Arrrgh Essay

It's back to school to learn how to do multiplication and division for RSA

by Julian Bucknall

I unlocked the airport baggage locker. The key had arrived in the mail with a terse note detailing how to find this particular locker at this particular airport. Inside the locker was a portable tape player with a tape already mounted. I pressed play and the tape started.

'Good morning, Julian. One question I've always had is how the RSA encryption algorithm works. You get these two keys, one private and one public, and miraculously you can encrypt with one of them and decrypt with the other. It's like magic. It is magic! Just *how* does it work? Your mission, should you choose to accept it, is to explain the RSA public key algorithm in terms I can understand. This tape will self-destruct in 5 seconds. Good luck, Julian.'

I recognized the voice and intonation of Our Esteemed Editor, even though it sounded like he was trying to disguise it with a handkerchief. Either that or he was using those cheap factory-surplus cassettes again. Sweat started breaking out on my forehead: this was obviously a high priority mission, something that I couldn't ignore. My column was on the line.

The tape started fizzing and emitting smoke. I slammed the locker door shut before the cops came over.

Public Key Encryption

RSA, then. Let's start by explaining public key (or asymmetric key) algorithms and why they're important in the grand scheme of things. As I explained last month, private key algorithms like DES suffer from a vexing problem. Alice and Bob, our two encryption protagonists, must agree on and exchange a key before they can communicate with each other in a secure manner. If they can meet face-to-face, then they can just decide on the private key there and then. If they can't meet face-to-face, how can they exchange a key without their eavesdropping enemy, Eve, listening in? As I explained last time, if Eve manages to get hold of the key as Alice sends it to Bob, then she can send Bob a dummy key instead. At that point, Eve is in control, she can intercept encrypted messages from Alice, decrypt them, then either send the original or a fake message to Bob, encrypted with her dummy key. And vice-versa. Alice and Bob have no way of knowing that this is happening.

So what can Alice and Bob do if they are separated and cannot physically meet? Alice needs a way of encrypting the secret key so that she can send it to Bob, and he needs to be able to decrypt it safely. Of course, Eve cannot be able to decrypt this key exchange message. Alice can't use a private key encryption algorithm, otherwise it's just the same problem over again: how do Alice and Bob agree on the private key used to encrypt and exchange the message with the private key? What Alice and Bob use instead is public key cryptography. She encrypts the message holding the secret key with Bob's public key. He's published this key so that *anyone* can send him a message that only he can decrypt. When Bob gets a message encrypted with his public key, he decrypts it with the matching secret key that only he knows. It is impossible (or, rather, very, very hard) to deduce Bob's secret key from his well-known public key, so Eve is stuck. Once Alice has sent the encrypted message to Bob and he's decrypted it, they can ignore the asymmetric algorithm and start using the private key algorithm again, secure in the knowledge that they have bamboozled Eve again.

Sounds fabulous, but how does it work? What is the relationship between the public and secret key that gives them the ability to use this clever scheme? Why is it hard-to-impossible to calculate the secret key given the public key? And why bother with private key algorithms at all, given this miraculous public key algorithm?

The underpinning of the RSA algorithm (named after its inventors: Ron Rivest, Adi Shamir, and Leonard Adleman) is a branch of mathematics known as modulo arithmetic. Before we delve into the mathematics and the implementation of the algorithm, let us see an example.

Simple RSA Example

Bob's public key is expressed as two numbers, n and e, known as the *modulus* and the *exponent*. We'll use some small numbers to illustrate what's going on, such as n=437 and e=17 (in reality n is a number with about 155 decimal digits, not three). Don't worry about where these numbers come from yet. To encrypt, Alice takes her message and converts it into numbers. Each of the numbers that she chooses must be less than 437. Suppose the message she wanted to send was the password 'SECRET'. One method for her to encode it as decimal numbers would be to take the ordinal value of the ASCII characters: 53, 45, 43, 52, 45, and 54. For each of these numbers, x, she would calculate the encrypted value, y, using the following formula: $y = x^e \bmod n$; that is, take x, raise it to the power e, and calculate the remainder after dividing by n. A pretty nasty calculation to do by hand. For example, taking the first value, 53, raising it to the power 17 is a 30-digit number, which equals 318 modulo 437 (take this from me!). The other values will get converted to 68, 99, 357, 68, and 82.

She then sends these six numbers to Bob. Bob takes out his secret key. This is expressed as two numbers again, this time known as *n* and *d*. *n* is the same number that Alice used (the modulus), 437. *d*, however, is different than *e* although it is an exponent again. Its value is 233 and it's the really secret part of Bob's secret key. For each of the encrypted numbers, *y*, Bob calculates the original *x* by using the formula: $x = y^d \bmod n$. Using the first value, 318, we raise it to the power 233, giving a 584-digit number, which happens to be 53 modulo 437 (again, take this from me!). Wow, it's the original value again!

