

In recent years, much CPU time on sophisticated computers has been devoted to the task of factoring large composite numbers. One reason for this interest is the RSA public-key encryption system developed in 1977 by Rivest, Shamir, and Adleman. This system is only as secure as the infactorability of the modulus used. Therefore, to form a secure RSA modulus, we need to understand how various factorization techniques, such as the Pollard P-1, work and then create a modulus which will cause these techniques to fail.

RSA Public Key Encryption - Background Info:

Before we start creating moduli, it's probably a good idea to show how the modulus is used. At the heart of the RSA technique is Euler's formula. The quantity $\Phi(n)$ is related to the modulus n by the equation:

$$a^{\Phi(n)} \equiv 1 \pmod{n}$$

For *any* integer base a , such that $\gcd(a,n)=1$. For example, if $n=10$ then $\Phi(10) = 4$ and we can compute:

$$3^4 \equiv 81 \pmod{10} \equiv 1$$

The quantity $\Phi(n)$ has another feature which makes computing it trivial for small moduli; it is related to the modulus n in that it is a count of the number of integers less than n which are relatively prime to n ¹. For example, the modulus 10 has four integers less than it which are relatively prime, namely 1,3,7, and 9; therefore $\Phi(10) = 4$. Notice that if the modulus is a prime number, then all of the integers less than it will be relatively prime. i.e. if p is a prime integer then $\Phi(p) = p-1$. Furthermore, if the modulus is the product of two primes, say $n=p*q$ where both p and q are prime, then $\Phi(n)=(p-1)*(q-1)$. For example, if $p=3$ and $q=7$ then $\Phi(21)=(3-1)*(7-1)=12$. Now we're ready for the RSA engine.

¹The integer a is relatively prime to the integer b if $\gcd(a,b)=1$.

RSA Public Key Encryption - The Engine:

Given a modulus, say n , which is the product of two primes, say p and q , we know that $\Phi(n)=(p-1)*(q-1)$. The only other thing we need to setup the RSA engine are two 'keys', the encryption key and the decryption key, call them e and d . These keys need to have the property that:

$$d(e/1(\text{mod}M(n)))$$

Or, in less succinct notation, the keys need to be related by:

$$d(e' 1\%t(M(n)))$$

For some integer t . Once we have n, d , and e , we make the modulus n and *ONE* of the keys, say d , known to the public. It is very important that no one else knows both keys as will be seen shortly. Likewise, it is very important that no one *EVER* knows the values of p, q , or $\Phi(n)$ because this knowledge will allow someone to crack your code. Now, given a numerical message², say H , we create an encrypted message, say E , as follows:

$$E' H^e(\text{mod}n)$$

The person receiving this information can now decrypt it with the decryption key as follows:

$$E^d(\text{mod}n)' (H^e)^d(\text{mod}n)' H^{de}(\text{mod}n)$$

But, the keys were chosen with the property that:

$$H^{de}(\text{mod}n)' H^{M'}(H^1(\text{mod}n)' 1^t(H^1(\text{mod}n)' H$$

So, the original message is returned. If someone uses the wrong key, they will very likely get garbage. Notice that since the modulus and one key is known to the public, the only remaining piece of information to crack the code is the other key. But, finding the other key relies on the ability to find the factorization of the modulus. Thus, in the interest of generating secure schemes, we are not free to pick just any two primes p and q because the resulting modulus may be very easy to factor. Consequently, we must examine how factorization is achieved and then pick 'safe' primes for our code.

²Given a textual message, we can always create a numerical message by replacing each letter of the alphabet and a few symbols such as spaces, with integers.

Factorization : Pollard P-1

The simplest form of factorization is trial division, or perhaps the Sieve of Erastosthenes, both of which rely on the ability to 'try' dividing the modulus by all numbers less than the square root until a divisor is found. To overcome these tests, we can simply choose a modulus which is large enough to make this brute force task impossible within, say a lifetime or two³. However, there are factorization techniques such as the Pollard P-1 which will produce factors relatively quickly for large moduli - It is these tests which require us to choose the modulus n for RSA very carefully. In particular, the Pollard P-1 is very good at factoring numbers, say $n=p*q$, when either p or q have the property that $p-1$ or $q-1$ is comprised of small factors. For example if $n=679$ ($679=97*7$) then we can see that $97-1=2*2*2*2*2*3$. And thus pollard should pick out the 97 very easily because $97-1$ is comprised of small factors.

To conquer this factorization technique, we need to pick both p and q such that $p-1$ and $q-1$ have very large factors. Ideally, $p-1$ would be the product of several large primes, but as experimentation has shown⁴, this is a fairly rare occurrence, too rare in fact to be a practical guideline for generating p and q . Instead, start with a large prime p_2 and then multiply it by even numbers, starting at two, until 2^i*p_2+1 is prime (for some integer i). Let this new prime be p_1 and repeat the process once more to generate p . Thus, the prime p has the property that $p-1$ is the product of one very large prime and a few small primes. How well does this technique work against Pollard P-1 factorization? Let's get experimental!

