CHAPTER 7

Key Length

7.1 SYMMETRIC KEY LENGTH

The security of a symmetric cryptosystem is a function of two things: the strength of the algorithm and the length of the key. The former is more important, but the latter is easier to demonstrate.

Assume that the strength of the algorithm is perfect. This is extremely difficult to achieve in practice, but easy enough for this example. By perfect, I mean that there is no better way to break the cryptosystem other than trying every possible key in a brute-force attack.

To launch this attack, a cryptanalyst needs a small amount of ciphertext and the corresponding plaintext; a brute-force attack is a known-plaintext attack. For a block cipher, the cryptanalyst would need a block of ciphertext and corresponding plaintext: generally 64 bits. Getting this plaintext and ciphertext is easier than you might imagine. A cryptanalyst might get a copy of a plaintext message by some means and intercept the corresponding ciphertext. He may know something about the format of the ciphertext: For example, it is a WordPerfect file, it has a standard electronic-mail message header, it is a UNIX directory file, it is a TIFF image, or it is a standard record in a customer database. All of these formats have some predefined bytes. The cryptanalyst doesn't need much plaintext to launch this attack.

Calculating the complexity of a brute-force attack is easy. If the key is 8 bits long, there are 2^8, or 256, possible keys. Therefore, it will take 256 attempts to find the correct key, with a 50 percent chance of finding the key after half of the attempts. If the key is 56 bits long, then there are 2^{56} possible keys. Assuming a supercomputer can try a million keys a second, it will take 2285 years to find the correct key. If the key is 64 bits long, then it will take the same supercomputer about 585,000 years to find the correct key among the 2^{64} possible keys. If the key is 128 bits long, it will take 10^{25} years. The universe is only 10^{10} years old, so 10^{25} years is a long time. With a 2048-bit key, a million million-attempts-per-second computers working in paral-

lel will spend 10^{597} years finding the key. By that time the universe will have long collapsed or expanded into nothingness.

Before you rush to invent a cryptosystem with an 8-kilobyte key, remember the other side to the strength question: The algorithm must be so secure that there is no better way to break it than with a brute-force attack. This is not as easy as it might seem. Cryptography is a subtle art. Cryptosystems that look perfect are often extremely weak. Strong cryptosystems, with a couple of minor changes, can become weak. The warning to the amateur cryptographer is to have a healthy, almost paranoid, suspicion of any new algorithm. It is best to trust algorithms that professional cryptographers have scrutinized for years without cracking them and to be suspicious of algorithm designers' grandiose claims of security.

Recall an important point from Section 1.1: The security of a cryptosystem should rest in the key, not in the details of the algorithm. Assume that any cryptanalyst has access to all the details of your algorithm. Assume he has access to as much ciphertext as he wants and can mount an intensive ciphertext-only attack. Assume that he can mount a plaintext attack with as much data as he needs. Even assume that he can mount a chosen-plaintext attack. If your cryptosystem can remain secure, even in the face of all that knowledge, then you've got something.

That warning aside, there is still plenty of room in cryptography to maneuver. In reality, this kind of security isn't really necessary in many situations. Most adversaries don't have the knowledge and computing resources of a major government, and even the ones who do probably aren't that interested in breaking your cryptosystem. If you're plotting to overthrow a major government, stick with the tried and true algorithms in the back of the book. The rest of you, have fun.

Time and Cost Estimates for Brute-Force Attack

Remember that a brute-force attack is typically a known-plaintext attack; it requires a small amount of ciphertext and corresponding plaintext. If you assume that a brute-force attack is the most efficient attack possible against an algorithm—a big assumption—then the key must be long enough to make the attack infeasible. How long is that?

Two parameters determine the speed of a brute-force attack: the number of keys to be tested and the speed of each test. Most symmetric algorithms accept any fixed-length bit pattern as the key. DES has a 56-bit key; it has 2^{56} possible keys. Some algorithms discussed in this book have a 64-bit key; these have 2^{64} possible keys. Others have a 128-bit key.

The speed at which each possible key can be tested is also a factor, but a less important one. For the purposes of this analysis, I will assume that each different algorithm can be tested in the same amount of time. The reality may be that one algorithm may be tested two, three, or even ten times faster than another. But since we are looking for key lengths that are millions of times more difficult to crack than would be feasible, small differences due to test speed are irrelevant.

Most of the debate in the cryptologic community about the efficiency of brute-force attacks has centered on the DES algorithm. In 1977, Whitfield Diffie and Martin Hellman [497] postulated the existence of a special-purpose DES-cracking

machine. This machine consisted of a million chips, each capable of testing a million keys per second. Such a machine could test 2^{56} keys in 20 hours. If built to attack an algorithm with a 64-bit key, it could test all 2^{64} keys in 214 days.

A brute-force attack is tailor-made for parallel processors. Each processor can test a subset of the keyspace. The processors do not have to communicate among themselves; the only communication required at all is a single message signifying success. There are no shared memory requirements. It is easy to design a machine with a million parallel processors, each working independent of the others.

