Analysis of the RSA Encryption Algorithm

Betty Huang

April 8, 2010

Abstract

The RSA encryption algorithm is commonly used in public security due to the asymmetric nature of the cipher. The procedure is deceptively simple, though; given two random (large) prime numbers p and q, of which n = pq, and message m, the encrypted text is defined as c = me (mod n). E is some number that is coprime to the totient(n). The public key (n, e), however, makes it difficult for the user to find the private key (n, d), due to the fact that given only n, it is extremely difficult to find the prime factors p and q. The fastest methods currently have O(sqrt(n)) complexity, but require expensive resources and technology (Kaliski). The aim of this paper is to improve on the factorization process required by the RSA encryption algorithm.

1 Introduction

My project aims to research and compare the various factorization methods available, including a proposal towards improving the current methods already available. There is an industry demand towards improving the speed of the RSA cryptosystem, as it is widely used by agencies that require secure transfer of information between clientele and administrators. Since the security of the algorithm is depended on the mathematical difficulty and computationally ?hard? problem of factoring, it is important to recognize potential exploits of factorization.

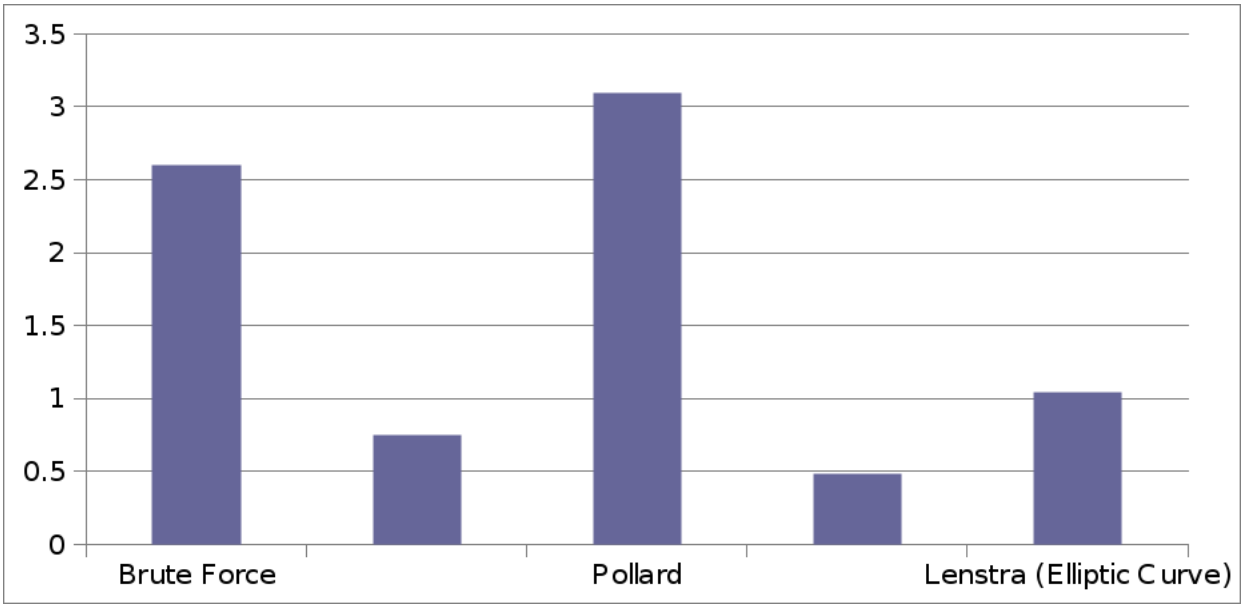

Figure 1: comparison of the various methods of integer factorization.

1.1 Background

$The field of public key cryptography is rapidly advancing, due to the growth of internet connections arou$
$1) multiplying numbers together is easy, and 2) factoring numbers is hard (Kaliski). Current methods$
$(a?b)(a+b)), and is better than brute force. There have been other methods since developed, including Le$

1.2 Scope of Study

The field of study will be in encryption and cryptanalysis in C.

2 Review of Literature

Rivest's original paper on the RC5 algorithm has been helpful in my research and understanding of the encryption mechanism in order to approach it from an angle of attack. Since I wanted to experience the thought process on breaking a cipher, I did not review other, widely available literature on RC5 cryptanalysis. After that, I studied two papers that detailed two different methods in breaking the RC5. The first paper, titled "On Differential and Linear Cryptanalysis of the RC5 Encryption Algorithm," offered the first

approach taken towards attacking the RC5. Authors Burton S. Kaliski Jr. and Yiqun Lisa Yin focused efforts on recovering the expanded S table, rather than the input (thus, their results are key-length independent). Figure two explains the algorithm procedure that Yin et al. used. Refer to Appendix A for the version that I used for RC5.

In conclusion Yin et al suggest using a 12 round implementation of the RC5, which helps prevent against most common differential/linear cryptanalysis attacks. However, this paper had been published in 1995, and I expect that recent innovations would make it possible to use attacks against the RC5, so I continued looking for more recent papers. The most recent update was in 1998, also authored by Yin and Kaliski, which further expanded on their research.