I can see that some of you are shaking your heads. What I've just done seems just as magical as before, and now you're taking a lot on trust as well. How the heck did I calculate the remainder of 318 to the power 233 when divided by 437? I can assure you I didn't calculate the 584-digit intermediate result! And where did I come up with the values for *n*, *e*, and *d*?

At this point, there's nothing for it but to introduce some mathematics. First I'll show some identities for modulo arithmetic (the proof of which is beyond this article: indeed I only learnt the various theorems to prove them in the first year of my maths degree). Figure 1 has the identities we shall need. In particular take note of the first identity. It says, in plain English, that if you take two numbers, multiply them and then take the modulus with respect to *n*, you'll get the same answer as if you'd taken the individual numbers, taken their modulus, multiplied them and then taken the modulus once again. And this is how I did those large calculations; for example, 318 to the power of 233, modulus 437. At every multiplication, you take the modulus; the intermediary results can all be stored in a longint, so you just keep on multiplying the intermediary result by 318 and finding the remainder after dividing by 437.

```
(a*b) mod n = ((a mod n) * (b mod n)) mod n

Fermat's Little Theorem:
if p is prime and 1 < a < p-1 then a^p-1 mod p = 1
```

➤ *Figure 1: Modulo arithmetic theorems required for RSA.*

Easy Exponentiation

In fact, there is another optimization we can do: an algorithm known as the binary square and multiply method. Suppose you wanted to work out 5 to the power 8. The simplistic way is to take the 5 and multiply it by 5 seven more times (usually coding it as a little loop):

$$5^8 = 5*5*5*5*5*5*5*5$$

In other words, we'd be performing 7 multiplications. Doesn't seem too bad, but it's likely we'd be calculating a larger power with RSA (for example, in our simple illustration we had an exponent of 233 at one point). Is there anything we can do to reduce the number of multiplications? Yes: for an exponent of 8: square the 5, square the result, and then square that second result. In other words we'd be calculating $((5^2)^2)^2$. This takes 3 multiplications: the original square, the square of that result and then the square of *that* result. In fact, any exponent that is a power of 2 can be broken down in the same way; for example 5^{16} would require 4 successive squaring operations. (If you have a calculator with an x^2 key, calculating y^x when *x* is a power of 2 means pressing that key the right number of times.)

Well that's all fine and dandy, but what happens if the exponent is not so obliging, and is *not* a power of 2? Well, we rewrite it so that it becomes a series of multiplications of intermediary results that have exponents that are powers of 2. For example, we can rewrite 5^7 as $5^4 * 5^2 * 5$ and we only have to do 5 multiplications instead of 6 originally (OK, I know it doesn't sound like an improvement with these small numbers, but bear with me).

In fact, we can make another optimization and remove another multiplication:

$$5^7 = (5^2 * 5)^2 * 5$$

A total of four multiplications. How do we take advantage of this in code? How do we know how to split the exponent 7 into a 4, a 2, and a 1? Well, this is nothing more than the binary representation of 7, with each clear bit meaning 'square' and each set bit 'square, then multiply by the number'. We start off with the most significant bit of the exponent. This gives rise to the implementation of the binary square and multiply algorithm shown in Listing 1. (A note for those readers who don't have Delphi 5: all the code this month was written using Delphi 5; I'm afraid I didn't have time to try it with other compilers, or to convert it to Delphi 1.) Notice that this listing actually shows the algorithm using modulo arithmetic, and this was how I really calculated that 53^{17} mod 437 equals 318, and 318^{233} mod 437 equals 53.

But where did I get the numbers 437, 17 and 233? More mathematics. 437 is the product of two primes: 19 and 23 (I can see some of you are waking up here: this is something you've heard about with RSA). The two primes are usually known in the computer science trade as *p* and *q*. The number 17 is entirely arbitrary but it cannot divide *p*-1 (18) or *q*-1 (22). The best bet is a prime again; one of the smallest that satisfies this condition (I could equally well have chosen 5 or 7, for example). The reason we choose one of the smallest is that it is going to be part of our public key, and we're being nice to people who want to send us messages: this is the exponent *e* (meaning *encrypt*) with which they'll be raising the numbers that they're encoding. The smaller the better for them. We now calculate *d* (meaning *decrypt*) such that:

```
function PowerAndMod(a, e, n : integer) : integer;
{-calculate a^n mod n}
var
  Exponent : integer;
  BitCount : integer;
  i        : integer;
begin
  {reverse the bits in e and put them in Exponent}
  Exponent := 0;
  for i := 1 to 32 do begin
    if Odd(e) then
      Exponent := (Exponent shl 1) or 1
    else
      Exponent := (Exponent shl 1);
    e := e shr 1;
  end;
  {pop off clear bits until reach first set bit; this will
   be the most significant bit of the original exponent}
  BitCount := 32;
  while not Odd(Exponent) do begin
    Exponent := Exponent shr 1;
    dec(BitCount);
  end;
  {OK, we're ready for the loop}
  Result := 1;
  {for all the bits in the original exponent...}
  for i := pred(BitCount) downto 0 do begin
    {sqaure the intermediate result}
    Result := (Result * Result) mod n;
    {if the current bit is set, multiply by a}
    if Odd(Exponent) then
      Result := (Result * a) mod n;
    Exponent := Exponent shr 1;
  end;
end;
```

➤ *Listing 1: Binary square and multiply algorithm.*

$$(d*e) \bmod (p\text{-}1)\ (q\text{-}1) = 1$$

For our simple example, we can work out that 233 * 17 mod (18*22) equals 1, and so $d = 233$.

