Department of Mathematics

Integer Factorization Problem
An Attack on the RSA Public-Key Encryption Scheme

Maria Chumpitaz, Chad Cole, Haley Hamer, Alisa Iduma, Lucero Morales

Advisor: Dr. Charles Lam Assistant: Frank Madrid

CSU Bakersfield

Chevron

Research Experience Vitalizing Science — University Program

Partial support for this work was provided by the National Science Foundation's Federal Cyber Service: Scholarship for Service (SFS) program under Award No. 1241636.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Introduction:

RSA is one of the encryption systems that depend upon the complexity of the integer factorization problem to prevent an unwelcomed third party from decoding the message. The function of this system involves a public key (*n,e*), *n* being a product of two large prime numbers that are not easily factored and *e* being a random number between 1 and *Φn,* and a private key *(d),* which can be found using the formula: $e \times d \equiv 1(mod(\Phi n))$. For example, Alice creates a public and private key, then she makes the public key available to anyone who wants to send her a message. Once Alice gets an encrypted message back, only she is able to decrypt it with the private key. However if an eavesdropper, Eve, is able to factor *n* then she could possibly find the private key *d* by using the previous formula, then plug *d* into $m' = c^d(mod\ n)$, m' being the decrypted message and *c* being the encrypted message.

Background:

Three possible ways to break the RSA algorithm are Trial Division, Pollard's Rho Algorithm, and Pollard's p-1 Factoring Algorithm. Trial division factors smaller numbers (less than one million) and does not work well with semi primes because they can be fairly large numbers. The Pollard Rho Algorithm allows the division process to be much quicker and allows the possibility of finding the two numbers that divide into a composite number. Pollard's p-1 Factoring Algorithm finds prime factors p by dealing with p-1. Each of the algorithms are efficient in certain cases.

Method & Data:

**The computer program Maple16 was used to find the calculations..*
**The computer generates two random prime integers between the interval $2^i..2^{i+1}$. Then the algorithm multiplies these two primes together to create n. Next, the algorithm tries to factor the n back into the two random primes p and q.*

Trial Division

```
> trial := proc(p, q)
  local n, count, i;
  n := p·q :
  count := 0 :
  for i from 2 to ceil(evalf(sqrt(n))) do
    count := count + 1;
    if n mod i = 0 then
      RETURN("factor", i, "count", count);
      break;
    fi;
    if i = ceil(sqrt(n)) then
      RETURN("fail", count);
      break;
    fi;
  od:
  end:
> i := 10 :
  generate := rand(2^i..2^(i+1)) :
  st := time( ) :
  for i from 1 to 1000 do
    a := nextprime(generate( )) :
    b := nextprime(generate( )) :
    trial(a, b);
  od:
  time( ) − st;
```

Trial Division (1000 runs)		
Size*(bits)*	Time(s)	Time per run(s)
0	0.062	0.00062
1	0.327	0.000343
2	1.591	0.0015974
3	3.385	0.0035412
4	8.985	0.0091666
5	20.217	0.0206574
6	41.839	0.0424666
7	89.030	0.09039
8	182.022	0.1849202
9	359.239	0.3694976
10	736.029	0.7393542

Trial division worked extremely fast for very small numbers. It depends on the factorization of the smallest primes, and it divides the number by the next prime until it works and is fully factored. According to the data gathered, the time grows exponentially as the bits increased. So with larger numbers, the time taken will increase greatly between two consecutive bits.

Pollard's Rho Algorithm

```
> rho := proc(p, q)
  local n, count, a, b, d;
  n := p·q :
  count := 0 :
  a := 2 :
  b := 2 :
  while(true) do
    count := count + 1;
    a := a^2 + 1 mod n;
    b := b^2 + 1 mod n;
    b := b^2 + 1 mod n;
    d := gcd(a − b, n);
    if 1 < d < n then
      RETURN("factor", d, "count", count);
      break;
    elif d = n then
      RETURN("fail", count);
      break;
    fi;
  od:
  end:
> i := 30 :
  generate := rand(2^i..2^(i+1)) :
  st := time( ) :
  for i from 1 to 1000 do
    p := nextprime(generate( ));
    q := nextprime(generate( ));
    rho(p, q);
  od:
  time( ) − st;
```

Pollard Rho's Algorithm (1000 runs)		
Size*(bits)*	Time(s)	Time per run(s)
0	.078	.000078
1	.109	.0001218
2	.110	.0000968
3	.203	.0002214
4	.234	.000231
5	.515	.000312
6	.328	.000365
7	.500	.000499
8	.842	.0008922
9	1.180	.0012386
10	1.638	.0015694
11	1.950	.002081
12	3.198	.0032324
13	4.430	.0043898
14	6.396	.0066052
15	9.173	.0093162
16	13.213	.0136
17	19.531	.0192444
18	27.035	.0269786
19	39.062	.0389348
20	54.726	.055633
21	77.922	.0786558
22	106.549	.1116748
23	154.335	.1565498
24	208.979	.21477433
25	295.996	.304534
26	440.126	.43169233
27	597.390	.6116485
28	890.127	.8772565
29	1223.781	1.232044
30	1744.263	1.748438

In order to factor numbers, there are easier ways to factor than just dividing by the smaller prime numbers such as 2, 3, and 5 like trial division. Once the numbers begin to get larger, this becomes a lengthy and time consuming process. The Pollard Rho Algorithm allows the division process to be much quicker and allows the possibility of finding two numbers that can divide into the number. Based on the data obtained, the time increased as the numbers became larger. If one were to test further, one would expect the difference in time, between two consecutive numbers, to be even more drastic.