There are several methods in dealing with block ciphers:

1. The exhaustive search - this is the most intuitive and brute-force type of attack. Simply put, the attacker finds one plaintext/ciphertext combination, and then essentially runs through all possible combinations until a match is found.

2. Statistical searches - the attacker analyzes patterns and matches between plaintext/ciphertext combinations

3. Differential Cryptanalysis - the attacker choosese two plaintexts, P1 and P2, with altercations ("difference") P' between the two. After P1 and P2 are encrypted, their ciphertexts C1 and C2 are compared, and the difference between the two is identified as C'. The goal of differential cryptanalysis is to find a a pair of (P', C') that occur with more than normal probaility.

4. Linear Cryptanalysis - the idea here is to find items (plaintexts, ciphertexts, etc.) that occur over several iterations with probaility != 1/2 (Kaliski).

This research has led me since to Rivest's RSA Encryption Algorithm; The first paper I encountered was published by B. Kaliski, which described the mathematical properties that functioned as the backbone of the RSA encryption algorithm. He details several mathematical properties that are mentioned in my literature review of the paper. In summary, the core aspect of the security of the RSA is that factorization is difficult, or expensive. In

terms of time spent/resources required, factoring large numbers has gained notoriety in the mathematic community since the advent of these usage. The RSA exploits this inherent property in order to increase security, but is considerably slower than other competing asymmetric ciphers (ex. DES).

3 Development

For the first part of the quarter, I refined the code implementation of the RSA cryptosystem in C and reading the mathematical concepts behind the code. In addition, I read several papers outlining the core concepts and foundations that the algorithm rests upon. When choosing large prime numbers, it?s often expensive for computer systems to process 200+ digits, which increases the difficulty (ie. memory and time spent) of the problem. Upon further examination, I found that the fastest method available was the Quadratic Sieve, explained in Appendix B. The QS lends itself to parallel implementation, and reigns as champion in terms of speed for factorization fewer than 110 digits. Numbers that exceed this bound are better approximated by General Number Field Sieve (Gerver). As of now, I have tested code implementations of several various prominent prime factorization techniques in order to determine the fastest on several conditions. Although many authors (Gerver, Pearson, Pomerance) agree that the QS is the fastest algorithm up to 110 digits, it is not as successful for ?medium large? numbers, as shown in Figure A.

3.1 Overview

The second part of my development during this quarter was the implementation and testing of the RSA algorithm in C. The code is included in Appendix B, and the results are as follows:

4 Results

(message modification generated from CrypTool version 1.4.21)

Original Text:
00000 44 65 61 72 20 4D 72 20 53 68 6F 70 61 68 6F 6C 69 63 Dear Mr Shopaholic

00012 2C 0D 0A 0D 0A 70 6C 65 61 73 65 20 6F 72 64 65 72 20 ,....please order
00024 61 20 50 6F 72 73 63 68 65 20 61 6E 64 20 61 20 70 72 a Porsche and a pr
00036 65 70 61 69 64 20 69 6E 73 75 72 61 6E 63 65 20 73 63 epaid insurance sc
00048 68 65 6D 65 20 66 6F 72 20 4D 72 2E 20 44 6F 64 67 79 heme for Mr. Dodgy
0005A 2E 0D 0A 0D 0A 52 65 67 61 72 64 73 0D 0A 48 6F 6E 65 .....Regards..Hone
0006C 73 74 20 4A 6F 68 6E 0D 0A st John..

Name: SHA-1
Length in bit: 160
Algorithm ID: 30 21 30 09 06 05 2B 0E 03 02 1A 05 00 04 14

SHA-1 HASH - 2C 6B 70 15 B2 59 7A A6 43 44 43 80 08 B2 9B A9 8F EE 24 88

RSA KEY
Bit length of N: 304
RSA modulus N: 642950776111283768964376349927412243443720271264342437889420595499271
phi(N) = (p-1)(q-1):
642950776111283768964376349927412243443720270757089183452636785524690895455913897335556
Public key: 65537
Private key:
258791194638746884073291174981082557483587851477810329589348089863066987064277472234 23

ENCRYPTED HASH VALUE
Padding string: 01 00
Algorithm ID: 30 21 30 09 06 05 2B 0E 03 02 1A 05 00 04 14
Hash value: 2C 6B 70 15 B2 59 7A A6 43 44 43 80 08 B2 9B A9 8F EE 24 88

ASN-1 hash value: 01 00 30 21 30 09 06 05 2B 0E 03 02 1A 05 00 04 14 2C 6B 70 15 B2 59 7A A6
43 44 43 80 08 B2 9B A9 8F EE 24 88
Length in bit: 296

Encrypted hash value: 15 45 EF 40 D3 49 DB 69 48 6D 1B 2E F5 A4 EC D6 51 EC AC 99 10 F3 78 E2 CF
45 0C E4 74 3C 03 FD 30 BA 07 5D B8 02
Length in bit: 304

Result from program:

Enter N: 642950776111283768964376349927412243443720271264342437889420595499271490891