Yet more magic, eh? Where does this other modulus come from? Why does it work?

Suppose we start out with a plaintext number x and encode it to a number y:.

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

Now decode this mess and see what falls out. Remember, we want to calculate $y^d \bmod n$.

$$\begin{aligned} &y^d \bmod n \\ &= (x^e \bmod n)^d \bmod n \\ &= x^{ed} \bmod n \end{aligned}$$

Now what? This doesn't seem to help us, mainly because we need a further piece of the mathematics puzzle. Enter *The Chinese Remainder Theorem* and another couple of theorems, one from Leonard Euler (pronounced *Oiler*) and another from Pierre de Fermat (pronounced *Fairmah*). I'm not going to bother using them here; suffice it to say that, using these results, we can prove that $x^{ed} \bmod n$ is equal to $x \bmod n$ for all possible values of x if we know that $n = pq$ and $de \bmod (p\text{-}1)(q\text{-}1) = 1$. Any mathematics book can prove the equality. Anyway, it is possible to show that the basic encryption and decryption work.

Wide Word Arithmetic

By now you may have realized that we will need to have some chunky large integer arithmetic routines. The key for RSA encryption is usually expressed as the number of bits in its modulus n. The standard these days is 512 bits at a minimum. We're going to need to multiply two 512-bit numbers to produce a 1,024-bit number, divide a 1,024-bit number by a 512-bit number to give a 512-bit number and a 512-bit remainder. So, for fun, let's take a detour now into *Algorithms Alfresco Wide Word Arithmetic* (AAWWA, the scream you make when you try and write the code the first time). It's instructive and we'll definitely need it in a moment.

We'll first declare the Wide Word type: `TaaWideWord`. This will be a class with which we can do 1,024-bit unsigned integer arithmetic. For ease of computation and understanding, at the expense of efficiency, I'll make it an array of bytes (the alternative might be an array of words or longwords) because it makes the arithmetic much easier: I can use longwords throughout. I'll follow the Intel standard and make the least significant byte the first byte in the array,

```
procedure AddPrim(x, y : PByteArray; aCount : integer);
var
  Temp  : LongWord;
  Carry : LongWord;
  i     : integer;
begin
  Carry := 0;
  for i := 0 to pred(aCount) do begin
    Temp := x^[i] + y^[i] + Carry;
    x^[i] := Temp mod 256;
    Carry := Temp div 256;
  end;
  x^[aCount] := Carry;
end;
procedure TaaWideWord.Add(x : TaaWideWord);
begin
  FDigitCount := MaxI(DigitCount, x.DigitCount);
  AddPrim(Digits, x.Digits, DigitCount);
  if (Digits^[DigitCount] <> 0) then
    inc(FDigitCount);
  if (DigitCount > 128) then begin
    SetToMax;
    raise Exception.Create('TaaWideWord.Add: overflow')
  end;
end;
```

➤ *Listing 2: Wide Word addition.*

and the most significant byte the last byte. In the following discussion I shall call the individual bytes the *digits* of the number; as you can see, each digit has 256 possible values, like decimal digits have 10 possible values, 0 to 9.

The first and easiest arithmetic operation is addition, adding two Wide Words together. This is just like you did it at school. Starting from the least significant place, add the digits from the two numbers together, and make sure you carry overflows onto the next place. Here, of course, an overflow is the part of the intermediary result greater than 256. Listing 2 has this simple routine. (The `DigitCount` property is used to maintain the number of significant digits in the Wide Word.)

Subtraction is a little more involved: instead of carries you have borrows. Listing 3 shows this implementation of Wide Word school arithmetic. At this coding rate we'll be finished before bedtime!

Multiplication, then. If your school was like mine you learnt to do it like this:

```
   654
   321 *
 ------
   654
  1308
 1962
 ------
 209934
```

Well, in Wide Word School, we do it the same way, except that this time we don't keep the individual multiplication sub-results ready to add them all together at the end; we do the addition immediately after the multiplication of the individual digits.

If you follow through the routine in Listing 4 with the numbers in our example (the Y number is 654, the X number is 321) you'll see the multiplication result being built up as we go along:

```
1*4:  000004
1*5:  000054
1*6:  000654
2*4:  000734
2*5:  001734
2*6:  013734
3*4:  014934
3*5:  029934
3*6:  209934
```

In Wide Word School, coming across the next operation is when most pupils think about becoming accountants instead of programmers. The very word is enough to strike terror in the heart of the most fearless student: division. It pretty well did for me, too.

