price of the stock you hit with the closing price of the stock directly above it. If the one you hit is more, output 0; if it less, output 1.

Hook a Geiger counter up to your computer, count emissions over a fixed time interval, and keep the least significant bit. Or measure the time between successive ticks. (Since the radioactive source is decaying, the average time between successive ticks is continuously getting longer. You want to choose a source with the half life long enough to make this negligible—like plutonium. Or, if you're worried about your health, you can apply appropriate statistical corrections.)

G. B. Agnew proposed a real random-bit generator, suitable for integration into a VLSI device [21]. It is a metal insulator semiconduction capacitor (MISC). Two of them are placed in close proximity, and the random bit is a function of the difference in charge between the two. Another random-number generator generates a random-bit stream based on the frequency instability in a free-running oscillator [535]. A commercial chip from AT&T generates random numbers from the same phenomenon [67]. M. Gude built a random-number generator that collected random bits from physical phenomena, such as radioactive decay [668,669]. Manfield Richter developed a random-number generator based on thermal noise from a semiconductor diode [1309].

Supposedly the time intervals between successive 2e4 light emissions from a trapped mercury atom are random. Use that. Better yet, find a semiconductor company that makes random-number-generation chips; they are out there.

There is also a random-number generator that uses the computer's disk drive [439]. It measures the time required to read a disk block and uses the variation in that time as a random number source. It filters the timing data to remove structure that comes from quantization, then applies a fast Fourier transform to vectors of the numbers. This removes bias and correlation. Finally, it uses the spectral angles for frequencies in $(0, \pi)$, normalized to the unit interval, as the random bits. A large part of the variation in disk rotation speed is caused by air turbulence, so there is randomness in the system. There are caveats, though. If you keep too many bits of the output, you are using the fast Fourier transform as a random-number generator and risk predictability. And it's best to read the same disk block over and over, so that your filtering doesn't have to remove structure that comes from the disk-scheduler. An implementation of this system was able to collect about 100 bits per minute [439].

Using the Computer's Clock

If you want a single random bit (or even a few), take the least significant bit from any clock register. This might not be terribly random in a UNIX system because of various potential synchronizations, but it works on some personal computers.

Beware of getting too many bits this way. Executing the same subroutine several times in succession could easily skew bits generated in this manner. For example, if each bit generation subroutine takes an even number of clock ticks to execute, you will get an endless stream of the same bit out of the generator. If each subroutine takes an odd number of clock ticks to execute, you will get an endless stream of alternating bits out of the generator. Even if the resonance isn't this obvious, the resultant bit stream will be far from random.

One random-number generator works this way [918]:

> Our truly random number generator . . . works by setting an alarm and then incrementing a counter register rapidly in the CPU until an interrupt occurs. The contents of the register are then XORed with the contents of an output buffer byte (truncating the register's data to 8 bits). After each byte of the output buffer is filled, the buffer is further processed by doing a right, circular shift of each character by 2 bits. This has the effect of moving the most active (and random) least significant bits into the most significant positions. The entire process is then repeated 3 times. Finally each character of the buffer has been touched by the two most random bits of the counter register after interrupts. That is $4n$ interrupts have occurred where n is the number of desired random bytes.

This method is very sensitive to the randomness of system interrupts and the granularity of the clock. The output looked pretty good when tested on real UNIX machines.

Measuring Keyboard Latency

People's typing patterns are both random and nonrandom. They are nonrandom enough that they can be used as a means of identification, but they are random enough that they can be used to generate random bits. Measure the time between

successive keystrokes, then take the least significant bits of those measurements. These bits are going to be pretty random. This technique may not work on a UNIX terminal, since the keystrokes pass through filters and other mechanisms before they get to your program, but it will work on most personal computers.

Ideally, you only want to collect one random bit per keystroke. Collecting more may skew the results, depending on how good a typist is sitting at the keyboard. This technique is limited, though. While it's easy to have someone type 100 words or so when it is time to generate a key, it isn't reasonable to ask the typist to type a 100,000-word essay to generate a keystream for a one-time pad.

Biases and Correlations

A major problem with all these systems is that there could be nonrandomness in the generated sequence. The underlying physical processes might be random, but many kinds of measuring instruments are between the digital part of the computer and the physical process. Those instruments could easily introduce problems.

A way to eliminate **bias**, or skew, is to XOR several bits together. If a random bit is biased toward 0 by a factor e, then the probability of 0 can be written as:

$$P(0) = .5 + e$$

XORing two of these bits together yields:

$$P(0) = (.5 + e)^2 + (.5 - e)^2 = .5 + 2e^2$$

By the same calculation, XORing 4 bits together yields:

$$P(0) = .5 + 8e^4$$

XORing m bits will exponentially converge to an equal probability of 0 and 1. If you know the maximum bias you are willing to accept for your application, you can calculate how many bits you need to XOR together to get random bits below that bias.

An even better method is to look at the bits in pairs. If the 2 bits are the same, discard them and look at the next pair. If the 2 bits are different, take the first bit as the output of the generator. This eliminates bias completely. Other techniques for reducing bias use transition mappings, compression, and fast Fourier transforms [511].

The potential problem with both methods is that if there is a **correlation** between adjacent bits, then these methods will increase the bias. One way to correct this is to use multiple random sources. Take four different random sources and XOR the bits together; or take two random sources, and look at those bits in pairs.

For example, take a radioactive source and hook a Geiger counter to your computer. Take a pair of noisy diodes and record as an event every time the noise exceeds a certain peak. Measure atmospheric noise. Get a random bit from each and XOR them together to produce the random bit. The possibilities are endless.

The mere fact that a random-number generator has a bias does not necessarily mean that it is unusable. It just means that it is less secure. For example, consider the problem of Alice generating a triple-DES 168-bit key. All she has is a random-bit generator with a bias toward 0: It produces 55 percent 0s and 45 percent 1s. This means that there are only 0.99277 bits of entropy per key bit, as opposed to 1 bit of

entropy if the generator were perfect. Mallory, trying to break the key, can optimize his brute-force search to try the most probable key first (000 . . . 0), and work toward the least probable key (111 . . . 1). Because of the bias, Mallory can expect to find the key in 2^{109} attempts. If there were no bias, Mallory would expect to make 2^{111} attempts. The resultant key is less secure, but not appreciably so.

