Contemporary Mathematics
https://ojs.wiserpub.com/index.php/CM/

Research Article

RSA Cryptanalysis: A Novel Acceleration for Euler-Based Fermat Factorization Algorithm

Ibrahim M. Alseadoon[1], Khaled A. Fathy[2], Mohamed A.G. Hazber[1], Yasser Kotb[3], Hazem M. Bahig[1,4*]

[1]Department of Information and Computer Science, College of Computer Science and Engineering, University of Hail, Hail, 81481, Saudi Arabia
[2]Department of Mathematics, Faculty of Science, Al-Azhar University, Cairo, 11884, Egypt
[3]Department of Information Systems, College of Computer and Information Sciences, Imam Mohammad Ibn Saud Islamic University (IMSIU), Riyadh, 13318, Saudi Arabia
[4]Computer Science Division, Department of Mathematics, Faculty of Science, Ain Shams University, Cairo, 11566, Egypt
E-mail: h.bahig@uoh.edu.sa

Received: 30 Ocotber 2025; **Revised:** 3 December 2025; **Accepted:** 8 December 2025

Abstract: The Rivest-Shamir-Adleman (RSA) cryptosystem is an efficient and secure method for transmitting data over the Internet. Breaking this system primarily relies on the integer factorization problem, which involves factoring a composite odd number, n, into two prime factors, p and q. Euler-based Fermat Factorization (EFF) is one of the factoring methods based on the modular multiplication operation and is efficient when $\delta = p - q \leq n^{0.25}$. However, the execution times of the EFF algorithm increase significantly with large values of n and $\delta > n^{0.25}$. In this paper, we propose a novel technique for optimizing the number of modular multiplication operations required to find the two factors. The method can factor a large odd number in a fast time, even when $\delta > n^{0.25}$. For different values of the length of the primes and δ, the experimental results indicate that the proposed algorithm is, on average, 85.5% faster than the previous improvements on the Fermat factorization method.

Keywords: Rivest-Shamir-Adleman (RSA) cryptanalysis, integer factorization, Fermat's method, Euler's theorem, modular multiplication

MSC: 11A51, 11Y05, 68Q25

1. Introduction

The widespread use of computers and the internet makes information security an increasingly important problem. The most important tool for achieving information security is cryptography. There are two types of cryptosystems: (1) secret-key cryptosystems, in which the two entities have to share a common secret key; and (2) public-key cryptosystems, in which each entity has two keys: one is a public key, and the other is a private key. Public-key cryptosystems have three main advantages over secret-key cryptosystems: (1) Key distribution under secret-key cryptosystems has a problem; (2) There is no digital signature, in general, using secret-key cryptosystems; and (3) The number of keys in secret-key

DOI: https://doi.org/10.37256/cm.7220269032

cryptosystems for a network is larger than in public-key cryptosystems. In contrast, secret-key cryptosystems are faster than public-key cryptosystems.

The security of many public-key cryptosystems is based on some problems in number theory, such as integer factorization. Integer factorization is a problem of finding two factors, p and q, for a positive odd composite integer, n, such that $n = pq$ [1].

This problem is significant in public-key cryptosystems such as the Rivest-Shamir-Adleman (RSA) cryptosystem. Moreover, integer factorization presents a challenge because of the considerable computational time required to find the prime factors when the value of n is large. This difficulty leads to challenges in breaking the RSA cryptosystem within polynomial time in the general case [2].

Owing to the importance of the integer factorization problem, many researchers have proposed various methods to address it. These methods differ from one another in terms of several criteria.

• Is the factoring method deterministic or randomized? Examples of deterministic methods include trial division [2], Fermat Factorization (FF) [3–5], wheel factorization [6, 7], Lehman's method [8], quadratic Sieves [9], and number field sieving [10]. An example of a randomized factorization method is provided in [11, 12].

• Is the factorization method effective when the value of n is large or small? An example of an effective method for a large value of n is thegeneral number field sieving [10], whereas the trial division method is effective when one of the two prime factors is small.

• Is the factorization method effective when n satisfies a specific condition? For example, the Fermat factorization method is efficient when the difference between the two factors is less than or equal to $n^{0.25}$. Pollard's rho [13] is another example of an effective factorization method, particularly when all prime factors of $p-1$ and $q-1$ are small.

