Journal of Theoretical and Applied Information Technology
15th June 2021. Vol.99. No 11
© 2021 Little Lion Scientific

ISSN: **1992-8645** www.jatit.org E-ISSN: **1817-3195**

FACTORIZATION OF SMALL RPRIME RSA MODULUS USING FERMAT'S DIFFERENCE OF SQUARES AND KRAITCHIK'S ALGORITHMS IN PYTHON

[1] MAYA SILVI LYDIA, [2] MOHAMMAD ANDRI BUDIMAN, [3] DIAN RACHMAWATI
[1, 2, 3] Departemen Ilmu Komputer, Fakultas Ilmu Komputer dan Teknologi Informasi, Universitas Sumatera Utara, Jl. Universitas No. 9-A, Kampus USU, Medan 20155, Indonesia
E-mail: [1] maya.silvi@usu.ac.id, [2] mandrib@usu.ac.id, [3] dian.rachmawati@usu.ac.id

ABSTRACT

The RPrime RSA is one among many public key algorithms that relies its security on the hardness of finding prime factors of a very large integer. While its predecessor, the RSA, utilizes two large prime numbers to formulate its modulus, the RPrime RSA can utilize two or more prime numbers. Therefore, the RPrime RSA is intuitively taken into account to be securer the than the RSA. The modulus of the RPrime RSA, *n*, is the public key whose size determines the security of the entire cryptosystem: the larger the modulus, the securer the cryptosystem. In this study, we shall show the way to factorize small modulus of the RPrime RSA using two factorization algorithms, which are the Fermat's difference of squares algorithm and the Kraitchik's algorithm. The programming of the factorization is completed using Python programming language. Our study shows that both of Fermat's difference of squares and Kraitchik's algorithms can be used effectively as methods to factorize small modulus of RPrime RSA. The time both algorithms take to factorize is mostly directly proportional to the size of *n*. However, Fermat's difference of squares is much faster than Kraitchik's algorithm: while factoring six digits to eleven digits of *n*, Fermat's is about 24 to 550 times faster than Kraitchik.

Keywords: *Cryptography, Public Key Cryptosystem, Cryptanalysis, Rprime RSA, Difference of Squares, Kraitchik*

1. INTRODUCTION

Cryptography is generally viewed as a science and strategy of using mathematical operations to secure communication [1]. One aspect to maintain secure communication is to maintain confidentiality. Maintaining confidentiality is done by encryption algorithm which consists of two mathematical functions: *encryption function* and *decryption function*. The decryption function is the inverse of the encryption function.

There are two kinds of encryption algorithms: *symmetric* and *public key*. In symmetric encryption algorithm, message is passed into encryption function, obfuscated into absurd form using a secret key. This secret key is generated by the sender. The obfuscated message is sent to a legitimate recipient via a communication channel. The secret key is also sent to this recipient, but a secure channel must be used to send it. Upon receiving the key, the recipient uses the decryption function to convert the obfuscated message into the original message.

Public key encryption is a part of *public key cryptography*, a term brought forward by Whitfield Diffie and Martin E. Hellman in 1976 [2]. In public key encryption, a recipient generates two kinds of keys: *public key* and *private key*. The public key is published by electronic meanings and the private key is kept as a secret. Anyone who wants to send a message to this recipient must know his or her public key. The encryption function makes use of this public key to encrypt the message. The encrypted message is then sent to the recipient. With his or her private key, the recipient can then decrypt the message using the decryption function.

The Rivest-Shamir-Adleman (RSA) algorithm [3] was among the first algorithm to implement the Diffie-Hellman concept of public key cryptography. Formulated in 1978 by Ronald Rivest, Adi Shamir, and Leonard Adleman, the RSA has been the most popular and the most widely used public key encryption algorithm until now. The RSA utilizes two very large prime numbers to formulate its modulus. The larger the prime numbers, the securer the RSA.

The RSA modulus, n, is simply the multiplication of p and q, two large distinct prime numbers. The modulus is published, so every party knows its value. However, the two prime numbers are kept private. If a cryptanalyst wants to break the RSA cryptosystem, he or she must be able to factor the modulus into its prime factors. If the modulus is too small, for example, $n = 1691$, it only takes seconds for the cryptanalyst to determine that $1691 = 89 \times 19$ using brute force search, and the RSA cryptosystem is compromised. Therefore, only two very large distinct prime numbers should be used to formulate the modulus.