Enter Message (in hexadecimal – post message modification):
2C6B7015B2597AA64344438008B29BA98FEE2488

Public Key (304,65537)
Private Key
(204,258791194638746884073291174981082557483587851477810329589348089863066987064277472

Ciphertext: 1545EF40D349DB69486D1B2EF5A4ECD651ECAC9910F378E2CF450CE4743C03FD
Time: 1.324 seconds

Appendix A. A Simple Example of the RSA Encryption Algorithm

The algorithm ?
$c = M^e(modn)$
$M = c^d(modn)$
Public key: (n, e)
Private key: (n, d)
Where C represents the ciphertext, and M represents the plaintext message in numeric format.
Choose two large primes (100 digits or more) p and q. For simplicity, small primes will be chosen so the math is easier to follow: p = 11 and q = 7. Multiply p*q, and set N equal to the result. This N is part of the public key.
N = p*q = 11 * 7 = 77
Choose a number e that is coprime (shares no common factors) with k=(p -

1)(q - 1).
k=(p - 1)(q - 1) = (10)(6) = 60.
We choose e to be 13, which is also part of the public key. This information is sufficient to encode the message.
For the decryption, one must find d such that ed = 1 (mod k). Since the number k is not released to the public, it is extremely difficult for someone to find d. Since we do know what k is, we find that:
13d = 1 (mod 60), or the inverse modular function
d = 37
Suppose a friend passes the message m = 53 (note that 1 ¡ m ¡ N). In order to encrypt the message,
$C = m^e(modN)$
$C = 53^{13}(mod77)$
The subsequent decryption:

$M = c^d(modN)$
M = 53

Appendix B. Comparison table of various integer factorization methods

1.55E+45 Time Success? Resultant

Brute Force 2.6 no 2 * 3 * 607 * 7669 * 553303237539342315529035815346023343 49
Brent (p-1) 0.748 yes 2 * 3 * 607 * 7669 * 20947 * 413551 * 51083807 * 370770947 *
337227786373 Pollard 3.099 no 2 * 3 * 4655083 * 553303237539342315529035815346023343 49
Williams (p+1) 0.483 yes 2 * 3 * 607 * 7669 * 20947 * 413551 * 51083807 * 370770947 *
337227786373 Lenstra (Elliptic Curve) 1.044 yes 2 * 3 * 607 * 7669 * 20947 * 413551 * 51083807 *
370770947 * 337227786373

Appendix C. Code

```
                                                                                1
#include <stdio.h>                                                              2
#include <math.h>                                                               3
                                                                                4
int phi,M,n,e,d,C,FLAG,m,D;                                                     5
                                                                                6
int gcd(int a,int b) //greatest common divisor                                  7
{                                                                               8
  int c;                                                                        9
                                                                                10
  if(a<b)                                                                       11
  {                                                                             12
    c = a;                                                                      13
    a = b;                                                                      14
    b = c;                                                                      15
  }                                                                             16
                                                                                17
  while(1)                                                                      18
  {                                                                             19
    c = a%b;                                                                    20
    if(c==0)                                                                    21
      return b;                                                                 22
    a = b;                                                                      23
    b = c;                                                                      24
  }                                                                             25
}                                                                               26
                                                                                27
// int privatekey(int val)                                                      28
// {                                                                            29
//  int d;                                                                      30
//  d= e                                                                        31
                                                                                32
}                                                                               33
                                                                                34
int totient(int X) // calculates how many numbers between 1 and                 35
   N- 1 which are relatively prime to
N.                                                                              36
{                                                                               37
  int i;                                                                        38
        phi = 1;                                                                39
        for (i = 2 ; i < X ; ++i)                                               40
            if (gcd(i, X) == 1)                                                 41
                ++phi;                                                          42
```

```
     return phi;
int encrypt(int message)
{

  int result;
  result=message;
  int x;
  for (x=0;x<e;x++)
  {

    result=result*message;

  }

  result=result%n;
  return result;

}

int decrypt(int ciphertext)
{
  int x;

  int plainres=plainres;
  for (x=0;x<d;x++)
  {
    plainres=plainres*ciphertext;
  }
  plainres=plainres%n;
  return plainres;
}

}
```

References

[1] D. Pearson, A parallel implementation of RSA,? Cornell University (July 1996).

[2] C. Pomerance, Analysis and comparison of some integer factoring algorithms,? Mathematisch Centrum Computational Methods in Number Theory, Pt. 1 p 89-139(SEE N 84-17990 08-67) (1982).

[3] H. W. Lenstra Jr, Factoring integers with elliptic curves, Annals of mathematics 126, no. 3 (1987): 649-673.

[4] Factoring large numbers with a quadratic sieve, Mathematics of Computation 41, no.163 (1983): 287-294.

[5] T. Denny et al., On the factorization of RSA-120, in Advances in Cryptology?CRYPTO?93, 166?174.

[6] Parallel implementation of the RSA public-key cryptosystem, International Journal of Computer Mathematics 48, no. 3 (1993): 153-155.

[7] The Mathematics of the RSA Public-Key Cryptosystem, RSA Laboratories. April 9 (2006).

RC5 v1.1,