In this paper, we address the FF method for several reasons. (1) In the RSA, the modulus, n, is constructed as the product of two primes, p and q, of the same size, meaning that the size of each prime factor is $|n|/2$, where $|n|$ represents the number of bits in n. If the difference between the two factors is less than or equal to $|n|/4$, then the FF method can be solved in polynomial time [1, 2]. (2) Many general-purpose factoring methods, such as the quadratic sieve method and the general number field sieve method, feature the FF method as a step in their general solutions. (3) Many algorithms have been proposed to accelerate the computational time required by the FF method when the difference between the two prime factors exceeds the size of n divided by 4, as detailed in [4]. Most of these algorithms focus on reducing the number of integers, u, in the search space to minimise perfect square testing. Different operations are used in these algorithms, such as (i) using one modulo operation, u mod 10, (ii) using two moduli operations, $(u \bmod 10)$ and $(n \bmod 20)$, and (iii) testing if $n = 4k \pm 1$ to ignore half the numbers in the search space. Another proposed method involves replacing the perfect square operation with a modular multiplication operation to reduce the computational cost associated with a perfect square operation [5].

In this work, we develop a novel algorithm that aims to reduce the number of modular multiplication operations in the Euler-based Fermat Factorization algorithm (EFF) [5]. The suggested technique uses modular multiplication rather than perfect square calculations and optimizes the total number of modular multiplication operations required to obtain the result. Furthermore, we reduce the number of modular multiplications by applying an effective masking method and the binary search algorithm to accelerate the identification of the prime factors. Consequently, the developed algorithm can handle large values of n, specifically $n = 1,024$, and $2,048$, as well as cases in which the difference between the two prime factors exceeds $n^{0.25}$.

The remainder of this work is organized into the following sections: In the second section, we provide a detailed review of the most important algorithms related to the subject of the study. In the third section, we present the proposed algorithm in detail, and in the fourth section, we present an experimental study of the proposed algorithm. Finally, the conclusions of this work are presented in Section five.

2. Related algorithms

This section describes the key algorithms on the subject of study, including Fermat Factorization (FF) [1], Fermat Factorization using two Moduli (FF2M) [3], and Euler-based Fermat Factorization (EFF) algorithms [5]. The FF algorithm is the original algorithm for the Fermat method, whereas the FF2M algorithm is a recently modified algorithm for the Fermat method. The EFF algorithm is a factorization method that combines the Fermat method with the Euler theorem.

2.1 *The FF algorithm*

The main idea of the FF algorithm is to test whether the value of $v^2 = u^2 - n$ is a perfect square, where the value of u starts with $\lceil \sqrt{n} \rceil$. If the value of v^2 is a perfect square, the algorithm terminates, and the two main factors are $p = u + v$ and $q = u - v$. Otherwise, the algorithm increases the value of u by one and tests v again. The complete code for this method is shown in Algorithm 1.

Algorithm 1: Fermat Factorization (FF)

1. **Input:** An odd number n.
2. $u = \lceil \sqrt{n} \rceil$
3. $v = \sqrt{u^2 - n}$
4. **while** v is not integer **do**
5. $\quad u = u + 1$
6. $\quad v = \sqrt{u^2 - n}$
7. **end**
8. $p = u + v$
9. $q = u - v$
10. **return** p and q
11. **Output:** Two prime factors p and q of n.

2.2 *The FF2M algorithm*

The author of [3] proposed a significant improvement to the FF method by reducing the number of values examined for the perfect square operation, depending on the values of (u mod 10) and (n mod 20). The first step of the algorithm begins with $u = \lfloor \sqrt{n} \rfloor + 1$ and then searches for the first value of u that is divisible by 10 or that satisfies the condition that $u^2 - n$ is a perfect square. If $u^2 - n$ is a perfect square, then $p = u + \sqrt{u^2 - n}$ and $v = u - \sqrt{u^2 - n}$ are calculated. The second step begins when $u^2 - n$ is not a perfect square, at which point the value of n mod 20 is computed. Based on values of n mod 20, specifically 1, 3, 7, 9, 11, 13, 17, and 19, the algorithm tests either 3 or 4 values of u in cycles of length 10 (u mod 10 = 0). Table 1 presents the increments considered for each case during one cycle iteration.

