On using Pollard’s p-1 Algorithm to Factor RPrime RSA Modulus

Maya Silvi Lydia¹, Mohammad Andri Budiman¹ and Dian Rachmawati¹

¹Departemen Ilmu Komputer, Fakultas Ilmu Komputer dan Teknologi Informasi, Universitas Sumatera Utara, Medan, Indonesia

Keywords: Public Key Cryptography, Cryptanalysis, Factorization, RPrime RSA, Pollard’s p-1

Abstract: RPrime RSA is a variant of RSA public key algorithm that uses the multiplication of two or more prime numbers to construct its modulus. The larger the prime numbers are being used, the better the security of the RPrime RSA becomes. Thus, the security of RPrime RSA depends on the hardness of factoring one big integer into its prime factors. In this study, we attempt to factorize the modulus of RPrime RSA using a modified version of Pollard’s p-1 algorithm, an exact algorithm used to factor an integer into its factors. The modified version of Pollard’s p-1 algorithm makes use of Fermat’s algorithm in order to make sure that all of the factors are primes. The results show that the correlation between RPrime RSA modulus and the factoring time is directly proportional, but the value of RPrime RSA modulus does not always reflect the number of iterations the Pollard’s p-1 algorithm is going through.

1 INTRODUCTION

The concept of public key cryptography (Diffie and Hellman, 1976) was introduced in 1976 and the Rivest-Shamir-Adleman (RSA) algorithm (Rivest, Shamir, and Adleman, 1978) is one of the oldest algorithms that implement the concept. Nowadays, the use of RSA is still very popular since the RSA is easy to implement and it can also be utilized in both encryption scheme and digital signature scheme (Verma, Dutta, and Vig, 2018).

The RSA has a lot of variants; one of them is the RPrime RSA (Paixao and Filho, 2003). Both the RSA and the RPrime RSA base their security on the hardness of factoring a very large integer into its prime factors. The difference is that there are exactly two prime factors that make the modulus of the RSA; while in the case of RPrime RSA, there can be two or more prime factors. Therefore, it is intuitively clear that the RPrime RSA is harder to cryptanalyze than the original RSA.

The Pollard’s p-1 factorization algorithm (Pollard, 1974) is an exact algorithm studied in the field of number theory whose purpose is to factorize an integer into its two factors. This algorithm makes use of Fermat’s Little Theorem (Beatty, Barry, and Orsini, 2018), B-smooth integers (Monaco and Vindiola, 2017), and Euclidean GCD (Marouf, 2017) to quicken its process. Pollard’s rho algorithm, which is the other Pollard’s factorization algorithm, has been known to be more efficient to factorize the RSA modulus than random restart hill-climbing, a metaheuristic algorithm (Budiman and Rachmawati, 2017).

In our study, we use the Pollard’s p-1 algorithm to factor the modulus of the RPrime RSA. Factoring the RPrime RSA modulus can be expected to be harder and slower than factoring the RSA modulus since the RPrime RSA modulus can have more than two prime factors. Therefore, the Pollard’s p-1 algorithm should be modified so it can factor a large integer into infinite numbers of prime factors. The graphical relationships amongst factoring time, the size of the modulus, and the size of its prime factors will be shown as a result.

2 METHODS

In this section we give explanations about the RPrime RSA key generation, the original Pollard’s p-1 algorithm, and the modified version of Pollard’s p-1 to factor the RPrime RSA modulus in Python programming language. The example of each algorithm is explained.
2.1 RPrime RSA Key Generation

As with any other public key cryptography algorithm (Batten, 2013), the RPrime RSA has three stages: key generation, encryption, and decryption. In this study, the key generation of the modulus is the most relevant, and, therefore, it is put forward as follows (Paixao and Filho, 2003):

1. Choose \( k \), the number of prime numbers which will be used in forming the modulus.
2. Generate \( k \) random prime numbers, namely, \( p_1, p_2, \ldots, p_k \), so that \( \gcd(p_1 - 1, p_2 - 1, \ldots, p_k - 1) = 2 \).
3. Compute \( n = p_1 \times p_2 \times \ldots \times p_k \).

As an example, let us select \( k = 3 \). We then generate 3 random prime numbers, \( p_1 = 37, p_2 = 47, p_3 = 71 \), and we check that \( \gcd(37 - 1, 47 - 1) = \gcd(47 - 1, 71 - 1) = \gcd(37 - 1, 71 - 1) = 2 \), so they all can be used as the prime numbers for the RPrime RSA. Lastly, we compute \( n = 37 \times 47 \times 71 = 123469 \).