The RPrime RSA cryptosystem [4] is one of various variants of the RSA cryptosystem. The RPrime RSA is actually a combination of two previous variants of RSA: the Mprime RSA and the Rebalanced RSA [5]. The encryption method is the same as the original RSA, while the key generation method is similar to the Rebalanced RSA and the decryption method is similar to the Mprime RSA.

Unlike the original RSA, the RPrime RSA uses three or more large distinct prime numbers to construct its modulus. Therefore, one may notice that RPrime RSA is securer than the original RSA, since a cryptanalyst will find that factoring an integer that has three or more prime factors is certainly harder than factoring another integer that has only two prime factors.

In order to understand how hard it is to cryptanalize the RPrime RSA, in this study we try to factor some small RPrime RSA moduli using two factorization algorithms: the difference of squares algorithm and the Kraitchik algorithm.

The difference of squares algorithm is a classic integer factorization algorithm proposed by Pierre de Fermat in the 1600s. The difference of squares algorithm is shown to be efficient if the difference between the two factors is small [7].

The Kraitchik algorithm is a factoring algorithm that is based on congruence [8]. The algorithm employs the concept of perfect squares and Euclidean greatest common divisors at the core of its computations.

It is noted that both Fermat's difference of squares and Kraitchik's algorithms can factor an integer to its two factors. However, the RPrime RSA modulus can consist of two or more factors — and these factors must be prime numbers. Therefore, if these two algorithms are going to be utilized to factor the RPrime RSA modulus, we need to make some appropriate adjustments.

In this study, several RPrime RSA moduli ranging from 44745833 (8 digits) to 171464936134901 (15 digits) are factored by the Fermat's difference of squares and Kraitchik's algorithms. These RPrime RSA moduli (n) are products of five Rprime RSA private keys, so that $n = p \times q \times r \times s \times t$. The parameters $p, q, r, s,$ and t are distinct prime numbers that were generated randomly. The factoring time of both algorithms is recorded to conclude which of these two algorithms can factor the Rprime RSA modulus most efficiently.

2. THE RPRIME RSA

The RPrime RSA [4] is a variant of the RSA cryptosystem [3] that combines the key generation method of the Rebalanced RSA [5] and the decryption method of the Mprime RSA [8]. As many other public key cryptosystems, the RPrime RSA cryptosystem consists of three stages, namely key generation, encryption, and decryption.

The key generation stage is conducted by the recipient. The steps are as follows [4]:

1. Choose k, the numbers of prime numbers to be employed.
2. Generate k random and distinct prime numbers, $p_1, p_2, ..., p_k$. This can be done using primality test, such as Fermat Little Theorem [10] or Agrawal-Biswas algorithm [11]. The larger the primes, the securer the cryptosystem, but the slower the encryption and decryption processes.
3. Compute $n = p_1 \times p_2 \times ... \times p_k$.
4. Choose $dp_1, dp_2, ... dp_k$, such that:
   a. $dp_1, dp_2, ... dp_k$ are odd numbers.
   b. $gcd(dp_1, p_1 - 1) = gcd(dp_2, p_2 - 1) = ... = gcd(dp_k, p_k - 1) = 1$.
5. Find the solution of these Chinese Remainder Theorem (CRT) equations (for examples on CRT, see [12]):
$$d \equiv dp_1 \pmod{p_1 - 1}$$
$$d \equiv dp_2 \pmod{p_2 - 1}$$
$$\ldots$$
$$d \equiv dp_k \pmod{p_k - 1}$$
   The value of d will be the decryption key.
6. Calculate $\Phi(n) = (p_1 - 1) \times (p_2 - 1) \times ... \times (p_k - 1)$.
7. Calculate $e \equiv d^{-1} \pmod{\Phi(n)}$. Examples on how to compute a multiplicative inverse can be found on [13]).
8. Publish the public key (n, e).
9. Keep the private key $(p_1, p_2, ... p_k, dp_1, dp_2, ... dp_k, d, \Phi(n))$.

The encryption stage is conducted by the sender. The steps are as follows [4]:

1. Obtain the public key *(n, e)* from the recipient.
2. Let m be the message the sender wants the recipient to transmit securely.
3. Calculate the ciphertext $c = m^e \bmod n$.
4. Send the ciphertext c to the recipient.

