PEARSON SERIES IN COMPUTER NETWORKING AND DISTRIBUTED SYSTEMS

THIRD EDITION

NETWORK SECURITY

PRIVATE Communication in a PUBLIC World

NEW CONTENT
Quantum computing, post-quantum algorithms, multiparty computation, fully homomorphic encryption, and more!

CHARLIE KAUFMAN • RADIA PERLMAN
MIKE SPECINER • RAY PERLNER

Network Security

PRIVATE Communication in a PUBLIC World

THIRD EDITION

Charlie Kaufman • Radia Perlman
Mike Speciner • Ray Perlner

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

capacity of all existing computers for more than the expected life of the universe and thus would not be cost-effective.

Luckily, you can do better than that. If you do the modular reduction after each multiplication, it keeps the number from getting really ridiculous. To illustrate:

$$123^2 = 123 \times 123 = 15129 = 213 \bmod 678$$
$$123^3 = 123 \times 213 = 26199 = 435 \bmod 678$$
$$123^4 = 123 \times 435 = 53505 = 621 \bmod 678$$

This reduces the problem to 54 small multiplications and 54 small divisions, but it would still be unacceptable for exponents of the size used with RSA.

However, there is a much more efficient method. To raise a number *m* to an exponent that is a power of 2, say 32, you could start with *m* and multiply by *m* 31 times, which is reasonable if you have nothing better to do with your time. A much better scheme is to first square *m*, then square the result, and so on. Then you'll be done after five squarings (five multiplications and five divisions):

$$123^2 = 123 \times 123 = 15129 = 213 \bmod 678$$
$$123^4 = 213 \times 213 = 45369 = 621 \bmod 678$$
$$123^8 = 621 \times 621 = 385641 = 537 \bmod 678$$
$$123^{16} = 537 \times 537 = 288369 = 219 \bmod 678$$
$$123^{32} = 219 \times 219 = 47961 = 501 \bmod 678$$

What if you're not lucky enough to be raising something to a power of 2? First note that if you know what 123^x is, then it's easy to compute 123^{2x}—you get that by squaring 123^x. It's also easy to compute 123^{2x+1}—you get that by multiplying 123^{2x} by 123. Now you use this observation to compute 123^{54}.

Well, 54 is 110110_2 (represented in binary). You'll compute 123 raised to a sequence of powers—1_2, 11_2, 110_2, 1101_2, 11011_2, 110110_2. Each successive power concatenates one more bit of the desired exponent. And each successive power is either twice the preceding power or one more than twice the preceding power:

$$123^2 = 123 \times 123 = 15129 = 213 \bmod 678$$
$$123^3 = 123^2 \times 123 = 213 \times 123 = 26199 = 435 \bmod 678$$
$$123^6 = (123^3)^2 = 435^2 = 189225 = 63 \bmod 678$$
$$123^{12} = (123^6)^2 = 63^2 = 3969 = 579 \bmod 678$$
$$123^{13} = 123^{12} \times 123 = 579 \times 123 = 71217 = 27 \bmod 678$$
$$123^{26} = (123^{13})^2 = 27^2 = 729 = 51 \bmod 678$$
$$123^{27} = 123^{26} \times 123 = 51 \times 123 = 6273 = 171 \bmod 678$$
$$123^{54} = (123^{27})^2 = 171^2 = 29241 = 87 \bmod 678$$

The idea is that squaring is the same as doubling the exponent, which, in turn, is the same as shifting the exponent left by one bit. And multiplying by the base is the same as adding one to the exponent.

In general, to perform exponentiation of a base to an exponent, you start with your value set to 1. As you read the exponent in binary bit by bit from high-order bit to low-order bit, you square your value, and if the bit is a 1, you then multiply by the base. You perform modular reduction after each operation to keep the intermediate results small.

By this method you've reduced the computation of 123^{54} to eight multiplications and eight divisions. More importantly, the number of multiplications and divisions rises linearly with the length of the exponent in bits rather than with the value of the exponent itself.

RSA operations using this technique are sufficiently efficient to be practical.

Note that as we explained in §2.7.3 *Avoiding a Side-Channel Attack*, this implementation would allow a side-channel attack, because the implementation would behave differently on each bit of the exponent, depending on whether the bit was a 0 or a 1. An implementation should avoid providing a side channel by behaving the same way on each bit, as explained in §2.7.3.

