Some Security Issues of Public Key Cryptosystems using DERIVE

Johann Wiesenbauer, Vienna University of Technology

Abstract

The paper deals with the proper generation of primes for RSA and other cryptographic purposes. It turns out that there are a number of pitfalls which should be avoided. In particular, the randomness and the unpredictibility of the generated primes is definitely a key issue when it comes to the proper implementation of a public key cryptosystem based on them. Furthermore, primes with certain special properties pose a security problem. It is shown how "safe" primes can be generated using DERIVE.

1. Introduction

I will assume that the reader is already familiar with the principles of public key cryptography (cf. e.g. [1],[2]). Even though, I will put together some basic facts in the following in order to introduce some notations. Basically, public-key cryptography involves the use of two keys:

- a **public key**, which is widely available as its name says, and can be used to encrypt messages and verify signatures
- a **private key**, known only to the recipient, that is used to decrypt messages and sign signatures

The generation of these keys makes use of a so-called **trapdoor function**, i.e. a one-way function $f:X\rightarrow Y$ with the additional property that given some **extra information** it becomes feasible to find for any $y\in \mathrm{Im}(f)$ an $x\in X$ such that $f(x)=y$.

The public key gives all the information to compute the values f(x) for any $x\in X$, whereas the private key gives the extra information for computing the inverse images. Given these pieces of information all computations should be possible in polynomial time.

To make things simpler I will only refer to the most widely used public key cryptosystem RSA in the following, although some of the subsequent considerations apply also to others, where primes are used (e.g. the ElGamal cryptosystem). Here, the trapdoor function has the form

$$f: \{0,1,..n-1\} \rightarrow \{0,1,..,n-1\}$$

$$x \mapsto x^e \bmod n$$

where n is the product of two „big" primes p and q of roughly the same size. Furthermore, e is an integer in the interval $(1,\nu)$ with $\nu=\mathrm{lcm}(p-1,q-1)$ such that $\gcd(e,\nu)=1$. (Note that in the "classical" RSA as introduced by its inventors, $\nu=(p-1)*(q-1)$ is used instead, which works as well, but is not optimal as pointed out in [2].)

The latter condition ensures that the mapping

$$f^*: \{0,1,..n-1\} \rightarrow \{0,1,..,n-1\}$$

$$x \mapsto x^d \bmod n$$

where $d\in(1,\nu)$ and $de\equiv 1 \bmod \nu$, is inverse to f.

Here f is used for encryption and f* for decryption. Hence, it should be clear that the public key of RSA consists of the pair (n,e), whereas d is the private key. (The primes p and q are no longer needed and should be destroyed!) The computation of d can be done in $O(n^3)$ time using the **extended Euclidean algorithm** by anyone who knows the factorization n=p*q. e can be small (even e=3, if „salting" of messages is used), whereas d should be large to be safe. (If d has up to 1/4 as many bits as n then the so-called **Wiener attack** will be successful, which uses convergents of the continued fraction for e/n, cf. [2])

Basically, any attacker faces the so-called **RSA-problem**: Given n and e as above, for any integer $c \in [0,n)$ he must be able to solve the congruence

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

If he can factor n though, the RSA-problem can be easily solved as well (via the computation of d). Hence, the RSA-problem is at most as hard as the so-called **integer factorization problem**. It is widely believed, though not proven, that these two problems are actually computationally equivalent.

The checking, whether a given k-bit prime (where k=512 or greater for RSA) is prime or not is very fast and can be done in $O(k^3)$ time using **probabilistic methods** (usually by carrying out a fixed number of **Rabin-Miller tests** as this is also done by Derive internally, cf. [3])

A far bigger problem is to provide truly random primes of the desired size, e.g. with about 512 bits, which is a very common size nowadays. In particular, a generation of such a prime p like

p:= next_prime(random(2^512))

should not be used for "serious applications" as is shown in the following.

2. What is wrong with p:= next_prime(random(2^512)) ?

Basically, in order to get a random number with at most 512 bits using random(2^512) Derive performs the iteration

$s := 2654435721 \cdot s + 1 \bmod 2^{32}$

16 times starting with a 32-bit random seed s and concatenates all resulting 16 values of s (in binary representation). In the following Derive demonstration rand(s) emulates the generation of a random number with 32k bits (where k=16 by default) using the 32-bit seed s. (As for the details see also the online help for random(n).)

```
rand(s, k := 16) :=
  Prog
    s := MOD(2654435721·s + 1, 2^32)
    Loop
      k :- 1
      If k = 0
         RETURN s
      s := 2^32·s + MOD(2654435721·s + 1, 2^32)

rand(123456789) =
469131636682700919707560162755174507870917390811084621728479526702581757055044653776136102611630766118708250817469331956555615895347514304632385513376189 3

                                    RANDOM(-123456789) = 123456789
        512
RANDOM(2   )=
469131636682700919707560162755174507870917390811084621728479526702581757055044653776136102611630766118708250817469331956555615895347514304632385513376189 3
```