Table 1. The increments that are implemented in each case of $r = n$ mod 20

r	1	3	7	9	11	13	17	19
C	[1, 4, 4, 1]	[2, 6, 2, -]	[4, 2, 4, -]	[3, 2, 2, 3]	[0, 4, 2, 4]	[3, 4, 3, -]	[1, 8, 1, -]	[0, 2, 6, 2]

Table 1 illustrates the increment values, C, on u to generate three or four values. This table can be represented as an array of vectors, D, of size 4. Therefore, the increment number for the position r in the array D is $D[r,\ j]$, where $r = 1,\ 3,\ 7,\ 9,\ 11,\ 13,\ 17,\ 19$ and $j = 1,\ 2,\ 3,\ 4$. The algorithm performs a loop to search for the values of u and tests the perfect square for the value of $u^2 - n$. At the end of the loop, the values of p and q are calculated as follows: $p = u + \sqrt{u^2 - n}$ and $v = u - \sqrt{u^2 - n}$. The complete code for the FF2M algorithm is provided in Algorithm 2.

Algorithm 2: Fermat Factorization using two Moduli (FF2M)

1. **Input:** An odd number n.
2. $u = \lfloor \sqrt{n} \rfloor$
3. *found* = *false*
4. **while** *found* = *false* and $u \bmod 10 \neq 0$ **do**
5. **If** $u^2 - n$ is perfect square **then**
6. *found* = *true*
7. **else**
8. $u = u + 1$
9. **end**
10. **end**
11. **if** *found* = *true* **then**
12. $p = u + \sqrt{u^2 - n}$
13. $q = u - \sqrt{u^2 - n}$
14. **else**
15. $r = n \bmod 20$
16. **while** *found* = *false* **do**
17. $u = u + D[r,\ 1]$
18. **if** $u^2 - n$ is perfect square **then**
19. *found* = *true*
20. **else**
21. $u = u + D[r,\ 2]$
22. **if** $u^2 - n$ is perfect square **then**
23. *found* = *true*
24. **else**
25. $u = u + D[r,\ 3]$
26. **if** $u^2 - n$ is perfect square **then**
27. *found* = *true*
28. **else**
29. **if** $r \neq 3,\ 7,\ 13,\ 17$ **then**
30. $u = u + D[r,\ 4]$
31. **end**
32. **end**
33. **end**
34. **end**
35. **end**
36. $p = u + \sqrt{u^2 - n}$
37. $q = u - \sqrt{u^2 - n}$
38. **end**
39. **return** p and q
40. **Output:** Two prime factors p and q of n.

2.3 *The EFF algorithm*

The author of [5] proposed an improvement to the FF algorithm by replacing the perfect square operation test with a low-cost modular multiplication operation based on Euler's theorem. Euler's theorem states that $a^{\Phi(n)} \equiv 1 \bmod n$, where n is a positive integer, a is an integer that is coprime to n, and $\Phi(n)$ is the Euler totient function, defined as $(p-1)*(q-1) = n - (p+q) + 1$. Algorithm 3 contains the pseudocode for the EFF algorithm.

Algorithm 3: Euler-based Fermat Factorization (EFF)

1. **Input:** An odd number n.
2. $u = 2\lceil\sqrt{n}\rceil$
3. Choose a positive integer c such that $GCD(c,\ n) = 1$, say $c = 2$
4. $a = c^{-1} \bmod n$
5. $s = c^2 \bmod n$
6. $t = a^{n-u+1} \bmod n$
7. **if** $t = 1$ **then**
8. $\quad u = \dfrac{u}{2}$
9. $\quad v = \sqrt{u^2 - n}$
10. **else**
11. $\quad$ **while** $t \neq 1$ **do**
12. $\quad\quad t = t * s$
13. $\quad\quad t = t \bmod n$
14. $\quad\quad u = u + 2$
15. $\quad\quad$ **if** $t = 1$ **then**
16. $\quad\quad\quad u = \dfrac{u}{2}$
17. $\quad\quad\quad v = \sqrt{u^2 - n}$
18. $\quad\quad$ **end**
19. $\quad$ **end**
20. **end**
21. $p = u + v$
22. $q = u - v$
23. **return** p and q
24. **Output:** Two prime factors p and q of n.

