Fourth International Derive TI-89/92 Conference
Liverpool John Moores University, July 12 – 15, 2000

Using DERIVE to Explore the Mathematics Behind the RSA Cryptosystem

Johann Wiesenbauer, Vienna, Austria.
e-mail: **j.wiesenbauer@tuwien.ac.at**

Summary

The goal of this workshop is to investigate the mathematical pillars on which the RSA cryptosystem rests, namely the (extended) Euclidean Algorithm, Fermat's Little Theorem and the Square-and-Multiply method. Furthermore, it is shown how the Chinese Remainder Theorem (CRT) can be used to speed up the decryption and the generation of signatures in RSA considerably. A number of common attacks on RSA are discussed. All algorithms and attacks are illustrated by using the new powerful programming language of Derive 5.

Introduction

Although the RSA cryptosystem, invented by R. Rivest, A. Shamir and L. Adleman (cf. [3]), has been around for little more than two decades, it is today ubiquitous in modern telecommunication: RSA is used by Web browsers to ensure Web traffic, it is used to ensure "pretty good privacy" (PGP) and authenticity of e-mails and, last but not least, it is widely used in electronic credit card payment systems. (If you registered for this conference via Internet, you will know that its organizers also used RSA for the secure transmission of your data.)

The importance of RSA in modern telecommunication is certainly one of the reasons why it should be represented in our educational system. Even more important is the fact, the mathematical theory behind it is for one thing very appealing and for another involves some of the most fundamental theorems and algorithms of all mathematics. Take this as a kind of apology for dealing with this subject once more, although it has already been treated by many authors including myself (cf. [4], [5]). Another justification is the fact that unlike previous versions the new Derive for Windows 5 (DfW5 for short) now offers everything that is needed to treat this topic properly due to a lot of new powerful features. In fact, one of the main goals of this workshop is exactly to prove this claim.

What is the RSA cryptosystem all about?

Although dealing with the question in the headline might be carrying coals to Newcastle in this forum, I will do it all the same - not only for the sake of completeness, but also in order to introduce some basic notations for the following.

Basically, RSA belongs to the so-called public key cryptosystems. As the name suggests, the public key is public and is used to set up the encryption E of messages. Here, E is a so-called one-way trapdoor function, i.e. it is virtually infeasible in a reasonable time to invert E without some additional information – the "trapdoor"-, which is the private key.

To be more specific, suppose Bob wants to create an RSA public key and a corresponding private key. Then he should do the following:

1. Generate two large random (and distinct) primes p and q, each roughly the same size and compute n: = pq and v: = lcm(p-1,q-1).
2. Select integers e,d such that $1 < e,d < v$ and $ed \equiv 1 \bmod v$, where d should be large (roughly the size of v).
3. Publicize the pair (n,e), which is his public key, and keep secret his private key d.

If Alice wants to send a message to Bob, she is supposed to do the following:

1. Obtain Bob's authentic public key (n,e).
2. Represent the message as an integer m in the interval [0,n-1].
3. Compute $c = m^e \bmod n$.

Bob in turn can easily recover m from c, by computing $m = c^d \bmod n$.

Before dealing with the whys and wherefores in detail, let's simply set up an RSA environment using Derive and compute an example.

In the first place, we need a routine text_to_number(t) that converts an alphanumerical string t into a decimal number. By default we assume that t uses the full ASCII character set. The corresponding decimal number is then simply the decimal number whose digits with respect to the base 256 are exactly the ASCII-numbers of the characters of t (both read from left to right). In some textbooks though, a more primitive source coding is used with the following correspondence: A=01, B=02,...,Z=26 and space = 00 (no other characters are allowed). If you want to use this kind of source coding, you should set the optional parameter o to "plain". (Note that currently inverted commas are left out in listings of programs!)

```
text_to_number(t, o := full, n_ := 0) :=
  Prog
    t := NAME_TO_CODES(t)
    Loop
      If t = []
        RETURN n_
      If o = plain
        n_ := 100·n_ + MOD(FIRST(t) - 32, 32)
      If o = full
        n_ := 256·n_ + FIRST(t)
      t := REST(t)
```

What follows are two simple examples. The second one yields a decimal number m with 238 digits, which will be used as message in the following.

```
text_to_number(ONLY CAPITAL LETTERS AND SPACES ARE ALLOWED USING THE OPTION PLAIN, plain)

151412250003011609200112001205202005181900011404001916010305190001180500011212152305040007
  211909140700200805001516200915140016120109 14

text_to_number(By default the full ASCII character set, including e.g. ☺, ♥, ♀, †, <, @
  etc., is at your disposal.)

n := text_to_number(By default the full ASCII character set, including e.g. ☺, ♥, ♀, †,
  <, @ etc., is at your disposal.)

676336558586382958568276672176313130372310707197296015566791272650194857518117329860050700
  261142905609849648634753457806632916496542350921775854241331122809839589856646731242610
  363950835440957784653689881987868466942952015824567804715292574 2

                                                                    237
                                APPROX(n) = 6.763365585·10
```

If we choose the modulus n to be a number with about 1024 bits, i.e. about 309 decimal digits, which is a very common size nowadays, we can encrypt the full text in one run only. (In general, if n is given beforehand, it might be necessary to split up m into several decimal blocks each < n.)