Distilling Randomness

In general, the best way to generate random numbers is to find a whole lot of seemingly random events and distill randomness from them. This randomness can then be stored in a pool or reservoir that applications can draw on as needed. One-way hash functions are ready-made for the job; they're fast, so you can shovel quite a bit through them without worrying too much about performance or the actual randomness of each observation. Hash almost anything you can find that has at least some randomness. Try:

- A copy of every keystroke
- Mouse commands
- The sector number, time of day, and seek latency for every disk operation
- Actual mouse position
- Number of current scanline of monitor
- Contents of the actually displayed image
- Contents of FATs, kernel tables, and so on
- Access/modify times of /dev/tty
- CPU load
- Arrival times of network packets
- Input from a microphone
- /dev/audio without a microphone attached

If your system uses separate crystal oscillators for its CPU and time-of-day clocks, try reading the time of day in a tight loop. On some (but not all) systems this will reflect the random phase jitter between the two oscillators.

Since much of the randomness in these events is in their timing, use the most finely grained time-of-day clock you can find. A standard PC uses an Intel 8254 clock chip (or equivalent) driven at 1.1931818 megahertz, so reading the counter register directly gives you 838-nanosecond resolution. To avoid skewing the results, avoid taking your event samples on a timer interrupt.

Here is the process in C with MD5 (see Section 18.5) as the hash function:

```
char Randpool[16];

/* Call early and call often on a wide variety of random or semi-
 * random system events to churn the randomness pool.
```

```
 * The exact format and length of randevent doesn't matter as long as
 * its contents are at least somewhat unpredictable.
 */
void churnrand(char *randevent,unsigned int randlen)
{
     MD5_CTX md5;
     MD5Init(&md5);
     MD5Update(&md5,Randpool,sizeof(Randpool));
     MD5Update(&md5,randevent,randlen);
     MD5Final(Randpool,&md5);
}
```

After calling churnrand() enough to build up sufficient randomness in Randpool, you can now generate random bits from it. MD5 again comes in handy, this time as a counter-mode pseudo-random byte-stream generator.

```
long Randcnt;
void genrand(char *buf,unsigned int buflen)
{
     MD5_CTX md5;
     char tmp[16];
     unsigned int n;

     while(buflen != 0) {
          /* Hash the pool with a counter */
          MD5Init(&md5);
          MD5Update(&md5,Randpool,sizeof(Randpool));
          MD5Update(&md5,(unsigned char *)&Randcnt,sizeof(Randcnt));
          MD5Final(tmp,&md5);
          Randcnt++; /* Increment counter */

          /* Copy 16 bytes or requested amount, whichever is less,
          * to the user's buffer */
          n = (buflen < 16) ? buflen : 16;
          memcpy(buf,tmp,n);
          buf += n;
          buflen -= n;
     }
}
```

The hash function is crucial here for several reasons. First, it provides an easy way to generate an arbitrary amount of pseudo-random data without having to call churnrand() each time. In effect, the system degrades gracefully from perfect to practical randomness when the demand exceeds the supply. In this case it becomes *theoretically* possible to use the result from one genrand() call to determine a previous or subsequent result. But this requires inverting MD5, which is computationally infeasible.

This is important since the routine doesn't know what each caller will do with the random data it returns. One call might generate a random number for a protocol that is sent in the clear, perhaps in response to a direct request by an attacker. The very next call might generate a secret key for an unrelated session that the attacker

wishes to penetrate. Obviously, it is very important that an attacker not be able to deduce the secret key from the nonce.

One problem remains. There must be sufficient randomness in the Randpool[] array before the first call to genrand(). If the system has been running for a while with a local user typing on the keyboard, no problem. But what about a standalone system that reboots automatically without seeing any keyboard or mouse input?

This is a tough one. A partial solution would require the operator to type for a while after the very first reboot, and to create a seed file on disk before shutting down to carry the randomness in Randseed[] across reboots. But do not save the Randseed[] array directly. An attacker who steals this file could determine all of the results from genrand() after the last call to churnrand() prior to the file being created.

The fix to this problem is to hash the Randseed[] array before storing it, perhaps by just calling genrand(). When the system reboots, you read in the seed file, pass it to churnrand(), then promptly destroy it. Unfortunately, this does not deal with the threat of someone stealing the seed file between reboots and using it to guess future values of the genrand() function. I see no solution to this problem other than to wait until enough external random events have taken place after a reboot before allowing genrand() to produce results.

CHAPTER 20

Public-Key Digital Signature Algorithms

20.1 DIGITAL SIGNATURE ALGORITHM (DSA)

In August 1991, The National Institute of Standards and Technology (NIST) proposed the Digital Signature Algorithm (DSA) for use in their Digital Signature Standard (DSS). According to the *Federal Register* [538]:

> A Federal Information Processing Standard (FIPS) for Digital Signature Standard (DSS) is being proposed. This proposed standard specifies a public-key digital signature algorithm (DSA) appropriate for Federal digital signature applications. The proposed DSS uses a public key to verify to a recipient the integrity of data and identity of the sender of the data. The DSS can also be used by a third party to ascertain the authenticity of a signature and the data associated with it.
>
> This proposed standard adopts a public-key signature scheme that uses a pair of transformations to generate and verify a digital value called a signature.

And:

> This proposed FIPS is the result of evaluating a number of alternative digital signature techniques. In making the selection NIST has followed the mandate contained in section 2 of the Computer Security Act of 1987 that NIST develop standards to ". . . assure the cost-effective security and privacy of Federal information and, among technologies offering comparable protection, on selecting the option with the most desirable operating and use characteristics."
>
> Among the factors that were considered during this process were the level of security provided, the ease of implementation in both hardware and software, the ease of export from the U.S., the applicability of patents, impact on national security and law enforcement and the level of efficiency in both the signing and verification functions. A number of techniques were deemed to provide appropriate protection for Federal systems. The technique selected has the following desirable characteristics:

> NIST expects it to be available on a royalty-free basis. Broader use of this technique resulting from public availability should be an economic benefit to the government and the public.
>
> The technique selected provides for efficient implementation of the signature operations in smart card applications. In these applications the signing operations are performed in the computationally modest environment of the smart card while the verification process is implemented in a more computationally rich environment such as a personal computer, a hardware cryptographic module, or a mainframe computer.

Before it gets too confusing, let me review the nomenclature: DSA is the algorithm; the DSS is the standard. The standard employs the algorithm. The algorithm is part of the standard.