Way back when we learnt it like this:

```
         654
321 ) 209944
      1926
       1734
       1605
        1294
        1284
          10
```

In other words, 209,944 divided by 321 is 654 with remainder 10. But, just think about the actual mechanics of it for a moment. In the first step, we have to *guess* how many times 321 goes into 2,099. Get it wrong, we then have to increment or decrement our guess until we have it right. And it's still not over! We then have to guess how many times 321 goes into 1,734, and so on. Each step by laborious step requires a guess and some refinement until we have it right when subtracting the new quotient digit times the divisor from the intermediate value is less than the divisor. Nasty.

How can we implement an algorithm that has this element of 'guessing'? Use random numbers, perhaps? Actually, it turns out to be a little more structured than that. Firstly we need to make sure that the most significant digit of the divisor is 5 or more. If it isn't, multiply the divisor by 2 (maybe more than once) until it is. Double the dividend the same number of times. The quotient of this new division will be the same as the original; however, the division will have been simplified to a large degree. In our case, we need to double the divisor and the dividend once to make the most significant digit of the divisor greater then or equal to 5. We are now dividing 419,888 by 642.

We can now make our guess to the quotient digit as follows. Take the most significant digit of the divisor (in our case 6). Take the two most significant digits of the dividend (41). Divide to give 6 (notice that this division is nothing more than knowing the multiplication tables of the digits). If the answer happened to be greater than 9, the largest digit, we force the answer to 9. This is our guess at the first digit of the quotient. Amazingly enough, it can be proven that, if the most significant digit of the divisor is greater than 5, the actual quotient digit we just calculated is either this one, call it *q*, or it will be *q*-1, or maybe *q*-2 at a pinch. That's it. Period. Interested readers can check out Knuth's *The Art of Computer Programming Volume II* for the proof. We see that 6 is the correct answer. At this point the division looks like this:

```
         6??
642 ) 419888
      3852
       3468
```

We do the same thing: divide 6 into 34 to make 5. It's correct again:

```
         65?
642 ) 419888
      3852
       3468
       3210
        2588
```

The final 'guess' is the result of dividing 25 by 6, which is 4. Correct again. The final division is:

➤ *Listing 3: Wide Word subtraction.*

```
function SubtractPrim(x, y : PByteArray; aCount : integer) : integer;
var
  Temp   : integer;
  Borrow : integer;
  i      : integer;
begin
  Borrow := 0;
  for i := 0 to pred(aCount) do begin
    Temp := x^[i] - (y^[i] + Borrow);
    if (Temp < 0) then begin
      inc(Temp, 256);
      Borrow := 1;
    end else
      Borrow := 0;
    x^[i] := Temp;
  end;
  Result := Borrow;
end;
procedure TaaWideWord.Subtract(x : TaaWideWord);
var Borrow : integer;
begin
  if (DigitCount < x.DigitCount) then
    Borrow := 1
  else
    Borrow := SubtractPrim(Digits, x.Digits, DigitCount);
  if (Borrow <> 0) then
    raise Exception.Create('TaaWideWord.Subtract: negative result');
  wwRecalcDigitCount;
end;
```

```
procedure MultiplyPrim(x, y : PByteArray; var aXCount :
  integer; aYCount : integer);
var
  InxX    : integer;
  InxY    : integer;
  MaxX    : integer;
  Carry   : LongWord;
  Temp    : LongWord;
  Result  : array [0..128] of byte;
begin
  {initialize the result}
  FillChar(Result, sizeof(Result), 0);
  {if either x or y had no digits it was zero;
   answer is zero}
  if (aXCount = 0) or (aYCount = 0) then begin
    Move(Result, x^, sizeof(Result));
    aXCount := 0;
    Exit;
  end;
  {for every Y digit we shall multiply by all the X digits}
  MaxX := aXCount;
  for InxY := 0 to pred(aYCount) do begin
    {if the Y digit is zero there'll be nothing to do in
      this loop, so only do work if it is non-zero}
    if (y[InxY] <> 0) then begin
      {start off with a carry of zero}
      Carry := 0;
      {for each X digit, we multiply it with the current Y
         digit, add in the carry from the previous step, and
         add the current value of the result digit}
      for InxX := 0 to pred(MaxX) do begin
        Temp := (LongWord(x[InxX]) * y[InxY]) +
          Result[InxX + InxY] + Carry;
        {strip off the carry and set the result digit}
        Result[InxX + InxY] := Temp mod 256;
        Carry := Temp div 256;
      end;
      {make sure that any remaining carry is saved}
      Result[InxY + MaxX] := Carry;
    end;
  end;
  {return the answer}
  Move(Result, x^, sizeof(Result));
  {return the new number of digits}
  inc(MaxX, aYCount);
  while (x^[pred(MaxX)] = 0) do
    dec(MaxX);
  aXCount := MaxX;
end;
procedure TaaWideWord.Multiply(x : TaaWideWord);
begin
  if ((DigitCount + x.DigitCount) > 128) then begin
    SetToMax;
    raise Exception.Create('TaaWideWord.Multiply: overflow');
  end;
  MultiplyPrim(Digits, x.Digits, FDigitCount, x.DigitCount);
end;
```

➤ *Listing 4: Wide Word multiplication.*

```
          654
     ________
642 ) 419888
      3852
       3468
       3210
        2588
        2568
          20
```

But what about the remainder? Well, we take the remainder from this division and halve it the number of times we had to double the original divisor. We doubled once, and so we halve the remainder of 20 once to give 10. Ta-da!