The following routine generates an RSA environment, such that that modulus n has k bits and its prime factors p and q have about half as many bits. Furthermore, e is chosen to be the fixed Fermat prime 65537. Then e is almost certainly coprime to p-1 and q-1 and we can use the built-in function inverse_mod(a,m) to find the unique positive solution d < v of the congruence $ex \equiv 1 \bmod v$ with v=lcm(p-1,q-1). Furthermore, the special of e is obviously advantageous when it comes to forming powers m^e mod n, as m^{e-1} mod n is simply the result of 16 squarings mod n starting with m.

```
RSA_init(k) :=
  Prog
    e := 2^16 + 1
    Loop
      p := NEXT_PRIME(RANDOM(2^CEILING(k/2)))
      If GCD(p - 1, e) = 1 exit
    q := FLOOR((2^(k - 1) + RANDOM(2^(k - 1)))/p)
    Loop
      q := NEXT_PRIME(q)
      If p ≠ q ∧ GCD(q - 1, e) = 1 exit
    n := p·q
    d := INVERSE_MOD(e, LCM(p - 1, q - 1))
    ok
```

Before using this routine for the very first time, it is very important that the global variables p,q,n,e,d are initialised by a statement like

```
[p :=, q :=, n :=, e :=, d :=]
```

otherwise Derive won't recognise these variables after exiting the routine RSA_init(k). (Is this a bug or just a "feature" of DfW5? Frankly, I don't know.)

```
                                RSA_init(1024) - ok
p =

  69540162702087335381707491708798571077716165396424054469712010183639489504141095151103~
  69897754884106663232215611443409569663906790449485803251035237163647

q =

  19542285746985436566263637425667039203732812340294740481126741538734268139109162201750~
  59201862344663835706349682899137882264509804188571099053053243835018 9

n =

  13589737304160495978276233986481529901846077203497657914949743942721427511433991236562~
  75290826698656227076580342818064247765272062917193587899691724254213636206199651530112~
  33417597032001796315162802551681792078447985985562184669946612672830877282622202236246~
  16766487332652144738863450061073154462654848637928 3

                                    e - 65537

d =

  47736292854277549768206842333764331579168796756293327869761822375969297151337704236842~
  31368558437182813300876767629683544837927912540505910712849716046850789537091230847432~
  18383888099821366684803119302429923596149274099621584248479076166720206366327670780064~
  42135235936960597234968866340709606774834112744457
```

Above you can see how the values of p,q,n,e and d typically look like after calling our routine RSA_init(k) with k=1024. In particular, it is easy to see that n really has exactly 1024 bits:

```
CEILING(LOG(n, 2)) = 1024
```

Given these values encryption and decryption is actually very easy now.

```
                e
encrypt(m) := MOD(m , n)

                d
decrypt(m) := MOD(c , n)

(c := encrypt(m)) =

  11767261248955595570542026666383714474267919279487743217289339709676088540796744609278~
  13000514903530046875674177872189577937170520941918021650756975689664653798849769427447~
  35999760329200521779041263951413018680761170879125577304546407966997045548696235486197~
  28177707938237855815098637313001411811160673772144 9

(M := decrypt(c)) =

  67633655858638295856827667217631313037231070719729601556679127265019485751811732986005~
  07261142905609849648634753457806632916496542350921775854241331122809839589856646731242~
  61363950835440957784653689881987868466942952015824567804715292574 2

                                    M - m - 0
```

Well, I still owe you the routine that converts a decimal number back to a meaningful text.

```
number_to_text(n, o := full, t_ := ) :=
  Loop
    If n = 0
       RETURN t_
    If o = plain
       Prog
         t_ := ADJOIN(CODES_TO_NAME(MOD(n, 100) + 64 - 32·0^MOD(n, 100)), t_)
         n := FLOOR(n, 100)
    If o = full
       Prog
         t_ := ADJOIN(CODES_TO_NAME(MOD(n, 256)), t_)
         n := FLOOR(n, 256)
```

Will it pass the acid test?

```
number_to_text(m) - By default the full ASCII character set, including e.g. ⊟, ♥, ♀, †,
  <, @ etc., is at your disposal.
```

Yes! For the first time, it is possible now to go through all steps of the RSA encryption and decryption including the conversions of texts to decimal numbers and vice versa.

Just a few words to the use of RSA for digital signatures. When the RSA cryptosystem was first publicised in the August 1977 issue of *Scientific American*, the inventors posed a small problem to the readers. To prove their authenticity they also published an electronic signature s along with the public key (e,n), namely

```
e := 9007

n :=

  1143816257578888676692357799761466120102182967212423625625618429357069352457338978305 9~
  7123563958705058989075147599290026879543541

s :=

  1671786115038084424601527138916839824543690103235831121783503844692906265544879223711 4~
  4905095786086556624965779748400040570203 73
```

claiming that s is simply the text “FIRST SOLVER WINS ONE HUNDRED DOLLARS” encrypted by their private key d (known only to them!) rather than their public exponent e. And here is proof by Derive that this signature was valid:

```
number_to_text(MOD(s^e, n), plain) = FIRST SOLVER WINS ONE HUNDRED DOLLARS
```

In fact, by this exchange of public exponent e and private exponent d, everybody can read the resulting ciphertext using e as exponent for decryption and compare the outcome with what was claimed by the sender, but nobody can forge the signature without knowing d. Needless to see that this special option of RSA to prove the authenticity of the sender adds very much to its popularity.

The Extended Euclidean Algorithm or How Does INVERSE_MOD(a,m) work?

We now turn to the mathematics behind RSA. The first question regards the built-in function INVERSE_MOD(a,m) that computes the inverse of a mod m under the assumption that a and m are coprime and which was used above with a = e and m = lcm(p-1,q-1). How does it work?