2.2 Pollard’s p-1 Algorithm

The Pollard’s p-1 algorithm works as follows (see Pollard (1974), Batten (2013), and Yan (2009)):

1. Get \( n \), an odd integer to be factored.
2. Let \( a = 2 \) and \( i = 2 \).
3. Compute \( a = a^i \mod n \).
4. Compute \( d = \gcd(a - 1, n) \).
5. If \( 1 < d < n \), then output \( d \) as a factor of \( n \).
6. If \( d = 1 \), then \( i = i + 1 \), and go to step 3.

For example, let us factor \( n = 209 \). Let \( a = 2 \) and \( i = 2 \). Compute \( a = 2^2 \mod 209 = 4 \). Compute \( d = \gcd(4 - 1, 209) = 1 \). Since \( d = 1 \), compute \( i = 1 + 1 = 3 \), and go to step 3. Compute \( a = 4^3 \mod 209 = 64 \). Compute \( d = \gcd(64 - 1, 209) = 1 \). Since \( d = 1 \), compute \( i = 3 + 1 = 4 \), and go to step 3. Compute \( a = 64^4 \mod 209 = 159 \). Compute \( d = \gcd(159 - 1, 209) = 1 \). Since \( d = 1 \), compute \( i = 4 + 1 = 5 \), and go to step 3. Compute \( a = 159^5 \mod 159 = 144 \). Compute \( d = \gcd(144 - 1, 209) = 11 \). Since \( 1 < d < 209 \), \( d = 11 \) is a factor of 209. The other factor of 209 is 209/11 = 19.

2.3 A Modified Version of Pollard’s p-1 Algorithm to Factor the RPrime RSA Modulus

The original Pollard’s p-1 algorithm can handle factorization of an integer into its two factors. In order to factor RPrime RSA modulus, the Pollard’s p-1 algorithm has to be modified so that it can handle factorization of an integer into two or more factors and it can ensure that all of these factors are primes (by using Fermat’s algorithm to test the primality of those factors). Our modified version of Pollard’s p-1 algorithm to factor the RPrime RSA modulus is shown as a Python code as follows.

```python
iterations = 1

def Pollard(n):
    global iterations
    a = 2
    i = 2
    factor = [1]
    while (n % 2 == 0):
        factor.append(2)
        n = n // 2
    while (n != 1):
        print "Factoring ", n
        if Fermat(n):
            print n, "is already a prime, thus it is a factor"
            factor.append(n)
            factor.sort()
            return factor
        a_old = a
        a = modexp(a, i, n)
        print "iterations =", iterations
        print "a =", a_old, ", ^", i, "mod", n, ", ^" =", a
        d = gcd(a - 1, n)
        n_old = n
        if 1 < d < n:
            factor.append(d)
            n = n // d
```
i = 1
if d == 1:
    print "d = gcd(" + a + ", " + n_old + ") =", d
else:
    print "d = gcd(" + a + ", " + n_old + ") =", d, "is a factor"
    print "Now, factoring ", n_old, "/", d, ", n_old / d
print
iterations += 1
i += 1

d = gcd(108901 - 1, 123469) = 1

Factoring 123469
iterations = 4
a = 108901 ^ 5 mod 123469 = 32697
d = gcd(32697 - 1, 123469) = 1

Factoring 123469
iterations = 5
a = 32697 ^ 6 mod 123469 = 41441
d = gcd(41441 - 1, 123469) = 37 is a factor

Now, factoring 123469 / 37 = 3337

Factoring 3337
iterations = 6
a = 41441 ^ 2 mod 3337 = 2801
d = gcd(2801 - 1, 3337) = 1

Factoring 3337
iterations = 7
a = 2801 ^ 3 mod 3337 = 1883
d = gcd(1883 - 1, 3337) = 1

Factoring 3337
iterations = 8
a = 1883 ^ 4 mod 3337 = 1820
d = gcd(1820 - 1, 3337) = 1

Factoring 3337
iterations = 9
a = 1820 ^ 5 mod 3337 = 900
d = gcd(900 - 1, 3337) = 1

Factoring 3337
iterations = 10
\[ a = 900^6 \mod 3337 = 2593 \]
\[ d = \gcd(2593 - 1, 3337) = 1 \]

Factoring 3337
iterations = 11
\[ a = 2593^7 \mod 3337 = 1563 \]
\[ d = \gcd(1563 - 1, 3337) = 71 \] is a factor

Now, factoring 3337 / 71 = 47

Factoring 47
47 is already a prime, thus it is a factor

Thus, our code shows that the factors of RPrime RSA modulus \( n = 123469 \) are 37, 47, and 71, and these are the prime numbers we have generated in Section 2.1.

The code is then tested with RPrime RSA moduli of different sizes. The result is pictured in Table 1 and Table 2.

