On factoring RSA modulus using random-restart hill-climbing algorithm and Pollard’s rho algorithm

M A Budiman and D Rachmawati
Departemen Ilmu Komputer, Fakultas Ilmu Komputer dan Teknologi Informasi, Universitas Sumatera Utara, Jl. Universitas No. 9-A, Kampus USU, Medan 20155, Indonesia
E-mail: mandrib@usu.ac.id

Abstract. The security of the widely-used RSA public key cryptography algorithm depends on the difficulty of factoring a big integer into two large prime numbers. For many years, the integer factorization problem has been intensively and extensively studied in the field of number theory. As a result, a lot of deterministic algorithms such as Euler’s algorithm, Kraitchik’s, and variants of Pollard’s algorithms have been researched comprehensively. Our study takes a rather uncommon approach: rather than making use of intensive number theories, we attempt to factorize RSA modulus \( n \) by using random-restart hill-climbing algorithm, which belongs the class of metaheuristic algorithms. The factorization time of RSA moduli with different lengths is recorded and compared with the factorization time of Pollard’s rho algorithm, which is a deterministic algorithm. Our experimental results indicates that while random-restart hill-climbing algorithm is an acceptable candidate to factorize smaller RSA moduli, the factorization speed is much slower than that of Pollard’s rho algorithm.

1. Introduction
RSA is one of the first algorithms that employs Diffie-Hellman’s concept of asymmetrical cryptography [1] and has, until now, been one of the most widely used public key crypto-system since its publication in 1978 [2]. There are a considerable number of variants of RSA [3] such as Batch RSA [4], Multi-power RSA [5], Rebalanced RSA [6], RSA-CRT [7], and R-Prime RSA [8].

The original RSA and all of its variants have at least one thing in common: their security depends on the hardness of factoring a large integer into its prime factors. Although Boneh and Venkatesan [9] have pointed out that breaking RSA may be easier than factoring, they have also highlighted that this does not lead to any weakness of the RSA crypto-system. In fact, [10] has stated that there are no obliterating attacks on RSA crypto-system have been found. It should be noted, nevertheless, that RSA crypto-system may be vulnerable if the nowadays’ hypothetical quantum computers can eventually become real in the future [11].

A fair number of deterministic algorithms to factorize integers into their prime factors have been researched in the field of number theory, such as Fermat’s, Euler’s, Kraitchik’s, and some variants of Pollard’s algorithms [12]. Meanwhile, there are currently lack of researches regarding the use of metaheuristic algorithms to factorize integers. As argued by Siarry [13], metaheuristic algorithms can be applied to all kinds of discrete problems and can be adapted to continuous problems. Some
examples of metaheuristic algorithms are tabu search, hill-climbing, simulated annealing, and ant colony optimization algorithms.

The purpose of this work is to give some suggestions on how efficient metaheuristic algorithms can be used to factorize \( n \), i.e., the RSA public key. Here we test random-restart hill-climbing algorithm, which belongs to the class of metaheuristic algorithms, to factor small to large values of \( n \). Random-restart hill-climbing algorithm is an improvement of hill-climbing algorithm, in which hill-climbing are repeated with random initial candidate solutions that are distant from each other in order to escape from local optimum solution. As a benchmark, we also work with a deterministic integer factorization algorithm called Pollard’s rho algorithm [14] in order to determine which algorithm is faster to factorize \( n \).

2. Methods
In this section we explain the RSA computation, the hill-climbing and the random-restart hill-climbing techniques, and the Pollard’s rho algorithm. As an addition, we also provide our Python code for the Pollard’s rho algorithm.

2.1 RSA
The RSA public key cryptography algorithm has three stages: key generation, encryption, and decryption. Suppose we have two parties that want to communicate, which are a sender called “Alice” and a recipient called “Bob”. The computation involved in each stage, which can be found in [2] and [15], are explained below.