It simply makes use of one of the oldest algorithms of all mathematics, the Euclidean algorithm. In fact, it has been called by D. Knuth (cf. [2]) “the granddaddy of all algorithms, because it is the oldest nontrivial algorithm that has survived to the present day”. It can be used not only to compute d = gcd(a,b) of any two integers a and b in a very efficient way, but in its extended form also to find integers x,y, such that d = xa + yb. Assuming that $a \geq 0$ and $b > 0$, which is in view of gcd(a,b)=gcd($|a|,|b|$) no loss of generality, and setting $r_0 := a$ and $r_1 := b$ we can always form the following “chain of divisions” that eventually terminates with a division whose remainder is 0, as the numbers $r_1, r_2, ..., r_n$ form a strictly decreasing chain of positive integers.

$$r_0 = q_0 r_1 + r_2 \quad \text{with } 0 < r_2 < r_1$$

$$r_1 = q_1 r_2 + r_3 \quad \text{with} \quad 0 < r_3 < r_2$$

$$\ldots$$

$$r_{n-2} = q_{n-2} r_{n-1} + r_n \quad \text{with} \quad 0 < r_n < r_{n-1}$$

$$r_{n-1} = q_{n-1} r_n$$

It is easy to see that $\gcd(r_0, r_1) = \gcd(r_1, r_2) = \ldots = \gcd(r_{n-1}, r_n) = r_n$, hence $r_n = \gcd(a, b)$. Furthermore, all r_i, i = 0,1,..,n, can be expressed in the form $r_i = x_i a + y_i b$. This is trivial for i = 0 and i = 1, because of $r_0 = a = 1a + 0b$ and $r_1 = b = 0a + 1b$. Assuming that this had been already been proven for all i < k with $1 < k \le n$, it is then valid also for i = k, due to

$$r_k = r_{k-2} - q_{k-2} r_{k-1} = (x_{k-2} a + y_{k-2} b) - q_{k-2}(x_{k-1} a + y_{k-1} b) =$$
$$= (x_{k-2} - q_{k-2} x_{k-1}) a + (y_{k-2} - q_{k-2} y_{k-1}) b$$

In particular, we see that $x_k := x_{k-2} - q_{k-2} x_{k-1}$ and $y_k := y_{k-2} - q_{k-2} y_{k-1}$, k = 2,3,...,n, which means that the recursion formulas for x_k and y_k are exactly of the same form as for the r_k.

How could a DERIVE-program look like that computes d=gcd(a,b) along with integers x and y such d=xa+yb? First, let me point out that there are already two programs in the utility files NUMBER.MTH and NUMBER.DFW, respectively, that deal with this task, namely

```
EXTENDED_GCD(a, b, d_, x_) :=
  Prog
    If b = 0
       If a = 0
          RETURN [0, [0, 0]]
          RETURN SIGN(a)·[a, [1, 0]]
    d_ := GCD(a, b)
    a := a/d_
    b := b/d_
    x_ := INVERSE_MOD(a, ABS(b))
    [d_, [x_, (1 - a·x_)/b]]


EXTENDED_GCD(a, b, q_, r_) :=
  Prog
    a := [a, [1, 0]]
    b := [b, [0, 1]]
    Loop
      If FIRST(b) = 0 exit
      q_ := FIRST(a)/FIRST(b)
      q_ := ROUND(RE(q_)) + ROUND(IM(q_))·î
      r_ := a - q_·b
      a := b
      b := r_
    If FIRST(a) = 0
       RETURN [0, [0, 0]]
    Loop
      If 0 ≤ PHASE(FIRST(a)) < π/2
         RETURN a
      a :* î
```

Both compute [d, [x,y]] for integers a and b, but are not very satisfactory for our purposes. The first one is very fast by calling the built-in function INVERSE_MOD( ), which in turn calls an internal function like EXTENDED_GCD( ) on a LISP-level. Thus from a didactic point of view, we have a perfect

vicious circle here! The second also works for Gaussian integers (cf. [6] for a detailed discussion of these numbers) and without any reference to INVERSE_MOD( ), but is unnecessarily complicated, if you are interested in integers a, b only.

Hence let's make an extremely streamlined version of the second program that covers only the case, where a and b are nonnegative integers. It could look like this:

```
XGCD(a, b, q_, r_) :=
  Prog
    a := [a, [1, 0]]
    b := [b, [0, 1]]
    Loop
      If FIRST(b) = 0
        RETURN a
      q_ := FLOOR(FIRST(a)/FIRST(b))
      r_ := a - q_·b
      a := b
      b := r_
```

I hope that this form shows (at last!) that the Euclidean algorithm is incredibly simple even in its extended form!

Let's go back to the question how to compute the inverse of a mod m on condition that a and m are coprime and nonnegative integers. Since gcd(a,m)=1, we could use xgcd(a,m) to find integers x and y such that xa+ym=1. But this means that xa ≡ 1 mod m, in other words x is the inverse of a mod m we are looking for! Since y is not needed at all here, when designing a routine inv_mod(a,m), we could streamline our xgcd( ) even a bit further by leaving out the components referring to y in the vectors above. (This is what a good program is all about: Modifications are very easy due to its general structure!)

```
inv_mod(a, m, q_, r_) :=
  Prog
    a := [a, 1]
    m := [m, 0]
    Loop
      If FIRST(m) = 0
        If FIRST(a) = 1
          RETURN a↓2
          RETURN ?
      q_ := FLOOR(FIRST(a)/FIRST(m))
      r_ := a - q_·m
      a := m
      m := r_
```

Concluding this chapter let's note an important consequence of the Euclidean algorithm, which is needed in the following

Lemma (Euclid): If a,b,c are integers, such that $a \mid bc$ and gcd(a,b)=1, then $a \mid c$.