<table>
<thead>
<tr>
<th>n</th>
<th>digit</th>
<th>factors</th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
</tr>
</thead>
<tbody>
<tr>
<td>604</td>
<td>21</td>
<td>5</td>
<td>23</td>
<td>37</td>
<td>71</td>
</tr>
<tr>
<td>251905</td>
<td>905</td>
<td>6</td>
<td>5</td>
<td>83</td>
<td>607</td>
</tr>
<tr>
<td>402353</td>
<td>353</td>
<td>6</td>
<td>19</td>
<td>53</td>
<td>479</td>
</tr>
<tr>
<td>3531581</td>
<td>7</td>
<td>23</td>
<td>23</td>
<td>659</td>
<td></td>
</tr>
<tr>
<td>194585749</td>
<td>585749</td>
<td>9</td>
<td>563</td>
<td>577</td>
<td>599</td>
</tr>
<tr>
<td>539715601</td>
<td>715601</td>
<td>9</td>
<td>547</td>
<td>653</td>
<td>151</td>
</tr>
</tbody>
</table>

Table 2: Iterations and time to factor different RPrime RSA modulus

<table>
<thead>
<tr>
<th>n</th>
<th>time (seconds)</th>
<th>iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>60421</td>
<td>0.125804186</td>
<td>11</td>
</tr>
<tr>
<td>251905</td>
<td>0.451285144</td>
<td>43</td>
</tr>
<tr>
<td>402353</td>
<td>0.197597027</td>
<td>17</td>
</tr>
<tr>
<td>3531581</td>
<td>0.474765927</td>
<td>38</td>
</tr>
<tr>
<td>194585749</td>
<td>0.315865994</td>
<td>27</td>
</tr>
<tr>
<td>22985861</td>
<td>1.497702837</td>
<td>140</td>
</tr>
<tr>
<td>539715601</td>
<td>1.726655006</td>
<td>162</td>
</tr>
<tr>
<td>30917064649</td>
<td>0.375844002</td>
<td>35</td>
</tr>
<tr>
<td>861455342321</td>
<td>12.35400701</td>
<td>1048</td>
</tr>
<tr>
<td>152058701902</td>
<td>7.281822205</td>
<td>644</td>
</tr>
</tbody>
</table>

In Table 1, it is shown that the various sizes of \( n \) have been successfully factorized into p1, p2, and p3 which are the RPrime RSA prime numbers. One may check that \( n = p1 \times p2 \times p3 \) for every \( n \) shown in that table. The trend shown in Table 2 shows that while it is intuitively true that the larger the value of the RPrime RSA modulus, the longer it takes time to factorize it, sometimes irregularities do happen. One example of the irregularities is that factoring \( n = 539715601 \) (9 digits) takes 1.726655006 seconds, while factoring \( n = 30917064649 \) (11 digits) takes 0.375844002 seconds. This irregularity is due to the fact that factoring \( n = 539715601 \) takes 162 iterations, while factoring \( n = 30917064649 \) only takes 35 iterations. The number of iterations depends on the relationship amongst the prime numbers that form the RPrime RSA modulus.
4 CONCLUSIONS

The conclusions of our study are as follows. First, the modified version of Pollard’s p-1 algorithm which makes use of Fermat’s algorithm is able to factor RPrime RSA modulus into its all its prime factors. Second, the correlation between RPrime RSA modulus and the time to factor it with Pollard’s p-1 tends to be directly proportional. Third, the value of RPrime RSA modulus does not always reflect the number of iterations the Pollard’s p-1 algorithm is going through.

ACKNOWLEDGEMENTS

We gratefully acknowledge that this research is funded by Kemenristekdikti Republik Indonesia via Lembaga Penelitian Universitas Sumatera Utara. The support is under the research grant DRPM Kemenristekdikti of Year 2018 Contract Number: 59/UN5.2.3.1/PPM/KP-DRPM/2018.

REFERENCES

Batten L M 2013 Public key cryptography applications and attacks (Hoboken, N.J: Wiley-Blackwell)
Beatty T, Barry M and Orsini A 2018 A Geometric Proof of Fermat’s Little Theorem Advances in Pure Mathematics 08 41–4
Paixao C A M and Filho D L G 2003 An efficient variant of the RSA cryptosystem IACR Cryptology ePrint Archive
Pollard J M 1974 Theorems on factorization and primality testing Mathematical Proceedings of the Cambridge Philosophical Society 76 521

Verna R, Dutta M and Vig R 2018 RSA Cryptosystem Based on Early Word Based Montgomery Modular Multiplication Services – SERVICES 2018 Lecture Notes in Computer Science 33–47
Yan S Y 2009 Primality testing and integer factorization in public-key cryptography (Boston: Springer)