Reaction to the Announcement

NIST's announcement created a maelstrom of criticisms and accusations. Unfortunately, it was more political than academic. RSA Data Security, Inc., purveyors of the RSA algorithm, led the criticism against DSS. They wanted RSA, and not another algorithm, used as the standard. RSADSI makes a lot of money licensing the RSA algorithm, and a royalty-free digital signature standard would directly affect their bottom line. (Note: DSA is not necessarily free of patent infringements; I'll discuss that later.)

Before the algorithm was announced, RSADSI campaigned against a "common modulus," which might have given the government the ability to forge signatures. When the algorithm was announced without this common modulus, they attacked it on other grounds [154], both in letters to NIST and statements to the press. (Four letters to NIST appeared in [1326]. When reading them, keep in mind that at least two of the authors, Rivest and Hellman, had a financial interest in DSS's not being approved.)

Many large software companies that already licensed the RSA algorithm came out against the DSS. In 1982, the government had solicited public-key algorithms for a standard [537]. After that, there wasn't a peep out of NIST for nine years. Companies such as IBM, Apple, Novell, Lotus, Northern Telecom, Microsoft, DEC, and Sun had already spent large amounts of money implementing the RSA algorithm. They were not interested in losing their investment.

In all, NIST received 109 comments by the end of the first comment period on February 28, 1992.

Let's look at the criticisms against DSA, one by one.

1. DSA cannot be used for encryption or key distribution.

   True, but not the point of the standard. This is a signature standard. NIST should have a standard for public-key encryption. NIST is committing a grave injustice to the American people by not implementing a public-key

encryption standard. It is suspicious that this proposed digital signature standard cannot be used for encryption. (As it turns out, though, it can—see Section 23.3.) That does not mean that a signature standard is useless.

2. DSA was developed by the NSA, and there may be a trapdoor in the algorithm.

   Much of the initial comments were just paranoia: "NIST's denial of information with no apparent justification does not inspire confidence in DSS, but intensifies concern that there is a hidden agenda, such as laying the groundwork for a national public-key cryptosystem that is in fact vulnerable to being broken by NIST and/or NSA" [154]. One serious question about the security of DSA was raised by Arjen Lenstra and Stuart Haber at Bellcore. This will be discussed later.

3. DSA is slower than RSA [800].

   True, more or less. Signature generation speeds are the same, but signature verification can be 10 to 40 times slower with DSA. Key generation, however, is faster. But key generation is irrelevant; a user rarely does it. On the other hand, signature verification is the most common operation.

   The problem with this criticism is that there are many ways to play with the test parameters, depending on the results you want. Precomputations can speed up DSA signature generation, but don't always apply. Proponents of RSA use numbers optimized to make their calculations easier; proponents of DSA use their own optimizations. In any case, computers are getting faster all the time. While there is a speed difference, it will not be noticeable in most applications.

4. RSA is a *de facto* standard.

   Here are two examples of this complaint. From Robert Follett, the program director of standards at IBM [570]:

   > IBM is concerned that NIST has proposed a standard with a different digital signature scheme rather than adopting the international standard. We have been convinced by users and user organizations that the international standards using RSA will be a prerequisite to the sales of security products in the very near future.

   From Les Shroyer, vice president and director, corporate MIS and telecommunications, at Motorola [1444]:

   > We must have a single, robust, politically-accepted digital signature standard that is usable throughout the world, between both U.S. and non-U.S., and Motorola and non-Motorola entities. The lack of other viable digital signature technology for the last eight years has made RSA a de facto standard. . . . Motorola and many other companies . . . have committed millions of dollars to RSA. We have concern over the interoperability and support of two different standards, as that situation will lead to added costs, delays in deployment, and complication. . . .

Many companies wanted NIST to adopt the ISO 9796, the international digital signature standard that uses RSA [762]. While this is a valid complaint, it is not a sufficient justification to make it a standard. A royalty-free standard would better serve the U.S. public interest.

5. The DSA selection process was not public; sufficient time for analysis has not been provided.

   First NIST claimed that they designed the DSA; then they admitted that NSA helped them. Finally, they confirmed that NSA designed the algorithm. This worries many people; the NSA doesn't inspire trust. Even so, the algorithm is public and available for analysis; and NIST extended the time for analysis and comment.

6. DSA may infringe on other patents.

   It may. This will be discussed in the section on patent issues.

7. The key size is too small.

   This was the only valid criticism of DSS. The original implementation set the modulus at 512 bits [1149]. Since the algorithm gets its security from the difficulty of computing discrete logs in that modulus, this worried most cryptographers. There have since been advances in the problem of calculating discrete logarithms in a finite field, and 512 bits is too short for long-term security (see Section 7.2). According to Brian LaMacchia and Andrew Odlyzko, ". . . even 512-bit primes appear to offer only marginal security . . ." [934]. In response to this criticism, NIST made the key size variable, from 512 bits to 1024 bits. Not great, but better.

On May 19, 1994, the standard was finally issued [1154]. The issuing statement said [542]:

> This standard is applicable to all Federal departments and agencies for the protection of unclassified information. . . . This standard shall be used in designing and implementing public-key based signature schemes which Federal departments and agencies operate or which are operated for them under contract. Adoption and use of this standard is available to private and commercial organizations.

Before you run out and implement this standard in your next product, read the section on patent issues below.

Description of DSA

DSA is a variant of the Schnorr and ElGamal signature algorithms, and is fully described in [1154]. The algorithm uses the following parameters:

> p = a prime number L bits long, when L ranges from 512 to 1024 and is a multiple of 64. (In the original standard, the size of p was fixed at 512 bits [1149]. This was the source of much criticism and was changed by NIST [1154].)

q = a 160-bit prime factor of $p - 1$.

$g = h^{(p-1)/q} \bmod p$, where h is any number less than $p - 1$ such that $h^{(p-1)/q} \bmod p$ is greater than 1.

x = a number less than q.

$y = g^x \bmod p$.

The algorithm also makes use of a one-way hash function: $H(m)$. The standard specifies the Secure Hash Algorithm, discussed in Section 18.7.

The first three parameters, p, q, and g, are public and can be common across a network of users. The private key is x; the public key is y.

To sign a message, m:

(1) Alice generates a random number, k, less than q.