The decryption stage is conducted by the recipient. The steps are as follows [4]:

1. Get the ciphertext c from the sender.
2. Calculate the original message $m = c^d \bmod n$.

Now, let us illustrates how the RPrime RSA cryptosystem works. Suppose a sender, Alice, would like to send a simple message to a recipient, Bob. In the key generation stage, Bob will generate his public and private keys as follows:

1. Bob chooses $k = 3$.
2. With his chosen primality test algorithm, Bob generates 3 primes: $p_1 = 907$, $p_2 = 149$, and $p_3 = 359$.
3. Bob computes $n = p_1 \times p_2 \times p_3 = 907 \times 149 \times 359 = 48516337$.
4. Obeying the mathematical requirements, Bob chooses $dp1 = 77$, $dp2 = 95$, and $dp3 = 243$.
5. Using CRT, Bob computes the decryption key $d = 31578707$.
6. Bob calculates $\Phi(n) = (p_1 - 1) \times (p_2 - 1) \times (p_3 - 1) = (907 - 1) \times (149 - 1) \times (359 - 1) = 48003504$.
7. Bob calculates $e \equiv d^{-1} \pmod{\Phi(n)}$. Thus, $e \equiv 31578707^{-1} \equiv 41046683 \pmod{48003504}$.
8. Bob publishes his public key $(n, e) = (48516337, 48003504)$.
9. Bob keeps his private key $(p_1, p_2, p_3, dp_1, dp_2, dp_3, d, \Phi(n)) = (907, 149, 359, 77, 95, 243, 31578707, 48003504)$.

In the encryption stage, Alice will do as follows:

1. From Bob, Alice obtains Bob's public key $(n, e) = (48516337, 48003504)$.
2. Suppose Alice wants to send a simple message that only contains one letter *"B"*. Alice looks up the ASCII table and finds out that the letter *"B"* corresponds to the value of *66*. Therefore, Alice lets the message $m = 66$.
3. Alice computes the ciphertext $c = m^e \bmod n = 66^{48003504} \bmod 48516337 = 21723622$.
4. Via a channel, which can be secured or unsecured, Alice transmits the ciphertext c to Bob.

In the decryption stage, Bob will do as follows:

1. From Alice, Bob gets the ciphertext $c = 21723622$.
2. Bob calculates the original message $m = c^d \bmod n = 21723622^{31578707} \bmod 48516337 = 66$. Bob looks up the ASCII table for the value of 66 and gets the letter "B" which is the message Alice wants him to read.

3. THE FERMAT'S DIFFERENCE OF SQUARES FACTORIZATION ALGORITHM

The difference of squares algorithm is a classic integer factorization algorithm made by French mathematician Pierre de Fermat. The difference of squares algorithm can factor an integer into its two factors. The algorithm works as follows [7][14]:

1. Let n be the integer to be factorized.
2. Compute $r = \lceil \sqrt{n} \rceil$.
3. Compute $s = r^2 - n$.
4. While s is not a perfect square, then do step 4a, 4b, 4c:
   a. Increment r by *1*, *i.e.*, $r = r + 1$.
   b. Compute $s = r^2 - n$.
   c. If s is a perfect square, then go to step 5, else go to step 4.
5. Output $r - \sqrt{s}$ as a factor of n.

For example, let us find the factor of 85. The algorithm works as follows:

1. Let n = 85.
2. Compute $r = \lceil \sqrt{n} \rceil = \lceil \sqrt{85} \rceil = \lceil 9.22 \rceil = 10$.
3. Compute $s = r^2 - n = 10^2 - 85 = 100 - 85 = 15$.
4. Since s is not a perfect square, we do:
   a. r = r + 1 = 10 + 1 = 11.
   b. $s = r^2 - n = 11^2 - 85 = 121 - 85 = 36$
   c. s is a perfect square, so we go to step 5.
5. Output $r - \sqrt{s} = 11 - \sqrt{36} = 11 - 6 = 5$.

Here we have 5 as a factor of 85. The other factor is 85/5 = 17.

From this example, we have 5 and 17 as the factors of 85. Interestingly, 5 and 17 are prime numbers. Thus, it begs the question: does the

Fermat's difference of squares always return prime number as the factors?

To answer the question, let us try to factorize n = 135. So, r=$\lceil \sqrt{n} \rceil = \lceil \sqrt{135} \rceil = \lceil 11{,}62 \rceil = 12$. Compute $s = r^2 - n = 12^2 - 135 = 144 - 135 = 9$. Since s is a perfect square, the algorithm terminates, and we output $r - \sqrt{s} = 12 - \sqrt{9} = 12 - 3 = 9$ as the factor of n. The other factor is 135/9 = 15. Thus, the factors of 135 are 9 and 15, and both are not prime numbers. Therefore, we conclude that:

1. The Fermat's difference of squares does not always return prime number as the factors.
2. As the modulus of RPrime RSA are constructed as a multiplication of some distinct prime numbers, the Fermat's difference of square algorithm must be modified so it can factorize the modulus.