In Wide Word School though we're not using decimal arithmetic: our digits have 256 different possible values. No problem, we do exactly the same thing. First double the divisor until the most significant digit (ie, byte) is greater than or equal to 128. Double the dividend the same number of times. To calculate the quotient digit, we divide the most significant byte of the divisor into the word value formed from the two most significant bytes of the dividend. (The theorem above still applies even though the radix has changed from 10 to 256: we are guaranteed to only check at most three digits, *q*, *q*-1, or *q*-2, for the current quotient digit).

Proceed like this until we have a remainder less than the divisor. At this point we have the correct quotient, but the remainder may need some work. Halve this remainder the same number of times as we originally doubled the divisor, and we can return this resulting value as the remainder of the original division. Listing 5 shows the full implementation. Despite having to create three temporaries on the heap, the routine is fairly nippy.

Finding Primes

RSA boils down to first finding two large primes *p* and *q* (we select primes expressible in 256 bits), multiplying them together to give *n* (512 bits), finding an *e* such that it does not divide either (*p*-1) or (*q*-1), and then calculating the *d* that goes with the *e*. At that point we publish *n* (=*p***q*) and *e* and wait for the encrypted messages to come flooding in, when we can decrypt them with *n* and *d*. We can, if we want, throw away our *p* and *q* since they're not needed any more. Simple, huh?

Well, no, not really. How on earth do we find primes of length 256 bits? This is not exactly going to be the same as testing that 1,021 is a prime (which is a mere 10-bit number). Recall that Fermat's Little Theorem (as it's usually known) tells us that $a^{p-1} \bmod p = 1$ for a prime modulus *p* and for any *a* between 1 and (*p*-1). So, if we take a random number between 1 and *p*-1, where *p* is our supposed prime number (note that we first make sure that *p* is odd!), raise it to the power of (*p*-1), calculate the remainder after dividing by *p*, then we can conclude that *p* is not prime (is *composite*) if the answer is not 1. If the answer is 1, then we could have a prime, or we could have been unlucky and just hit a value that just works for that *p*. We can then try again, and again, and again for a while with different initial random numbers. If we continually calculate the answer 1, it becomes more and more likely that we have a prime number. This process is known as *probabilistic primality testing*. It does not explicitly *prove* that a given number is prime, but we can reduce the probability that it is composite to an arbitrarily small value after several tests.

If Fermat's Little Theorem shows our chosen *p* is composite, we try *p*+2 and then test *p*+4, and so on. According to the Prime Number Theorem (the number of primes less than *n* can be approximated by *n*/ln(*n*)), we are assured of hitting upon a prime number fairly quickly. Also the Prime Number Theorem tells us that the number of 256-bit primes is extremely large (being 2^{247}, a 75-digit number) so it's not likely that we'll hit upon a prime number that someone's seen before.

Unfortunately, although an extremely powerful result, Fermat's Little Theorem does not tell us that if $a^{n-1} \bmod n = 1$ for all *a* then *n* is prime (there is a series of composite numbers, known as *Carmichael numbers*, where this property is true; the first such is 561 = 3*11*17). So we could have found a Carmichael number with the above algorithm, and that is composite by definition. How can we remove the possibility that we latch onto a Carmichael number, a number that masquerades as a prime number, at least as far as Fermat is concerned?

Enter the Miller-Rabin algorithm. This algorithm is a variant on the probabilistic primality testing algorithm I've just given, except that it makes another test in each loop. Again it tries several values of *a* to apply to Fermat's Theorem. The extra test it makes is to see whether a non-trivial square root of 1 is ever found (the trivial square roots of 1 modulus *p* are either 1 or *p*-1). (There's another theorem which states that if $x^2 \bmod p = 1$ and *p* is prime, then *x*=1 or *p*-1.) So if we find in our tests an *x* such that *x* is neither 1 nor *p*-1 and yet $x^2 \bmod p = 1$ then obviously *p* is composite. An example using the first Carmichael number 561 is 67: $67^2 \bmod 561$ is 1.

Listing 6 shows this remarkable algorithm. It first calculates a 256-bit random number by using the standard `Random` procedure, and makes sure that it is odd by setting the least significant bit. It then sets the most significant bit to ensure that it is a proper 256-bit number. At that point, we start into the primality testing loop. We generate a random 255-bit odd number and run this through Fermat's Theorem together with a test for a non-trivial square root of 1 (this latter part is done by the `wwIsWitness` method: we're testing to see if *a* is a witness to *p* being composite). And that is that.