(2) Alice generates

$$r = (g^k \bmod p) \bmod q$$
$$s = (k^{-1}(H(m) + xr)) \bmod q$$

The parameters r and s are her signature; she sends these to Bob.

(3) Bob verifies the signature by computing

$$w = s^{-1} \bmod q$$
$$u_1 = (H(m) * w) \bmod q$$
$$u_2 = (rw) \bmod q$$
$$v = ((g^{u_1} * y^{u_2}) \bmod p) \bmod q$$

If $v = r$, then the signature is verified.

Proofs for the mathematical relationships are found in [1154]. Table 20.1 provides a summary.

Speed Precomputations

Table 20.2 gives sample software speeds of DSA [918].

Real-world implementations of DSA can often be speeded up through precomputations. Notice that the value r does not depend on the message. You can create a string of random k values, and then recompute r values for each of them. You can also recompute k^{-1} for each of those k values. Then, when a message comes along, you can compute s for a given r and k^{-1}.

This precomputation speeds up DSA considerably. Table 20.3 is a comparison of DSA and RSA computation times for a particular smart card implementation [1479].

Table 20.1
DSA Signatures

Public Key:

p 512-bit to 1024-bit prime (can be shared among a group of users)
q 160-bit prime factor of $p - 1$ (can be shared among a group of users)
g $= h^{(p-1)/q} \bmod p$, where h is less than $p - 1$ and $h^{(p-1)/q} \bmod p > 1$ (can be shared among a group of users)
y $= g^x \bmod p$ (a p-bit number)

Private Key:

x $< q$ (a 160-bit number)

Signing:

k choose at random, less than q
r (signature) $= (g^k \bmod p) \bmod q$
s (signature) $= (k^{-1} (H(m) + xr)) \bmod q$

Verifying:

$w = s^{-1} \bmod q$
$u_1 = (H(m) * w) \bmod q$
$u_2 = (rw) \bmod q$
$v = ((g^{u1} * y^{u2}) \bmod p) \bmod q$
If $v = r$, then the signature is verified.

DSA Prime Generation

Lenstra and Haber pointed out that certain moduli are much easier to crack than others [950]. If someone forced a network to use one of these "cooked" moduli, then their signatures would be easier to forge. This isn't a problem for two reasons: These moduli are easy to detect and they are so rare that the chances of using one when choosing a modulus randomly are almost negligible—smaller, in fact, than the chances of accidentally generating a composite number using a probabilistic prime generation routine.

In [1154] NIST recommended a specific method for generating the two primes, p and q, where q divides $p - 1$. The prime p is L bits long, between 512 and 1024 bits

Table 20.2
DSA Speeds for Different Modulus Lengths with a 160-bit Exponent (on a SPARC II)

	512 bits	768 bits	1024 bits
Sign	0.20 sec	0.43 sec	0.57 sec
Verify	0.35 sec	0.80 sec	1.27 sec

Table 20.3
Comparison of RSA and DSA Computation Times

	DSA	RSA	DSA with Common p, q, g
Global Computations	Off-card (P)	N/A	Off-card (P)
Key Generation	14 sec	Off-card (S)	4 sec
Precomputation	14 sec	N/A	4 sec
Signature	.03 sec	15 sec	.03 sec
Verification	16 sec	1.5 sec	10 sec
	1–5 sec off-card (P)	1–3 sec off-card (P)	

Off-card computations were performed on an 8038633 MHz, personal computer. (P) indicates public parameters off-card and (S) indicates secret parameters off-card. Both algorithms use a 512-bit modulus.

long, in some multiple of 64 bits. The prime q is 160 bits long. Let $L - 1 = 160n + b$, where L is the length of p, and n and b are two numbers and b is less than 160.

(1) Choose an arbitrary sequence of at least 160 bits and call it S. Let g be the length of S in bits.

(2) Compute $U = \text{SHA}(S) \oplus \text{SHA}((S + 1) \bmod 2^g)$, where SHA is the Secure Hash Algorithm (see Section 18.7).

(3) Form q by setting the most significant bit and the least significant bit of U to 1.

(4) Check whether q is prime.

(5) If q is not prime, go back to step (1).

(6) Let $C = 0$ and $N = 2$.

(7) For $k = 0, 1, \ldots, n$, let $V_k = \text{SHA}((S + N + k) \bmod 2^g)$

(8) Let W be the integer

$$W = V_0 + 2^{160}V_1 + \ldots + 2^{160(n-1)}V_{n-1} + 2^{160n}(V_n \bmod 2^b)$$

and let

$$X = W + 2^{L-1}$$

Note that X is an L-bit number.

(9) Let $p = X - ((X \bmod 2q) - 1)$. Note that p is congruent to 1 mod $2q$.

(10) If $p < 2^{L-1}$, then go to step (13).

(11) Check whether p is prime.

(12) If p is prime, go to step (15).

(13) Let $C = C + 1$ and $N = N + n + 1$.

(14) If $C = 4096$, then go to step (1). Otherwise, go to step (7).

(15) Save the value of S and the value of C used to generate p and q.

In [1154], the variable S is called the "seed," C is called the "counter," and N the "offset."

The point of this exercise is that there is a public means of generating p and q. For all practical purposes, this method prevents cooked values of p and q. If someone hands you a p and a q, you might wonder where that person got them. However, if someone hands you a value for S and C that generated the random p and q, you can go through this routine yourself. Using a one-way hash function, SHA in the standard, prevents someone from working backwards from a p and q to generate an S and C.

This security is better than what you get with RSA. In RSA, the prime numbers are kept secret. Someone could generate a fake prime or one of a special form that makes factoring easier. Unless you know the private key, you won't know that. Here, even if you don't know a person's private key, you can confirm that p and q have been generated randomly.

ElGamal Encryption with DSA

There have been allegations that the government likes the DSA because it is only a digital signature algorithm and can't be used for encryption. It is, however, possible to use the DSA function call to do ElGamal encryption.

Assume that the DSA algorithm is implemented with a single function call:

```
DSAsign (p,q,g,k,x,h,r,s)
```

You supply the numbers p, q, g, k, x, and h, and the function returns the signature parameters: r and s.

To do ElGamal encryption of message m with public key y, choose a random number, k, and call

```
DSAsign (p,p,g,k,0,0,r,s)
```

The value of r returned is a in the ElGamal scheme. Throw s away. Then, call

```
DSAsign (p,p,y,k,0,0,r,s)
```

Rename the value of r to be u; throw s away. Call

```
DSAsign (p,p,m,1,u,0,r,s)
```

Throw r away. The value of s returned is b in the ElGamal scheme. You now have the ciphertext, a and b.