4. THE KRAITCHIK'S FACTORIZATION ALGORITHM

The Kraitchik's algorithm is a newer integer factorization algorithm proposed by Maurice Kraitchik in the 1920's [8]. Kraitchik's algorithm is an improvement to Fermat's difference of square algorithm [15] [16] [17] [18] [19] [20]: while Fermat pondered at the relation $n = x^2 - y^2 = (x + y)(x - y)$ to get the intuitive idea of factoring n, Kraitchik reformulated the relation into $x^2 - y^2 \equiv 0 \pmod{n}$. Kraitchik detected that $x^2 - y^2$ was indeed a multiple of n, so that it can be written as $x^2 - y^2 = kn$, assuming that $x > y$.

The Kraitchik's factorization algorithm is as follows [8]:

```
1. Let $n$ be the integer to factor.
2. Compute $x = \lceil \sqrt{n} \rceil$
3. While the factor of $n$ is not found do:
    a. Let $k = 1$
    b. While $x^2 - kn \geq 0$ do the following:
        i. $y = \sqrt{x^2 - kn}$
        ii. If y is a perfect square and $(x + y) \bmod n \neq 0$ and $(x - y) \bmod n \neq 0$ then the factor of $n$ is $p = \gcd(x + y, n)$ and $q = \gcd(x - y, n)$, then stop.
        iii. Else let $k = k + 2$
    c. Let $x = x + 1$
```

Let us again find the factors of 85. So we let $n = 85$. We compute $x = \lceil \sqrt{n} \rceil = \lceil \sqrt{85} \rceil = \lceil 9.22 \rceil = 10$

At the outer loop, we let $k = 1$

Since $x^2 - kn = 10^2 - 1 \times 85 = 15 \geq 0$ we get into the inner loop.

$y = \sqrt{x^2 - kn} = \sqrt{15} = 3.87$ so y is not a perfect square then we let $k = k + 2 = 1 + 2 = 3$.

Since $x^2 - kn = 10^2 - 3 \times 85 = -155 < 0$ we exit the inner loop and we compute $x = x + 1 = 10 + 1 = 11$.

The factor is still not found so we get into the outer loop and we let $k = 1$.

Since $x^2 - kn = 11^2 - 1 \times 85 = 36 \geq 0$ we get into the inner loop.

$y = \sqrt{x^2 - kn} = \sqrt{36} = 6$ so y is not a perfect square then we let $k = k + 2 = 1 + 2 = 3$.

Since $x^2 - kn = 11^2 - 3 \times 85 = -134 < 0$ we exit the inner loop and we compute $x = x + 1 = 11 + 1 = 12$.

The factor is still not found so again we get into the outer loop and we let $k = 1$.

Since $x^2 - kn = 12^2 - 1 \times 85 = 59 \geq 0$ we get into the inner loop.

$y = \sqrt{x^2 - kn} = \sqrt{59} = 7.68$ so y is not a perfect square then we let $k = k + 2 = 1 + 2 = 3$.

Since $x^2 - kn = 12^2 - 3 \times 85 = -111 < 0$ we exit the inner loop and we compute $x = x + 1 = 12 + 1 = 13$.

The factor is still not found so again we get into the outer loop and we let $k = 1$.

Since $x^2 - kn = 13^2 - 1 \times 85 = 84 \geq 0$ we get into the inner loop.

$y = \sqrt{x^2 - kn} = \sqrt{84} = 9.16$ so y is not a perfect square then we let $k = k + 2 = 1 + 2 = 3$.

Since $x^2 - kn = 13^2 - 3 \times 85 = -86 < 0$ we exit the inner loop and we compute $x = x + 1 = 13 + 1 = 14$.

The factor is still not found so again we get into the outer loop and we let $k = 1$.

Since $x^2 - kn = 14^2 - 1 \times 85 = 111 \geq 0$ we get into the inner loop.

$y = \sqrt{x^2 - kn} = \sqrt{111} = 10.54$ so y is not a perfect square then we let $k = k + 2 = 1 + 2 = 3$.

Since $x^2 - kn = 14^2 - 3 \times 85 = -59 < 0$ we exit the inner loop and we compute $x = x + 1 = 14 + 1 = 15$.

