Journal of Computing and Applied Informatics (JoCAI) Vol. 02, No. 1, 2018 | 45 – 52

talenta PUBLISHER

DATA SCIENCE

Journal of Computing and Applied Informatics

Using Random Search and Brute Force Algorithm in Factoring the RSA Modulus

Mohammad Andri Budiman[1] and Dian Rachmawati[2]

[1,2]Departemen Ilmu Komputer, Fakultas Ilmu Komputer dan Teknologi Informasi, Universitas Sumatera Utara, Jl. Universitas No. 9-A, Kampus USU, Medan 20155, Indonesia

Abstract. The security of the RSA cryptosystem is directly proportional to the size of its modulus, n. The modulus n is a multiplication of two very large prime numbers, notated as p and q. Since modulus n is public, a cryptanalyst can use factorization algorithms such as Euler's and Pollard's algorithms to derive the private keys, p and q. Brute force is an algorithm that searches a solution to a problem by generating all the possible candidate solutions and testing those candidates one by one in order to get the most relevant solution. Random search is a numerical optimization algorithm that starts its search by generating one candidate solution randomly and iteratively compares it with other random candidate solution in order to get the most suitable solution. This work aims to compare the performance of brute force algorithm and random search in factoring the RSA modulus into its two prime factors by experimental means in Python programming language. The primality test is done by Fermat algorithm and the sieve of Eratosthenes.

Keyword: RSA Modulus, Factorization, Random Search, Brute Force, Primality Test.

Abstrak. *Tingkat keamanan sistem kunci publik RSA sangat tergantung pada besar modulusnya, yaitu n. Semakin besar n, maka tingkat keamanan RSA semakin tinggi. Modulus n adalah hasil perkalian dari dua buah bilangan prima yang sangat besar yang bersifat privat, yang dinotasikan sebagai p dan q. Karena bersifat publik, modulus n dapat diketahui oleh seorang kriptanalis, dan dengan menggunakan beberapa metode faktorisasi bilangan bulat seperti metode faktorisasi Euler dan Pollard, ia dapat mengetahui nilai kunci privat p dan q tersebut. Brute force adalah sebuah algoritma yang mencari solusi suatu permasalahan dengan dengan membangkitkan seluruh kandidat solusi yang mungkin dan menguji kandidat-kandidat solusi tersebut satu per satu untuk mengetahui yang mana yang menghasilkan solusi yang paling relevan. Random search adalah sebuah algoritma optimasi numerik yang memulai pencariannya dengan membangkitkan secara acak sebuah kandidat solusi, dan secara iteratif membandingkannya dengan kandidat-kandidat solusi acak yang lain untuk mengetahui yang mana yang paling mendekati solusi yang diinginkan. Penelitian ini bertujuan untuk membandingkan performansi kedua algoritma tersebut dalam memfaktorkan modulus RSA menjadi dua buah faktor primanya dengan menggunakan pendekatan eksperimental dalam bahasa pemrograman Python. Uji keprimaan dilakukan dengan algoritma Fermat dan algoritma saringan Eratosthenes*.

Kata Kunci: *Modulus RSA, Faktorisasi, Random Search, Brute Force, Uji Keprimaan.*

Received 15 January 2018 | Revised 20 January 2018 | Accepted 28 January 2018

*Corresponding author at: Departemen Ilmu Komputer, Fakultas Ilmu Komputer dan Teknologi Informasi, Universitas Sumatera Utara, Jl. Universitas No. 9-A, Kampus USU, Medan 20155, Indonesia

E-mail address: mandrib@usu.ac.id

Journal Homepage: https://talenta.usu.ac.id/JoCAI

1. Introduction

The RSA cryptosystem was proposed by Ronald Rivest, Adi Shamir, and Leonard Adleman in 1978 [1]. It was amongst the first cryptosystems that implements the Diffie-Hellman key exchange protocol [2]. The Diffie-Hellman key exchange protocol enables a sender of a message (plaintext) to send the secret version of that message (ciphertext) via a channel without sending any key (which is used to decrypt the ciphertext) via another channel. This is done by publishing the key that is used to encrypt the message in a server or any other electronic means. Meanwhile, the key that is used to decrypt the ciphertext is kept private. Both keys are generated by the recipient of the message. The key that is used in the encryption process is called public key and the key that is used in the decryption process is called private key.