Decryption is just as easy. Using secret key x, and ciphertext messages a and b, call

```
DSAsign (p,p,a,x,0,0,r,s)
```

The value r is $a^x \bmod p$. Call that e. Then call

```
DSAsign (p,p,1,e,b,0,r,s)
```

The value s is the plaintext message, m.

This method will not work with all implementations of DSA. Some may fix the values of p and q, or the lengths of some of the other parameters. Still, if the implementation is general enough, this is a way to encrypt using nothing more than digital signature function.

RSA Encryption with DSA

RSA encryption is even easier. With a modulus n, message m, and public key e, call

```
DSAsign (n,n,m,e,0,0,r,s)
```

The value of r returned is the ciphertext.

RSA decryption is the same thing. If d is the private key, then

```
DSAsign (n,n,m,d,0,0,r,s)
```

returns the plaintext as the value of r.

Security of DSA

At 512-bits, DSA wasn't strong enough for long-term security. At 1024 bits, it is.

The NSA, in its first public interview on the subject, commented to Joe Abernathy of *The Houston Chronicle* on allegations about a trapdoor in DSS [363]:

> Regarding the alleged trapdoor in the DSS. We find the term trapdoor somewhat misleading since it implies that the messages sent by the DSS are encrypted and with access via a trapdoor one could somehow decrypt (read) the message without the sender's knowledge.
>
> The DSS does not encrypt any data. The real issue is whether the DSS is susceptible to someone forging a signature and therefore discrediting the entire system. We state categorically that the chances of anyone—including NSA—forging a signature with the DSS when it is properly used and implemented is infinitesimally small.
>
> Furthermore, the alleged trapdoor vulnerability is true for *any* public key-based authentication system, including RSA. To imply somehow that this only affects the DSS (a popular argument in the press) is totally misleading. The issue is one of implementation and how one goes about selecting prime numbers. We call your attention to a recent EUROCRYPT conference which had a panel discussion on the issue of trapdoors in the DSS. Included on the panel was one of the Bellcore researchers who initially raised the trapdoor allegation, and our understanding is that the panel—including the person from Bellcore—concluded that the alleged trapdoor was not an issue for the DSS. Furthermore, the general consensus appeared to be that the trapdoor issue was trivial and had been overblown in the

press. However, to try to respond to the trapdoor allegation, at NIST's request, we have designed a prime generation process which will ensure that one can avoid selection of the relatively few weak primes which could lead to weakness in using the DSS. Additionally, NIST intends to allow for larger modulus sizes up to 1024 which effectively negates the need to even use the prime generation process to avoid weak primes. An additional very important point that is often overlooked is that with the DSS the primes are *public* and therefore can be subject to public examination. Not all public key systems provide for this same type of examination.

The integrity of any information security system requires attention to proper implementation. With the myriad of vulnerabilities possible given the differences among users, NSA has traditionally insisted on centralized trusted centers as a way to minimize risk to the system. While we have designed technical modifications to the DSS to meet NIST's requests for a more decentralized approach, we still would emphasize that portion of the *Federal Register* notice for the DSS which states:

> "While it is the intent of this standard to specify general security requirements for generating digital signatures, conformance to this standard does not assure that a particular implementation is secure. The responsible authority in each agency or department shall assure that an overall implementation provides an acceptable level of security. NIST will be working with government users to ensure appropriate implementations."

Finally, we have read all the arguments purporting insecurities with the DSS, and we remain unconvinced of their validity. The DSS has been subjected to intense evaluation within NSA which led to its being endorsed by our Director of Information Systems Security for use in signing unclassified data processed in certain intelligence systems and even for signing classified data in selected systems. We believe that this approval speaks to the lack of any credible attack on the integrity provided by the DSS given proper use and implementation. Based on the technical and security requirements of the U.S. government for digital signatures, we believe the DSS is the best choice. In fact, the DSS is being used in a pilot project for the Defense Message System to assure the authenticity of electronic messages of vital command and control information. This initial demonstration includes participation from the Joint Chiefs of Staff, the military services, and Defense Agencies and is being done in cooperation with NIST.

I'm not going to comment on the trustworthiness of the NSA. Take their comments for what you think they're worth.

Attacks against k

Each signature requires a new value of k, and that value must be chosen randomly. If Eve ever recovers a k that Alice used to sign a message, perhaps by exploiting some properties of the random-number generator that generated k, she can recover Alice's private key, x. If Eve ever gets two messages signed using the same k, even if she doesn't know what it is, she can recover x. And with x, Eve can generate undetectable forgeries of Alice's signature. In any implementation of the DSA, a good random-number generator is essential to the system's security [1468].

Dangers of a Common Modulus

Even though the DSS does not specify a common modulus to be shared by everyone, different implementations may. For example, the Internal Revenue Service is considering using the DSS for the electronic submission of tax returns. What if they require every taxpayer in the country to use a common p and q? Even though the standard doesn't require a common modulus, such an implementation accomplishes the same thing. A common modulus too easily becomes a tempting target for cryptanalysis. It is still too early to tell much about different DSS implementations, but there is some cause for concern.

Subliminal Channel in DSA

Gus Simmons discovered a subliminal channel in DSA [1468,1469] (see Section 23.3). This subliminal channel allows someone to embed a secret message in his signature that can only be read by another person who knows the key. According to Simmons, it is a "remarkable coincidence" that the "apparently inherent shortcomings of subliminal channels using the ElGamal scheme can all be overcome" in the DSS, and that the DSS "provides the most hospitable setting for subliminal communications discovered to date." NIST and NSA have not commented on this subliminal channel; no one knows if they even knew about it. Since this subliminal channel allows an unscrupulous implementer of DSS to leak a piece of the private key with each signature, it is important to never use an implementation of DSS if you don't trust the implementer.

Patents

David Kravitz, formerly of the NSA, holds a patent on DSA [897]. According to NIST [538]:

> NIST intends to make this DSS technique available world-wide on a royalty-free basis to the public interest. We believe this technique is patentable and that no other patents would apply to the DSS, but we cannot give firm assurances to such effect in advance of issuance of the patent.