The factor is still not found so again we get into the outer loop and we let $k = 1$.

Since $x^2 - kn = 15^2 - 1 \times 85 = 140 \geq 0$ we get into the inner loop.

$y = \sqrt{x^2 - kn} = \sqrt{140} = 11.83$ so y is not a perfect square then we let $k = k + 2 = 1 + 2 = 3$.

Since $x^2 - kn = 15^2 - 3 \times 85 = -30 < 0$ we exit the inner loop and we compute $x = x + 1 = 15 + 1 = 16$.

The factor is still not found so again we get into the outer loop and we let $k = 1$.

Since $x^2 - kn = 16^2 - 1 \times 85 = 171 \geq 0$ we get into the inner loop.

$y = \sqrt{x^2 - kn} = \sqrt{171} = 13.08$ so y is not a perfect square then we let $k = k + 2 = 1 + 2 = 3$.

Since $x^2 - kn = 16^2 - 3 \times 85 = 1 \geq 0$ we are still entering the inner loop for the next iteration.

$y = \sqrt{x^2 - kn} = \sqrt{1} = 1$ so y is a perfect square. We also know that $(x + y) \bmod n = (16 + 1) \bmod 85 = 17 \neq 0$ and $(x - y) \bmod n = (16 - 1) \bmod 85 = 15 \neq 0$.

Thus, the factor of n is $p = \gcd(x + y, n) = \gcd(16 + 1, 85) = 17$ and $q = \gcd(x - y, n) = \gcd(16 - 1, 85) = 5$.

The result is indeed correct since $n = 85 = p \times q = 17 \times 5$; this is exactly the same result we got with Fermat's algorithm before.

Since p and q are prime numbers, it could make one wonder if Kraitchik's algorithm always returns prime numbers. To answer this question, we again try to factor $n = 135$ with Kraitchik. Following the similar steps as above, we got $p = 27$ and $q = 5$, which is correct since $135 = 27 \times 5$. However, $27 = 3 \times 3 \times 3$, so it is not a prime number. Thus, we conclude that:

1. The Kraitchik's algorithm does not always return prime number as the factors.
2. Since the modulus of RPrime RSA are formulated as a multiplication of two or more distinct prime numbers, the Kraitchik's algorithm must also be altered so it can factorize the modulus.

5. FACTORIZATION OF RPRIME RSA MODULUS USING FERMAT'S DIFFERENCE OF SQUARES IN PYTHON

The entire computation of our proposed scheme is done in Python programming language. The version of Python is 2.7.17 and the IDE is Wing Personal version 7.2.1.0. The operating system is MacOS 11.1 (20C69), the processor is 1.4 GHz Dual-Core Intel Core i5, and the memory is 4 GB 1600 MHz DDR3. The Python codes of factoring RPrime RSA modulus using Fermat's difference of squares is as follows.

```
#title: Factoring RPrime RSA modulus with
Fermat's Difference of square
#author: Mohammad Andri Budiman, Maya
Silvi Lydia, Dian Rachmawati
#version: 1.07
#date: October 7th 2020
#time: 21.30

import math, random, time

def rnd(mini, maxi): #get random number
between mini and maxi (and including mini
and maxi)
    return random.randint(mini, maxi)

def isFermatPrime(p):
    t = 5 * len(str(p))
    for i in range(t):
        w = rnd(2, p - 1)
        if pow(w, p - 1, p) != 1:
            return False
    return True

def issquare(b):
    if math.fabs(math.sqrt(b) - int(math.sqrt(b))) > 0:
        return False
    return True

def FermatDiff(n):
    assert n % 2 == 1
    r = int(math.ceil(math.sqrt(n)))
    s = r * r - n
    #print "\nr =", r
    #print "\ns =", s
    while not(issquare(s)):
        r += 1
        s = r * r - n
        #print "\nr =", r
        #print "\ns =", s
    return r - int(math.sqrt(s))

def getAllFactorsWithFermatDiff(n):
    all_factors = []
    while n % 2 == 0:
        all_factors.append(2)
        n = n // 2
    s = 1
    while not(isFermatPrime(n)):
        s = FermatDiff(n)
        if isFermatPrime(s):
            all_factors.append(s)
        else:
            all_factors.extend( getAllFactorsWithFermatDiff (s) )
        n = n // s
        #print all_factors
    all_factors.append(n)
    return all_factors

def getAllFactorsWithFermatDiff2(n):
    all_factors = []
    while n % 2 == 0:
        all_.append(2)
        n = n // 2
    s = 1
    while not(isFermatPrime(n)):
        s = FermatDiff(n)
        all_factors.append(s)
        n = n // s
            print all_factors
    all_factors.append(n)
    return all_factors


print "Factoring RPrime RSA with Fermat's Difference of Squares Factorization Algorithm\n"

#test
n = 837233419


print "n =", n
print
print "Calculating..."

start = time.time()
fs = getAllFactorsWithFermatDiff (n)
stop = time.time()
print
print "factors of ", n, "are", fs
print
print "time =", stop - start, "secs"
print
```