In the key generation stage, Bob will do the following:
1. Randomly generate two very large prime numbers, \( p \) and \( q \), such that \( p \neq q \). The values of \( p \) and \( q \) are Bob’s private keys and, therefore, should be kept secret.
2. Compute \( n = pq \). The value of \( n \) is Bob’s public key and should be published so anyone who wants to send a message to Bob knows it.
3. Compute \( \Phi(n) = (p – 1)(q – 1) \). The value of \( \Phi(n) \) should also be kept secret.
4. Randomly select a value of \( e \), such that \( 1 < e < \Phi(n) \) and \( \gcd(\Phi(n), e) = 1 \). The value of \( e \) will be used in the encryption stage by Alice. Therefore, \( e \) should be published.
5. Compute \( d \), such that \( de \equiv 1 \pmod{\Phi(n)} \). The value of \( d \) should be kept private. It will be used in the decryption stage by Bob.

In the encryption stage, assuming that Alice has a message \( m \), in this stage, she will do the following:
1. Get Bob’s public keys, \( n \) and \( e \).
2. Compute the cipher-text, \( c = m^e \mod n \).
3. Send \( c \) to Bob.

In the decryption stage, Bob will do the following:
1. Receive \( c \) from Alice.
2. Compute \( m = c^d \mod n \). Thus, Bob can now read the message from Alice.

2.2 Hill-climbing algorithm and random-restart hill-climbing algorithm
Hill-climbing is a simple metaheuristic optimization algorithm that works as follows [16].
1. Take a candidate solution and test the quality.
2. Generate a new random and slightly different candidate solution and test the quality.
3. If the a quality of the new candidate solution is better than the quality of the previous candidate solution, then this new candidate solution becomes the candidate solution.
4. Repeat step 2 and 3 until time is up or the ideal solution is found.

The second step (generating a new candidate solution that is only slightly different from the candidate solution is called ‘tweak’ or ‘walk’). Since hill-climbing only takes a slightly different new candidate solution in each trial, it often only leads to a local optimum solution.
To minimize the probability of getting only a local optimum solution, hill-climbing is repeated for some random amounts of time. When the time is up, a new random solution that is more distant from the previous best solution will be generated, and hill-climbing is done again for some random amounts of time, and so on until the ideal solution is found. This whole technique is known as random-restart hill-climbing [16].

The step of generating a new random solution that is more distant from the previous best solution is called “explore”. Doing the ‘exploration’ more than once is expected to maximize the chance of getting the global optimum solution, since there are many ‘hills’ to be explored to see whether or not the currently explored hill is better than the previously explored hills.

2.3 Pollard’s rho algorithm

Pollard’s rho algorithm is a method to factorize an odd integer. It works as follows [12].
1. Input \( n \) (the odd integer to be factored).
2. Select a polynomial with degree \( \geq 2 \) and with all coefficients modulo \( n \). In general, one may simply use \( f(x) \equiv x^2 + 1 \mod n \).
3. Select a random value \( x_0 \). In general, simply use \( x_0 = 2 \).
4. Set \( i = 0 \), \( x_i = x_0 \), \( y_i = x_0 \), and \( d = 1 \).
5. Compute \( x_i = f(x_{i-1}) \) and \( y_i = f(f(y_{i-1})) \).
6. Compute \( d = \gcd(\text{abs}(x_i - y_i)) \).
7. Set \( i = i + 1 \).
8. If \( d = 1 \), go to step 5. If not, go to step 9.
9. If \( d = n \), then output failure or go back to step 2 to choose a different polynomial. If not, output \( d \) as one factor of \( n \).

If \( n \) is RSA modulus and if in the last step \( d \neq n \), then the value of \( d \) is either the value of \( p \) or \( q \).