More recently, Michael Wiener decided to design a brute-force cracking machine [1597,1598]. (He designed the machine for DES, but the analysis holds for most any algorithm.) He designed specialized chips, boards, and racks. He estimated prices. And he discovered that for $1 million, someone could build a machine that could crack a 56-bit DES key in an average of 3.5 hours (results guaranteed in 7 hours). And that the price/speed ratio is linear. Table 7.1 generalizes these numbers to a variety of key lengths. Remember Moore's Law: Computing power doubles approximately every 18 months. This means costs go down a factor of 10 every five years; what cost $1 million to build in 1995 will cost a mere $100,000 in the year 2000. Pipelined computers might do even better [724].

For 56-bit keys, these numbers are within the budgets of most large companies and many criminal organizations. The military budgets of most industrialized nations can afford to break 64-bit keys. Breaking an 80-bit key is still beyond the realm of possibility, but if current trends continue that will change in only 30 years.

Of course, it is ludicrous to estimate computing power 35 years in the future. Breakthroughs in some science-fiction technology could make these numbers look like a joke. Conversely, physical limitations unknown at the present time could make them unrealistically optimistic. In cryptography it is wise to be pessimistic. Fielding an algorithm with an 80-bit key seems extremely short-sighted. Insist on at least 112-bit keys.

Table 7.1
Average Time Estimates for a Hardware Brute-Force Attack in 1995

	Length of Key in Bits					
Cost	40	56	64	80	112	128
$100 K	2 seconds	35 hours	1 year	70,000 years	10^{14} years	10^{19} years
$1 M	.2 seconds	3.5 hours	37 days	7000 years	10^{13} years	10^{18} years
$10 M	.02 seconds	21 minutes	4 days	700 years	10^{12} years	10^{17} years
$100 M	2 milliseconds	2 minutes	9 hours	70 years	10^{11} years	10^{16} years
$1 G	.2 milliseconds	13 seconds	1 hour	7 years	10^{10} years	10^{15} years
$10 G	.02 milliseconds	1 second	5.4 minutes	245 days	10^{9} years	10^{14} years
$100 G	2 microseconds	.1 second	32 seconds	24 days	10^{8} years	10^{13} years
$1 T	.2 microseconds	.01 second	3 seconds	2.4 days	10^{7} years	10^{12} years
$10 T	.02 microseconds	1 millisecond	.3 second	6 hours	10^{6} years	10^{11} years

If an attacker wants to break a key badly enough, all he has to do is spend money. Consequently, it seems prudent to try to estimate the minimum "value" of a key: How much value can be trusted to a single key before it makes economic sense to try to break? To give an extreme example, if an encrypted message is worth $1.39, then it wouldn't make much financial sense to set a $10-million cracker to the task of recovering the key. On the other hand, if the plaintext message is worth $100 million, then decrypting that single message would justify the cost of building the cracker. Also, the value of some messages decreases rapidly with time.

Software Crackers

Without special-purpose hardware and massively parallel machines, brute-force attacks are significantly harder. A software attack is about a thousand times slower than a hardware attack.

The real threat of a software-based brute-force attack is not that it is certain, but that it is "free." It costs nothing to set up a microcomputer to test possible keys whenever it is idle. If it finds the correct key—great. If it doesn't, then nothing is lost. It costs nothing to set up an entire microcomputer network to do that. A recent experiment with DES used the collective idle time of 40 workstations to test 2^{34} keys in a single day [603]. At this speed, it will take four million days to test all keys, but if enough people try attacks like this, then someone somewhere will get lucky. As was said in [603]:

> The crux of the software threat is sheer bad luck. Imagine a university computer network of 512 workstations, networked together. On some campuses this would be a medium-sized network. They could even be spread around the world, coordinating their activity through electronic mail. Assume each workstation is capable of running [the algorithm] at a rate of 15,000 encryptions per second. . . . Allowing for the overhead of testing and changing keys, this comes down to . . . 8192 tests per second per machine. To exhaust [a 56-bit] keyspace with this setup would take 545 years (assuming the network was dedicated to the task twenty-four hours per day). Notice, however, that the same calculations give our hypothetical student hackers one chance in 200,000 of cracking a key in one day. Over a long weekend their odds increase to one chance in sixty-six thousand. The faster their hardware, or the more machines involved, the better their chance becomes. These are not good odds for earning a living from horse racing, but they're not the stuff of good press releases either. They are much better odds than the Government gives on its lotteries, for instance. "One-in-a-million"? "Couldn't happen again in a thousand years"? It is no longer possible to say such things honestly. Is this an acceptable ongoing risk?

Using an algorithm with a 64-bit key instead of a 56-bit key makes this attack 256 times more difficult. With a 40-bit key, the picture is far more bleak. A network of 400 computers, each capable of performing 32,000 encryptions per second, can complete a brute-force attack against a 40-bit key in a single day. (In 1992, the RC2 and RC4 algorithms were approved for export with a 40-bit key—see Section 13.8.)