6. FACTORIZATION OF RPRIME RSA MODULUS USING KRAITCHIK'S DIFFERENCE OF SQUARES IN PYTHON

The Python codes of factoring RPrime RSA modulus using Kraitchik difference of squares is as follows.

```
#title: Factoring RPrime RSA Modulus with Kraitchik Algorithm
#author: Mohammad Andri Budiman, Maya Silvi Lydia, Dian Rachmawati
#version: 1.07
#date: November 5th 2020
#time: 21:00

import math, random, time

def rnd(mini, maxi):
    return random.randint(mini, maxi)

def isFermatPrime(p):
    t = 5 * len(str(p))
    for i in range(t):
          w = rnd(2, p - 1)
          if pow(w, p - 1, p) != 1:
               return False
     return True

def euclid_gcd(m, n):
     if m % n == 0:
          return n
     return euclid_gcd(n, m % n)

def isPerfectSquare(z):
     if        math.fabs(math.sqrt(z)        -
int(math.sqrt(z))) > 0:
          return False
     return True

def Kraitchik(n):
     x = int(math.ceil(math.sqrt(n)))
     while True:
          k = 1
          while x * x - k * n >= 0:
               y = math.sqrt(x * x - k * n)
               if isPerfectSquare(y):
                    y = int(y)
                    if (x + y) % n != 0 and (x -
y) % n != 0:
                         p = euclid_gcd (x + y,
n)
                         q = euclid_gcd (x - y,
n)
                         return p
               k += 2
          x += 1

def getAllFactorsWithKraitchik(n):
     all_factors = []
     while n % 2 == 0:
          all_factors.append(2)
          n = n // 2
     s = 1
     while not(isFermatPrime(n)):
          s = Kraitchik(n)
          if isFermatPrime(s):
               all_factors.append(s)
          else:
               all_factors.extend(
getAllFactorsWithKraitchik (s) )
          n = n // s
          #print all_factors
          all_factors.append(n)
          return all_factors

#test
n = 837233419

print "Factoring RPrime RSA with Kraitchik's
Factorization Algorithm\n"


print "n =", n
print
print "Calculating..."

start = time.time()
fs = getAllFactorsWithKraitchik (n)
stop = time.time()
print
print "factors of ", n, "are", fs
print
print "time =", stop - start, "secs"
print
```

7. THE RESULTS OF FACTORING RPRIME RSA USING FERMAT'S DIFFERENCE OF SQUARE

The results of factoring RPrime RSA modulus using Fermat's difference of squares is shown in Table 1 and the relation between modulus and factoring time is depicted in Figure 1 as follows.

Table 1. Factoring RPrime RSA modulus using Fermat's difference of square

RPrime RSA Modulus	The factors					Factoring time
n	p	q	r	s	t	time (seconds)
44745833	53	71	47	11	23	0.002947092
114070363	53	181	47	11	23	0.000624895
837233419	47	389	181	11	23	0.007280111
3544881923	199	11	23	181	389	0.002234936
77525026403	389	503	199	11	181	0.037840128

1148779936699	389	503	199	163	181	3.65061903
2633286813149	389	1153	199	163	181	3.03720808
13653829054297	1153	2017	199	163	181	0.787917852
171464936134901	2017	2273	1153	163	199	14.90988517

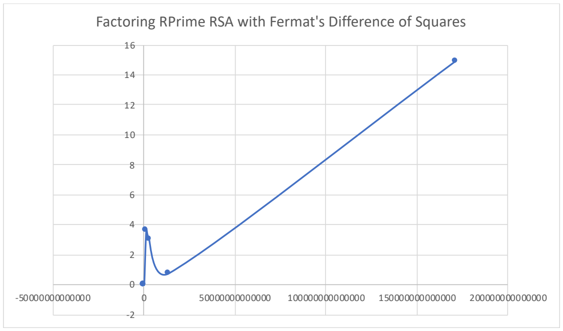

Figure 1 Factoring RPrime RSA with Fermat's Difference of Squares
(the x axis is the modulus and the y axis is the factoring time in seconds)

In Table 1, small RPrime RSA moduli are factored using Fermat's difference of squares, with $n = p \times q \times r \times s \times t$. The n ranges between 44745833 (eight digits) and 171464936134901 (fifteen digits), with p, q, r, s, t range between two digits and four digits.

