Prime Number Hide-and-Seek: How the RSA Cipher Works

Table of Contents

Preface: What is This?

The RSA cipher is a fascinating example of how some of the most abstract mathematical subjects find applications in the real world. Few are the mathematicians who study creatures like the prime numbers with the hope or even desire for their discoveries to be useful outside of their own domain. But every now and then that is exactly what happens.

This text explains the mathematics behind RSA -- how and why it works. The intended audience is just about anyone who is interested in the topic and who can remember a few basic facts from algebra: what a variable is, the difference between a prime number and a composite number, and the like.

The most important mathematical facts necessary for understanding RSA's foundations are reviewed near the beginning. Even if you are familiar with everything covered in these sections, I would recommend that you at least skim through them.

In one or two places, I have specifically targeted an explanation to what I consider to be the average computer programmer, leveraging analogous concepts in programming and general mathematics.

Before getting started, I should make some observations on the mathematical notation used here.

For the most part, where notations for the same idea differ between standard mathematics and the common practices among computer programmers, I have stuck with the mathematicians. This is, after all, a mathematical subject. However, I have deviated in a few places where there was too much opportunity for confusion. I have used * to denote multiplication, and have completely avoided "implied" multiplication (i.e., using PQ as shorthand for P * Q). Since not all web browsers can display superscripts, I have used ^ to denote exponentiation. (This necessitates more parenthesizing than would normally be used.) The mathematician's three-bar congruency symbol is not available, so I have made do with = instead. Variables are always named with a single capital letter.

Finally, please note that throughout the text I use the word number to refer specifically to a positive integer -- what are sometimes referred to as the natural numbers, or counting numbers.

Introduction: The Idea of a Trapdoor Function

What a mathematician refers to as a function is very similar to a function in computer programming. It is, in essence, an abbreviation. For example:

F(X) = 7 * X + 43.

If X happens to be 3, then F(X) will be 64. So, "F(3)" is shorthand for "7 * 3 + 43".

The same function in a C program might look like:

```
int F(int x)
{
    return 7 * x + 43;
}
```

Of course, in a computer program, functions are used to encapsulate all kinds of algorithms, and frequently make use of external variables and the like. In mathematics, however, a function is used solely for the number it returns. And, given a certain number as input, they will always return the same output. (Thus, rand() would not qualify as a mathematical function, unless it were written so that the seed value was passed in as an input parameter.)

Mathematicians often consider how to construct a function's inverse -- taking a function and making a new one that "goes in the other direction", so to speak:

G(X) = (X - 43) / 7.

G(64) is equal to 3, and in general, G(F(X)) is equal to X. Therefore, G is F's inverse. Not all functions are invertible, of course. Clearly, the function:

F(X) = X * 0

cannot be inverted. (Because how could G(F(X)) return X when F(X) is always zero?)

Usually, when you have a mathematical function for which an inverse does exist, constructing it is not too difficult. In fact, it is often transparent. Typically, you can just run through the steps backwards, subtracting where the original function adds, and so on. But can it be done for *every* invertible function?

To put the question in terms of programming, imagine that there are two functions:

```
        int foo(int x);
        int bar(int x);
```

foo() and bar() work like mathematical functions -- they do nothing but compute a return value, and given the same number for input, they will always produce the same output. (And pretend for the moment that this is on a machine where integers can be arbitrarily large.) Suppose you are told that bar() is the inverse of foo(). The statement:

```
        x == bar(foo(x))
```

is always true, as long as x meets foo()'s requirements for a valid argument.

Now, imagine that you have the source code for foo(), but not for bar(). Can you write your own replacement for bar(), just by examining foo()?

It seems that you ought to be able to. There are no secrets as to what foo() does, after all. You can run foo() with different inputs as many times as you like. You already know that bar() exists, somewhere, so you know that it is *possible* to write. Is it guaranteed that you can reconstruct it?

Theoretically speaking, the answer is yes. Given such an function, it is always possible to construct its inverse. However, if we also throw in the tiny constraint that you have to finish before the heat-death of the universe, the answer subtly changes.

There are some special functions that, though what they do is simple enough, and how they do what they do is utterly transparent, figuring out how to *undo* what they do is a diabolical task. Such a creature is a trapdoor function. Anyone can fall through a trapdoor, but only those who know where the hidden lever is can climb back out again.