Proof: Since gcd(a,b)=1 we can find integers x and y such that xa+yb =1. If we multiply this equation with c, we get xac+ybc=c, where a is a divisor of xac and y(bc), hence also of xac+y(bc)= c.

Corollary: If p is a prime, then $p \mid ab$ implies $p \mid a$ or $p \mid b$. More generally: If a prime p divides a product, it must divide one of its factors.

Proof: If $p|ab$, then either $p|a$ or gcd(p,a)=1. In the latter case we have $p|b$ according to Euclid's Lemma.

Fermat's (Little) Theorem and Its Important Consequences

Now that we know how to construct the RSA-keys e and d by means of the Extended Euclidean Algorithm, the question arises why RSA works with these numbers as it did in our example. This is where the following important theorem, often referred to as Fermat's (Little) Theorem, comes into play:

Theorem: If p is any prime and a any integer not divisible p, then $a^{p-1} \equiv 1 \mod p$.

Proof: Let's consider the multiples a, 2a, 3a, ... , (p-1)a. We first claim that these numbers are all incongruent mod p. Assuming on the contrary that $ia \equiv ja \mod p$, where $0 < j < i < p$ w.l.o.g., leads to $p|(i-j)a$ and hence to $p|a$ or $p|i-j$ according to the corollary above. Both cases are clearly impossible. In a similar way one can see that none of the elements is 0 mod p, i.e. divisible by p.

Since a, 2a, 3a, ... , (p-1)a are all different mod p and incongruent to 0, they must be exactly the elements 1,2,...,p-1 apart from the order. Hence, if we form the product of a,2a,...,(p-1)a and 1,2,...,p-1 the results should be equal mod p, i.e.

$$a(2a)(3a)...((p-1)a) \equiv 1 \cdot 2 \cdot 3 ... \cdot (p-1) \mod p.$$

By regrouping this implies that

$$p|(a^{p-1}-1)(p-1)!$$

Since p doesn't divide any of the factors of (p-1)!, it must divide $a^{p-1}-1$ according to the last corollary, which is exactly what we wanted to prove.

Fermat's Little Theorem can be slightly generalised to

Corollary: For any prime p and any integer a the congruence

$$a^{1+k(p-1)} \equiv a \mod p$$

holds for all $k \geq 0$.

Proof: This is trivial, if p is a divisor of a, because in this case both sides of the congruence are 0 mod p. On the other hand, if a isn't divisible by p, this follows from

$$a^{1+k(p-1)} \equiv a(a^{p-1})^k \equiv a \mod$$

where $a^{p-1} \equiv 1 \mod p$ was used.

At last we are ready now to prove that RSA works with our choice of e and d, i.e. that the mappings $m \mapsto m^e \bmod n$ and $c \mapsto c^d \bmod n$ are inverse to each other. Since $(m^e)^d = m^{ed}$ and $(c^d)^e) = c^{ed}$ and ed $\equiv 1 \bmod \text{lcm}(p-1,q-1)$, all we have to prove is

Theorem: For all primes p and q, $p \neq q$, and all integers a the congruence

$$a^{1+k\,lcm(p-1,q-1)} \equiv a \bmod pq$$

holds for all $k \geq 0$.

Proof: Obviously it suffices to prove that

$$a^{1+k\,lcm(p-1,q-1)} \equiv a \bmod p \text{ and } a^{1+k\,lcm(p-1,q-1)} \equiv a \bmod q.$$

But this is an immediate consequence of the corollary to Fermat's Little Theorem, since lcm(p-1,q-1) is a multiple both of p-1 and q-1.

By the way, the inventors of RSA (and sadly enough, many authors of textbooks on cryptography thereafter) used the product (p-1)(q-1) instead of lcm(p-1,q-1). Of course, the decryption exponent d you get in this way will also do the trick, but it is usually a few bits larger than necessary. In fact, it can be proved that our d is the smallest possible one.

As we have seen, Fermat's Theorem is at the heart of RSA. But it also proves very useful when it comes to generating large primes as they are needed for RSA (and also some other cryptosystems).

The idea behind the use of Fermat's Theorem as primality test (or rather compositeness test) is simple: If you can find for any given n an integer a with 0<a<n such that

$$a^{n-1} \neq 1 \bmod n$$

then n must be composite. Unfortunately, this is not a strict primality test as the following computation with fixed a=2 shows:

```
SELECT(MOD(2^(n - 1), n) = 1 ∧ ¬ PRIME(n), n, 3, 10000, 2)
[341, 561, 645, 1105, 1387, 1729, 1905, 2047, 2465, 2701, 2821, 3277, 4033, 4369, 4371,
  4681, 5461, 6601, 7957, 8321, 8481, 8911]
```

22 composite numbers below 10000 pass the so-called Fermat test for the base a=2 without being prime! What is more, there are composite numbers n (called Carmichael numbers) which pass the Fermat test for all a in the range 0<a<n except for those with gcd(a,n) ≠ 1. For example, $561 = 3 \cdot 11 \cdot 17$ is such a number and even the smallest one.

```
SELECT(MOD(a^560, 561) ≠ 1 ∧ GCD(a, 561) = 1, a, 1, 560) = []
```

As was shown in 1992, there are even infinitely many of these numbers. The following theorem (without proof) gives a nice characterisation of Carmichael numbers.