Some Code:

To test our moduli generating scheme, we need some routines to do computer crunching. The following programs are written in 'Maple' and are preceded with an explanation of the inputs and outputs. The Pollard P-1 code was adapted from an algorithm in the book "Factorization and Primality Testing" by David Bressoud.

³If n has 100 digits, and it's smallest prime factor p is 40 digits, (with the density of primes roughly equal to $p/\log(p)$) then trial division of ONLY primes at a trillion divisions per second would take roughly $3.44E18$ years, or 172 million estimated lifespans of the universe.

⁴My attempt to generate such primes (such that $p-1$ had only large factors) resulting in consuming my computers available memory and locking the keyboard indefinitely.

Pollard P-1 Factorization:

The following routine is the Pollard P-1 factorization code. The program takes a single integer as input - the number to be factored, and returns the first factor it finds:

```
pmo:=proc(n:integer) local max,g,i;
max:=100000;
c:=2;
i:=1;
g:=1;
if not isprime(n) then
  while g=1 and c<50 do
    m:=c;
    i:=1;
    while i<max and g=1 do
      i:=i+1;
      m:=m &^ i mod n;
      if i mod 10 = 0 then g:=igcd(m-1,n); fi;
    od;
    if not (g>1 and g<n) then g:=1; fi;
    c:=c+1;
    if (g=1) then printf(`Now trying base %d\n`,c); fi;
  od;
  if (g>1 and g<n) then printf(`WOW I found a factor! \n`);
  g;
else printf(`Grrrr, I've failed to find a factor for the bases up
to : %d\n`,c);
fi;
else lprint(`You clutz! you've entered a prime number!!`);
fi;
end;
```

Pollard Generator:

The following code takes two inputs, integers a and b , and uses them to create a composite which the Pollard P-1 factorization code should be able to factor effortlessly. It does this by randomly selecting 'a' many primes in the interval $1..b$, multiplying them together and then successively multiplying the result by 2 and adding one until a factor is found. The resulting modulus is returned as n and the factors are p and q . This code will be used to generate composites by which we can compare the factorability of the RSA moduli.

```
mpollard:=proc(i:integer,k:integer) local j,k;
j:=0:p:=1;
random:=rand(3..k);
while (j<i) do p:=p*prevprime(random());
j:=j+1;
od;
j:=0;
while (not isprime(p+1)) and j<100 do
p:=p*2;
j:=j+1;
od;
p:=p+1;
q:=prevprime(floor(p^.5));
n:=p*q;
if isprime(p) then
lprint(`Results are p, q, and n=p*q where q is on the order
of p^.5`);
lprint(`Success! the factors of p-1 are:`);
else
lprint(`Failed to find a prime (p+1) within 100 iterations
of p=2*p...`);
fi;
if isprime(p) then ifactor(p-1); fi;
end;
```

RSA Moduli Generator:

This routine takes two integers as inputs which are to be considered as 'seeds'. The first prime larger than the seed (if the seed is not already prime) is used in the method described previously, to generate primes. In general, the resulting primes will be roughly 100~1000 times larger than the seed. Once two primes, p and q , have been generated, the modulus n and the $\Phi(n)$ are computed. The routine then makes use of a random number generator to randomly select an encryption key in the range $1..p$. The decryption key is then computed by finding integer solutions to the equation $d*e-t*\Phi(n)=1$, and using the first positive solution found for d (Maple returns not only the trivial solution, but the complete solution, i.e $d=r+i*w$ where r and w are two fixed integer, but i can be any integer and d will still be a solution.) The results are returned as the modulus, n , the encryption key, $ekey$, and the decryption key, $dkey$. For factorization testing, the routine also returns p and q , although normally this information is unnecessary (not to mention

dangerous!)

```
MakeKey := proc(a:integer, b:integer) local
p2,q2,p1,q1,i,w,u,v,phin;
if not isprime(a) then p2:=nextprime(a) else p2:=a fi;
if not isprime(b) then q2:=nextprime(b) else q2:=b fi;
i:=1;
while not isprime(2*i*p2+1) do i:=i+1 od;
p1:=2*i*p2+1;
i:=1;
while not isprime(2*i*q2+1) do i:=i+1 od;
q1:=2*i*q2+1;
i:=1;
while not isprime(2*i*p1+1) do i:=i+1 od;
p:=2*i*p1+1;
i:=1;
while not isprime(2*i*q1+1) do i:=i+1 od;
q:=2*i*q1+1;
n:=p*q;
phin:=(p-1)*(q-1);
randome:=rand(1..(p-1)/2);
ekey:=2*randome()+1;
while not igcd(ekey,phin) = 1 do ekey:=randome() od;
w:='w';
dkey:='dkey';
s:=isolve(dkey*ekey-w*phin=1);
if op(1,op(1,s))=dkey then s:=op(2,op(1,s)) else s:=op(2,op(2,s))
fi;
if whattype(op(1,s))=integer then u:=op(1,s); v:=op(1,op(2,s));
else u:=op(2,s); v:=op(1,op(1,s)) fi;
i:=1;
while u+v*i < 0 do i:=i+1 od;
dkey:=u+v*i;
lprint(`Computations Complete.... Checking Results....`);
i:=dkey*ekey mod phin;
printf(`dkey*ekey mod phi(n) = %d \n`,i);
lprint(`Results: n (modulus), ekey (encrypt key) and dkey
(decrypt key)`);
end;
```