Pollard's p-1 Factoring Algorithm

```
> poll := proc(p, q)
  local n, a, B, m, l, d;
  n := p·q :
  a := 2 :
  B := 1 :
  m := 1 :
  while(true) do
  while(m ≤ B) do
  m := nextprime(m);
  l := floor((ln(n))/(ln(m)));
  a := power(power(a, m) mod n, l) mod n;
  od:
  d := gcd(a − 1, n);
  if 1 < d < n then RETURN("Count", B, "Factor", d);
  else B := B + 1;
  fi;
  if d = n then RETURN("Fail"); fi;
  od:
  end:
> i := 0 :
  generate := rand(2^i..2^(i+1)) :
  st := time( ) :
  for i from 1 to 100 do
  p := nextprime(generate( ));
  q := nextprime(generate( ));
  poll(p, q);
  od:
  time( ) − st;
```

Pollard's p-1 (100 runs)		
Size*(bits)*	Time (s)	Time per run(s)
0	0	0
1	.124	.00124
2	.202	.00202
3	.250	.00250
4	.343	.00343
5	.468	.00468
6	.468	.00468
7	1.185	.01185
8	1.123	.01123
9	1.170	.01170
10	1.966	.01966
11	4.353	.04353
12	6.536	.06536
13	6.927	.06927
14	14.399	.14399
15	21.450	.21450
16	34.117	.34117
17	50.981	.50981
18	129.902	1.29902
19	214.829	2.14829
20	247.855	2.47855

Pollard's p-1 Factoring Algorithm is effective, but not to the extent that the Rho Algorithm is because of the complexity of the calculating process that this algorithm undergoes. All of the numbers tested in the data above were successfully factored, and longer numbers did result in a longer run time. Based on the data collected, as the bits get larger, the time it will take to break the numbers will grow at a steadily increasing rate. At the beginning it is reasonably slow, however, as the number grows, this algorithm becomes more useful as it is intended for a certain range of numbers.

Conclusion:

Trial division worked well for smaller numbers but became extremely time consuming for large numbers. Pollard's p-1 Factoring Algorithm was moderately better than trial division, however, as the numbers became even larger, it, too, took more time. Overall, Pollard's Rho Algorithm was the fastest and most effective method out of the three tested. Even though the algorithm was able to factor reasonably large numbers, there is a point where the value of *n* could become so large that no method could factor the number fast enough to crack the code for Eve before it becomes useless.

Future Work:

Aside from the algorithms researched in this project, there are many different methods that are more complex that can be used to solve the integer factorization problem faster. These include but are not limited to: elliptic curve factoring, random square factoring methods, quadratic sieve factoring, and number field sieve factoring. Random square factoring method finds the factors by finding the congruence of squared modulo of n, however, it has not been completely developed. The quadratic sieve factoring method is one of the faster ways to factor and is similar to the random square factoring method except it requires large amounts of memory. The recommended size for RSA is 4096- bits, basic unit of information, because it would take an eavesdropper years to crack. Therefore, people with adequate resources could further explore this method.

References:

1. "Courses: Sriram Sankaranarayanan." *A Quick Tutorial on Pollard's Rho Algorithm []*. N.p., n.d. Web. 29 July 2014.
2. "Eavesdrop Stock Illustrations." *Eavesdrop Stock Illustrations*. N.p., n.d. Web. 04 Aug. 2014.
3. Menezes, A. J., Van Oorschot Paul C., and Scott A. Vanstone. *Handbook of Applied Cryptography*. Boca Raton: CRC, 1997. Print.
4. "Pictures Of..." *A Young Boy Working On A Computer*. N.p., n.d. Web. 04 Aug. 2014.
5. "Record 232-digit Number from Cryptography Challenge Factored | Observations, Scientific American Blog Network." *Scientific American Global RSS*. N.p., n.d. Web. 04 Aug. 2014.
6. "RSA Key Sizes: 2048 or 4096 Bits?" *Welcome*. N.p., 18 June 2013. Web. 05 Aug. 2014.
7. "Stock Photography and Stock Footage." *Clipart of Two Women Talking through Tin Cans Jba0871*. N.p., n.d. Web. 04 Aug. 2014.
8. "Incrementando La Productividad. Utilizando MÃºltiples Monitores." Incrementando La Productividad. N.p., n.d. Web. 06 Aug. 2014.