The RSA has two public keys, which are n and e. n is the RSA modulus and e is the RSA exponent. The RSA has four private keys, which are p, q, d, and $\Phi(n)$. The modulus n has a relation with p and q, i.e., $n = pq$. p and q are very large prime numbers. Since n is published, anyone, including cryptanalysts, can use some integer factorization algorithms to get the value of p and q. If any cryptanalyst can obtain the value of p and q, the whole RSA cryptosystem may be compromised.

The problem of factoring an integer is considered hard on a classical computer [3]. A lot of computational efforts are needed to do this, and the larger the integer to be factored, the time increases exponentially [4]. Shor [3] has pointed out that integer factorization can be done in polynomial time if quantum computer exists in the future. However, twenty years after Shor published his paper, quantum computers are still under development and are not yet ready for common use. Therefore, nowadays, a cryptanalyst can only factorize the RSA modulus in exponential time using exact factorization algorithms such as Fermat's difference of squares, Euler's, Brent's, Kraitchik's, or Pollard's algorithms.

Rather than using the exact factorization algorithms as mentioned above, this study uses a metaheuristic called random search to factor the RSA modulus. Metaheuristic is usually a stochastic procedure [5], which is oftentimes nature-inspired [6]. It is devised to solve hard optimization problems. Random search is the simplest form of metaheuristics [7]: it tries some random candidate solutions until the best solution is found or until the allowed time is up. In order to gain more perspectives, the performance of the random search is compared with the performance of brute force algorithm. Brute force algorithm is an exhaustive search algorithm that checks every possible candidate solutions one by one until it found the best solution regardless of time. This work is performed in Python programming language. The primality testing is done with the sieve of Eratosthenes and Fermat's little theorem.

2. Method

In order to format a paragraph, authors (writer) can click on the appropriate style name in the style toolbar. In this part, we describe the RSA cryptosystem, the Fermat's little theorem, the sieve of Eratosthenes, the brute force algorithm as well as the random search to factor the modulus, and give the main part of the Python code that we use to implement the algorithms.

2.1. RSA cryptosystem

RSA cryptosystem can be used to protect confidentiality with its encryption scheme and also to ensure authentication with its digital signature scheme. In this work, we are only interested in the encryption scheme. There are three main parts of the RSA encryption which are as key generation, encryption, and decryption which are explained as follows [1].

Key generation part is done by the recipient of the message. The steps are as follows.

1. Two large distinct prime numbers, named p and q are generated. This can be done by generating two large odd integers, and test them with primality test algorithm such as Fermat's little theorem.
2. Compute the public key, n = pq.
3. Compute the totient, $\Phi(n) = (p - 1)(q-1)$.
4. Choose an odd integer e, so that $\gcd(\Phi(n), e) = 1$ and $1 < e < \Phi(n)$.
5. Compute the private key, d, so that $de \equiv 1 \pmod{\Phi(n)}$. This can be done by testing the value of d = 1 to $d = \Phi(n)$ that satisfies the equation or can also be done much faster by extended Euclidean algorithm. The readers are invited to read [8] and [9].
6. The value of e and n are published, and the other remaining values are kept secret.

The encryption part is done by the sender of the message. The steps are the followings.

1. Let the message to be sent is m.
2. Obtain e and n.
3. Encrypt message m into ciphertext c, by computing $c = m^e \bmod n$.
4. Send c to the recipient.

The decryption part is done by the recipient. The steps are as follows.

1. Obtain c.
2. Decrypt the ciphertext c into message m, by computing $m = c^d \bmod n$.

In this work, we are interested in factoring n into p and q by using brute force algorithm and random search.

2.2. The Fermat's little theorem

The Fermat's little theorem is used to test whether or not an odd integer is a prime or a composite. The theorem states that if p is a prime, then $F = a^{p-1} \bmod p = 1$ for $1 < a < p$, where 'a' and 'p' are

integers and p is odd. [10]. Value of 'a' should be tested with many distinct values to avoid Fermat's liars [11]. A Fermat liar is a value of 'a' that causes F = 1, even though p is a composite. Our Python code for the Fermat's little theorem is as follows.

```
def Fermat(p):
      trial = len(str(p)) * 3
      for i in range(trial):
            if pow(rnd(2, p - 1), p - 1, p) != 1:
                  return False
      return True
```

In this code, we try the Fermat's little theorem with some values of a three times the digits of p. If, for example, p = 2333, then the digits of p is 4, so we try the Fermat's little theorem 4 * 3 = 12 values of a. If all a's result in F = 1, then we conclude that p is prime.