In 1975, Whitfield Diffie, Martin E. Hellman, and Ralph Merkle realized that a trapdoor function could be the basis for an entirely new kind of cipher -- one in which the decoding method could remain secret even when the encoding method was public knowledge. Diffie and Hellman published a paper in 1976 that described this idea, and offered some examples of weak trapdoor functions. And in 1977, Ronald L. Rivest, Adi Shamir, and Leonard Adleman outlined, in an MIT technical memo, an excellent candidate that became the basis for the RSA cipher.

What follows is a description of that function.

Background, Part I: How to Calculate with Exponents

Here's a quick refresher on how to combine exponents. Recall that:

N^2 = N * N,
N^3 = N * N * N,
N^4 = N * N * N * N,

and so on. For example:

2^7 = 2 * 2 * 2 * 2 * 2 * 2 * 2 = 128.

If we fiddle with exponents for a bit, we will quickly realize that:

N^E * N = N^(E + 1).

So, for example:

2^7 * 2 = 128 * 2 = 256 = 2^8.

Building upon this, we can also see that:

N^E * N * N = N^(E + 2).

But N * N can also be written as N^2:

N^E * N^2 = N^(E + 2).

We can extrapolate from this, and derive a more general rule -- namely:

N^A * N^B = N^(A + B).

And, if we repeated this process on the next level up, we would find that:

(N^A)^B = N^(A * B).

These two simple facts are very useful when handling exponent-laden formulas.

Background, Part II: Modulus Arithmetic

Most computer programmers are familiar with modulus as a "remainder" operator, usually denoted by "%", which gives the remainder of an integer division instead of the quotient. For example:

27 % 12 = 3.

Though the idea is the same, the mechanics here are slightly different from what mathematicians refer to as modulus arithmetic. In essence, modulus arithmetic consists of taking the infinitely long number-line and coiling it around a finite circle. All the numbers that land on the same point along the circle's edge are considered interchangeable, or congruent. Thus, the analogue to the above example in modulus arithmetic would be expressed as:

27 = 3 (mod 12),

or, in words:

27 is congruent to 3, modulo 12.

(Though note that mathematicians actually use a three-barred version of the equal sign to indicate congruency.) In this case, 12 is the modulus that we are working under, and the equation simply tells us that, under a modulus of 12, 27 and 3 are considered to be the same number. Likewise:

11 + 16 = 3 (mod 12)

reads as:

11 plus 16 is congruent to 3, modulo 12.

Modulus arithmetic is sometimes called clockface arithmetic -- if it's currently 11 o'clock, then 16 hours later it will be 3 o'clock. (Of course, the analogy is less perfect when the modulus is something other than 12.)

An important feature of modulus arithmetic is that you can replace the terms of an addition operation with congruent values, and still get the right answer:

16 = 4 (mod 12), therefore
11 + 16 = 11 + 4 = 3 (mod 12).

Another example:

9835 = 7 (mod 12), and
1176 = 0 (mod 12), therefore
9835 + 1176 = 7 + 0 = 7 (mod 12).

Even better, this trick also works with multiplication:

9835 * 1176 = 7 * 0 = 0 (mod 12)

(and, if we check, we will see that, yes, 9835 * 1176 is 11565960, and 11565960 = 0 (mod 12)).

If our modulus was 10, then modulus arithmetic would be equivalent to ignoring all but the last digit in our numbers:

37 = 7 (mod 10),
287 + 482 = 9 (mod 10), and
895 * 9836 = 0 (mod 10).

And, in a sense, a C program does all of its calculations in modulus arithmetic. Since integer calculations in C are permitted to overflow, the high bits silently falling off into the bit bucket, a C program using 32-bit integers is really doing all of its arithmetic modulo 2^32.

As you might imagine, some calculations that are time-consuming and produce huge numbers become trivial in modulus arithmetic. The ability to reduce values to their remainders before doing the actual calculation keeps the calculations from getting out of hand.

Background, Part III: The Fundamental Theorem of Arithmetic

The Fundamental Theorem of Arithmetic states that for every number, there is exactly one way to factor that number into primes -- and vice versa: every selection of primes multiplies into a different number. For example:

1176 = 2 * 2 * 2 * 3 * 7 * 7, or
1176 = 2^3 * 3^1 * 7^2.

It is guaranteed that there is no other way to break 1176 into prime factors. And, certainly, any time you take three 2s, two 7s, and a three, you're always going to get 1176 when you multiply them together. The Fundamental Theorem of Arithmetic assures us that both these things are true for *every* number.