It can be seen from Table 1 and Figure 1 that the time the algorithm takes to factorize is mostly directly proportional to the size of n

8. THE RESULTS OF FACTORING RPRIME RSA USING KRAITCHIK'S ALGORITHM

The results of factoring RPrime RSA modulus using Kraitchick's algorithm is shown in Table 2 and the relation between modulus and factoring time is depicted in Figure 2 as follows.

Table 2. Factoring RPrime RSA modulus using Kraitchick's Algorithm

RPrime RSA Modulus	The factors					Factoring time
n	p	q	r	s	t	time (seconds)
44745833	53	71	47	11	23	0.07245898
114070363	53	181	47	11	23	3.73909402
837233419	47	389	181	11	23	3.51274085
3544881923	199	11	23	181	389	103.12642217
77525026403	389	503	199	11	181	1240.25034404

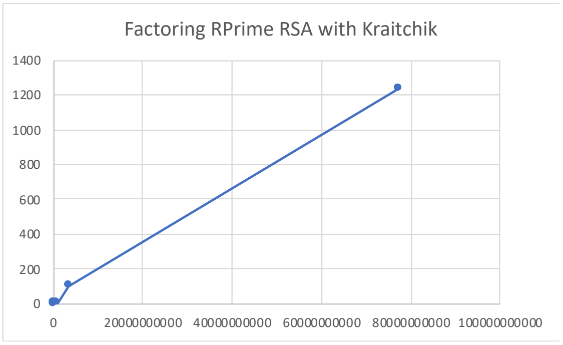

Figure 2 Factoring RPrime RSA with Kraitchik algorithm
(the x axis is the modulus and the y axis is the factoring time in seconds)

In Table 2, small RPrime RSA modulus are factored using Kraitchik, with $n = p \times q \times r \times s \times t$. The n ranges between 44745833 (eight digits) and 77525026403 (eleven digits), with p, q, r, s, t range between two digits and three digits.

It can be concluded from Table 2 and Figure 2 that the time this algorithm takes to factorize is also mostly directly proportional to the size of n

9. THE COMPARISON OF RESULTS

The Fermat's difference of squares and the Kraitchik's algorithms were primarily designed by their authors (Pierre de Fermat in the 1600s and Maurice Kraitchik in 1920s, correspondingly) to factor an integer to its two factors. These two factors may be both prime numbers, both composite numbers, or a mixed of a prime number and a composite number. In contrast, our Python codes can be used to factor RPrime RSA modulus whose factors are two or more prime numbers.

Although it seems that Kraitchik does a fair job factoring the modulus, it should be noted that the performance of Kraitchik's algorithm is worse than that of Fermat's difference of squares. The explanation is below.

When factoring n = 44745833, Fermat's difference of squares only needs 0.002947092 seconds, while Kratchik's algorithm needs 0.07245898 seconds. Thus, here Fermat's is about 24 times faster than Kraitchik's.

When factoring n = 77525026403, Fermat's difference of squares only needs 0.002234936 seconds, while Kraitchik's algorithm needs 1240.25034404 seconds. Therefore, here Fermat's is about 550 times faster than Kraitchik's.

In this study, Fermat's is used to factor up to n = 171464936134901, and it needs 14.90988517 seconds to find all the five factors. Meanwhile, Kraitchik is only used to factor up to n = 77525026403, since here it already needs more than one thousand of seconds to compute all the factors.

10. CONCLUSION

Our major findings are as follows:

1. Two Python codes based on the Fermat's difference of squares and the Kraitchik's algorithms were developed to factor the modulus of RPrime RSA. The Fermat's difference of squares and the Kraitchik's algorithms were originally introduced by their corresponding inventors to factor an integer to its two factors that can be prime or composite numbers; meanwhile, the two Python codes are able to handle integers that have more than two factors and these factors are guaranteed to be prime numbers. Therefore, these codes can factor the modulus RPrime RSA, since the modulus are products of two or more prime numbers.
2. Both the Fermat's difference of squares and the Kraitchik's algorithms can be used as methods to factorize small n, the modulus of RPrime RSA.
3. Based on the experiments using $n = p \times q \times r \times s \times t$ (with p, q, r, s, t ranges between two and four digits and n ranges from eight and fifteen digits), the time both algorithms take to factorize is mostly directly proportional to the size of n.
4. By comparison, Fermat's difference of squares is much faster than Kraitchik's algorithm: while factoring six digits to eleven digits of n, Fermat's is about 24 to 550 times faster than Kraitchik.