A 128-bit key makes a brute-force attack ridiculous even to contemplate. Industry experts estimate that by 1996 there will be 200 million computers in use world-

wide. This estimate includes everything from giant Cray mainframes to subnotebooks. If every one of those computers worked together on this brute-force attack, and each computer performed a million encryptions per second every second, it would still take a million times the age of the universe to recover the key.

Neural Networks

Neural nets aren't terribly useful for cryptanalysis, primarily because of the shape of the solution space. Neural nets work best with problems that have a continuity of solutions, some better than others. This allows a neural net to learn, proposing better and better solutions as it does. Breaking an algorithm provides for very little in the way of learning opportunities: You either recover the key or you don't. (At least this is true if the algorithm is any good.) Neural nets work well in structured environments where there is something to learn, but not in the high-entropy, seemingly random world of cryptography.

Viruses

The greatest difficulty in getting millions of computers to work on a brute-force attack is convincing millions of computer owners to participate. You could ask politely, but that's time-consuming and they might say no. You could try breaking into their machines, but that's even more time-consuming and you might get arrested. You could also use a computer virus to spread the cracking program more efficiently over as many computers as possible.

This is a particularly insidious idea, first presented in [1593]. The attacker writes and lets loose a computer virus. This virus doesn't reformat the hard drive or delete files; it works on a brute-force cryptanalysis problem whenever the computer is idle. Various studies have shown that microcomputers are idle between 70 percent and 90 percent of the time, so the virus shouldn't have any trouble finding time to work on its task. If it is otherwise benign, it might even escape notice while it does its work.

Eventually, one machine will stumble on the correct key. At this point there are two ways of proceeding. First, the virus could spawn a different virus. It wouldn't do anything but reproduce and delete any copies of the cracking virus it finds but would contain the information about the correct key. This new virus would simply propagate through the computer world until it lands on the computer of the person who wrote the original virus.

A second, sneakier approach would be for the virus to display this message on the screen:

```
There is a serious bug in this computer.
Please call 1-800-123-4567 and read the
following 64 bit number to the operator:

xxxx xxxx xxxx xxxx

There is a $100 reward for the first
person to report this bug.
```

How efficient is this attack? Assume the typical infected computer tries a thousand keys per second. This rate is far less than the computer's maximum potential,

because we assume it will be doing other things occasionally. Also assume that the typical virus infects 10 million machines. This virus can break a 56-bit key in 83 days and a 64-bit key in 58 years. You might have to bribe the antiviral software makers, but that's your problem. Any increase in computer speeds or the virus infection rate would, of course, make this attack more efficient.

The Chinese Lottery

The Chinese Lottery is an eclectic, but possible, suggestion for a massively parallel cryptanalysis machine [1278]. Imagine that a brute-force, million-test-per-second cracking chip was built into every radio and television sold. Each chip is programmed to test a different set of keys automatically upon receiving a plaintext/ciphertext pair over the airwaves. Every time the Chinese government wants to break a key, it broadcasts the data. All the radios and televisions in the country start chugging away. Eventually, the correct key will appear on someone's display, somewhere in the country. The Chinese government pays a prize to that person; this makes sure that the result is reported promptly and properly, and also helps the sale of radios and televisions with the cracking chips.

If every man, woman, and child in China owns a radio or television, then the correct key to a 56-bit algorithm will appear in 61 seconds. If only 1 in 10 Chinese owns a radio or television—closer to reality—the correct key will appear in 10 minutes. The correct key for a 64-bit algorithm will appear in 4.3 hours—43 hours if only 1 in 10 owns a radio or television.

Some modifications are required to make this attack practical. First, it would be easier to have each chip try random keys instead of a unique set of keys. This would make the attack about 39 percent slower—not much in light of the numbers we're working with. Also, the Chinese Communist party would have to mandate that every person listen to or watch a certain show at a certain time, just to make sure that all of the radios and televisions are operating when the plaintext/ciphertext pair is broadcast. Finally, everyone would have to be instructed to call a Central-Party-Whatever-It's-Called if a key ever shows up on their screen, and then to read off the string of numbers appearing there.

Table 7.2 shows the effectiveness of the Chinese Lottery for different countries and different key lengths. China would clearly be in the best position to launch such an attack if they have to outfit every man, woman, and child with their own television or radio. The United States has fewer people but a lot more equipment per capita. The state of Wyoming could break a 56-bit key all by itself in less than a day.

Biotechnology

If biochips are possible, then it would be foolish not to use them as a distributed brute-force cryptanalysis tool. Consider a hypothetical animal, unfortunately called a "DESosaur" [1278]. It consists of biological cells capable of testing possible keys. The plaintext/ciphertext pair is broadcast to the cells via some optical channel (these cells are transparent, you see). Solutions are carried to the DESosaur's speech organ via special cells that travel through the animal's circulatory system.