For those of you who are wondering, this routine is pretty slow. The first few times I ran it, I had to add debugging `writeln` statements just to make sure I hadn't entered an infinite loop. It's dreadful. On my 550MHz machine, it takes about 7 seconds to calculate and test a 256-bit prime. Nasty. Luckily though, we don't have to do this all that often. In fact, for a single RSA key that we can use again and again, we only have to calculate two primes. So, who cares if it takes a good 15 seconds? Once it's done, it's done.

Calculating Exponents

So, what's next on the RSA front? Calculating *e*, that's what. Pretty easy, this one. Try dividing (*p*-1) and (*q*-1) by 3, and then by 5, and then by 7, etc, until we reach a small prime that leaves a remainder for both divisions. We can do this one in our sleep. However, the standard RSA key generation algorithm always sets *e* to 3 and then generates *p* and *q* such that both (*p*-1) and (*q*-1) are not multiples of 3. This means that you won't suffer from the small probability that our sequence of possible *e* values continues for some long series of primes. Why do it this way?

It turns out that testing a binary number for division by 3 is a simple affair, just as it is for decimal numbers. In the latter case, we add up all the digits in the number we're testing. Add up all the digits in the number thus produced, and continue like this, adding up the digits in our intermediate results,

➤ *Listing 5: Wide Word division.*

```
procedure TaaWideWord.Divide(x : TaaWideWord;
  aRem : TaaWideWord);
var
  Divisor, Dividend, Test : TaaWideWord;
  TestCompare : integer;
  InxQ, InxX, Factor : integer;
  SigDigit : byte;
  q        : longint;
begin
  {first there's some code to get rid of the easy cases...}
  ..omitted code..
  {if we reach this point we actually have to do some work:
    the dividend is greater than the divisor, neither is
    zero, and the divisor is not 1}
  {allocate and initialize some temporaries}
  Test := nil;
  Divisor := nil;
  Dividend := nil;
  try
    Divisor := TaaWideWord.Create;
    Divisor.Assign(x);
    Dividend := TaaWideWord.Create;
    Dividend.Assign(Self);
    Test := TaaWideWord.Create;
    {after the division we shall be holding the quotient;
      make sure we're zeroed out for now}
    SetToZero;
    {make sure most significant digit of divisor is greater
     than 128; retain the multiplicative factor to do that}
    Factor := Divisor.wwNormalize;
    {multiply the dividend by the same factor}
    if (Factor <> 1) then
      Dividend.wwMultiplyByDigit(Factor);
    {if the most sigdigit of the dividend is greater than or
      equal to that of the divisor, increment the number of
      digits in the dividend; this'll help once we jump into
      the division itself}
    if (Dividend.Digits^[pred(Dividend.DigitCount)] >=
      Divisor.Digits^[pred(Divisor.DigitCount)]) then
      inc(Dividend.FDigitCount);
    {note that InxQ will be the position of the start of the
      part of the dividend we're looking at; in other words
      digits InxQ..InxX of the dividend form the part of the
      dividend we're dividing by the divisor}
    {calculate the position of the most significant digit of
      both the quotient and dividend}
    InxQ := Dividend.DigitCount - Divisor.DigitCount;
    FDigitCount := InxQ;
    InxX := Dividend.DigitCount;
    {get the most significant digit of the divisor}
    SigDigit := Divisor.Digits^[pred(Divisor.DigitCount)];
    {while we are still calculating quotient digits...}
    while InxQ >= 0 do begin
      {calculate our first estimate for this quotient digit
        q (it may be too large of course but the final value
        will be within 2 of this)}
      q := ((LongWord(Dividend.Digits^[InxX]) * 256) +
        Dividend.Digits^[InxX-1]) div SigDigit;
      {refine q if necessary}
      if (q <> 0) then begin
        {it's only a digit so force it in range}
        if (q >= 256) then
          q := 255;
        Test.Assign(Divisor);
        Test.wwMultiplyByDigit(q);
        while (ComparePrim(@Dividend.Digits^[InxQ],
          Test.Digits, Test.DigitCount+1) < 0) do begin
          dec(q);
          Test.Assign(Divisor);
          Test.wwMultiplyByDigit(q);
        end;
      end;
      {save this digit for the quotient}
      Digits^[InxQ] := q;
      {subtract}
      if (q <> 0) then begin
        SubtractPrim(@Dividend.Digits^[InxQ], Test.Digits,
          Test.DigitCount+1);
      end;
      {we've done this digit, now do the next}
      dec(InxX);
      dec(InxQ);
    end;
    {we now have the quotient, calculate number of digits}
    wwRecalcDigitCount;
    {set up the remainder}
    Dividend.wwRecalcDigitCount;
    aRem.Assign(Dividend);
    if (Factor <> 1) then
      aRem.wwDivideByDigit(Factor);
  finally
    Divisor.Free;
    Dividend.Free;
    Test.Free;
  end;
end;
```

until we end up with a single digit. If this is 0, 3, 6, or 9, then the original number was divisible by 3. For example, let's test 123,456,789 to be divisible by 3. Add up the digits: 1+2+3+...+9. This is 45. Add the 4 and the 5 to make 9. This is divisible by 3 so the original number also was (123,456,789 = 3 * 41,152,263). Simple, huh?