(By the way, this is one of the reasons that 1 is not considered to be a prime number: if it were, then each number would have an infinite number of prime factorizations, all differing by how many 1s were included. Instead, 1 is considered to have no prime factors at all, and we say that a number is prime if it has exactly one prime factor -- namely itself.)

Put another way, the Fundamental Theorem of Arithmetic states that the set of all numbers and the set of all selections of prime numbers are "isomorphic" -- there is a perfect one-to-one mapping between the two. A number is therefore *defined* by its prime factorization.

Background, Part IV: Relatively Prime Numbers

The greatest common divisor (abbreviated GCD) of two numbers is the largest number that evenly divides both of them. For example:

GCD(15, 10) = 5,
GCD(18, 10) = 2,
GCD(21, 10) = 1, and
GCD(170, 102) = 34.

Or, another way to look at it is to say that the GCD is the intersection of the two numbers' set of prime factors:

GCD((2^3 * 3^1 * 7^2), (2^2 * 5^1 * 7^3)) = 2^2 * 7^2, so
GCD(1176, 6860) = 196.

When two numbers have no common factors, their GCD will be 1, and the two numbers are said to be relatively prime (or coprime). For example, we can see in our list up above that 21 and 10 are relatively prime.

Since a prime number has no factors besides itself, clearly a prime number is relatively prime to every other number (except for multiples of itself). And the same can be said of the number 1.

Okay. Enough background material. Let's get to the good stuff.

Euler's Totient Function

Euler's Totient Function is denoted by the Greek letter phi, and is defined as follows:

phi(N) = how many numbers between 1 and N - 1 which are relatively prime to N.

Thus:

phi(4) = 2   (1 and 3 are relatively prime to 4),

phi(5) = 4   (1, 2, 3, and 4 are relatively prime to 5),
phi(6) = 2   (1 and 5 are relatively prime to 6),
phi(7) = 6   (1, 2, 3, 4, 5, and 6 are relatively prime to 7),
phi(8) = 4   (1, 3, 5, and 7 are relatively prime to 8), and
phi(9) = 6   (1, 2, 4, 5, 7, and 8 are relatively prime to 9).

Here is the same definition expressed as C code:

```
phi = 1;
for (i = 2 ; i < N ; ++i)
    if (gcd(i, N) == 1)
        ++phi;
```

(By the way, notice that phi(1) is specially defined to be 1.)

It should be easy to see that phi(N) will be N - 1 whenever N is prime. Somewhat less obvious is the useful fact that phi(N) is also easy to calculate when N has exactly two different prime factors:

phi(P * Q) = (P - 1) * (Q - 1), if P and Q are prime.

(The proof of this fact is left as an exercise for the reader. It's actually not too hard.) Thus, for example:

phi(15) = 2 * 4 = 8   (1, 2, 4, 7, 8, 11, 13, and 14).

The two prime factors cannot be the same number for this to work, and in fact you can see above that phi(9) does not equal 4.

Euler's Totient Theorem

This theorem is one of the important keys to the RSA algorithm:

> If GCD(T, R) = 1 and T < R, then T^(phi(R)) = 1 (mod R).

Or, in words:

> If T and R are relatively prime, with T being the smaller number, then when we multiply T with itself phi(R) times and divide the result by R, the remainder will always be 1.

We can test this theorem on some smaller numbers for which we have already calculated the totient. Using 5 for T and 6 for R, we get:

phi(6) = (2 - 1) * (3 - 1) = 1 * 2 = 2, so
5^(phi(6)) = 5^2 = 25, and
25 = 24 + 1 = 6 * 4 + 1, therefore
25 = 1 (mod 6).

Using 2 for T and 15 for R, we have:

phi(15) = (3 - 1) * (5 - 1) = 2 * 4 = 8, so

2^(phi(15)) = 2^8 = 256, and
256 = 255 + 1 = 17 * 15 + 1, therefore
256 = 1 (mod 15).

Variations on a Theme

Here again is the equation of Euler's Totient Theorem:

T^(phi(R)) = 1 (mod R)

(remembering that T < R, and T and R are relatively prime). Thanks to the way that modulus arithmetic works on multiplication, we can easily see that:

T^(phi(R)) * T^(phi(R)) = 1 * 1 (mod R),

which can be rewritten, using the laws of exponents, as:

T^(phi(R) + phi(R)) = 1 * 1 (mod R),

or:

T^(2 * phi(R)) = 1 (mod R).

If we ran through this sequence again, we would get:

T^(3 * phi(R)) = 1 (mod R).