The typical dinosaur had about 10^{14} cells (excluding bacteria). If each of them can perform a million encryptions per second (granted, this is a big if), breaking a 56-bit

Table 7.2
Brute-Force Cracking Estimates for Chinese Lottery

Country	Population	# of Televisions/Radios	Time to Break	
			56-bit	64-bit
China	1,190,431,000	257,000,000	280 seconds	20 hours
U.S.	260,714,000	739,000,000	97 seconds	6.9 hours
Iraq	19,890,000	4,730,000	4.2 hours	44 days
Israel	5,051,000	3,640,000	5.5 hours	58 days
Wyoming	470,000	1,330,000	15 hours	160 days
Winnemucca, NV	6,100	17,300	48 days	34 years

(All data is from the *1995 World Almanac and Book of Facts.*)

key would take seven ten-thousandths of a second. Breaking a 64-bit key would take less than two tenths of a second. Breaking a 128-bit key would still take 10^{11} years, though.

Another biological approach is to use genetically engineered cryptanalytic algae that are capable of performing brute-force attacks against cryptographic algorithms [1278]. These organisms would make it possible to construct a distributed machine with more processors because they could cover a larger area. The plaintext/ciphertext pair could be broadcast by satellite. If an organism found the result, it could induce the nearby cells to change color to communicate the solution back to the satellite.

Assume the typical algae cell is the size of a cube 10 microns on a side (this is probably a large estimate), then 10^{15} of them can fill a cubic meter. Pump them into the ocean and cover 200 square miles (518 square kilometers) of water to a meter deep (you figure out how to do it—I'm just the idea man), and you'd have 10^{23} (over a hundred billion gallons) of them floating in the ocean. (For comparison, the Exxon *Valdez* spilled 10 million gallons of oil.) If each of them can try a million keys per second, they will recover the key for a 128-bit algorithm in just over 100 years. (The resulting algae bloom is your problem.) Breakthroughs in algae processing speed, algae diameter, or even the size puddle one could spread across the ocean, would reduce these numbers significantly.

Don't even ask me about nanotechnology.

Thermodynamic Limitations

One of the consequences of the second law of thermodynamics is that a certain amount of energy is necessary to represent information. To record a single bit by changing the state of a system requires an amount of energy no less than kT, where T is the absolute temperature of the system and k is the Boltzman constant. (Stick with me; the physics lesson is almost over.)

Given that $k = 1.38 * 10^{-16}$ erg/°Kelvin, and that the ambient temperature of the universe is 3.2°K, an ideal computer running at 3.2°K would consume $4.4 * 10^{-16}$ ergs every time it set or cleared a bit. To run a computer any colder than the cosmic background radiation would require extra energy to run a heat pump.

Now, the annual energy output of our sun is about $1.21 * 10^{41}$ ergs. This is enough to power about $2.7 * 10^{56}$ single bit changes on our ideal computer; enough state changes to put a 187-bit counter through all its values. If we built a Dyson sphere around the sun and captured all of its energy for 32 years, without any loss, we could power a computer to count up to 2^{192}. Of course, it wouldn't have the energy left over to perform any useful calculations with this counter.

But that's just one star, and a measly one at that. A typical supernova releases something like 10^{51} ergs. (About a hundred times as much energy would be released in the form of neutrinos, but let them go for now.) If all of this energy could be channeled into a single orgy of computation, a 219-bit counter could be cycled through all of its states.

These numbers have nothing to do with the technology of the devices; they are the maximums that thermodynamics will allow. And they strongly imply that brute-force attacks against 256-bit keys will be infeasible until computers are built from something other than matter and occupy something other than space.

7.2 PUBLIC-KEY KEY LENGTH

One-way functions were discussed in Section 2.3. Multiplying two large primes is a one-way function; it's easy to multiply the numbers to get a product but hard to factor the product and recover the two large primes (see Section 11.3). Public-key cryptography uses this idea to make a trap-door one-way function. Actually, that's a lie; factoring is *conjectured to be* a hard problem (see Section 11.4). As far as anyone knows, it seems to be. Even if it is, no one can prove that hard problems are actually hard. Most everyone assumes that factoring is hard, but it has never been mathematically proven one way or the other.

This is worth dwelling on. It is easy to imagine that 50 years in the future we will all sit around, reminiscing about the good old days when people used to think factoring was hard, cryptography was based on factoring, and companies actually made money from this stuff. It is easy to imagine that future developments in number theory will make factoring easier or that developments in complexity theory will make factoring trivial. There's no reason to believe this will happen—and most people who know enough to have an opinion will tell you that it is unlikely—but there's also no reason to believe it won't.