The algorithm begins by setting the initial values of many variables used to define and update the value of the term t as in Lines 2-6. These variables are u, c, a, and s. Then the objective of the algorithm is to find the value of t that is equal to 1. Lines 11-17 outline the main steps of the EFF algorithm and include a loop that continues as long as t is not equal to 1. During each iteration, modular multiplication is performed on t and s until t equals 1. If $t = 1$, then the algorithm calculates the two prime factors, p and q, as in Lines 21-23.

3. The proposed algorithm

The main advantage of the EFF algorithm is that it replaces perfect square testing with modular multiplication. The main objective of the proposed algorithm is to minimize the number of modular multiplication operations required to find the two factors and to substitute the modular multiplication operation with a simple efficient searching method.

The basic concept and steps of the proposed algorithm are detailed in the following subsections. Additionally, we provide the complexity analysis of the proposed algorithm and a numerical example to illustrate its operations.

3.1 *The main idea*

In the EFF algorithm, we continue multiplying t by s and calculating the remainder of the division by n until we reach 1. After each multiplication, we increase the value of u by 2.

The idea behind the proposed algorithm is that instead of multiplying t repeatedly by s, we can multiply it by a mask value, s^l, one time, where $l = \log_s n$. We then calculate the remainder of this multiplication divided by n and check whether the result of the multiplication has reached the target value, which is $t = 1$.

If the multiplication of t by the mask, s^l, reaches or exceeds 1, then the result of the multiplication is one of the powers of s, ranging from $s^0 = 1$ to s^l. To achieve this outcome, we create an incremental array, B, that contains the powers of s up to $l = \log_s n$.

$$B(i) = 2^i,\ 0 \leq i \leq l.$$

In general, the algorithm continues to multiply t by $b_l = s^{\log_s n}$ and tests the resulting value to determine if it is in B. If we find it, this indicates that the value has either reached or passed $t = 1$. If we fail to find it, this indicates that the value has not yet reached 1, prompting us to multiply by b_l once more. To test whether the resulting value in the array B, we use the binary search algorithm because the array B is sorted.

If the value does not reach 1 or passes it, the algorithm increases the previously variable, i, by l to keep track of the number of multiplication operations. We then perform the multiplication again, noting that i was initially set to zero. When we find the result of the current multiplication in B, we multiply the result of the previous iteration by s until it reaches 1, and increase the value of i by 1 each time. Finally, we calculate the final value of u by incrementing its old value by $2i$ and dividing the sum by 2. Next, we determine the value of v, which is equal to $\sqrt{u^2 - n}$, and then calculate the values of $p = u + v$ and $q = u - v$.

3.2 *The algorithm*

The pseudocode for the newly proposed algorithm, the Euler-based Fermat Factorization using Masking (EFFM) algorithm, is presented in Algorithm 4. Lines 2-10 are similar to the original EFF algorithm. In lines 11-12, the values of i and l are set. In lines 13-16, the incremental array of powers of s, B, is created. In lines 17-29, a loop runs until t equals 1. In each iteration, the previous value of t is multiplied by b_l, and the remainder of this multiplication result after it is divided by n is stored in $temp$ (line 18). The binary search algorithm, the best-known method for searching sorted data, is applied to find the value of $temp$ in the array, B, and store its index in j. If $temp$ is not found in B, then the returned value is -1 (line 20). In this case, the value of i is incremented by l, and the value of t is equal to $temp$ (lines 20-22). Therefore, the number of modular multiplication operations is reduced by $l - 1$ for one iteration in the while loop. Additionally, the time required for binary search is $\log l = \log \log_s n$, which is very small. Conversely, if $temp$ is found in B, then an inner loop executes until $t = 1$ (lines 24-27). In each iteration, t is updated by $t * s \bmod n$, and the value of i is incremented by 1. The value of the increment to be added to u is computed as shown in line 30. The values of u, v, p, and q are computed in lines 31-35.

In summary, we have reduced the number of multiplication operations done by the EFF algorithm. Now, for each multiplication operation, we perform substitutions for l multiplication operations that were executed in the EFF algorithm. While we have added a search operation with each multiplication in the EFFM algorithm, it is very inexpensive and costs only $\log l$. Consequently, we assume that the EFF algorithm executes αl total number of modular multiplication operations, where $\alpha \gg l$. Then, the EFFM algorithm executes $\alpha - 1 + l$ modular multiplications plus $\alpha \log l$ search operations. This means that the number of modular multiplication operations is reduced, approximately, by a factor l. Consequently, the number of modular multiplication operations has been significantly reduced, as well as the cost of the search operation is very low compared to the cost of the modular multiplication operation.