Maple Session - Comparing Moduli and Their Factorability:

The following data was generated in Maple using the above procedures. Four techniques were used to generate 10 composite numbers of 10 digits each: The four techniques were the MakeKey procedure, the MPollard procedure, the product of two 'safepimes' (a function built into maple : a safeprime, say q , has the property that $(q-1)/2$ is also prime), and the product of

two random primes chosen with as little bias as possible.

| | MakeKey | MPollard | Safeprimes | Random |
|-------------|------------|------------|------------|------------|
| # | 4586802581 | 1060820779 | 1244300401 | 1923270827 |
| Pollard | 46 sec | 0 sec | 13 sec | 0 sec |
| Ifactor | 1 sec | 0 sec | 1 sec | 0 sec |
| # | 5194201487 | 7588650571 | 5246099701 | 2437311593 |
| Pollard P-1 | 46 sec | 0 sec | 14 sec | 2 sec |
| Ifactor | 1 sec | 0 sec | 0 sec | 0 sec |
| # | 8007054647 | 1909310737 | 8910001201 | 4128667579 |
| Pollard P-1 | 4 sec | 0 sec | 108 sec | 0 sec |
| Ifactor | 0 sec | 0 sec | 1 sec | 0 sec |
| # | 1496001289 | 7886404987 | 1458977869 | 6217502983 |
| Pollard P-1 | 10 sec | 0 sec | 28 sec | 0 sec |
| Ifactor | 1 sec | 0 sec | 0 sec | 0 sec |
| # | 2360564209 | 7588650571 | 4230038389 | 6419625727 |
| Pollard P-1 | 14 sec | 0 sec | 51 sec | 0 sec |
| Ifactor | 0 sec | 0 sec | 1 sec | 0 sec |
| # | 3484400113 | 3270016483 | 2280008473 | 9664174141 |
| Pollard P-1 | 22 sec | 0 sec | 49 sec | 1 sec |
| Ifactor | 0 sec | 0 sec | 1 sec | 0 sec |
| # | 6057535657 | 2247910607 | 1329560041 | 9705832013 |
| Pollard P-1 | 9 sec | 0 sec | 13 sec | 0 sec |
| Ifactor | 0 sec | 0 sec | 0 sec | 0 sec |
| # | 8004078413 | 6217528531 | 1647469741 | 5160428521 |
| Pollard P-1 | 16 sec | 0 sec | 48 sec | 0 sec |
| Ifactor | 0 sec | 0 sec | 1 sec | 0 sec |
| # | 6776672657 | 9265848689 | 1445203861 | 1234748479 |
| Pollard P-1 | 65 sec | 0 sec | 22 sec | 0 sec |

| | | | | |
|-------------|------------|------------|------------|------------|
| Ifactor | 1 sec | 0 sec | 0 sec | 0 sec |
| # | 6582132017 | 3080268461 | 2947274101 | 5310474643 |
| Pollard P-1 | 62 sec | 0 sec | 8 sec | 0 sec |
| Ifactor | 0 sec | 0 sec | 1 sec | 0 sec |

Taking averages of factorization time for the above data we get:

| | MakeKey | MPollard | Safeprimes | Random |
|-------------|----------|----------|------------|---------|
| Pollard P-1 | 29.4 sec | 0 sec | 35.4 sec | 0.3 sec |
| Ifactor | 0.5 sec | 0 sec | 0.75 sec | 0 sec |

Conclusion:

It is interesting to note that the MakeKey moduli which Pollard P-1 factored the fastest (4 seconds) was comprised of a large prime and a small prime, namely $n = 8007054647 = 2781193 * 2879$ - evidently small factors are undesirable when creating RSA moduli (as we already knew - this is just a nice reinforcement). However, from the above table we can conclude that the MakeKey procedure for designing moduli has overcome the Pollard P-1 factorization technique - as long as neither 'seed' is chosen to small!! Experimentation with larger moduli revealed no increase in factorization time for either of the primes generated randomly or using the MPollard procedure (these composites remained 'factorable'). However, the moduli created by either the safeprime method or the MakeKey procedure rapidly become infactorable. At a mere 15 digits, Pollard P-1 cranked away, never returning an answer in any span of time which I was willing to wait. At 30 digits, even Maple's Ifactor started slowing up, averaging 500 seconds per factorization. At 40 digits, I got tired of waiting. Ideally, for maximum security in RSA moduli, this experiment should be repeated for other common factorization techniques such as the Pollard Phi to ensure that the composites generated by MakeKey truly are infactorable within any reasonable amount of time.