In any case, today's dominant public-key encryption algorithms are based on the difficulty of factoring large numbers that are the product of two large primes. (Other algorithms are based on something called the Discrete Logarithm Problem, but for the moment assume the same discussion applies.) These algorithms are also susceptible to a brute-force attack, but of a different type. Breaking these algorithms does not involve trying every possible key; breaking these algorithms involves trying to factor the large number (or taking discrete logarithms in a very large finite field—a similar problem). If the number is too small, you have no security. If the number is large enough, you have security against all the computing power in the world working from now until the sun goes nova—given today's understanding of

the mathematics. Section 11.3 discusses factoring in more mathematical detail; here I will limit the discussion to how long it takes to factor numbers of various lengths.

Factoring large numbers is hard. Unfortunately for algorithm designers, it is getting easier. Even worse, it is getting easier faster than mathematicians expected. In 1976 Richard Guy wrote: "I shall be surprised if anyone regularly factors numbers of size 10^{80} without special form during the present century" [680]. In 1977 Ron Rivest said that factoring a 125-digit number would take 40 quadrillion years [599]. In 1994 a 129-digit number was factored [66]. If there is any lesson in all this, it is that making predictions is foolish.

Table 7.3 shows factoring records over the past dozen years. The fastest factoring algorithm during the time was the quadratic sieve (see Section 11.3).

These numbers are pretty frightening. Today it is not uncommon to see 512-bit numbers used in operational systems. Factoring them, and thereby completely compromising their security, is well in the range of possibility: A weekend-long worm on the Internet could do it.

Computing power is generally measured in mips-years: a one-million-instruction-per-second (mips) computer running for one year, or about $3*10^{13}$ instructions. By convention, a 1-mips machine is equivalent to the DEC VAX 11/780. Hence, a mips-year is a VAX 11/780 running for a year, or the equivalent. (A 100 MHz Pentium is about a 50 mips machine; a 1800-node Intel Paragon is about 50,000.)

The 1983 factorization of a 71-digit number required 0.1 mips-years; the 1994 factorization of a 129-digit number required 5000. This dramatic increase in computing power resulted largely from the introduction of distributed computing, using the idle time on a network of workstations. This trend was started by Bob Silverman and fully developed by Arjen Lenstra and Mark Manasse. The 1983 factorization used 9.5 CPU hours on a single Cray X-MP; the 1994 factorization took 5000 mips-years and used the idle time on 1600 computers around the world for about eight months. Modern factoring methods lend themselves to this kind of distributed implementation.

The picture gets even worse. A new factoring algorithm has taken over from the quadratic sieve: the general number field sieve. In 1989 mathematicians would have

Table 7.3
Factoring Using the Quadratic Sieve

Year	# of decimal digits factored	How many times harder to factor a 512-bit number
1983	71	>20 million
1985	80	>2 million
1988	90	250,000
1989	100	30,000
1993	120	500
1994	129	100

told you that the general number field sieve would never be practical. In 1992 they would have told you that it was practical, but only faster than the quadratic sieve for numbers greater than 130 to 150 digits or so. Today it is known to be faster than the quadratic sieve for numbers well below 116 digits [472,635]. The general number field sieve can factor a 512-bit number over 10 times faster than the quadratic sieve. The algorithm would require less than a year to run on an 1800-node Intel Paragon. Table 7.4 gives the number of mips-years required to factor numbers of different sizes, given current implementations of the general number field sieve [1190].

And the general number field sieve is still getting faster. Mathematicians keep coming up with new tricks, new optimizations, new techniques. There's no reason to think this trend won't continue. A related algorithm, the special number field sieve, can already factor numbers of a certain specialized form—numbers not generally used for cryptography—much faster than the general number field sieve can factor general numbers of the same size. It is not unreasonable to assume that the general number field sieve can be optimized to run this fast [1190]; it is possible that the NSA already knows how to do this. Table 7.5 gives the number of mips-years required for the special number field sieve to factor numbers of different lengths [1190].

At a European Institute for System Security workshop in 1991, the participants agreed that a 1024-bit modulus should be sufficient for long-term secrets through 2002 [150]. However, they warned: "Although the participants of this workshop feel best qualified in their respective areas, this statement [with respect to lasting security] should be taken with caution." This is good advice.

The wise cryptographer is ultra-conservative when choosing public-key key lengths. To determine how long a key you need requires you to look at both the intended security and lifetime of the key, and the current state-of-the-art of factoring. Today you need a 1024-bit number to get the level of security you got from a 512-bit number in the early 1980s. If you want your keys to remain secure for 20 years, 1024 bits is likely too short.

Even if your particular secrets aren't worth the effort required to factor your modulus, you may be at risk. Imagine an automatic banking system that uses RSA for security. Mallory can stand up in court and say: "Did you read in the newspaper in 1994 that RSA-129 was broken, and that 512-bit numbers can be factored by any

Table 7.4
Factoring Using the General Number Field Sieve

# of bits	Mips-years required to factor
512	30,000
768	$2*10^8$
1024	$3*10^{11}$
1280	$1*10^{14}$
1536	$3*10^{16}$
2048	$3*10^{20}$

Table 7.5
Factoring Using the Special Number Field Sieve

# of bits	Mips-years required to factor
512	<200
768	100,000
1024	$3*10^7$
1280	$3*10^9$
1536	$2*10^{11}$
2048	$4*10^{14}$

organization willing to spend a few million dollars and wait a few months? My bank uses 512-bit numbers for security and, by the way, I didn't make these seven withdrawals." Even if Mallory is lying, the judge will probably put the onus on the bank to prove it.