2.3. The sieve of Eratosthenes

The sieve of Eratosthenes is an ancient algorithm to generate the table of prime numbers from 2 to up to a specific bound [12]. The algorithm works by assuming all the numbers from 2 to that bound to be primes, and then 'sieving' the composites by striking out (or eliminating) the multiples of the first primes up until the squares root of the bound. Our Python code for the sieve of Eratosthenes is as follow.

```
def GeneratePrimeTable(start, stop):
      isPrime = {}
      isPrime[1] = False
      for i in range(2, stop + 1):
            isPrime[i] = True
      for i in range(2, int(math.ceil(math.sqrt(stop)) + 1)):
            if isPrime[i]:
                  for j in range(i, int(math.ceil((stop)/i) + 1)):
                        isPrime[i * j] = False
      primes = []
      count = 0
      for i in range(start, stop + 1):
            if isPrime[i]:
                  count += 1
                  primes.append(i)
      return primes
```

The function GeneratePrimeTable(start, stop) tabulates all prime numbers from start to stop with the sieve of Eratosthenes and returns them as a set.

2.4. The brute force algorithm

Brute force is an approach which searches a solution to a problem by generating and testing all the possible candidate solutions. The most relevant candidate solution is then returned. Because this algorithm generates and tests all of the possible candidate solution, the brute force algorithm is also called exhaustive search or generate and test algorithm. Its time performance is the lowest of all other algorithms, but it is guaranteed to find the most relevant solution for a discrete

problem, and therefore, brute force is often used as a baseline [13] to compare other algorithms. Our brute force implementation in Python is as follows.

```
def BruteForce(n):
      #get RSA private keys with brute force algorithm
      primes = GeneratePrimeTable(1, int(math.ceil(math.sqrt(n))))
      for p in primes:
            if n % p == 0:
                  return p, n // p
```

The function BruteForce (n) takes n (the RSA modulus), and generates the prime numbers from 1 to the square root of n as candidate solutions, and check whether n is divisible by one of this prime numbers by checking it one by one. The function returns two values, which are the values of the RSA private keys, p and q.

2.5. The random search

The random search is the most rudimentary example of metaheuristics [7]. Random search generates and checks some random candidate solutions until the allotted time is up, the best (or, sometimes, just good enough solution) is found, or the other stop criteria is met. Our Python code implementation of the random search is as follows.

```
max_time = 10
LIMIT = 10 ** (len(str(n)) // 2 + 1)

p = getRandomPrime(1, LIMIT)
q = getRandomPrime(1, LIMIT)

best = [p, q]

start = time.time()

while True:
      p = getRandomPrime(1, LIMIT)
      q = getRandomPrime(1, LIMIT)
      S = [p, q]
      if delta(S) < delta(best):
            best = S
      if delta(best) == 0.0:
            print "success"
            print S
            print "running time =", time.time() - start, "secs"
            break
      if time.time() - start > max_time:
            print "time's up"
            break
```

3. Results and Discussions

In factoring the RSA modulus with random search, we limit the computation to 10 seconds. If there are no suitable p and q are found in ten seconds, it simply prints "time's up". Otherwise, the value of p and q are returned.