ACKNOWLEDGEMENTS

We gratefully acknowledge that this research is funded by Lembaga Penelitian Universitas Sumatera Utara. The support is under the research grant DRPM Kemenristekdikti of Year 2019 Contract Number: 222/UN5.2.3.1/PPM/KP-DRPM/2019.

REFERENCES

[1] Rachmawati, Dian, and Mohammad Andri Budiman. "Using the RSA as as an asymmetric non-public key encryption algorithm in the Shamir three-pass protocol." *Journal of Theoretical & Applied Information Technology* 96, no. 17 (2018).

[2] Diffie, Whitfield, and Martin Hellman. "New directions in cryptography." *IEEE transactions on Information Theory* 22, no. 6 (1976): 644-654.

[3] Rivest, Ronald L., Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems." *Communications of the ACM* 21, no. 2 (1978): 120-126.

[4] Paixao, Cesar Alison Monteiro, and Decio Luiz Gazzoni Filho. "An efficient variant of the RSA cryptosystem." *IACR Cryptology ePrint Archive 2003* (2003): 159.

[5] Boneh, Dan, and Hovav Shacham. "Fast variants of RSA." *CryptoBytes* 5, no. 1 (2002): 1-9.

[6] Budiman, M. A., D. Rachmawati, and R. Utami. "The cryptanalysis of the Rabin public key algorithm using the Fermat factorization method." *Journal of Physics: Conference Series*, vol. 1235, no. 1, p. 012084. IOP Publishing, 2019.

[7] Somsuk, Kritsanapong. "The improvement of initial value closer to the target for Fermat's factorization algorithm." *Journal of Discrete Mathematical Sciences and Cryptography* 21, no. 7-8 (2018): 1573-1580.

[8] Pomerance, Carl. "The quadratic sieve factoring algorithm." In *Workshop on the*

Theory and Application of Cryptographic Techniques, pp. 169-182. Springer, Berlin, Heidelberg, 1984.

[9] Collins, Thomas, Dale Hopkins, Susan Langford, and Michael Sabin. “Public key cryptographic apparatus and method.” U.S. Patent 5,848,159, issued December 8, 1998.

[10] Rosenthal, Daniel, David Rosenthal, and Peter Rosenthal. “Fermat’s Little Theorem and Wilson’s Theorem.” In *A Readable Introduction to Real Mathematics*, pp. 37-42. Springer, Cham, 2018.

[11] Kim, Hyun Jong. “On a Modification of the Agrawal-Biswas Primality Test.” *arXiv preprint arXiv*:1810.09651 (2018).

[12] Yu, Gavin. "Proof Methods, Computational Algorithms and Applications of the Chinese Remainder Theorem." *Global Journal of Pure and Applied Mathematics* 15, no. 5 (2019): 667-673.

[13] Rosenthal, Daniel, David Rosenthal, and Peter Rosenthal. “The Euclidean Algorithm and Applications.” In *A Readable Introduction to Real Mathematics*, pp. 49-61. Springer, Cham, 2018.

[14] Kraft, James, and Lawrence Washington. *An introduction to number theory with cryptography*. Chapman and Hall/CRC, 2018.

[15] Pieprzyk, Josef. "Integer Factorization–Cryptology Meets Number Theory." *Scientific Journal of Gdynia Maritime University* (2019).

[16] Lenstra, Arjen K. "General purpose integer factoring." IACR Cryptol. ePrint Arch. 2017 (2017): 1087.

[17] Garrett, S. L. (2008). On the quadratic sieve. The University of North Carolina at Greensboro.

[18] Nguyen, S. T., Ghebregiorgish, S. T., Alabbasi, N., & Rong, C. (2011, November). Integer factorization using Hadoop. In 2011 IEEE Third International Conference on Cloud Computing Technology and Science (pp. 628-633). IEEE.

[19] Buchanan, W., & Woodward, A. (2017). Will quantum computers be the end of public key encryption?. Journal of Cyber Security Technology, 1(1), 1-22.

[20] Bressoud, D. M. (2012). Factorization and primality testing. Springer Science & Business Media.