Why not use 10,000-bit keys? You can, but remember that you pay a price in computation time as your keys get longer. You want a key long enough to be secure, but short enough to be computationally usable.

Earlier in this section I called making predictions foolish. Now I am about to make some. Table 7.6 gives my recommendations for public-key lengths, depending on how long you require the key to be secure. There are three key lengths for each year, one secure against an individual, one secure against a major corporation, and the third secure against a major government.

Here are some assumptions from [66]:

> We believe that we could acquire 100 thousand machines without superhuman or unethical efforts. That is, we would *not* set free an Internet worm or virus to find resources for us. Many organizations have several thousand machines each on the net. Making use of their facilities would require skillful diplomacy, but should not be impossible. Assuming the 5 mips average power, and one year elapsed time, it is not too unreasonable to embark on a project which would require half a million mips years.

The project to factor the 129-digit number harnessed an estimated 0.03 percent of the total computing power of the Internet [1190], and they didn't even try very hard. It isn't unreasonable to assume that a well-publicized project can harness 2 percent of the world's computing power for a year.

Assume a dedicated cryptanalyst can get his hands on 10,000 mips-years, a large corporation can get 10^7 mips-years, and that a large government can get 10^9 mips-years. Also assume that computing power will increase by a factor of 10 every five years. And finally, assume that advances in factoring mathematics allow us to factor general numbers at the speeds of the special number field sieve. (This isn't possible yet, but the breakthrough could occur at any time.) Table 7.6 recommends different key lengths for security during different years.

Table 7.6
Recommended Public-key Key Lengths (in bits)

Year	vs. Individual	vs. Corporation	vs. Government
1995	768	1280	1536
2000	1024	1280	1536
2005	1280	1536	2048
2010	1280	1536	2048
2015	1536	2048	2048

Remember to take the value of the key into account. Public keys are often used to secure things of great value for a long time: the bank's master key for a digital cash system, the key the government uses to certify its passports, or a notary public's digital signature key. It probably isn't worth the effort to spend months of computing time to break an individual's private key, but if you can print your own money with a broken key the idea becomes more attractive. A 1024-bit key is long enough to sign something that will be verified within the week, or month, or even a few years. But you don't want to stand up in court 20 years from now with a digitally signed document and have the opposition demonstrate how to forge documents with the same signature.

Making predictions beyond the near future is even more foolish. Who knows what kind of advances in computing, networking, and mathematics are going to happen by 2020? However, if you look at the broad picture, in every decade we can factor numbers twice as long as in the previous decade. This leads to Table 7.7.

On the other hand, factoring technology may reach its Omega point long before 2045. Twenty years from now, we may be able to factor anything. I think that is unlikely, though.

Not everyone will agree with my recommendations. The NSA has mandated 512-bit to 1024-bit keys for their Digital Signature Standard (see Section 20.1)—far less than I recommend for long-term security. Pretty Good Privacy (see Section 24.12) has a maximum RSA key length of 2047 bits. Arjen Lenstra, the world's most suc-

Table 7.7
Long-range Factoring Predictions

Year	Key Length (in bits)
1995	1024
2005	2048
2015	4096
2025	8192
2035	16,384
2045	32,768

cessful factorer, refuses to make predictions past 10 years [949]. Table 7.8 gives Ron Rivest's key-length recommendations, originally made in 1990, which I consider much too optimistic [1323]. While his analysis looks fine on paper, recent history illustrates that surprises regularly happen. It makes sense to choose your keys to be resilient against future surprises.

Low estimates assume a budget of $25,000, the quadratic sieve algorithm, and a technology advance of 20 percent per year. Average estimates assume a budget of $25 million, the general number field sieve algorithm, and a technology advance of 33 percent per year. High estimates assume a budget of $25 billion, a general quadratic sieve algorithm running at the speed of the special number field sieve, and a technology advance of 45 percent per year.

There is always the possibility that an advance in factoring will surprise me as well, but I factored that into my calculations. But why trust me? I just proved my own foolishness by making predictions.

DNA Computing

Now it gets weird. In 1994 Leonard M. Adleman actually demonstrated a method for solving an **NP-complete** problem (see Section 11.2) in a biochemistry laboratory, using DNA molecules to represent guesses at solutions to the problem [17]. (That's "solutions" meaning "answers," not meaning "liquids containing solutes." Terminology in this field is going to be awkward.) The problem that Adleman solved was an instance of the Directed Hamiltonian Path problem: Given a map of cities connected by one-way roads, find a path from City A to City Z that passes exactly once through all other cities on the map. Each city was represented by a different random 20-base string of DNA; with conventional molecular biology techniques, Adleman synthesized 50 picomols (30 million million molecules) of the DNA string representing each city. Each road was also represented by a 20-base DNA string, but these strings were not chosen randomly: They were cleverly chosen so that the "beginning" end of the DNA string representing the road from City P to City K ("Road PK") would tend to stick to the DNA string representing City P, and the end of Road PK would tend to stick to City K.