Clearly, we could keep doing this as many times as we like. So, we can expand on Euler's Totient Theorem, and state a more general corollary:

> If GCD(T, R) = 1 and T < R, then T^(K * phi(R)) = 1 (mod R), where K can be any number.

However, we can state this corrollary another way. Notice that if K can be any number, then K * phi(R) is just the set of numbers that are evenly divisible by phi(R). Or, in other words, the numbers that are congruent to zero, modulo phi(R). So:

> If GCD(T, R) = 1 and T < R, then T^S = 1 (mod R) whenever S = 0 (mod phi(R)).

Now, let's tweak our equation further by multiplying both sides by T:

T^S * T = 1 * T (mod R).

Simplifying leaves us with:

T^(S + 1) = T (mod R).

If we repeat this, we will get:

T^(S + 1) * T = T * T (mod R),

or:

$T^{(S + 2)} = T^2 \pmod{R}$.

Doing this yet again will give us:

$T^{(S + 3)} = T^3 \pmod{R}$,

and so on. This pattern looks familiar, doesn't it?

What makes it interesting this time is that S is *not* a multiple of R, but of phi(R). In other words, we have the rather surprising rule:

$T^E = T^F \pmod{R}$ whenever $E = F \pmod{\phi(R)}$.

(once again, only as long as $T < R$, and T and R are relatively prime).

The Plot Thickens

We are on the edge of something very important. Let's back up a bit and look at this equation more closely:

$T^{(S + 1)} = T \pmod{R}$.

Notice what we have here. We take a number, T, and raise it to a power, and when we do the calculation in modulus arithmetic, we wind up with T again. In short, we have a recipe for a function that returns its own input (presuming that R has been chosen ahead of time, and that T is verified to be relatively prime to R).

If you're thinking to yourself, "What's so interesting about that?", then consider what we would have if we broke this function up into two separate steps. Specifically, let's imagine that we can find two new numbers P and Q such that:

$P * Q = S + 1$, for one of the possible values of S.

Or, more to the point:

$P * Q = 1 \pmod{\phi(R)}$

Then we could write:

$T^{(P * Q)} = T \pmod{R}$,

which is equivalent to:

$(T^P)^Q = T \pmod{R}$,

and *this* is something that can be broken up into two steps:

$T^P = X \pmod{R}$, and then

X^Q = T (mod R).

Now, if you don't see the value in doing this, imagine now that the two steps are performed on separate computers. And that X is sent from the first computer to the second over an insecure phone line....

Does This Really Work?

T stands for the plaintext, the message that is to be sent. P, Q, and R together form the cipher's keys -- P and R make up the public key, and Q and R make up the private key. And X becomes the encrypted message.

Here, again, is the central equation that makes it work:

P * Q = 1 (mod phi(R))

Note here that P and Q will both automatically be relatively prime to phi(R). (Why? Because if either P or Q had a factor in common with phi(R), then P * Q would also have that as a factor. But we know that P * Q divided by phi(R) leaves a remainder of one.) This is important.

Imagine a clockface, with just an hour hand, and imagine yourself placing the hour hand on midnight and then moving it forward by jumps, over and over, each jump covering N hours. If you pick a value for N that is divisible by 2 or 3 (the prime factors of 12), then you will find that you will only hit certain numbers before you return to midnight, and the sequence will then repeat. If N is 2, then the hour hand will visit 12, 2, 4, 6, 8, 10, 12, 2, 4, 6, 8, 10, 12 ...

If, however, your N is relatively prime with 12, then you will wind up hitting every number exactly once before finally returning to midnight 12 jumps later. For example, using 7 for your N, the itinerary would be: 12, 7, 2, 9, 4, 11, 6, 1, 8, 3, 10, 5, 12, ... In addition, the *order* in which you visit the numbers is entirely dependent on what value you pick for N.

In a similar vein, it is important that both P and Q be relatively prime to phi(R). Because of this, we know that every possible value for T, when raised to the power P modulo R, will land on a different number. (Remember that when doing exponents in modulus arithmetic, it is actually phi(R), and not R itself, that determines the length of the cycles.) If this weren't true -- if P, for example, shared a factor in common with phi(R) -- then some values for T could get mapped to the same value for X, and it would clearly be impossible to tell which was originally which. There could not be one value for Q that would correctly map X back to T every time.

The question of which T-values will wind up going to which X-values depends entirely on the value used for P -- and here's the rub for the would-be codebreaker: Just about every possible mapping of T-values to X-values does in fact exist. Somewhere out there is a P that will make that mapping.