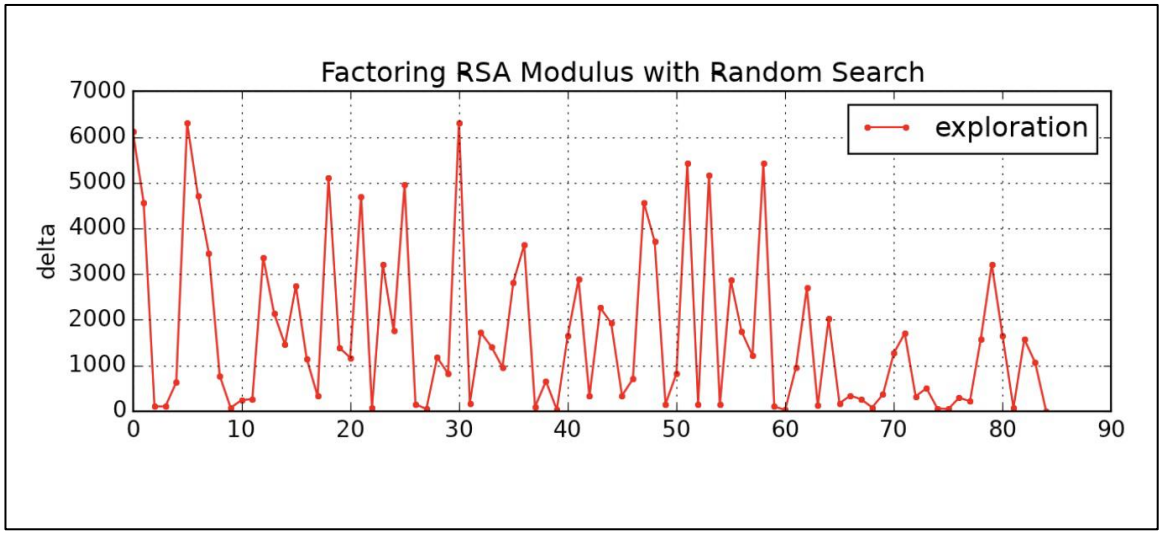

Figure 1. Factoring n = 187 with random search

Figure 1 shows that random search needs 84 explorations to factor n = 187 into p = 11 and q =17. The running time needed is 0.00374317169189 seconds.

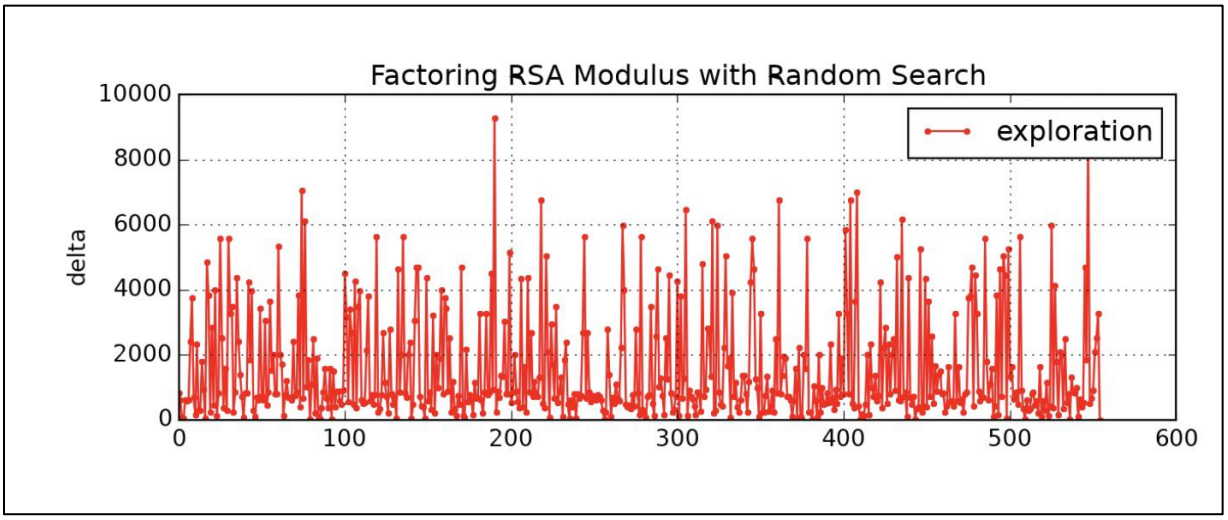

Figure 2. Factoring n = 913 with random search

Figure 2 shows that random search needs 554 explorations to factor n = 913 into p = 11 and q = 83. The running time needed is 0.0202369689941 seconds.

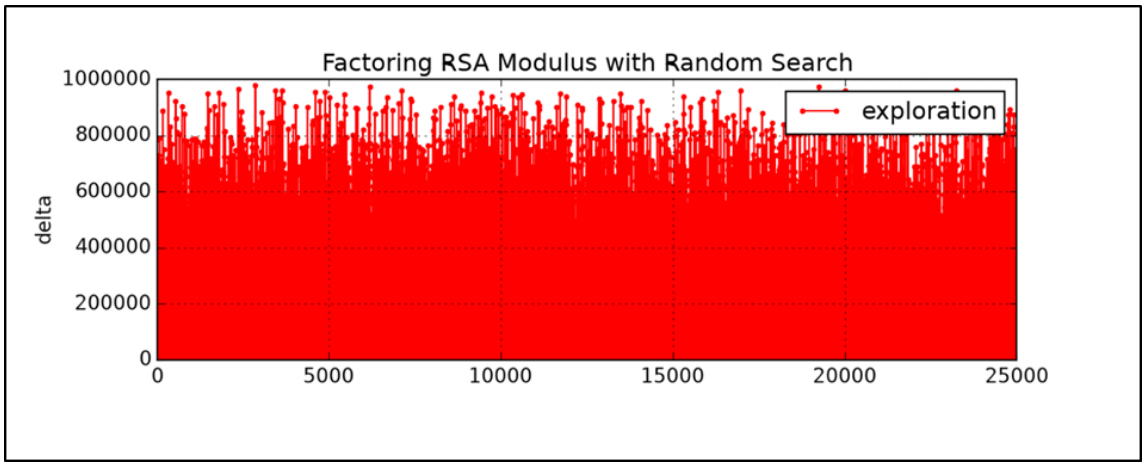

Figure 3. Factoring n = 14041 with random search

Figure 3 shows that random search needs 24988 explorations to factor n = 14041 into p = 739 and q = 19. The running time needed is 1.27728009224 seconds.