Table 7.8
Rivest's Optimistic Key-length Recommendations (in bits)

Year	Low	Average	High
1990	398	515	1289
1995	405	542	1399
2000	422	572	1512
2005	439	602	1628
2010	455	631	1754
2015	472	661	1884
2020	489	677	2017

Adleman synthesized 50 picomols of the DNA representing each road, mixed them all together with the DNA representing all the cities, and added a ligase enzyme, which links together the ends of DNA molecules. The clever relationship between the road DNA strings and the city DNA strings causes the ligase to link the road DNA strings together in a legal fashion. That is, the "exit" end of the road from P to K will always be linked to the "entrance" end of some road that originates at City K, never to the "exit" end of any road and never to the "entrance" end of a road that originates at some city other than K. After a carefully limited reaction time, the ligase has built a large number of DNA strings representing legal but otherwise random multiroad paths within the map.

From this soup of random paths, Adleman can find the tiniest trace—perhaps even a single molecule—of the DNA that represents the answer to the problem. Using common techniques of molecular biology, he discards all the DNA strings representing paths that are too long or too short. (The number of roads in the desired path must equal the number of cities minus one.) Next he discards all the DNA strings that do not pass through City A, then those that miss City B, and so forth. If any DNA survives this screening, it is examined to find the sequence of roads that it represents: This is the solution to the directed Hamiltonian path problem.

By definition, an instance of any **NP-complete** problem can be transformed, in polynomial time, into an instance of any other **NP-complete** problem, and therefore into an instance of the directed Hamiltonian path problem. Since the 1970s, cryptologists have been trying to use **NP-complete** problems for public-key cryptography.

While the instance that Adleman solved was very modest (seven cities on his map, a problem that can be solved by inspection in a few minutes), the technique is in its infancy and has no forbidding obstacles keeping it from being extended to larger problems. Thus, arguments about the security of cryptographic protocols based on **NP-complete** problems, arguments that heretofore have begun, "Suppose an adversary has a million processors, each of which can perform a million tests each second," may soon have to be replaced with, "Suppose an adversary has a thousand fermentation vats, each 20,000 liters in capacity."

Quantum Computing

Now, it gets even weirder. The underlying principle behind quantum computing involves Einstein's wave-particle duality. A photon can simultaneously exist in a large number of states. A classic example is that a photon behaves like a wave when it encounters a partially silvered mirror; it is both reflected and transmitted, just as an ocean wave striking a seawall with a small opening in it will both reflect off the wall and pass through it. However, when a photon is measured, it behaves like a particle and only a single state can be detected.

In [1443], Peter Shor outlines a design for a factoring machine based on quantum mechanical principles. Unlike a classical computer, which can be thought of as having a single, fixed state at a given time, a quantum computer has an internal wave function, which is a superposition of a combination of the possible basis states. Computations transform the wave function, altering the entire set of states in a single operation. In this way, a quantum computer is an improvement over classical finite-state automata: It uses quantum properties to allow it to factor in polynomial

time, theoretically allowing one to break cryptosystems based on factoring or the discrete logarithm problem.

The consensus is that quantum computers are compatible with the fundamental laws of quantum mechanics. However, it is unlikely that a quantum factoring machine will be built in the foreseeable future . . . if ever. One major obstacle is the problem of decoherence, which causes superimposed waveforms to lose their distinctness and makes the computer fail. Decoherence will make a quantum computer running at 1° Kelvin fail after just one nanosecond. Additionally, an enormous number of gates would be required to build a quantum factoring device; this may render the machine impossible to build. Shor's design requires a complete modular exponentiator. No internal clock can be used, so millions or possibly billions of individual gates would be required to factor cryptographically significant numbers. If n quantum gates have some minimum probability p of failure, the average number of trials required per successful run is $(1/(1-p))^n$. The number of gates required presumably grows polynomially with the length (in bits) of the number, so the number of trials required would be superexponential with the length of the numbers used—worse than factoring by trial division!

So, while quantum factorization is an area of great academic excitement, it is extremely unlikely that it will be practical in the foreseeable future. But don't say I didn't warn you.

7.3 Comparing Symmetric and Public-Key Key Length

A system is going to be attacked at its weakest point. If you are designing a system that uses both symmetric and public-key cryptography, the key lengths for each type of cryptography should be chosen so that it is equally difficult to attack the system via each mechanism. It makes no sense to use a symmetric algorithm with a 128-bit key together with a public-key algorithm with a 386-bit key, just as it makes no sense to use a symmetric algorithm with a 56-bit key together with a public-key algorithm with a 1024-bit key.