Even so, three patent holders claim that the DSA infringes on their patents: Diffie-Hellman (see Section 22.1) [718], Merkle-Hellman (see Section 19.2) [720], and Schnorr (see Section 21.3) [1398]. The Schnorr patent is the most troublesome. The other two patents expire in 1997; the Schnorr patent is valid until 2008. The Schnorr algorithm was not developed with government money; unlike the PKP patents, the U.S. government has no rights to the Schnorr patent; and Schnorr patented his algorithm worldwide. Even if the U.S. courts rule in favor of DSA, it is unclear what other courts around the world would do. Is an international company going to adopt a standard that may be legal in some countries but infringes on a patent in others? This issue will take time to resolve; at the time of this writing it isn't even resolved in the United States.

In June 1993 NIST proposed to give PKP an exclusive patent license to DSA [541]. The agreement fell through after public outcry and the standard was issued without any deal. NIST said [542]:

> . . . NIST has addressed the possible patent infringement claims, and has concluded that there are no valid claims.

So the standard is official, lawsuits are threatened, and no one knows what to do. NIST has said that it would help defend people sued for patent infringement, if they were using DSA to satisfy a government contract. Everyone else, it seems, is on their own. ANSI has a draft banking standard that uses DSA [60]. NIST is working to standardize DSA within the government. Shell Oil has made DSA their international standard. I know of no other proposed DSA standards.

20.2 DSA VARIANTS

This variant makes computation easier on the signer by not forcing him to compute k^{-1} [1135]. All the parameters are as in DSA. To sign a message, m, Alice generates two random numbers, k and d, both less than q. The signature is

$$r = (g^k \bmod p) \bmod q$$
$$s = (H(m) + xr) * d \bmod q$$
$$t = kd \bmod q$$

Bob verifies the signature by computing

$$w = t/s \bmod q$$
$$u_1 = (H(m) * w) \bmod q$$
$$u_2 = (rw) \bmod q$$

If $r = ((g^{u1} * y^{u2}) \bmod p) \bmod q$, then the signature is verified.

This next variant makes computation easier on the verifier [1040,1629]. All the parameters are as in DSA. To sign a message, m, Alice generates a random number, k, less than q. The signature is

$$r = (g^k \bmod p) \bmod q$$
$$s = k * (H(m) + xr)^{-1} \bmod q$$

Bob verifies the signature by computing

$$u_1 = (H(m) * s) \bmod q$$
$$u_2 = (sr) \bmod q$$

If $r = ((g^{u1} * y^{u2}) \bmod p) \bmod q$, then the signature is verified.

Another DSA variant allows for batch verification; Bob can verify signatures in batches [1135]. If they are all valid, he is done. If one isn't valid, then he still has to find it. Unfortunately, it is not secure; either the signer or the verifier can easily create a set of bogus signatures that satisfy the batch criteria [974].

There is also a variant for DSA prime generation, one that embeds q and the parameters used to generate the primes within p. Whether this scheme reduces the security of DSA is still unknown.

(1) Choose an arbitrary sequence of at least 160 bits and call it S. Let g be the length of S in bits.

(2) Compute $U = \text{SHA}(S) \oplus \text{SHA}\ ((S + 1) \bmod 2^g)$, where SHA is the Secure Hash Algorithm (see Section 18.7).

(3) Form q by setting the most significant bit and the least significant bit of U to 1.

(4) Check whether q is prime.

(5) Let p be the concatenation of q, S, C, and SHA(S). C is set to 32 zero bits.

(6) $p = p - (p \bmod q) + 1$.

(7) $p = p + q$.

(8) If the C in p is $0 \times 7fffffff$, go to step (1).

(9) Check whether p is prime.

(10) If p is composite, go to step (7).

The neat thing about this variant is that you don't have to store the values of C and S used to generate p and q; they are embedded within p. For applications without a whole lot of memory, like smart cards, this can be a big deal.

20.3 GOST Digital Signature Algorithm

This is a Russian digital signature standard, officially called GOST R 34.10-94 [656]. The algorithm is very similar to DSA, and uses the following parameters

p = a prime number, either between 509 and 512 bits long, or between 1020 and 1024 bits long.

q = a 254- to 256-bit prime factor of $p - 1$.

a = any number less than $p - 1$ such that $a^q \bmod p = 1$.

x = a number less than q.

$y = a^x \bmod p$.

The algorithm also makes use of a one-way hash function: $H(x)$. The standard specifies GOST R 34.11-94 (see Section 18.11), a function based on the GOST symmetric algorithm (see Section 14.1) [657].

The first three parameters, p, q, and a, are public and can be common across a network of users. The private key is x; the public key is y.

To sign a message, m

(1) Alice generates a random number, k, less than q

(2) Alice generates

$$r = (a^k \bmod p) \bmod q$$
$$s = (xr + k(H(m))) \bmod q$$

If $H(m) \bmod q = 0$, then set it equal to 1. If $r = 0$, then choose another k and start again. The signature is two numbers: $r \bmod 2^{256}$ and $s \bmod 2^{256}$. She sends these to Bob.

(3) Bob verifies the signature by computing

$$v = H(m)^{q-2} \bmod q$$
$$z_1 = (sv) \bmod q$$
$$z_2 = ((q - r) * v) \bmod q$$
$$u = ((a^{z_1} * y^{z_2}) \bmod p) \bmod q$$

If $u = r$, then the signature is verified.

The difference between this scheme and DSA is that with DSA $s = (xr + k^{-1}(H(m))) \bmod q$, which leads to a different verification equation. Curious, though, is that q is 256 bits. Most Western cryptographers seem satisfied with a q of around 160 bits. Perhaps this is just a reflection of the Russian tendency to play it ultrasafe.

The standard has been in use since the beginning of 1995, and is not classified "for special use"—whatever that means.

20.4 DISCRETE LOGARITHM SIGNATURE SCHEMES

ElGamal, Schnorr (see Section 21.3), and DSA signature schemes are very similar. In fact, they are just three examples of a general digital signature scheme based on the Discrete Logarithm Problem. Along with thousands of other signature schemes, they are part of the same family [740,741,699,1184].

Choose p, a large prime number, and q, either $p - 1$ or a large prime factor of $p - 1$. Then choose g, a number between 1 and p such that $g^q \equiv 1 \pmod{p}$. All these numbers are public, and can be common to a group of users. The private key is x, less than q. The public key is $y = g^x \bmod q$.