6.3.4.2 Generating RSA Keys

Most uses of public key cryptography do not require frequent generation of RSA keys. If generation of an RSA key is only done, for instance, when an employee is hired, then it need not be as efficient as the operations that use the keys. However, it still has to be reasonably efficient.

6.3.4.2.1. Finding Big Primes p and q

There is an infinite supply of primes. However, they thin out as numbers get bigger and bigger. The probability of a randomly chosen number n being prime is approximately $1/\ln n$. The natural logarithm function, ln, rises linearly with the size of the number represented in digits or bits. For a ten-digit number, there is about one chance in 23 of it being prime. For a 300-digit number (a size of prime that would be useful for RSA), there is about one chance in 690.

So, we'll choose a random odd number and test if it is prime. On the average, we'll only have to try 690 of them before we find one that is a prime. So, how do we test if a number n is prime?

One naive method is to divide n by all numbers $\leq \sqrt{n}$ and see if there is always a nonzero remainder. The problem is that it would take several universe lifetimes (with numbers the size used in RSA) to verify that a candidate is prime. We said finding p and q didn't need to be as easy as generating or verifying a signature, but forever is too long.

For a long time, there was no sufficiently fast way for absolutely determining that a number of this size is prime. Mathematicians have now invented such an algorithm [AGRA04], but almost no one uses it. That's because there is a much faster test for determining that a number is *almost* certainly prime, and the more time we spend testing a number the more assured we can be that the number is prime.

Recall **Fermat's Theorem**: If p is prime and $0<a<p$, $a^{p-1} = 1 \bmod p$. Does $a^{n-1}=1 \bmod n$ hold even when n is not prime? The answer is—usually not! A primality test, then, for a number n is to pick a number $a<n$, compute $a^{n-1} \bmod n$, and see if the answer is 1. If it is not 1, n is certainly not prime. If it is 1, n may or may not be prime. If n is a randomly generated number of about 300 digits, the probability that n isn't prime but $a^{n-1} \bmod n=1$, is less than 1 in 10^{40} [POME81, CORM91]. Most people would decide they could live with that risk of falsely assuming n was prime when it wasn't. The cost of such a mistake would be either that (1) RSA might fail—they could not decrypt a message addressed to them, or (2) someone might be able to compute their private exponent with less effort than anticipated. There aren't many applications where a risk of failure of 1 in 10^{40} is a problem.

But if the risk of 1 in 10^{40} is unacceptable, the primality test can be made more reliable by using multiple values of a. If for any given n, each value of a had a probability of 1 in 10^{40} of falsely reporting primality, a few tests would assure even the most paranoid person. Unfortunately, there exist numbers n that are not prime but which satisfy $a^{n-1}=1 \bmod n$ for all values of a. They are called **Carmichael number**s. Carmichael numbers are sufficiently rare that the chance of selecting one at random is nothing to lose sleep over. Nevertheless, mathematicians have come up with an enhancement to the above primality test that will detect non-primes (even Carmichael numbers) with high probability and negligible additional computation, so we may as well use it.

The method of choice for testing whether a number is prime is due to Miller and Rabin [RABI80]. We can always express $n-1$ as a power of two times an odd number, say $2^b c$. We can then compute $a^{n-1} \bmod n$ by computing $a^c \bmod n$ and then squaring the result b times. If the result is not 1, then n is not prime and we're done. If the result is 1, we can go back and look at those last few intermediate squarings. (If we're really clever, we'll be checking the intermediate results as we compute them.) If $a^c \bmod n$ is not 1, then one of the squarings took a number that was not 1 and squared it to produce 1. That number is a mod n square root of 1. It turns out that if n is prime, then the only mod n square roots of 1 are ± 1. Further, if n is not a power of a prime, then 1 has multiple square roots, and all are equally likely to be found by this test. A square root other than ± 1 is known as a *nontrivial square root* of 1. So if you can find a nontrivial square root of n, you know n is not prime. For more on why, see §6.3.4.3 *Why a Non-Prime Has Multiple Square Roots of One*.

So if the Miller-Rabin test finds a square root of 1 that is not ± 1, then n is not prime. Furthermore, if n is not prime (even if it is a Carmichael number), at least ¾ of all possible values of a will fail the Miller-Rabin primality test. By trying many values for a, we can make the probability of falsely identifying n as prime inconceivably small. In actual implementations, how many values of a to try is a trade-off between performance and paranoia.