Our Python code of Pollard’s rho algorithm is as follows.

```python
#title: Factoring RSA Modulus with Pollard’s Rho Algorithm
#author: Mohammad Andri Budiman
#version: 1.01
#date: October 1st 2017
#time: 16:20

import time

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

def f(x, n):
    return (x * x + 1) % n

def PollardRho(n):
    iterations = 0
    x = 2
    y = 2
    d = 1
    start = time.time()
    while d == 1:
        x = f(x, n)
        y = f(f(y, n), n)
        d = gcd(abs(x - y), n)
        iterations += 1
    stop = time.time()
    if d == n:
```
print "failure"
  else:
    print n, "\times", d, "\times", n // d
print "iterations =", iterations
print "time =", stop - start, "secs"

3. Results and Discussions
In the case of factoring the RSA modulus, we set the random-restart hill-climbing to work as follows. Two candidate solutions are generated in each trial. These two candidate solutions are random prime numbers \( p \) and \( q \) that are generated using Fermat’s little theorem [15]. When testing the quality of these new candidate solutions, we actually multiply the \( p \) and the \( q \), and check whether or not the result of this multiplication is close enough to the value of \( n \). If it is close enough, then the ‘walk’ will be done for 10 times, but if it is not close enough, then in the next exploration, the new random values of \( p \) and \( q \) that are distant from the previous best \( p \) and \( q \) values will be generated.

Since RSA modulus is an exact number, then the only ideal solution that can be accepted is when \( pq = n \). If the time limit is reached and \( pq \) is still not equal to \( n \), then we let the algorithm end without a result.

The results of factoring RSA moduli with random-restart hill-climbing are shown in Figure 1-4.

![Figure 1. Factoring n = 989 with random-restart hill-climbing algorithm](image)

It can be seen from Figure 1, that only one red dot (which means one exploration) and four blue dots (which mean four walks) are needed to factor \( n = 989 = 23 \times 43 \). The running time needed is 0.00448393821716 secs.
Figure 2. Factoring $n = 13289$ with random-restart hill-climbing algorithm

It can be seen from Figure 2 that 16 explorations and more than 160 walks are needed to factor $n = 13289 = 137 \times 97$. The running time needed is 0.00215601921082 second. Note that the time needed to factor $n = 13289$ is lower than the time needed to factor $n = 989$. Therefore, the correlation between $n$ and running time may not be directly proportional.
Figure 3. Factoring $n = 70531$ with random-restart hill-climbing algorithm

It can be seen from Figure 3, that 6 explorations and more than 300 walks are needed to factor $n = 70531 = 251 * 281$. The running time needed is $0.0644018650055$ second. Note that the number of explorations to factor $n = 70531$ is lower than the number of explorations needed to factor $n = 13289$. Therefore, the correlation between $n$ and the number of explorations may not be directly proportional.
Figure 4. Factoring $n = 194111$ with random-restart hill-climbing algorithm

It can be seen from Figure 4 that 6 explorations and more than 250 walks are needed to factor $n = 194111 = 389 \times 499$. The running time needed is 0.000250101089478 second. Note that the number of walks needed to factor $n = 194111$ is lower than the number of explorations needed to factor $n = 70531$. More surprisingly, the time needed to factor $n = 194111$ is lower than the time needed to factor $n = 70531$. Therefore, the correlation between $n$ and the number of walks may not be directly proportional, and the correlation between $n$ and running time may also not be directly proportional.

We also try to use random-restart hill-climbing algorithm to factor a very large value of RSA modulus ($n = 1356992669$), but the algorithm fails to find the corresponding $p$ and $q$ within 20 seconds, so the program automatically ends.