To sign a message, m, first choose a random k less than and relatively prime to q. If q is also prime, any k less than q works. First compute

$$r = g^k \bmod p$$

The generalized **signature equation** now becomes

$$ak = b + cx \bmod q$$

The coefficients a, b, and c can be any of a variety of things. Each line in Table 20.4 gives six possibilities.

To verify the signature, the receiver must confirm that

$$r^a = g^b y^c \bmod p$$

This is called the **verification equation.**

Table 20.5 lists the signature and verifications possible from just the first line of potential values for a, b, and c, ignoring the effects of the $\pm$.

Table 20.4
Possible Permutations of a, b, and c ($r' = r \bmod q$)

$\pm r'$	$\pm s$	m
$\pm r'm$	$\pm s$	1
$\pm r'm$	$\pm ms$	1
$\pm mr'$	$\pm r's$	1
$\pm ms$	$\pm r's$	1

That's six different signature schemes. Adding the negative signs brings the total to 24. Using the other possible values listed for a, b, and c brings the total to 120.

ElGamal [518,519] and DSA [1154] are essentially based on equation (4). Other schemes are based on equation (2) [24,1629]. Schnorr [1396,1397] is closely related to equation (5), as is another scheme [1183]. And equation (1) can be modified to yield the scheme proposed in [1630]. The rest of the equations are new.

There's more. You can make any of these schemes more DSA-like by defining r as

$$r = (g^k \bmod p) \bmod q$$

Keep the same signature equation and make the verification equation

$$u_1 = a^{-1}b \bmod q$$
$$u_2 = a^{-1}c \bmod q$$
$$r = (g^{u_1}y^{u_2} \bmod p) \bmod q$$
$$(r \bmod q)^a = g^b y^c \bmod p$$

There are two other possibilities along these lines [740,741]; you can do this with each of the 120 schemes, bringing the total to 480 discrete-logarithm-based digital signature schemes.

But wait—there's more. Additional generalizations and variations can generate more than 13,000 variants (not all of them terribly efficient) [740,741].

One of the nice things about using RSA for digital signatures is a features called **message recovery**. When you verify an RSA signature you compute m. Then you compare the computed m with the message and see if the signature is valid for that

Table 20.5
Discrete Logarithm Signature Schemes

Signature Equation	Verification Equation
(1) $r'k = s + mx \bmod q$	$r^{r'} = g^s y^m \bmod p$
(2) $r'k = m + sx \bmod q$	$r^{r'} = g^m y^s \bmod p$
(3) $sk = r' + mx \bmod q$	$r^s = g^{r'} y^m \bmod p$
(4) $sk = m + r'x \bmod q$	$r^s = g^m y^{r'} \bmod p$
(5) $mk = s + r'x \bmod q$	$r^m = g^s y^{r'} \bmod p$
(6) $mk = r' + sx \bmod q$	$r^m = g^{r'} y^s \bmod p$

message. With the previous schemes, you can't recover m when you compute the signature; you need a candidate m that you use in a verification equation. Well, as it turns out it is possible to construct a message recovery variant for all the above signature schemes.

To sign, first compute

$$r = mg^k \bmod p$$

and replace m by 1 in the signature equation. Then you can reconstruct the verification equation such that m can be computed directly.

You can do the same with the DSA-like schemes:

$$r = (mg^k \bmod p) \bmod q$$

All the variants are equally secure, so it makes sense to choose a scheme that is easy to compute with. The requirement to compute inverses slows most of these schemes. As it turns out, a scheme in this pile allows computing both the signature equation and the verification equation without inverses and also gives message recovery. It is called the **p-NEW** scheme [1184].

$$r = mg^{-k} \bmod p$$
$$s = k - r'x \bmod q$$

And m is recovered (and the signature verified) by

$$m = g^s y^{r'} r \bmod p$$

Some variants sign two and three message blocks at the same time [740]; other variants can be used for blind signatures [741].

This is a remarkable piece of research. All of the various discrete-logarithm-based digital signature schemes have been put in one coherent framework. In my opinion this finally puts to rest any patent dispute between Schnorr [1398] and DSA [897]: DSA is not a derivative of Schnorr, nor even of ElGamal. All three are examples of this general construction, and this general construction is unpatented.

20.5 ONG-SCHNORR-SHAMIR

This signature scheme uses polynomials modulo n [1219,1220]. Choose a large integer n (you need not know the factorization of n). Then choose a random integer, k, such that k and n are relatively prime. Calculate h such that

$$h = -k^{-2} \bmod n = -(k^{-1})^2 \bmod n$$

The public key is h and n; k is the private key.

To sign a message, M, first generate a random number, r, such that r and n are relatively prime. Then calculate:

$$S_1 = 1/2 * (M/r + r) \bmod n$$
$$S_2 = k/2 * (M/r - r) \bmod n$$

The pair, S_1 and S_2, is the signature.

To verify a signature, confirm that

$$S_1^2 + h * S_2^2 \equiv M \pmod{n}$$

The version of the scheme described here is based on quadratic polynomials. When it was first proposed in [1217], a $100 reward was offered for successful cryptanalysis. It was proved insecure [1255,18], but its authors were not deterred. They proposed a modification of the algorithm based on cubic polynomials, which is also insecure [1255]. The authors then proposed a quartic version, which was also broken [524,1255]. A variant which fixes these problems is in [1134].

20.6 ESIGN

ESIGN is a digital signature scheme from NTT Japan [1205,583]. It is touted as being at least as secure and considerably faster than either RSA or DSA, with similar key and signature lengths.

The private key is a pair of large prime numbers, p and q. The public key is n, when

$$n = p^2q$$

H is a hash function that operates on a message, m, such that $H(m)$ is between 0 and $m - 1$. There is also a security parameter, k, which will be discussed shortly.

(1) Alice picks a random number x, where x is less than pq.

(2) Alice computes:

w, the least integer that is larger than or equal to

$(H(m) - x^k \bmod n)/pq$

$s = x + ((w/kx^{k-1}) \bmod p)pq$

(3) Alice sends s to Bob.

(4) To verify the signature, Bob computes $s^k \bmod n$. He also computes a, which is the least integer larger than or equal to two times the number of bits of n divided by 3. If $H(m)$ is less than or equal to $s^k \bmod n$, and if $s^k \bmod n$ is less than $H(m) + 2^a$, then the signature is considered valid.