If this:

T^P = X
X^Q = T

was the cipher's scheme, there'd be no cipher. With P already being public knowledge, it would be trivial to take an X and compute backwards to T. But, instead, we have this:

T^P = X (mod R)

$X^Q = T \pmod R$

as the cipher's scheme, and that changes everything. The modulus arithmetic erases too much information. There's no way to deduce how many times the hour hand needs to spin around the clockface when Q turns X back into T. Without knowing what Q is, a given X could wind up going to *any* of the possible values for T.

But what is really maddening to our would-be codebreaker is that even when T and P and X are all known, Q still can't be deduced! (Of course, it actually can -- but not necessarily within anyone's lifetime. But we're getting ahead of ourselves.)

So, let's see how to make this recipe work.

Making a Pair of Keys

To construct our own personal cipher keys, we need an appropriate value for R. So, we start by randomly picking two prime numbers, U and V, and multiplying them together:

$R = U * V.$

There are two good reasons for selecting a value for R that has exactly two prime factors. First of all, we have an easy formula for calculating phi(R):

$\mathrm{phi}(R) = (U - 1) * (V - 1).$

Secondly, we want R to be hard to factor. The fewer factors a number has, the longer it takes to find them.

We then need to find values for P and Q such that:

$P * Q = 1 \pmod{\mathrm{phi}(R)}.$

When the numbers have been chosen, P and R together become the public key, and Q and R make up the private key. U and V are no longer needed, and can be forgotten.

An Example

In order to see all this in action, we want to stick with numbers that we can actually work with. So, for our example, we will select the primes 5 and 11 to be our U and V. This gives R a value of 55, and:

$\mathrm{phi}(55) = (5 - 1) * (11 - 1) = 4 * 10 = 40.$

Now, we need to find numbers to fit the equation:

$P * Q = 1 \pmod{40}.$

There are, of course, an infinite number of pairs that will fit this equation. So, let's find one of them.

Our only initial constraint is that P and Q are both relatively prime to 40. So, we can't use numbers that are multiples of 2 and/or 5. We also don't want P and Q to be congruent mod 40, since that would turn our trapdoor cipher into a garden-variety symmetric cipher. Ideally, in fact, we'd prefer that P and Q be relatively prime to each other. Let's start with 7, which we'll assign to P:

  7 * Q = 1 (mod 40).

What would that make Q? If we rewrite this equation to get rid of the unfamiliar modulus arithmetic, we have:

  7 * Q = K * 40 + 1, where K can be any number.

The first value for Q that works is 23:

  7 * 23 = 161 = 4 * 40 + 1.

So we have 7 for P, our public key, and 23 for Q, our private key.

To make our cipher work, you may recall that the values we use for T must be less than R, and also relatively prime to R. We also don't want to use 1 for T, because 1 raised to any power whatsoever is going to remain 1. Finally, the same holds true for R - 1, because R - 1 is congruent to -1, modulo R.

So, we'll take what's left and create the following character set:

2	3	4	6	7	8	9	13	14	17	18	19	24	27
A	B	C	D	E	F	G	H	I	J	K	L	M	N
28	29	37	38	39	41	42	46	47	48	49	51	52	53
O	P	Q	R	S	T	U	V	W	X	Y	Z	sp	*

The message we will encrypt is "VENIO" (Latin for "I come"):

V	E	N	I	O
46	7	27	14	28

To encode it, we simply need to raise each number to the power of P modulo R.

```
 V: 46^7 (mod 55) = 435817657216 (mod 55) = 51
 E:   7^7 (mod 55) =           823543 (mod 55) = 28
 N: 27^7 (mod 55) =   10460353203 (mod 55) =   3
  I: 14^7 (mod 55) =       105413504 (mod 55) =   9
 O: 28^7 (mod 55) =   13492928512 (mod 55) = 52
```

So, our encrypted message is 51, 28, 3, 9, 52 -- or "ZOBG " in our personalized character set.

When the message "ZOBG " arrives on the other end of our insecure phone line, we can decrypt it simply by repeating the process -- this time using Q, our private key, in place of P.

```
 Z: 51^23 (mod 55) = 1879810409774061983350381163649003258651 (mod 55) = 46
 O: 28^23 (mod 55) =             1925904380037276068854119113162752 (mod 55) =   7
 B:   3^23 (mod 55) =                                                       94143178827 (mod 55) = 27
```

```
  G:   9^23 (mod 55) =                         886293811965250109592 9 (mod 55) = 14
  sp: 52^23 (mod 55) = 2938169888454847603243483631603792478208 (mod 55) = 28
```

The result is 46, 7, 27, 14, 28 -- or "VENIO", our original message.