To test that a binary number is divisible by three, it's a little more convoluted, but still pretty simple. Count all the set bits at the odd bit positions. Similarly count all the set bits at the even bit positions. Subtract one count from the other and if the result is divisible by 3 then the original number also was. For example, let's test 1100100. Count the set bits by looking at the first bit (we count from the right), the third bit, the fifth bit and so on. The bits we see are 0, 1, 0, 1, for a count of 2. Now start at the second bit, and look at it and the fourth bit, the sixth bit and so on. The bits we see are 0, 0, 1, for a total of 1. Subtracting 1 from 2 gives 1, which is not divisible by 3. Hence, 1100100 itself is not divisible by 3 (this binary number expressed in decimal is 100, so our conclusion is correct). Let's test 1100011 by this method. There are two counts of 2, and subtracting one count from the other makes 0, which is divisible by three and hence so is the original number (which was 99 in decimal).

We're moving right along now. Next? We need to calculate *d*. It sounds easy: recall that *d* is a number such that

$$(d*e) \bmod m = 1$$

where $m = (p-1)*(q-1)$. This is known as calculating the *multiplicative inverse* of *e* modulo *m*. Think about it for a while with our example numbers above, where m=396 (ie, 18*22) and e=17. One method with these small numbers would be to test all possible values of *d* (there are only 395 of them for our example) until we found one that worked. There's another theorem that tells us that this search will succeed because *e* is prime and doesn't divide *m*. But, with RSA, *m* is going to be a 512-bit or 155-digit number! Not only will you be able to make a cuppa coffee doing a lot of those kinds of divisions, you would also be able to grow the coffee bush from a seed, reap the harvest and roast your own beans. There must be a better way, and so it proves with Euclid coming to our rescue this time with Euclid's extended algorithm (sorry, there's no way I can show why this works in an article!).

At this point, we can publish our public key and lock away our secret key. (By the way, the standard method of publishing an RSA key is to use the ASN.1 format, which is also beyond the scope of this article: see the internet document RFC 2459 for details.)

Encryption Methodology

We still need to discuss how to encrypt and decrypt arbitrary messages, although we know the methodology in principle. I won't be going to the full code details, but essentially the process goes like this. We split the original message into blocks, each block having 53 bytes (or maybe less). We then pad these 53 bytes to 64 bytes by adding some extra bytes onto the beginning. The reason for this is to make the encryption more secure and to obviate certain types of attack. The first padding

➤ *Listing 6: Testing a Wide Word to be prime.*

```
function TaaWideWord.wwIsWitness(
  a, NMinus1 : TaaWideWord) : boolean;
var
  d : TaaWideWord;
  Rem : TaaWideWord;
  x : TaaWideWord;
  i : integer;
begin
  {allocate temporaries}
  d := nil;
  x := nil;
  Rem := nil;
  try
    d := TaaWideWord.Create;
    x := TaaWideWord.Create;
    Rem := TaaWideWord.Create;
    d.Assign(1);
    {we're going to be calculating a^(n-1) mod n by the
     square and multiply method}
    for i := pred(NMinus1.BitCount) downto 0 do begin
      x.Assign(d);
      d.Multiply(d);
      d.Divide(Self, Rem);
      if (Rem.DigitCount > Self.DigitCount) then
        writeln('error');
      d.Assign(Rem);
      if d.IsOne then begin
        if (not x.IsOne) and
          (x.Compare(NMinus1) <> 0) then begin
          Result := true;
          Exit;
        end;
      end;
      if NMinus1.GetBit(i) then begin
        d.Multiply(a);
        d.Divide(Self, Rem);
        if (Rem.DigitCount > Self.DigitCount) then
          writeln('error');
        d.Assign(Rem);
      end;
    end;
    Result := not d.IsOne;
  finally
    d.Free;
    x.Free;
    Rem.Free;
  end;
end;

function TaaWideWord.IsPrime : boolean;
var
  a       : TaaWideWord;
  NMinus1 : TaaWideWord;
  TestNum : integer;
  i       : integer;
begin
  {allocate temporaries}
  a := nil;
  NMinus1 := nil;
  try
    a := TaaWideWord.Create;
    NMinus1 := TaaWideWord.Create;
    NMinus1.Assign(Self);
    NMinus1.SubOne;
    {test PrimeTestCount times}
    for TestNum := 1 to PrimeTestCount do begin
      {set a to a random number between 1 and ourselves}
      repeat
        a.SetToZero;
        a.FDigitCount := Random(DigitCount) + 1;
        for i := 0 to pred(a.DigitCount) do
          a.Digits^[i] := Random(256);
        a.wwRecalcDigitCount;
      until (not a.IsZero) and (Compare(a) > 0);
      if wwIsWitness(a, NMinus1) then begin
        Result := false;
        Exit;
      end;
    end;
    Result := true;
  finally
    a.Free;
    NMinus1.Free;
  end;
end;
```