In Table 1, we show the results of factoring RSA moduli with Pollard’s rho algorithm.

| $n$          | $p \times q$ | iterations | time (seconds) |
|--------------|--------------|------------|----------------|
| 989          | 23 \times 43 | 4          | 0.00002.19345092773 |
| 13289        | 97 \times 137 | 3          | 0.0000181198120117 |
| 70531        | 281 \times 251 | 8          | 0.0000488758087158 |
| 194111       | 389 \times 499 | 13         | 0.0000498294830322 |
| 1356992669   | 24691 \times 54959 | 215       | 0.000957012176514 |

From Table 1, it can be seen that the larger the value $n$, the number of iterations and the running time needed to factor $n$ tend to increase.

| $n$          | Random- Restart Hill-Climbing (secs) | Pollard’s rho (secs) |
|--------------|--------------------------------------|----------------------|
| 989          | 0.00448393821716                    | 0.00002.19345092773  |
| 13289        | 0.00215601921082                    | 0.0000181198120117   |
| 70531        | 0.06440.18650055                    | 0.0000488758087158    |

Table 1. Factoring RSA moduli with Pollards’s rho algorithm

Table 2. Time comparison between random-restart hill-climbing and Pollard’s rho

Figure 4. Factoring $n = 194111$ with random-restart hill-climbing algorithm

It can be seen from Figure 4 that 6 explorations and more than 250 walks are needed to factor $n = 194111 = 389 \times 499$. The running time needed is 0.000250101089478 second. Note that the number of walks needed to factor $n = 194111$ is lower than the number of explorations needed to factor $n = 70531$. More surprisingly, the time needed to factor $n = 194111$ is lower than the time needed to factor $n = 70531$. Therefore, the correlation between $n$ and the number of walks may not be directly proportional, and the correlation between $n$ and running time may also not be directly proportional.

We also try to use random-restart hill-climbing algorithm to factor a very large value of RSA modulus ($n = 1356992669$), but the algorithm fails to find the corresponding $p$ and $q$ within 20 seconds, so the program automatically ends.

In Table 1, we show the results of factoring RSA moduli with Pollard’s rho algorithm.

| $n$          | $p \times q$ | iterations | time (seconds) |
|--------------|--------------|------------|----------------|
| 989          | 23 \times 43 | 4          | 0.00002.19345092773 |
| 13289        | 97 \times 137 | 3          | 0.0000181198120117 |
| 70531        | 281 \times 251 | 8          | 0.0000488758087158 |
| 194111       | 389 \times 499 | 13         | 0.0000498294830322 |
| 1356992669   | 24691 \times 54959 | 215       | 0.000957012176514 |

From Table 1, it can be seen that the larger the value $n$, the number of iterations and the running time needed to factor $n$ tend to increase.

| $n$          | Random- Restart Hill-Climbing (secs) | Pollard’s rho (secs) |
|--------------|--------------------------------------|----------------------|
| 989          | 0.00448393821716                    | 0.00002.19345092773  |
| 13289        | 0.00215601921082                    | 0.0000181198120117   |
| 70531        | 0.06440.18650055                    | 0.0000488758087158    |

Table 2. Time comparison between random-restart hill-climbing and Pollard’s rho

Figure 4. Factoring $n = 194111$ with random-restart hill-climbing algorithm

It can be seen from Figure 4 that 6 explorations and more than 250 walks are needed to factor $n = 194111 = 389 \times 499$. The running time needed is 0.000250101089478 second. Note that the number of walks needed to factor $n = 194111$ is lower than the number of explorations needed to factor $n = 70531$. More surprisingly, the time needed to factor $n = 194111$ is lower than the time needed to factor $n = 70531$. Therefore, the correlation between $n$ and the number of walks may not be directly proportional, and the correlation between $n$ and running time may also not be directly proportional.

We also try to use random-restart hill-climbing algorithm to factor a very large value of RSA modulus ($n = 1356992669$), but the algorithm fails to find the corresponding $p$ and $q$ within 20 seconds, so the program automatically ends.