How to Crack RSA

Now, let's switch hats. Imagine that we've just managed to pluck the message "ZOBG " off of our wiretap. By looking up the message's destination in the public-key directory, we find that our message was encrypted with a value of 55 for R and 7 for P. How do we go about decrypting it when we don't know the value for Q?

Well, we know that that:

```
  P * Q = 1 (mod phi(R)),
```

or, without the modulus arithmetic:

```
  P * Q = K * phi(R) + 1.
```

We can divide both sides of the equation by P, which gives us a formula for Q:

```
  Q = (K * phi(R) + 1) / P.
```

K is also unknown, though, so we will try plugging in different numbers for K, and look for values for Q that meet all the necessary constraints.

```
  (1 * 40 + 1) / 7 =   41 / 7          (doesn't divide evenly)
  (2 * 40 + 1) / 7 =   81 / 7          (ditto)
  (3 * 40 + 1) / 7 = 121 / 7           (ditto)
  (4 * 40 + 1) / 7 = 161 / 7 = 23      (this could be it!)
```

Each time we find a candidate for Q, we can test it out on the message. We might get gibberish, in which case we can continue searching. If 23 hadn't worked and we needed to continue the search, it would be pretty obvious that we only needed to test every seventh number, since those are the only numbers which will give us a result that is evenly divisible by 7. Furthermore, we only need to test values up 39, thanks to the modulus arithmetic. So, even though this process involves a brute-force search, it is very simple and very fast.

Well then, so what's the catch? Simply that, in order to do any of this, we first need to know the value of phi(R). Of course, we already know that R has exactly two prime factors, so calculating phi(R) is a snap once we know what those factors are.

Famous last words.

How to Make RSA Uncrackable

Of course, in our case the factors of R can be found by consulting a times table. So it's not much of a challenge. (For that matter, since we're encrypting one character at a time, our coded messages would also be vulnerable to good old-fashioned cryptanalysis).

To make it less easy to find R's factors, we need to pick larger prime numbers for U and V to begin with. If, instead of 5 and 11, we had chosen 673 and 24971, we would have a value of 16805483 for R, and phi(R) would be 16779840. (This would also give us enough room to encrypt more than one byte at a time, which seriously reduces the vulnerability to cryptanalysis.) Looking for a P and Q pair is no longer something you want to do with pencil and paper, of course, but it took me less than three minutes to find the usable pair 397 and 211333 -- including the time it took to write and debug a Perl script.

On the other hand, it also took me less than three *seconds* to run "factor" on 16805483 to obtain 673 and 24971. Armed with those numbers, it wouldn't take much longer to derive 211333 from 397. So even these numbers aren't close to being large enough. We need *really* big numbers.

Well, we can certainly find huge values for R that are difficult to factor. But how far can we go before it becomes too difficult for us to use the number in the first place?

Huge Exponents in Modulus Arithmetic

The problem is this: The bigger R gets, the bigger P and Q will be, and P and Q are to be used as exponents! Even the relatively tame-looking

```
9^(9^9)
```

produces a number over 350 million decimal digits long. How are we going to be able to encrypt anything without needing terabytes of storage?

The trick is that we only need to calculate these exponential values modulo R. As always, modulus arithmetic simplifies things a great deal.

Let's revisit our example, and look at how we could decrypt the number 28, remembering that R = 55 and Q = 23:

```
28^23 (mod 55) = ?
```

To start with, we look at Q's binary representation. 23 in binary is 10111, which means that:

```
23 = 16 + 4 + 2 + 1, or
23 = 2^4 + 2^2 + 2^1 + 2^0.
```

We can now break the exponential calculation apart into several smaller ones:

```
28^23 = 28^(2^4 + 2^2 + 2^1 + 2^0)
      = 28^(2^4) * 28^(2^2) * 28^(2^1) * 28^(2^0)
      = 28^(2 * 2 * 2 * 2) * 28^(2 * 2) * 28^2 * 28
      = (((28^2)^2)^2)^2 * (28^2)^2 * 28^2 * 28.
```

This may look like anything but an improvement, at first. But on a closer examination, you'll see that we actually have many repeated subterms. This simplifies matters, particularly when we take advantage of the fact that we are calculating in modulo 55.

We compute the first square in modulus arithmetic:

28^2 = 784 = 14 (mod 55).