To summarize, an efficient method of finding primes is:

1. Pick an odd random number n with the desired number of bits.
2. Test n's divisibility by small primes and go back to step 1 if you find a factor. (Obviously, this step isn't necessary, but it's worth it since it has a high enough probability of catching some non-primes and is much faster than the next step.)
3. Repeat the following until n is proven not prime (in which case go back to step 1) or as many times as you feel necessary to show that n is probably prime:

   Pick an a at random and compute a^c mod n (where c is the odd number for which $n-1 = 2^b c$). If $a^c = \pm 1$ mod n, pick a different a and repeat. Otherwise, keep squaring the result mod n until you get a value that equals ± 1 mod n, or until the exponent is $n-1$. If the value is $+1$, the value you just squared is a square root of 1 other than ± 1, so the number is definitely not prime. If the exponent is $n-1$, the number is definitely not prime because $a^{n-1} \bmod n \neq 1$. Otherwise, n has passed the primality test for this a, and if you want, you can try another a.

6.3.4.3 Why a Non-Prime Has Multiple Square Roots of One

By the Chinese remainder theorem (§2.7.6), there are four ways that a number y can be a square root of 1 mod n when $n=pq$:

1. y is 1 mod p and 1 mod q. In this case, y will be 1 mod n.
2. y is -1 mod p and -1 mod q. In this case, y will be -1 mod n.
3. y is 1 mod p and -1 mod q. In this case, y will be a nontrivial square root of 1 mod n.
4. y is -1 mod p and 1 mod q. In this case, y will be a nontrivial square root of 1 mod n.

Note that if you find a nontrivial square root of 1 mod n, you can factor n (Homework Problem 9).

6.3.4.3.1. Finding *d* and *e*

How do we find d and e given p and q? As we said earlier, for e we can choose any number that is relatively prime to $(p-1)(q-1)$, and then all we need to do is find the number d such that $ed = 1$ mod $\phi(n)$. This we can do with the extended Euclidean algorithm.

There are two strategies one can use to ensure that e and $(p-1)(q-1)$ are relatively prime.

1. After p and q are selected, choose e at random. Test to see if e is relatively prime to $(p-1)(q-1)$. If not, select another e.
2. Don't pick p and q first. Instead, first choose e, then select p and q carefully so that $(p-1)$ and $(q-1)$ are guaranteed to be relatively prime to e. The next section will explain why you'd want to do this.

6.3.4.4 Having a Small Constant *e*

A rather astonishing discovery is that RSA is no less secure (as far as anyone knows) if e is always chosen to be the same number. And if e is chosen to be small, then the operations of encryption and signature verification become much more efficient. Given that the procedure for finding a $\langle d, e\rangle$ pair is to pick one and then derive the other, it is straightforward to make e be a small constant. This makes public key operations faster while leaving private key operations unchanged. You might wonder whether it would be possible to select small values for d to make private key operations fast at the expense of public key operations. The answer is that you can't. If d were a constant, the scheme would not be secure because d is the secret. If d were small, an attacker could search small values to find d.

Two popular values of e are 3 and 65537.

Why 3? The number 2 doesn't work because it is not relatively prime to $(p-1)(q-1)$ (which must be even because p and q are both odd). 3 can work, and with 3, public key operations require only two multiplications. Using 3 as the public exponent maximizes performance.

As far as anyone knows, using 3 as a public exponent does not weaken the security of RSA if some practical constraints on its use are followed. Most dramatically, if a message m to be encrypted is small—in particular, smaller than $\sqrt[3]{n}$—then raising m to the power 3 and reducing mod n will simply produce the value m^3. Anyone seeing such an encrypted message could decrypt it simply by taking a cube root. This problem can be avoided by padding each message with a random number before encryption so that m^3 is always large enough to be guaranteed to need to be reduced mod n.

A second problem with using 3 as an exponent is that if the same message is sent encrypted to three or more recipients, each of whom has a public exponent of 3, the message can be derived from the three encrypted values and the three public keys $\langle 3,n_1\rangle$, $\langle 3,n_2\rangle$, $\langle 3,n_3\rangle$.