In Table 1, we show the results of factoring RSA moduli with Pollard’s rho algorithm.

| $n$          | $p \times q$ | iterations | time (seconds) |
|--------------|--------------|------------|----------------|
| 989          | 23 \times 43 | 4          | 0.00002.19345092773 |
| 13289        | 97 \times 137 | 3          | 0.0000181198120117 |
| 70531        | 281 \times 251 | 8          | 0.0000488758087158 |
| 194111       | 389 \times 499 | 13         | 0.0000498294830322 |
| 1356992669   | 24691 \times 54959 | 215       | 0.000957012176514 |

From Table 1, it can be seen that the larger the value $n$, the number of iterations and the running time needed to factor $n$ tend to increase.

| $n$          | Random- Restart Hill-Climbing (secs) | Pollard’s rho (secs) |
|--------------|--------------------------------------|----------------------|
| 989          | 0.00448393821716                    | 0.00002.19345092773  |
| 13289        | 0.00215601921082                    | 0.0000181198120117   |
| 70531        | 0.06440.18650055                    | 0.0000488758087158    |
From Table 2, it can be seen that in RSA modulus factorization, the Pollard’s rho algorithm is much faster than random-restart hill-climbing algorithm.

4. Conclusions
The conclusions of this work are as follows. Firstly, in random-restart hill-climbing algorithm, the correlation between RSA modulus $n$ and the number of explorations, the number of walks, or running time is not directly proportional. Secondly, in Pollard’s rho algorithm, the correlation between RSA modulus $n$ and the number of iterations or running time tends to be directly proportional. Thirdly, random-restart hill-climbing algorithm is only acceptable to factor smaller values of $n$, while Pollard’s rho algorithm is suitable to factor larger values of $n$. Fourthly, Pollard’s rho algorithm is much faster and more reliable than random-restart hill-climbing to factor $n$.

Acknowledgments
We gratefully acknowledge that this research is funded by Lembaga Penelitian Universitas Sumatera Utara. The support is under the research grant TALENTA USU of Year 2017 Contract Number : 5338/UN5.1/R/PPM/2017.

References
[1] Diffie W and Hellman M 1976 New directions in cryptography IEEE Transactions on Information Theory 22 644-54
[2] Rivest R L, Shamir A and Adleman L 1978 A method for obtaining digital signatures and public-key cryptosystems,” Communications of the ACM 21 120-26
[3] Boneh D and Shacham H 2002 Fast variants of RSA CryptoBytes 5 1-9
[4] Fiat A 1997 Batch RSA Journal of Cryptology 10 75–88
[5] Takagi T 1998 Fast RSA-type cryptosystem modulo $p^k$q Advances in Cryptology — CRYPTO 98 Lecture Notes in Computer Science 318–26
[6] Wiener M J 1990 Cryptanalysis of short RSA secret exponents IEEE Transactions on Information Theory 36 553-8
[7] Quisquater J J and Couvreur C 1982 Fast decipherment algorithm for RSA public-key cryptosystem Electronics Letters 18 905-07
[8] Paixao C A M and Filho D L G 2003 An efficient variant of the RSA cryptosystem IACR Cryptology ePrint Archive
[9] Boneh D and Venkatesan R 1998 Breaking RSA may not be equivalent to factoring Advances in Cryptology – EUROCRYPT ’98 1233 59–71
[10] Boneh D 1999 Twenty years of attacks on the RSA cryptosystem Notices of the AMS 46 203-13
[11] Kiviiharju M 2017 On the fog of RSA key lengths: Verifying public key cryptography strength recommendations International Conference on Military Communications and Information Systems (ICMCIS) IEEE
[12] Batten L M 2013 Public key cryptography applications and attacks (Hoboken, N.J: Wiley-Blackwell)
[13] Siarry P 2016 Metaheuristics (Cham: Springer)
[14] Sarnika S, Bhakad R and Desai C 2015 Comparative study on integer factorization algorithm-Pollard's rho and Pollard's P-1 2nd International Conference on Computing for Sustainable Global Development (INDIACom) IEEE 677-79
[15] Mollin R A 2007 An introduction to cryptography (Boca Raton, FL: Chapman & Hall/CRC)
[16] Luke S 2013 Essentials of Metaheuristics 2nd ed (Lulu) available at http://cs. gmu. edu/~sean/book/metaheuristics/