Theorem: A composite number n is a Carmichael number if and only if it is squarefree and $p-1 \mid n-1$ holds for all prime divisors p of n.

We can use it to determine all Carmichael numbers up to 10000 using Derive:

```
Carmichael?(n) :=
  Prog
    If n = 1 ∨ PRIME(n)
       RETURN false
    If SOME(e_ > 1, e_, FACTORS(n)'↓2)
       RETURN false
    If SOME(MOD(n - 1, p_ - 1) > 0, p_, FACTORS(n)'↓1)
       RETURN false
    true

SELECT(Carmichael?(n), n, 1, 10000)

                          [561, 1105, 1729, 2465, 2821, 6601, 8911]
```

Carmichael numbers are also directly related to RSA in a very interesting way. If one (or even both) of the primes p and q is substituted by a Carmicheal number and if additionally gcd(p,q)=1 holds, then RSA will still work with e and d chosen in the usual way! On the other hand, getting p and q by factoring the modulus n is much easier in this case, since a Carmichael number has at least 3 prime factors!

We have seen above that a simple Fermat test won't exclude Carmichael number n, unless we are extremely lucky by finding a base a with 0<a<n and gcd(a,n)>1. Fortunately there is another very simple condition for primes which can be combined with the Fermat test to make it stronger. It is the fact that $x^2 \equiv 1 \mod p$ has only the solutions ±1, if p is a prime, due to

$$p \mid x^2 - 1 = (x-1)(x+1) \Rightarrow p \mid x-1 \text{ or } p \mid x+1 \Rightarrow x \equiv \pm 1 \mod p.$$

Let's assume in the following that n is odd (otherwise the primality testing of n would be very simple, wouldn't it?) and n = $s2^t+1$ for positive integers s,t, where s is odd. If a is any integer in the range 0 < a < n, then consider the sequence

$$a^s, a^{2s}, a^{4s}, \ldots, a^{(n-1)/2}$$

mod n, which you get by repeated squaring starting with $a^s \mod n$. Since you also get these numbers (but in inverse order) by taking repeatedly the square root of $a^{n-1} \mod n$, which is supposed to be $\equiv 1 \mod n$ by Fermat's Theorem, this sequence should either contain –1 or consist of 1's only, if n is prime. This sharpening of the Fermat test leads to the so-called Rabin-Miller test, which is by far the most widely used probabilistic primality test. In particular, it is used by Derive (and most other CAS) along with other primality tests. An implementation in Derive could look like this:

```
RABIN_MILLER(n, a, s_) :=
  Prog
    s_ := n - 1
    Loop
      s_ :/ 2
      If ODD?(s_) exit
    a := - ABS(MODS(a^s_, n))
    Loop
      If a = -1 exit
      s_ :* 2
      If s_ = n - 1
         RETURN false
      a := MODS(a^2, n)
```

And here again the examples from above that clearly show its efficiency:

```
SELECT(RABIN_MILLER(n, 2) ∧ ¬ PRIME(n), n, 3, 10000, 2) = [2047, 3277, 4033, 4681, 8321]
      SELECT(RABIN_MILLER(n, 2), n, [561, 1105, 1729, 2465, 2821, 6601, 8911]) = []
```

The numbers tested so far have been relatively small. How about really large, say with 1000 digits or more? Have a look at the following example:

```
                  1000
RABIN_MILLER(10      + 453, 2) = true
```

It took only 3.42s on my Pentium 450 PC, which is surprisingly fast. Since the modular exponentiation plays an important role in RSA itself (note that in the decryption $c \mapsto c^d \mod n$ the exponent d has about the same size as n, that is several hundreds digits!), we should have a close look at the underlying algorithm.

Just like the Euclidean algorithm it is also a very old nontrivial algorithm (according to D. Knuth "its chief rival for this honour"). In fact, it was already used by the ancient Egyptians for multiplication. (Note that you can view a product of two positive integers as additive power, e.g. $23 \cdot 5 = 23+23+23+23+23$.)

The basic idea of this algorithm, which is usually called "Square and Multiply" method, is best illustrated by an example. Suppose we have a monoid (H,·) and would like to compute say a^{100} for an $a \in H$. We could do this by first computing the sequence

$$a, a^2, a^4, a^8, a^{16}, a^{32}, a^{64}$$

in H (note that you have only to square 6 times starting with a!) and then multiplying appropriate powers of this sequence according to binary representation $(1100100)_2$ of 100, namely

$$a^{100} = a^4 a^{32} a^{64}$$

Hence, a very general implementation of this method, which can be adapted to your needs, could look like this:

```
[op(a, b) :=, id :=]

SAM(a, n, b_ := id) :=
  Loop
    If n = 0
      RETURN b_
    If ODD?(n)
      b_ := op(a, b_)
    If n > 0
      a := op(a, a)
    n := FLOOR(n, 2)
```

To use it you only have to specify the operation op(a,b) in H and the identiy element of H. For example, the “Egyptian multiplication”, which should rather be called “Double and Sum” method, can be carried out in this way:

```
[op(a, b) := a + b, id := 0]
```

$$SAM(37, 46) = 1702$$

$$37 \cdot 46 = 1702$$

If we want to perform a Fermat test for the huge number $10^{1000}+453$ above, we should specify the operation and the identity in H in the following way:

```
[op(a, b) := MOD(a·b, 10^1000 + 453), id := 1]
```

$$SAM(2, 10^{1000} + 452) = 1$$

This took only 5.02s on my PC. Of course, a “tailor-made” power_mod( ) without any detours, like

```
power_mod(a, n, m, b_ := 1) :=
  Loop
    If n = 0
      RETURN b_
    If ODD?(n)
      b_ := MOD(a·b_, m)
    If n > 0
      a := MOD(a·a, m)
    n := FLOOR(n, 2)
```

should be even faster. Now the computation

$$power\_mod(2, 10^{1000} + 452, 10^{1000} + 453) = 1$$

takes only 4.33s, which is remarkably close to 3.42s, the time Derive needed for

$$MOD\left(2^{10^{1000}+452}, 10^{1000}+453\right) = 1$$

on my PC. (Even though Derive is so smart to use the “Square and Multiply” method internally as well, if the parser recognises that the first argument of MOD( ) is a power!)