Hence, this naïve approach

p := NEXT_PRIME(2 ^512)

will produce only

$$\text{APPROX}(2^{32}) = 4.294967296 \cdot 10^9$$

different primes out of a total of about

$$\text{APPROX}\left(\int_2^{2^{512}} \frac{1}{\text{LN}(t)}\, dt\right) = 3.788726978 \cdot 10^{151}$$

512-bit primes. Furthermore, all these 4.3 billion primes can be easily generated as shown above and therefore when using two of those primes for RSA a simple brute force factoring attack is highly likely to be successful within a few hours!!!

If you need 512-bit random numbers for genuine cryptographic applications then I recommend to concatenate the 4 least significant bits of the time between two keystrokes until you have the requested number of bits. In more dertail, this can be accomplished by first inputting the following two lines

s := 0

FLOOR(LOG(RHS(s := 16·s + MOD(RANDOM(0), 16)), 2) + 1)

(by the way, I am indebted here to Albert Rich for pointing out a slight improvement of my original approach which you can still see in the attached file on the CD) and by repeatedly simplifying the second line by clicking with the mouse on the "=" icon to left of the input line until s has the desired bit length. (Note that the current bit length of s is shown on the screen while clicking!) In order to get "good" randomness, don't make more than say 3 mouse clicks in a second, though, and avoid any "rhythm". To get a 512-bit random number you have to click 128 times, it is true, but even though this should not take longer than say 1 minute! After all, this has to be done only once, namely when setting up of your public key cryptosystem, so it surely pays off in terms of security!

This procedure has two advantages. First, it will yield all 2^{512} numbers of bit length 512 and not only a tiny subset of size 2^{32}. Second, the bits are much unpredictable and have good random properties. To test the latter statement I have implemented some popular statistical test such as e.g. the poker test and the runs test. Here is a short description of the poker test (see [1], also for the other tests). For all these test the decimal number s must be converted into a binary vector t of length 512 before, which could be done by the assignment

t := VECTOR((SIGN((2^k AND s) - 1/2) + 1)/2, k, 511, 0, -1)

Let m be any positive integer such that for k:=[n/m] the condition $k \geq 5 * 2^m$ holds. Now, divide the sequence s in k non-overlapping parts of length m. The poker test checks then, whether they occur with approximately the same frequency. If n_i denotes the number of binary strings of length m, whose conversions into a decimal number are exactly i for some $i \in \{0,1,..,2^k-1\}$ then

$$X = 2^m (n_0^2 + \ldots + n_{2^{\wedge}k-1}^2)/k - k$$

approximately follows a χ^2 distribution with 2^m-1 degrees of freedom. The program PokerTest(u,m) on the CD computes the second P-value for a given binary vector u and a given block length m. For good randomness it should be as high as possible. If it is however below a given significance level α of say $\alpha=5\%$ (or even $\alpha=1\%$), then the binary vector has surely failed the randomness test! In my examples this was far from being the case though!

3. Other pitfalls as to the form of the primes

The primes used in RSA should be so-called „strong primes", which can be generated by Gordon's algorithm (cf. [1]). A prime number p is called a strong prime, if

- p-1 has a large prime factor r
- p+1 has a large prime factor s
- r-1 has a large prime factor t

Here, the first two conditions will give protection against two factoring methods, viz. Pollard's p-1 method and Williams' p+1 method, respectively, which are highly efficient if those conditions are not fulfilled, whereas the third condition is the „antidote" as to the so-called „cycling attack" of RSA. (For this attack one has only apply the encryption a relatively small number of times to the cipher text to get the plain text,)

```
gordon(s, t, i_ := 1, j_ := 1, p_, r_) :=
  Prog
    Loop
      r_ := 2·i_·t + 1
      If PRIME?(r_) exit
      i_ :+ 1
    p_ := 2·MOD(s^(r_ - 2), r_)·s - 1
    Loop
      p_ :+ 2·j_·r_·s
      If PRIME?(p_)
         RETURN p_
      j_ :+ 1
```

```
                          246
   s := NEXT_PRIME(RANDOM(2   ))

   s := 702109008473154815562991914938871351531056162203416270300220472107353595 91

                          246
   t := NEXT_PRIME(RANDOM(2   ))

   t := 781225745890394120210276021260212683184042897625603523642975585448356314 49

   p := gordon(s, t)

   p :=
1037770967767059056916416582406246532666687475579663248988671538837743003554757139006338
5206209323920876687529562646720469466719917506240784088392916142404 1
```