This algorithm works faster with precomputation. This precomputation can be done at any time and has nothing to do with the message being signed. After picking x, Alice could break step (2) into two partial steps. The first can be precomputed.

(2a) Alice computes:

$u = x^k \bmod n$

$v = 1/(kx^{k-1}) \bmod p$

(2b) Alice computes:

w = the least integer that is larger than or equal to
$(H(m) - u)/pq$
$s = x + (wv \bmod p)pq$

For the size of numbers generally used, this precomputation speeds up the signature process by a factor of 10. Almost all the hard work is done in the precomputation stage. A discussion of modular arithmetic operations to speed ESIGN can be found in [1625,1624]. This algorithm can also be extended to work with elliptic curves [1206].

Security of ESIGN

When this algorithm was originally proposed, k was set to 2 [1215]. This was quickly broken by Ernie Brickell and John DeLaurentis [261], who then extended their attack to $k = 3$. A modified version of this algorithm [1203] was broken by Shamir [1204]. The variant proposed in [1204] was broken in [1553]. ESIGN is the current incarnation of this family of algorithms. Another new attack [963] does not work against ESIGN.

The authors currently recommend these values for k: 8, 16, 32, 64, 128, 256, 512, and 1024. They also recommend that p and q each be of at least 192 bits, making n at least 576 bits long. (I think n should be twice that length.) With these parameters, the authors conjecture that ESIGN is as secure as RSA or Rabin. And their analysis shows favorable speed comparison to RSA, ElGamal, and DSA [582].

Patents

ESIGN is patented in the United States [1208], Canada, England, France, Germany, and Italy. Anyone who wishes to license the algorithm should contact Intellectual Property Department, NTT, 1–6 Uchisaiwai-cho, 1-chome, Chiyada-ku, 100 Japan.

20.7 CELLULAR AUTOMATA

A new and novel idea, studied by Papua Guam [665], is the use of cellular automata in public-key cryptosystems. This system is still far too new and has not been studied extensively, but a preliminary examination suggests that it may have a cryptographic weakness similar to one seen in other cases [562]. Still, this is a promising area of research. Cellular automata have the property that, even if they are invertible, it is impossible to calculate the predecessor of an arbitrary state by reversing the rule for finding the successor. This sounds a whole lot like a trapdoor one-way function.

20.8 OTHER PUBLIC-KEY ALGORITHMS

Many other public-key algorithms have been proposed and broken over the years. The Matsumoto-Imai algorithm [1021] was broken in [450]. The Cade algorithm

was first proposed in 1985, broken in 1986 [774], and then strengthened in the same year [286]. In addition to these attacks, there are general attacks for decomposing polynomials over finite fields [605]. Any algorithm that gets its security from the composition of polynomials over a finite field should be looked upon with skepticism, if not outright suspicion.

The Yagisawa algorithm combines exponentiation mod p with arithmetic mod $p - 1$ [1623]; it was broken in [256]. Another public-key algorithm, Tsujii-Kurosawa-Itoh-Fujioka-Matsumoto [1548] is insecure [948]. A third system, Luccio-Mazzone [993], is insecure [717]. A signature scheme based on birational permutations [1425] was broken the day after it was presented [381]. Tatsuaki Okamoto has several signature schemes: one is provably as secure as the Discrete Logarithm Problem, and another is provably as secure as the Discrete Logarithm Problem *and* the Factoring Problem [1206]. Similar schemes are in [709].

Gustavus Simmons suggested J-algebras as a basis for public-key algorithms [1455,145]. This idea was abandoned after efficient methods for factoring polynomials were invented [951]. Special polynomial semigroups have also been studied [1619,962], but so far nothing has come of it. Harald Niederreiter proposed a public-key algorithm based on shift-register sequences [1166]. Another is based on Lyndon words [1476] and another on propositional calculus [817]. And a recent public-key algorithm gets its security from the matrix cover problem [82]. Tatsuaki Okamoto and Kazuo Ohta compare a number of digital signature schemes in [1212].

Prospects for creating radically new and different public-key cryptography algorithms seem dim. In 1988 Whitfield Diffie noted that most public-key algorithms are based on one of three hard problems [492,494]:

1. Knapsack: Given a set of unique numbers, find a subset whose sum is N.
2. Discrete logarithm: If p is a prime and g and M are integers, find x such that $g^x \equiv M \pmod{p}$.
3. Factoring: If N is the product of two primes, either
   a) factor N,
   b) given integers M and C, find d such that $M^d \equiv C \pmod{N}$,
   c) given integers e and C, find M such that $M^e \equiv C \pmod{N}$, or
   d) given an integer x, decide whether there exists an integer y such that $x \equiv y^2 \pmod{N}$.

According to Diffie [492,494], the Discrete Logarithm Problem was suggested by J. Gill, the Factoring Problem by Knuth, and the knapsack problem by Diffie himself.

This narrowness in the mathematical foundations of public-key cryptography is worrisome. A breakthrough in either the problem of factoring or of calculating discrete logarithms could render whole classes of public-key algorithms insecure. Diffie points out [492,494] that this risk is mitigated by two factors:

> 1. The operations on which public key cryptography currently depends—multiplying, exponentiating, and factoring—are all fundamental arithmetic phenom-

ena. They have been the subject of intense mathematical scrutiny for centuries and the increased attention that has resulted from their use in public key cryptosystems has on balance enhanced rather than diminished our confidence.

2. Our ability to carry out large arithmetic computations has grown steadily and now permits us to implement our systems with numbers sufficient in size to be vulnerable only to a dramatic breakthrough in factoring, logarithms, or root extraction.

As we have seen, not all public-key algorithms based on these problems are secure. The strength of any public-key algorithm depends on more than the computational complexity of the problem upon which it is based; a hard problem does not necessarily imply a strong algorithm. Adi Shamir listed three reasons why this is so [1415]:

1. Complexity theory usually deals with single isolated instances of a problem. A cryptanalyst often has a large collection of statistically related problems to solve—several ciphertexts encrypted with the same key.

2. The computational complexity of a problem is typically measured by its worst-case or average-case behavior. To be useful as a cipher, the problem must be hard to solve in almost all cases.

3. An arbitrarily difficult problem cannot necessarily be transformed into a cryptosystem, and it must be possible to insert trapdoor information into the problem so that a shortcut solution is possible with this information and only with this information.