By substituting this value into our equation:

28^23 = (((28^2)^2)^2)^2 * (28^2)^2 * 28^2 * 28 (mod 55),

we get:

28^23 = ((14^2)^2)^2 * 14^2 * 14 * 28 (mod 55).

Now by computing that square:

14^2 = 196 = 31 (mod 55),

we will have:

28^23 = (31^2)^2 * 31 * 14 * 28 (mod 55).

And, finally:

31^2 = 961 = 26 (mod 55), and
26^2 = 676 = 16 (mod 55);

and so:

28^23 = 16 * 31 * 14 * 28 (mod 55).

We can continue to take advantage of the modulus when we do the final multiplications:

```
28^23 = 16 * 31 * 14 * 28 (mod 55)
      = 16 * 31 * 392 (mod 55)
      = 16 * 31 * 7 (mod 55)
      = 16 * 217 (mod 55)
      = 16 * 52 (mod 55)
      = 832 (mod 55)
      = 7 (mod 55)
```

Lo and behold: 7, the same result as when we did it the hard way.

This binary technique is really no different than how computers normally compute integer powers. However, the fact that we can break the process down to successive multiplications allows us to apply the modulus at every step of the way. This assures us that at no point will our algorithm have to handle a number larger than (R - 1)^2.

Huge Factors in Modulus Arithmetic

The magic of modulus arithmetic will also ensure that it's possible to find our P and Q pair. Remember that, after we've selected some humongous value for R, we need to find values to fit:

```
  P * Q = 1 (mod phi(R)),
```

or, without the modulus arithmetic:

```
  P * Q = K * phi(R) + 1, where K is any number.
```

After picking a likely value for P -- which probably will *not* be a conveniently small number like 7 -- we will need to find a matching Q. By rewriting the above as:

```
  P * Q - phi(R) * K = 1,
```

with known values for P and phi(R), we have what is called a Diophantine equation. This really just means that we have more unknowns than equations. However, it also means that algorithms exist for solving it, the most well-known one being Euler's. (One thing you quickly discover when you dabble in number theory is that a lot of things are named after Euler.) While it's still not something you'd want to do with pencil and paper, it doesn't involve anything more advanced than a whole lot of long division. In short, it's something that a computer can do in a relatively brief amount of time.

(Another side benefit to using really big numbers: If we pick P and Q to be larger than the range of numbers that we plan to encrypt, we are automatically guaranteed that all of them will be relatively prime to R, and we won't need to consider that issue at encryption time. Which is good, because by then we won't have access to P and Q anymore.)

Safety in Numbers

Okay. So we know that the whole process is still practical, even if R is immense. But all of this is still moot unless we can select an R in the first place. R has to be the product of two prime numbers, don't forget. If we want R to be so big that it can't be factored easily, how are we going to find those factors to begin with?

It turns out that there is an interesting little asymmetry here. Determining whether or not a number is prime happens to be a relatively cheap process.

One of the most famous methods for testing a number for primality uses Fermat's Little Theorem. Here is the version of this Theorem that we're interested in:

> If P is prime, then N^(P - 1) = 1 (mod P) is true for every number N < P.

Does this seems suspiciously reminiscent of Euler's Totient Theorem? It should. Euler was the first person to publish a proof of Fermat's Little Theorem, and his Totient Theorem is a generalization of Fermat's. You can see this for yourself by remembering that phi(P) = P - 1 when P is prime.

Of course, as far as proofs go, this theorem is only useful for proving that a given number is composite. For example, it just so happens that:

```
  4^14 (mod 15) = 268435456 (mod 15) = 1,
```

even though 15 is no prime. Nonetheless, it is also true that:

3^14 (mod 15) = 4782969 (mod 15) = 9, and
5^14 (mod 15) = 6103515625 (mod 15) = 10.

On the other hand, 17, which *is* prime, results in 1 every time:

3^16 (mod 17) = 43046721 (mod 17) = 1,
4^16 (mod 17) = 4294967296 (mod 17) = 1,
5^16 (mod 17) = 152587890625 (mod 17) = 1, and so on.

So, if we want to know if a number is prime, we can run it through this test, using (say) 2 as the base. If anything besides 1 results, we know with certainty that the number is composite. If the answer is 1, however, we try the test again with 3, and then 4, and so on. If we keep getting back 1 as the result, it soon becomes rather unlikely that the number is anything but prime.