The generated prime p has slightly more bits than s*t. Furthermore p+1 is divisible by s and p-1 is divisible by a big prime of roughly the size of t.

```
FLOOR(LOG(s·t, 2) + 1) = 491

FLOOR(LOG(p, 2) + 1) = 515

MOD(p + 1, s) = 0

ITERATE(IF(MOD(p - 1, t_) = 0, t_, t_ + 2·t), t_, 2·t + 1)=

2734290110616379420735966074410744391144150141689612332750414549069247100715 1
```

Now let's create advertedly "unsafe" primes just to see what happens. The following routine will create a prime q with about n bits, such that q-e with e in {1,-1} has only prime factors with at most s bits.

```
unsafe(n, s, e := -1, r_ := 1, p_) :=
  Prog
    Loop
      r_ :* NEXT_PRIME(RANDOM(2^s))
      If FLOOR(LOG(r_, 2) + s) ≥ n exit
    p_ := 2·r_ - e
    Loop
      If PRIME?(p_)
         RETURN p_
      p_ :+ 2·r_

q := unsafe(512, 16)

FACTOR(q - 1)

q :=

7412086308680915919188982332449416283785874137603168047732406555392287036631887013427084
3044385668015166459875112899839135772867533038393143928496658814 41
n := p·q

DIM(n) = 309

FACTOR(n) =
```

```
741208630868091591918898233244941628378587413760316804773240655539228703663188701342708430443856680151664598751128998391357728675330383931439284966588144
1·10377709677670590569164165824062465326666874755796632489886715388377430035547571390063385206209323920876687529562646720469466719917506240784088392916142404
1
```

Using the "unsafe" prime q the 309-digit number n=p*q was factored in only 46.6s on my PC! Note that both Pollard's p-1 method and Williams' p+1 method are part of the factoring strategy of Derive's factoring algorithm, which you can also see by choosing the "display step" mode from the tool bar

As a rule of thumb, the factorization problem is hardest, if the two primes p and q are roughly of the same size. Strangely enough, the case where p and q are very close to each other must be avoided though by all means, as Fermat's factorization method is highly efficient otherwise. It makes use of the fact that there is a bijection between any two representations

$$n=a*b \text{ with } a\geq b>0 \text{ and } n=u^2-v^2 \text{ with } u>v\geq 0$$

using u=(a+b)/2, v=(a-b)/2 and a=u+v, b=u-v, respectively.

The details of the algorithm can be seen from the program below. It is given here only for didactic reasons, as Fermat's factorization has also be added to Derive's bundle of factoring strategies since version 6.01. By the way, the 243-digit number n below has actually been used by an Austrian criminal who was was in the bad habit of sending letter bombs to people he didn't like before he was finally caught.

```
fermat(n, u_, v_) :=
  Prog
    u_ := CEILING(√n)
    v_ := u_^2 - n
    Loop
      If INTEGER?(√v_)
         RETURN [u_ - √v_, u_ + √v_]
      v_ :+ 2·u_ + 1
      u_ :+ 1
      If u > (n + 9)/6
         RETURN [1, n]
```

```
n :=
630548215070129547156718332495889632234434145411971275888376987603260225252787926135276738944105689100036295535868141424386536403649578707699128189491432138631900590774729214990015369102760964884776344849717811484309528915040117952098061886881

fermat(n) =
[251107191269013549761909333958671246802408057112768448862509598241562051889494061847352957883875611351675294302430759487 99,
251107191269013549761909333958671246802408057112768448862509598241562051889494061847352957883875611351675294351184297803 19]
```

Another threat comes from the fact that the encryption mapping of RSA, viz. $x\mapsto x^e \mod pq$, has got a number of fixpoints, i.e. points which will be mapped onto themselves with any encryption! It is easy to see that the number of fixpoints is at least 9. More precisely, one can easily compute the exact number of fixpoints using Derive by inputting the formula

$$(1+\gcd(e-1,p-1))(1+\gcd(e-1,q-1))$$

as it is.

Note that the nontrivial fixpoints $x\neq -1,0,1$ must be kept secret by all means, as gcd(x,n) is one of the primes p or q. Hence, in order to keep the number of fixpoints at its absolute minimum 9, the primes p and q are sometimes chosen such that (p-1)/2 and (q-1)/2 are also primes. In practice, this is no real security risk though, as the probability of an x chosen at random to be a fixpoint is virtually zero.

References

[1] A. Menezes, P. van Oorschot and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, Boca Raton, FL, 1996 (cf. http://www.cacr.math.uwaterloo.ca/hac/)

[2] J. Wiesenbauer, *Public Key Kryptosysteme in Theorie und Programmierung*, Didaktikhefte der ÖMG, Heft 30, 1999(144-159)

[3] J. Wiesenbauer, *Primality Testing and Factoring Large Numbers with DERIVE*, Proceedings of VIST-ME-2002 (ed. by J.Böhm), bk teachware Nr. SR-31