Algorithm 4: Euler-based Fermat Factorization using Mask (EFFM)

1. **Input:** A composite integer n.
2. $u = \lceil 2\sqrt{n} \rceil$
3. Choose a positive integer c such that $gcd(c,\ n) = 1$, say $c = 2$
4. $a = c^{-1} \bmod n$
5. $s = c^2 \bmod n$
6. $t = a^{n-u+1} \bmod n$
7. **if** $t = 1$ **then**

8. $u = \frac{u}{2}$
9. $v = \sqrt{u^2 - n}$
10. **else**
11. $i = 0$
12. $l = \log_s n$
13. $b_0 = 1$
14. **for** $k \leftarrow 1$ to l **do**
15. $b_k = b_{k-1} * s$
16. **end**
17. **while** $t \neq 1$ **do**
18. $temp = (t * b_l) \bmod n$
19. $j = Binarysearch(temp, B)$
20. **if** $j = -1$ **then**
21. $i = i + l$
22. $t = temp$
23. **else**
24. **while** $t \neq 1$ **do**
25. $i = i + 1$
26. $t = (t * s) \bmod n$
27. **end**
28. **end**
29. **end**
30. $inc = 2i$
31. $u = \frac{u + inc}{2}$
32. $v = \sqrt{u^2 - n}$
33. **end**
34. $p = u + v$
35. $q = u - v$
36. **return** p and q
37. **Output:** Two prime factors p and q of n.

3.3 *Complexity analysis*

Assume that the total number of iterations in the EFF algorithm is represented by α. In each iteration, the algorithm performs one modular multiplication operation and one addition operation. Therefore, the running time is $O(\alpha(t_{mm} + t_+))$, where t_{mm} and t_+ are the time complexities for the modular multiplication and addition operations, respectively.

In the case of the EFFM algorithm, one modular multiplication operation will be executed in the main loop. Thereafter, the algorithm executes the Binary Search (BS) algorithm and then, based on the output of the BS algorithm, the EFFM algorithm executes either

(1) Case 1: the EFFM algorithm jumps l modular multiplication operations and then returns to the main loop, because the variable $temp$ does not exist in the array B.

(2) Case 2: the variable $temp$ matches with one element in the array B. In this case, the algorithm finds the variable t that satisfies $t \bmod n = 1$ and terminates the main loop to find the prime factors.

In case 1 and for every iteration in the main loop, the algorithm executes $O(\log l)$ iterations for the BS algorithm and one addition operation. Therefore, the time complexity for this case is $O(t_{mm} + \log l + t_+)$. In general, this case will be repeated many times inside the main loop. Therefore, the total number of operations is $O((\alpha/l)(t_{mm} + \log l + t_+))$. The term (α/l) came from jumping l modular multiplication operations in Case 1.

In case 2, the algorithm finds the term that verifies $t \bmod n = 1$, which takes a maximum of l iterations. For each iteration, the algorithm executes one modular multiplication and one addition operations. Therefore, the running time is $O(l(t_{mm} + t_+))$. This case required $O(((\alpha/l) - 1)(t_{mm} + \log l + t_+) + l(t_{mm} + t_+))$.

The final time complexity for the EFFM is $O((\alpha/l)(t_{mm} + \log l + t_+))$.

Note that the term $\log l$ is very small compared to the value of l. Moreover, the cost of the searching operation is less than the modular multiplication operation.

The space complexity of the EFFM algorithm is based on the size of the array B is $l = \log n$, which represents the number of bits in n. In the context of the RSA cryptosystem, the values of n can be 1,024 or 2,048 bits. Consequently, the size of the array B remains relatively small, with a maximum of 2,048 elements. Therefore, there is no significant memory overhead associated with storing the precomputation.

3.4 *Example of EFFM execution*

This part aims to demonstrate the functioning of the EFFM algorithm by executing it on an example.