byte is $00. The next padding byte is $00, $01 or $02 and is known as the block type. We'll discuss what the values mean in a moment. We then have at least eight bytes known as the padding string. The values of the bytes in the padding string depend on the block type value. Then we have a single $00 byte, followed immediately by the at most 53 bytes in our message block. We thus have a 64-byte block that looks like

```
<$00><BT><PS><$00><data>
```

where `BT` is the block type and `PS` the padding string.

If the block type is $00, the padding string consists of at least eight $00 bytes. If the block type is $01, the padding string is at least eight $FF bytes. Otherwise the padding string consists of at least eight random non-zero bytes. The choice of which to use is the encryptor's, he or she can use different block types for different blocks, or use the same one throughout. We'll see that the decryptor can work out what's going on. The final block of the message is likely to be less than 53 bytes, of course, and the block type to choose is $01 or $02. In this case the padding string must be more than eight bytes.

There are a couple of comments to make about this seemingly bizarre way to make up a 64-byte block. The first one is that you don't *have* to have 53 bytes of your data in each 64-byte block: you can have less if you want (indeed the final block almost certainly does have less), but you cannot have more. Notice that there is no explicit length of data value in the 64-byte block. The decryptor must be able to determine this from the block itself. With block type $00 this could be difficult; after all with this block type, all of the padding bytes turn out to be $00 and if the data part of the block starts off with $00 bytes we'll have difficulty determining where the padding stops and the data begins. With the other two block types we know that we'll see a $00 byte at the end of the padding bytes and that's just before the data. So, in general, we only use block type $00 with 53-byte blocks and never use it with the final block, or we only use block types $01 and $02.

Making the first byte $00 means that the 512-bit number formed from the 64-byte block is guaranteed to be less than *n*, our modulus (we'll swing it so that this initial byte is going to be the most significant byte in the number). This is a requirement of the RSA mathematical algorithm.

For block type $02, it's good to use different random streams every time we encrypt something. This makes it more difficult for the attacker to try and find similar encodings for similar messages.

Once we have our 64-byte block, we convert it into a 512-bit number (of which the most significant eight bits are guaranteed to be zero), raise it to the power of 3, modulus *n*. We then take the 512-bit result and convert it back into a 64-byte block, which we can output. The original message, when encrypted, thus grows in size: the encrypted size is going to be ((original size + 52) div 53) * 64 bytes.

And the decryptor? Well, he does the reverse operation of course. He divides up the encrypted message into 64-byte blocks. For each block, he converts it into a 512-bit number. He then raises that number to the power *d* (his private exponent), modulus *n*. The resulting 512-bit number is then converted back into a 64-byte block. He then starts to read the bytes in the block from the start, trying to extract the data. The first byte must be zero (if not, there's an error: either the public key used to encrypt the message was the wrong one, or the secret key he's using is wrong). The second byte must be zero, 1 or 2 (if not, an error occurred). For 2, he then scans until he reaches a zero byte and the data is the remaining part of the block that he can output. For block type 1, he must scan $FF bytes until he reaches a zero byte, and the data is formed from the remaining bytes (if there is a non $FF byte before the zero byte, there's an error). For block type $00, he must scan nine further $00 bytes and the data is the remaining part of the block (if those nine bytes are not zero, there's an error).

Conclusions

A quick note on security is in order. Breaking the RSA algorithm is understood to be equivalent to factoring the modulus *n* into its two prime factors *p* and *q* (once we have these values, we can easily work out *d*). This is mathematically hard: there is no simple way to establish the factors of a very large number, especially when it is known that there are two and the two are of the same magnitude. Current research and algorithms have made it possible to factor 100-digit numbers in a 'reasonable' amount of time. Factoring 155-digit moduli (such as this article's 512-bit modulus) *can* be done, but it would take an inordinate amount of time and computing power.

On a completely unrelated note, RSA is a patented algorithm and the patent expires on 20th September 2000.

And that's it. I'm bushed and I can imagine you are too. From a simple enough request: 'explain the RSA algorithm', we've skirted traditional modulo arithmetic, implemented Wide Word arithmetic operations, learned how to test numbers to be prime, saw the quick way of checking for divisibility by 3, and finally looked at how to encrypt and decrypt message blocks using RSA.

Next month, it's going to be much easier all round. I'm going to review some readers' messages that were sent to me and answering them. And, at long last, I shall be deciding who won January's competition (something I've been promising for a while).

Julian Bucknall has had it with arithmetic, especially division. Long live calculators! His email is julianb@turbopower.com