Emulating some attacks on RSA in Derive

D. Boneh, certainly an expert when it comes to RSA, says in [1] that “securely implementing RSA is a nontrivial task”. In this final chapter, I would like to illustrate a few of the possible dangers and pitfalls.

One of the most obvious attacks on RSA is trying to factor the modulus n, since it is clear that anyone who can find the prime factors p and q of n can compute the private key d in just the same way as we did. Embarrassingly enough, one of the main pillars on which the RSA construction rests is not a theorem, but a belief, namely the belief of most mathematicians in the difficulty of the factorisation problem. The situation is even worse: We cannot even prove, that the so-called RSA problem, namely to solve the basic congruence

$$x^e \equiv c \bmod n$$

is computationally equivalent to the factorisation problem, though there is some evidence for it. In other words, it is still possible that there might be an efficient algorithm to solve this congruence without the use of the prime factors of n.

It must be said though that there are variants of RSA, where the RSA problem is computationally equivalent to the factorisation problem. The most important one of this kind is the Rabin variant. Here the encryption exponent e is always 2, which is clearly forbidden in the classical RSA because of the requirement gcd(e,(p-1)(q-1)=1. Since this last equation is not fulfilled, the solution of the RSA problem is no longer unique mod n, but there are usually 4 solutions. This leads to the question how to solve congruencies

$$x^2 \equiv c \bmod pq$$

for large (and different) primes p and q. Let's consider the simpler case

$$x^2 \equiv c \bmod p$$

for any prime p first. We could do this by calling SQUARE_ROOT(c,p) from NUMBER.MTH, it is true, but if we assume that $p \equiv 3 \bmod 4$ (no problem, as about 50% of all primes should fulfil this condition!) , then assuming that $c \equiv a^2 \bmod p$ for some integer a, we see that $c^{(p+1)/4} \bmod p$ is a solution because of

$$((c^{(p+1)/4})^2 = c^{(p+1)/2} = c^{(p-1)/2}c \equiv a^{p-1}c \equiv c \bmod p$$

(Again we were referring to Fermat's Theorem!) Since p is a prime, all solutions are given by

$$\pm c^{(p+1)/4} \bmod p.$$

It should be clear by now that the four solutions of the original congruence mod pq are exactly the solutions of the four congruence systems

$$x \equiv \pm c^{(p+1)/4} \bmod p$$

$$x \equiv \pm c^{(q+1)/4} \bmod q$$

you get by the four independent choices of the signs. To get the unique solution mod pq now, we could by apply the routine CRT(a,m) from NUMBER.MTH (CRT, of course, refers to the Chinese Remainder Theorem), but I prefer to use Fermat's theorem – an all-purpose tool, isn't it? – to represent the four solutions by means of an explicit formula. In fact, they can be given in the form $\pm r, \pm s \bmod pq$ with

$$r = (c^{(p+1)/4} \bmod p)(q^{p-2} \bmod p)q + (c^{(q+1)/4} \bmod q)(p^{q-2} \bmod q)p$$

$$s = (c^{(p+1)/4} \bmod p)(q^{p-2} \bmod p)q - (c^{(q+1)/4} \bmod q)(p^{q-2} \bmod q)p$$

as you can easily check by reducing these expressions mod p and mod q, respectively.

And here again the implementation of the Rabin variant in Derive:

```
Rabin_init(k) :=
  Prog
    Loop
      p := NEXT_PRIME(RANDOM(2^CEILING(k/2)))
      If MOD(p, 4) = 3 exit
    q := FLOOR(2^(k - 1) + RANDOM(2^(k - 1)), p)
    Loop
      q := NEXT_PRIME(q)
      If p ≠ q ∧ MOD(q, 4) = 3 exit
    n := p·q
    a := MOD(q^(p - 2), p)
    b := MOD(p^(q - 2), q)
    ok

                              2
Rabin_encrypt(m) := MOD(m , n)

Rabin_decrypt(c, r_, s_) :=
  Prog
    r_ := MOD(c^((p + 1)/4), p)·a·q + MOD(c^((q + 1)/4), q)·b·p
    s_ := MOD(c^((p + 1)/4), p)·a·q - MOD(c^((q + 1)/4), q)·b·p
    MOD([r_, -r_, s_, -s_], n)

[p :=, q :=, n :=, a :=, b :=]
```

Just to save space, we use a very small number k of bits in the following example, namely k=64. (You should however set k to a realistic size such as k=1024 and check that it's only a matter of seconds in this case as well!)

```
Rabin_init(64) = ok

p = 1060682039

q = 8878489951

n = 9417254824467690089

(m := text_to_number(GO AHEAD, plain)) = 7150001080501O4

(c := Rabin_encrypt(m)) = 7916092294297412252

v := Rabin_decrypt(c)

[7059244053210802850, 2358010771256887239, 7150001080501O4, 9416539824359639985]

VECTOR([v_, number_to_text(v_, plain)], v_, v)

[ 7059244053210802850   GE▫lE'J↵↖r ]
[ 2358010771256887239   Bc↵JnLx→ƒg ]
[   7150001080501O4       GO AHEAD  ]
[ 9416539824359639985   Ii⌊g²k{o∎· ]
```

As you can see the plain text is among the 4 solutions. (Usually the other three solutions don't yield a meaningful text as in our example, but one could also add some redundancy to recognise the correct solution for sure.)