We conduct a test to factor n = 557009 with the random search and the algorithm fails to find the factors within the given time limit (10 seconds).

The result of RSA modulus factorization with brute force is tabulated as follows.

Table 1. Factoring RSA modulus with brute force algorithm

n	p * q	number of searches	time (seconds)
187	11 * 17	5	0.00344800949097
913	11 * 83	5	0.00358390808105
14041	19 * 739	8	0.004469871521
557009	653 * 853	119	0.00167393684387
9192907	937 * 9811	159	0.00201606750488
37675201	3907 * 9643	540	0.0139532089233
17614895377	40559 * 434303	4252	0.117401838303
599855115407	694789 * 863363	56166	0.712327957153
4684589242027	837533 * 5593319	66714	1.99000310898
6833740248499	2565161 * 2664059	187492	2.51408982277
91063247464523	2577907 * 35324489	188371	8.69724798203

From Table 1, it can clearly be seen that brute force algorithm runs significantly faster than random search algorithm. For example, to factor n = 14041, brute force algorithm only needs 0.004469871521seconds, while random search needs 1.27728009224 seconds. In this experiment, brute force algorithm can also factor up until n = 91063247464523 (14 digits or 47-bit RSA modulus). Meanwhile, random search can only factor up until n = 14041 (5 digits or 14-bit RSA modulus).

4. Conclusion

From our experiment, it can be concluded that random search is not a suitable candidate to factor the RSA modulus n into its respective private keys, p and q. Random search can only factor small sizes of n (14-bit) and runs significantly slower than the brute force algorithm as it needs too many explorations in which random candidate solutions are generated and tested without proper

order. In contrast, brute force algorithm can factor 47-bit n within time limit of ten seconds. However, in the real world, the RSA usually needs more than 1024-bit modulus to perform securely. Thus, we can only assert that brute force algorithm is an appropriate candidate as a baseline for comparing with other factorization algorithm such as Pollard's and Brent's algorithms, but not to be used to factor RSA with very large modulus.

5. 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] Rivest, Ronald L., Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems." Communications of the ACM 21.2, pages: 120-126. 1978

[2] Diffie, Whitfield, and Martin Hellman. "New directions in cryptography." IEEE transactions on Information Theory 22.6, pages: 644-654. 1976

[3] Shor, Peter W. "Algorithms for quantum computation: Discrete logarithms and factoring." Foundations of Computer Science, 1994 .Proceedings., 35th Annual Symposium on. Ieee, 1994.

[4] Band, Yehuda B., and Y. Avishai. Quantum mechanics with applications to nanotechnology and information science. Academic Press, 2013.

[5] Siarry, Patrick. Metaheuristics. Springer, 2016.

[6] Bianchi, Leonora, Marco Dorigo, Luca Maria Gambardella, and Walter J. Gutjahr. "A survey on metaheuristics for stochastic combinatorial optimization." Natural Computing 8.2, pages: 239-287. 2009

[7] Luke, Sean. Essentials of metaheuristics: a set of undergraduate lecture notes. Lulu Com, 2013.

[8] Shantz, Sheueling Chang. "From Euclid's GCD to Montgomery multiplication to the great divide." 2001.

[9] Wang, Xinmao, and Victor Y. Pan. "Acceleration of Euclidean algorithm and rational number reconstruction." SIAM Journal on Computing 32.2, pages: 548-556. 2003

[10] Smyth, Chris J. "A coloring proof of a generalisation of Fermat's Little Theorem." The American Mathematical Monthly 93(6) , pages:469-471. 1986

[11] Bach, Eric, and Andrew Shallue. "Counting composites with two strong liars." Mathematics of Computation 84.296, pages: 3069-3089.2015

[12] Mateos, Luis A. "Dynamical Sieve of Eratosthenes." arXiv preprint arXiv, pages:1206-2791. 2012.

[13] Lin, Jimmy. "Brute force and indexed approaches to pairwise document similarity comparisons with MapReduce." Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval. ACM, 2009.