Let $n = 63,062,851$, the initial value of u is 15,883 (line 2). Since $gcd(2,\ n) = 1$, the algorithm selects 2 as the value of c (line 3). The algorithm computes $a = c^{-1} \bmod n = 31,531,426$ (line 4) and $s = c^2 \bmod n = 4$ (line 5). The first value of t is $36,979,860 \neq 1$ (line 6). The algorithm will jump to lines 11 and 12, and set $i = 0$ and $l = 12$, respectively. The elements of the B are as follows (lines 13-15):

$$B = \{1,\ 4,\ 16,\ 64,\ 256,\ 1,024,\ 4,096,\ 16,384,\ 65,536,\ 262,144,\ 1,048,576,\ 4,194,304,\ 16,777,216\}.$$

In the while loop (lines 17-29) and after the first iteration, the value of $t = temp = (t * b_l) \bmod n = 23,006,703$, which does not exist in B. Therefore, i will be incremented by l to equal 12. After the second iteration, $t = temp = (t * b_l) \bmod n = 11,940,254$ (also not found in B), and i will be incremented by l to equal 24. The loop continues to multiply and search 289 times. The last value of t that is not found in B is 27,373,086, with $i = 3,456$. If this value of t is multiplied by s, the resulting value will fall within B. Consequently, we execute the else statement and multiply it by s again until it reaches 1, incrementing the value of i by 1 in each iteration. The values of t are as follows: 46,429,493, 59,592,270, 49,180,527, 7,533,555, 30,134,220, 57,474,029, 40,707,563, 36,704,550, 20,692,498, 19,707,141,15,765,713, and 1, with the final value of i being 3,468. In lines 26-30, the values of $inc,\ u,\ v,\ p,$ and q are computed, resulting in values of 6,936, 11,410, 8,193, 8,193, and 3,217, respectively.

4. The experimental study

In this section, we present a practical study to evaluate the EFF, FF2M, and EFFM algorithms on the basis of their execution time. We implemented the algorithms in C++, using the GMP library to handle large numbers [15]. The programs were executed on the Google Cloud platform. The specifications of the machine used are as follows: e2-medium type with 2 vCPUs, 4 GB of memory, and an Intel Broadwell CPU platform. The machine runs the Ubuntu 20.04 operating system.

The execution time of the algorithms is based on two parameters: (1) the number of bits in n, described as len, and (2) the value of $|\delta| = len/4 + k$, where $k \geq 0$. The len values selected for this study are 64, 128, 256, 512, 1,024, and 2,048, with each value applied to δ values, where the values of k are: 0, 5, 10, 15 and 20.

For each fixed value of n, we generate two prime numbers, p and q, such that $n = pq$ and $p - q = \delta$. Also, for fixed values of n and δ, the execution time of an algorithm is the average time for 25 different instances.

Table 2. The comparison between EFF, FF2M and EFFM in terms of execution time in seconds where $len = 64$

				Improvements of EFFM	
k	EFF	FF2M	EFFM	EFF	FF2M
0	6.86×10^{-6}	3.09×10^{-6}	1.50×10^{-5}	-	-
5	2.71×10^{-5}	1.06×10^{-5}	0.00027	-	-
10	0.019	0.00379	0.000919	95.16%	75.75%
15	19.874	3.868	0.701	96.47%	81.88%
20	40,136.5	7,644.78	1,429.56	96.44%	81.3%

Table 3. The comparison between EFF, FF2M and EFFM in terms of execution time in seconds where $len = 128$

				Improvements of EFFM	
k	EFF	FF2M	EFFM	EFF	FF2M
0	2.17×10^{-5}	2.64×10^{-5}	2.78×10^{-5}	-	-
5	4.75×10^{-5}	1.50×10^{-5}	6.72×10^{-5}	-	-
10	0.159	0.00412	0.000679	99.57%	83.53%
15	63.0969	5.255	1.176	98.13%	77.63%
20	64,519	5,064.01	1,110.5	98.28%	78.07%

Table 4. The comparison between EFF, FF2M and EFFM in terms of execution time in seconds where $len = 256$

				Improvements of EFFM	
k	EFF	FF2M	EFFM	EFF	FF2M
0	0.000105	5.34×10^{-6}	0.000114	-	-
5	0.000111	2.07×10^{-5}	0.000207	-	-
10	0.0341	0.00505	0.000602	98.24%	88.09%
15	69.6875	9.763	0.804	98.85%	91.76%
20	71,706.2	10,962.2	877.821	98.78%	91.99%