Suppose a bad guy sees m^3 mod n_1, m^3 mod n_2, and m^3 mod n_3 and knows $\langle 3,n_1\rangle$, $\langle 3,n_2\rangle$, $\langle 3,n_3\rangle$. Then by the Chinese Remainder computation, the bad guy can compute m^3 mod $n_1n_2n_3$. Since m is smaller than each of the n_is (because RSA can only encrypt messages smaller than the modulus), m^3 will be smaller than $n_1n_2n_3$, so m^3 mod $n_1n_2n_3$ will just be m^3. Therefore, the bad guy can compute the ordinary cube root of m^3 (which again is easy if you are a computer) to get m.

Now this isn't anything to get terribly upset about. In practical uses of RSA, the message to be encrypted is usually a key for a secret key encryption algorithm and in any case is much smaller than n. As a result, the message must be padded before it is encrypted. If the padding is randomly chosen (and it should be for a number of reasons), and if it is re-chosen for each recipient, then there is no threat from an exponent of 3 no matter how many recipients there are. The padding doesn't really have to be random—for example, the recipient's ID would work fine.

Finally, an exponent of 3 works only if 3 is relatively prime to $\phi(n)$ (in order for it to have an inverse d). How do we choose p and q so that 3 will be relatively prime to $\phi(n)=(p-1)(q-1)$? Clearly, $(p-1)$ and $(q-1)$ must each be relatively prime to 3. To ensure that $p-1$ is relatively prime to 3, we want p to be 2 mod 3. That will ensure $p-1$ is 1 mod 3. Similarly we want q to be 2 mod 3. We can make sure that the only primes we select are congruent to 2 mod 3 by choosing a random number, multiplying by 3 and adding 2, and using that as the number we will test for primality. Indeed, we want to make sure the number we test is odd (since if it's even it is unlikely to be prime), so we should start with an odd number, multiply by 3, and add 2. This is equivalent to starting with any random number, multiplying by 6, and then adding 5.

An even more popular value of e is 65537. Why 65537? The appeal of 65537 (as opposed to others of the same approximate size) is that $65537 = 2^{16}+1$, and it is prime. Because its binary representation contains only two 1s, it takes only 17 multiplications to exponentiate. While this is much slower than the two multiplications required with an exponent of 3, it is much faster than the 3072 (on average) required with a randomly chosen 2048-bit value (the typical size of an RSA modulus in practical use today). Also, using the number 65537 as a public exponent largely avoids the problems with the exponent 3.

The first problem with 3 occurs if $m^3 < n$. Unless n is much longer than the 2048 bits in typical use today, there aren't too many values of m for which $m^{65537} < n$, so being able to take a normal 65537^{th} root is not a threat.

The second problem with 3 occurs when the same message is sent to at least 3 recipients. In theory, with 65537 there is a threat if the same message is sent encrypted to at least 65537 recipients. A cynic would argue that under such circumstances, the message couldn't be very secret.

The third problem with 3 is that we have to choose n so that $\phi(n)$ is relatively prime to 3. For 65537, the easiest thing to do is just reject any p or q that is equal to 1 mod 65537. The probability of rejection is very small (2^{-16}) so this doesn't make finding n significantly harder.

6.3.4.5 Optimizing RSA Private Key Operations

There is a way to speed up RSA exponentiations in generating signatures and decrypting (the operations using the private key) by taking advantage of knowledge of p and q. Feel free to skip this section—it isn't a prerequisite for anything else in the book, and it requires more than the usual level of concentration.

In RSA, d and n are 2048-bit numbers, or about 617 digits. p and q are 1024 bits, or about 308 digits. RSA private key operations involve taking some c (usually a 2048-bit number) and computing c^d mod n. It's easy to say "raise a 2048-bit number to a 2048-bit exponent mod a 2048-bit number", but it's certainly processor intensive, even if you happen to be a silicon-based computer. A way to speed up RSA operations is to do all the computation mod p and mod q, then use the Chinese Remainder Theorem to compute what the answer is mod pq.

So suppose you want to compute $m = c^d$ mod n. Instead of computing c^d mod n, you can take $c_p = c$ mod p and $c_q = c$ mod q and compute $m_p = c_p{}^d$ mod p and $m_q = c_q{}^d$ mod q, then use the Chinese Remainder Theorem to convert back to what m would equal mod n, which would give you c^d mod n. Also, it is not necessary to raise to the dth power mod p, given that d is going to be bigger