Table 7.9 lists public-key modulus lengths whose factoring difficulty roughly equals the difficulty of a brute-force attack for popular symmetric key lengths.

This table says that if you are concerned enough about security to choose a symmetric algorithm with a 112-bit key, you should choose a modulus length for your public-key algorithm of about 1792 bits. In general, though, you should choose a public-key length that is more secure than your symmetric-key length. Public keys generally stay around longer, and are used to protect more information.

7.4 Birthday Attacks against One-Way Hash Functions

There are two brute-force attacks against a one-way hash function. The first is the most obvious: Given the hash of message, $H(M)$, an adversary would like to be able to create another document, M', such that $H(M) = H(M')$. The second attack is more

Table 7.9
Symmetric and Public-key Key Lengths with Similar Resistances to Brute-Force Attacks

Symmetric Key Length	Public-key Key Length
56 bits	384 bits
64 bits	512 bits
80 bits	768 bits
112 bits	1792 bits
128 bits	2304 bits

subtle: An adversary would like to find two random messages, M, and M', such that $H(M) = H(M')$. This is called a **collision**, and it is a far easier attack than the first one.

The birthday paradox is a standard statistics problem. How many people must be in a room for the chance to be greater than even that one of them shares your birthday? The answer is 253. Now, how many people must there be for the chance to be greater than even that at least two of them will share the same birthday? The answer is surprisingly low: 23. With only 23 people in the room, there are still 253 different *pairs* of people in the room.

Finding someone with a specific birthday is analogous to the first attack; finding two people with the same random birthday is analogous to the second attack. The second attack is commonly known as a **birthday attack**.

Assume that a one-way hash function is secure and the best way to attack it is by using brute force. It produces an m-bit output. Finding a message that hashes to a given hash value would require hashing 2^m random messages. Finding two messages that hash to the same value would only require hashing $2^{m/2}$ random messages. A machine that hashes a million messages per second would take 600,000 years to find a second message that matched a given 64-bit hash. The same machine could find a pair of messages that hashed to the same value in about an hour.

This means that if you are worried about a birthday attack, you should choose a hash-value twice as long as you otherwise might think you need. For example, if you want to drop the odds of someone breaking your system to less than 1 in 2^{80}, use a 160-bit one-way hash function.

7.5 How Long Should a Key Be?

There's no single answer to this question; it depends on the situation. To determine how much security you need, you must ask yourself some questions. How much is your data worth? How long does it need to be secure? What are your adversaries' resources?

A customer list might be worth $1000. Financial data for an acrimonious divorce case might be worth $10,000. Advertising and marketing data for a large corporation

might be worth $1 million. The master keys for a digital cash system might be worth billions.

In the world of commodities trading, secrets only need to be kept for minutes. In the newspaper business, today's secrets are tomorrow's headlines. Product development information might need to remain secret for a year or two. U.S. Census data are required by law to remain secret for 100 years.

The guest list for your sister's surprise birthday party is only interesting to your nosy relatives. Corporate trade secrets are interesting to rival companies. Military secrets are interesting to rival militaries.

You can even specify security requirements in these terms. For example:

> The key length must be such that there is a probability of no more than 1 in 2^{32} that an attacker with $100 million to spend could break the system within one year, even assuming technology advances at a rate of 30 percent per annum over the period.

Table 7.10, taken partially from [150], estimates the secrecy requirements for several kinds of information:

Future computing power is harder to estimate, but here is a reasonable rule of thumb: The efficiency of computing equipment divided by price doubles every 18 months and increases by a factor of 10 every five years. Thus, in 50 years the fastest computers will be 10 billion times faster than today's! Remember, too, that these numbers only relate to general-purpose computers; who knows what kind of specialized cryptosystem-breaking equipment will be developed in the next 50 years?

Assuming that a cryptographic algorithm will be in use for 30 years, you can get some idea how secure it must be. An algorithm designed today probably will not see general use until 2000, and will still be used in 2025 to encrypt messages that must remain secret until 2075 or later.

Table 7.10
Security Requirements for Different Information

Type of Traffic	Lifetime	Minimum Key Length
Tactical military information	minutes/hours	56–64 bits
Product announcements, mergers, interest rates	days/weeks	64 bits
Long-term business plans	years	64 bits
Trade secrets (e.g., recipe for Coca-Cola)	decades	112 bits
H-bomb secrets	>40 years	128 bits
Identities of spies	>50 years	128 bits
Personal affairs	>50 years	128 bits
Diplomatic embarrassments	>65 years	at least 128 bits
U.S. census data	100 years	at least 128 bits

7.6 CAVEAT EMPTOR

This entire chapter is a whole lot of nonsense. The very notion of predicting computing power 10 years in the future, let alone 50 years is absolutely ridiculous. These calculations are meant to be a guide, nothing more. If the past is any guide, the future will be vastly different from anything we can predict.

Be conservative. If your keys are longer than you imagine necessary, then fewer technological surprises can harm you.