Now, why is it that for the Rabin variant the RSA problem and the factorisation problem are computationally equivalent? Well, have a look at our example again. Take any two of the four numeric solutions such that their sum is not n (that is they shouldn't belong both to the same pair $\pm r$ or $\pm s$ mod n) and compute the gcd of their sum with n, e.g.

```
GCD(7059244053210802850 + 715000108050104, n) = 8878489951
```

The result is one of the two prime factors of n (here q)! I leave it to the reader to check that this is also true in the general case due to the special form of r and s.

Let's go back to the factorisation problem of n. It's widely believed to be a hard problem, if n is sufficiently large, say with several hundreds decimal digits, and the factoring attack should usually fail. On the other hand, it can be surprisingly easy to factor even a large n if the prime factors p and q had been chosen carelessly.

For example, if p – 1 or q – 1 contain only relatively small prime factors, there is the so-called (p – 1)-method by Pollard which takes advantage of this fact to factor n. It is again based – you won't be surprised by now – on Fermat's theorem. Suppose that r is a positive integer, which is a multiple of p – 1, say r = k(p-1) for some k, and that a is any integer coprime to p. Then according to Fermat's Theorem the congruence

$$a^r = (a^{p-1})^k \equiv 1 \bmod p$$

holds and hence $p \mid \gcd(a^r - 1, n)$. Hence, unless you are extremely unlucky and $\gcd(a^r - 1, n) = n$, you have got a nontrivial factor of n by computing this gcd!

The problem is to find such an r that is a multiple of p – 1 for one of the prime factor p of n, since you don't know p. But provided that p – 1 has no big prime factors (in which case it is sometimes called "smooth"), you might choose for r the lcm of all numbers up to a certain bound B. Let's consider as a small example the Mersenne number $n = 2^{67} - 1$ and B = 3000.

```
n := 2^67 - 1
r := LCM([2, ..., 3000])
GCD(MOD(3^r, n) - 1, n) = 193707721
FACTOR(193707721 - 1) = 2^3·3^3·5·67·2677
```

The last line shows why we have been successful: The prime factor 193 707 721 of n, which we found, has no prime factors larger than 2677!

There is no need to give here a real implementation of the (p – 1)-method. For one thing I have done this on other occasions (cf. [5]), and for another it is already available in DfW5 as part of the built-in

FACTOR( ). It must be said though that for a big prime p the chances of p – 1 to be smooth are very slim.

There is also a (p + 1)-counterpart of the Pollard's (p-1)-method which uses Lucas sequences and is very successful, whenever p+1 or q+1 are smooth. You can find a program for it in [5] and it is also described there in more detail. Again it has become a part of the built-in FACTOR( ) by now. For example, try to factor the 89-digit number

17770979169933523850386416233269078047297068804035044337550222804109610224994980199227924 7

by applying FACTOR( ) to see it at work!

As for the encryption, it is advisable to take a small e in order to speed it up. The smallest possible value for e is 3, of course, and it is perfectly okay as long as you take some precautions. For example, if the decimal number m, which you get after source coding, has less than one third of the digits of the modulus n, then c = m^e mod n would be the same number as m^e, i.e. you get m back by simply computing the cubic root of c! Hence, m should always be padded to have about the same length as n.

Another danger is involved, if you send the same message m to at least 3 recipients. Suppose that an intruder could get hold of three of the ciphertexts c_1, c_2, c_3 with the corresponding moduli n_1, n_2, n_3. Since

$$c_i \equiv m^3 \bmod n_i \text{ , i = 1,2,3}$$

we see that x = m^3 is the unique solution in the range $0 \le x < n_1 n_2 n_3$ of the system of congruences

$$x \equiv c_i \bmod n_i \text{ , i = 1,2,3.}$$

But this solution can be computed by applying the Chinese Remainder Theorem, that is in Derive by calling CRT($[c_1, c_2, c_3], [n_1, n_2, n_3]$). The antidote is again padding of m, this time with random digits, which should be generated for each encryption independently.

As for d, it is clear that it shouldn't be extremely small otherwise one could find it by trial and error. In fact, if the decryption exponent d has loosely speaking less than one fourth of the digits of n (cf. [1] for the details), then you can find it (or another suitable d) among the denominators of the convergents of e/n. (Unfortunately I have to assume here that you are familiar with the basics of simple chain fractions and their rational approximations called convergents.)

Look at the following emulation in Derive. We first called our routine RSA_init(k) with k=70 (again k is so small only to save space!) and then exchanged e and d in order to fulfil our assumptions. As you can see, our new d is actually among the denominators of the convergents of e/n as predicted.

```
RSA_init(70) = ok
e = 65537
d = 364584566323984101233
[e := 364584566323984101233, d := 65537]
CONVERGENTS(e/n, 10)
```

$$\left[0,\ \frac{1}{3},\ \frac{9}{28},\ \frac{10}{31},\ \frac{699}{2167},\ \frac{709}{2198},\ \frac{1408}{4365},\ \frac{4933}{15293},\ \frac{16207}{50244},\ \frac{21140}{65537},\ \frac{248747}{771151}\right]$$

As we have seen d should be large (usually the size of n) as opposed to e. As consequence the decryption (and the generation of signatures, where d is used as well) is considerably slower than encryption, even though the "Square and Multiply" method used for modular exponentiation is incredibly fast, as we have seen.