Unlikely, mind you, but not impossible. There are a handful of numbers which pass this test for every base, but which are not prime. Called Carmichael numbers, they are far more rare than the prime numbers -- but, like the primes numbers, there are still an infinite number of them. So we wouldn't want to rely on this test alone.

Fortunately, there are other tests for primality which are more reliable. But they all have at least one thing in common with this test: When they reject a number, they tell us only that the number can be factored. The test results give us no information at all as to what the factors might be. How unfortunate!

Unfortunate for the mathematicians, that is. Very fortunate for us.

Summing Up

The basic truth is that, in order to find the factors of a composite number, we're pretty much stuck with using brute force: Divide the number by all the primes you can get your hands on until one of them goes in evenly. There are plenty of ways to improve on this approach (the Number Field Sieve currently being the best), but they are complicated, and all they do is allow you to narrow the scope of the search. They don't reduce the search enough to make this problem tractable in general.

Nor is it likely that new approaches will, either! The real issue is that the encrypting and decrypting algorithms have a running time that is linear with respect to the length of R. That is to say, doubling the number of digits in R doubles the amount of time (roughly) needed to encrypt, decrypt, and to select the two primes to make a key with. But the algorithms for factoring R have a running time that is exponential with respect to the length of R. That is to say, the time (roughly) doubles with every few digits! (Because every digit added to R makes it ten times larger, and thus multiplies the number of potential candidates for its measly two factors.)

So if a new technique is suddenly found that makes it a trillion times faster to factor a number, all we have to do is increase the size of R we use by enough digits, and the situation will be right back where it started -- and all it means to us is that it takes a little bit longer to send and receive our messages. Already some people are using keys that, in order to factor with the Number Field Sieve, would require more energy than exists in the known universe.

An illustration: At the time of my writing, one of the largest general numbers that has been independently factored was the number used as the modulus for the RSA-140 challenge. (By "general numbers", I'm excluding creatures like Mersenne numbers and Fermat numbers, which have specialized factoring techniques that are inapplicable elsewhere.) It was completed on February 2, 1999. Now, the record previous to this was the RSA-130 number, and the process of factoring it was estimated as taking a total of 1000 MIPS-years of computer time. RSA-140, a number only 10 decimal digits longer, required *twice* that amount.

This, finally, is the heart of what makes RSA a trapdoor function: the gap between obtaining a number with two prime factors, and rediscovering the factors from the number itself. And the gap just keeps expanding as the numbers get larger.

The breakthrough that would completely destroy RSA's security would be an algorithm that actually produced a number's factors directly, instead of merely narrowing the search's scope. Such a thing has not been proven impossible, and it may well be that such a proof will never be found. But considering that prime numbers have been studied for thousands of years, and given the renewed attention that has been focused on this problem in the last few decades, the likelihood of the existence of such an algorithm appears very remote. Discovering one would change the face of number theory as much as RSA has changed the face of cryptography.

However -- if this *were* to happen, there are other trapdoor functions out there, waiting to be found. Whatever the future of RSA may be, the trapdoor cipher has certainly changed the face of cryptography forever.

References

1. Clawson, Calvin C.: "Mathematical Mysteries", 1996, Plenum Press. (Clawson devotes an entire chapter to the mathematics behind RSA, and it is this that gave me the inspiration to create this text.)

2. Benson, Donald C.: "The Moment of Proof", 1999, Oxford University Press. (Like the previous one, this fine book discusses the mathematics of RSA alongside of many other topics.)

3. Gardner, Martin: "Penrose Tiles to Trapdoor Ciphers", 1989, W.H. Freeman & Co. (This is another anthology of Gardner's wonderful columns for "Scientific American", and includes the column which was the first widely published description of the RSA cipher -- the one which set the NSA to frantically running around in circles.)

4. Ribenboim, Paulo: "The Little Book of Big Primes", 1991, Springer-Verlag. (The title should actually be "The Little Book of Big Number Theory" -- the book is chock full of theorems and conjectures that relate to prime numbers.)

5. Devlin, Keith: "All the Math that's Fit to Print", 1994, The Mathematical Association of America. (A collection of short columns from The Manchester Guardian, in which I learned that the set of Carmichael numbers has been proven to be infinite.)

6. Wells, David: "The Penguin Dictionary of Curious and Interesting Numbers", 1986, Penguin Books. (I had to pull this out at the last minute to find out how many digits were in 9^(9^9). For the curious whose libraries lack this little gem, the exact number of digits is 369693100.)

Thanks to readers Joel Sturman and Lee Sloan for pointing out errors and ommisions in previous drafts.

Texts