Therefore, mathematicians were looking for means to speed up decryption (and the generation of signatures) even further. It turned out that one of the most efficient tools on that score is again the Chinese Remainder Theorem.

Suppose that Bob wants to sign the number s with his private key d by computing $s^d \bmod n$. Obviously, he could also compute $s^d \bmod p$ and $s^d \bmod q$ first and then combine these results using the Chinese Remainder Theorem to get $s^d \bmod n$. As for the computation of $s^d \bmod p$ and $s^d \bmod q$, again Fermat's Theorem comes into play: Since $s^{p-1} \equiv 1 \bmod p$ and $s^{q-1} \equiv 1 \bmod q$, we could compute the much simpler, but equivalent expressions $s^{d_p} \bmod p$ and $s^{d_q} \bmod q$ using the precomputed values $d_p := d \bmod (p-1)$ and $d_q := d \bmod (q-1)$.

Now let's modify our RSA_init(k) accordingly:

```
RSA_CRT_init(k) :=
  Prog
    RSA_init(k)
    dp := MOD(d, p - 1)
    dq := MOD(d, q - 1)
    a := MOD(q^(p - 2), p)
    b := MOD(p^(q - 2), q)
    ok

[p :=, q :=, n :=, e :=, d :=, dp :=, dq :=, a :=, b :=]
```

And here is an example, where n has 1024 bits, and s is any number < n that is to be encrypted with the private key d to get the digital signature.

```
RSA_CRT_init(1024) = ok
a :=
  114704327712847055473781842945211954859687856797342502775616627267126771650448862283287~
  678291895259342383438977813221983599808948755332181393069849220063638888761284183485835~
  348565220506384585825235259424921912766347267717108722437771444976721682418164778340 72~
  672577162013600889819092893946710551955595040293723 9
```

```
      d
MOD(s , n)

830807355527247045925550087534128431633458306232786636712243756970544391555034328406435 7~
  065498631594022964421008405132799448342587812418677756953146047131889922387188985869 19~
  936453018976352069023925666874761978752803350412843712926123831676344950091273058218 10~
  705255233151570097193403460421990730741376794558

            dp                    dq
MOD(a·MOD(s  , p)·q + b·MOD(s  , q)·p, n)

830807355527247045925550087534128431633458306232786636712243756970544391555034328406435 7~
  065498631594022964421008405132799448342587812418677756953146047131889922387188985869 19~
  936453018976352069023925666874761978752803350412843712926123831676344950091273058218 10~
  705255233151570097193403460421990730741376794558
```

As expected these results coincide, but the second computation took only 0.04s on my PC (vs. 0.14s for the first), i.e. it is faster by a factor 3-4.

When using the Chinese Remainder Theorem in this way, you should be aware of a certain risk though. If exactly one of the exponentiations mod p or mod q failed for some reason (as an intruder you could try to provoke such a failure e.g. by exposing a computer chip to x-rays!) , then the recipient, who knows the correct signature could easily compute the secret prime factors of n!

Let's assume for example that the exponentiation mod p failed, which we emulate by applying the RANDOM-function to the regular outcome.

```
                  dp                    dq
t := MOD(a·RANDOM(MOD(s  , p))·q + b·MOD(s  , q)·p, n)

289642156361032729847808794778418569979802352139732544088930172603649625626805963021317 0~
  697414396694757015279547565758387012829538887572737567296953919035627521700312465290 26~
  004282605815856508347824379529886532776608979360562611428533811862257014640930290888 774~
  364033966488622768031283741532962739378397624725
```

But then q = gcd(s – mod(t^e, n), n) as the following computation shows:

```
GCD(s - MOD(t^e, n), n) - q = 0
```

The remedy is clear: The sender should check himself, whether s = mod(t^e, n) really holds, before giving t away. Since e is small, this check is very fast.

A lot more could be said about RSA, but I hope I have already succeeded in showing the very special flavour of this cryptosystem, which involves so many fundamental and beautiful mathematical ideas. In particular, I hope you enjoyed the Derive programs as much as I did when I wrote them.

References

[1] D.Boneh, Twenty Years of Attacks on the RSA Cryptosystem, Notices of the AMS, Vol. 46, Nr.2 (Feb. 1999) ,203-213.

[2] D.E.Knuth, The Art of Computer Programming, Vol 2: Seminumerical Algorithms, 3rd ed., Addison-Wesley, Reading, Mass.,1998

[3] R.L.Rivest, A.Shamir and L.Adleman, A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, Comm.ACM 21 (178), 120-126.

[4] J.Wiesenbauer, Number Theory with DERIVE – Some Suggestions for Classroom Teaching, appeared in: DERIVE in Education - opportunities and strategies (ed. By H.Heugl and B.Kutzler), Chartwell-Bratt (1994), 51-61.

[5] J.Wiesenbauer, Factoring and RSA Codes using DERIVE, Proceedings of the Derive Conference at Ghettysburg in 1998 (appeared on CD-Rom)

[6] J.Wiesenbauer, Titbits from Algebra and Number Theory (17), to appear in the Derive-Newsletter #38