Table 5. The comparison between EFF, FF2M and EFFM in terms of execution time in seconds where $len = 512$

				Improvements of EFFM	
k	EFF	FF2M	EFFM	EFF	FF2M
0	0.000428	3.88×10^{-5}	0.000478	-	-
5	0.000601	2.86×10^{-5}	0.000938	-	-
10	0.0380	0.00539	0.00118	96.88%	78.00%
15	77.94	11.676	0.851	98.91%	92.71%
20	80,160.7	18,919.5	971.557	98.78%	94.86%

Table 6. The comparison between EFF, FF2M and EFFM in terms of execution time in seconds where $len = 1,024$

				Improvements of EFFM	
k	EFF	FF2M	EFFM	EFF	FF2M
0	0.004	5.16×10^{-5}	0.00357	-	-
5	0.00359	3.71×10^{-5}	0.00408	-	-
10	0.174	0.0162	0.00455	97.38%	71.94%
15	96.79	22.288	1.385	98.57%	93.79%
20	97,999.3	25,135.1	1,292.19	98.68%	94.86%

Table 7. The comparison between EFF, FF2M and EFFM in terms of execution time in seconds where $len = 2,048$

				Improvements of EFFM	
k	EFF	FF2M	EFFM	EFF	FF2M
0	0.0252	8.30×10^{-5}	0.0254	-	-
5	0.0249	6.08×10^{-5}	0.0287	-	-
10	0.214	0.0926	0.0296	86.15%	68.1%
15	130.973	99.681	1.836	98.6%	98.15%
20	134,600	68,938.6	1,899.42	98.59%	97.24%

The results of the implementation of the EFF, FF2M, and EFFM algorithms, along with the rates of improvement of the EFFM algorithm compared with the EFF and FF2M algorithms, are shown in Tables 2-7. From Tables 2-7, we can draw the following conclusions:

1. In the case of $k = 0$ or 5 for all values of len, it is clear that the EFF and FF2M algorithms outperform the EFFM algorithm in almost all instances. However, it is notable that the execution times for all three algorithms are already very small, so these situations can be ignored.

2. For all values of len when $k = 10$, the execution times for the three methods are quite modest. However, the EFFM algorithm outperforms the other two algorithms in all scenarios. For example, the rates of improvement for the EFFM method are 95.16% and 75.75% when the len is 64 and k is 10 for the EFF and FF2M algorithms, respectively. Additionally, the rates of improvement for the EFFM method are 97.38% and 71.94% when the len is 1,024 and k is 10 for the EFF and FF2M algorithms, respectively. On the basis of the results of this case, we can conclude that the average improvement ratio of the EFFM algorithm compared with the EFF algorithm is 95.56 %, and the average improvement ratio of the EFFM algorithm compared with the FF2M algorithm is 77.56 %.

3. For all values of len when $k = 15$ or 20, the EFFM algorithm is clearly superior to the EFF and FF2M algorithms. For example, the rates of improvement for the EFFM method are 98.85% and 91.76% for the case in which the len is 256 and k is 15 for EFF and FF2M algorithms, respectively. Additionally, the rates of improvement for the EFFM method are 98.68% and 94.86% for the case in which the len is 1,024 and k is 20 for the EFF and FF2M algorithms, respectively. The average improvement ratio of the EFFM algorithm compared with the EFF algorithm for all values of len when $k = 15$ is 98.25%, and the average improvement ratio of the EFFM algorithm compared with the FF2M algorithm is 89.32%. The average improvement ratio of the EFFM algorithm compared with the EFF algorithm in the case in which $k = 20$ for all values of len is 98.26%, and the average improvement ratio of the EFFM algorithm compared with the FF2M algorithm is 89.72%.

4. The results clearly show that as the value of k increases, the implementation of the EFF and FF2M algorithms becomes more challenging because of a significant increase in their execution times. However, the execution time of the EFFM algorithm remains within an acceptable range.

5. Conclusion

In this paper, we present an ultrafast algorithm that improves the Euler-based Fermat algorithm. The proposed algorithm significantly reduces the number of modular multiplication operations. Consequently, the running time of the proposed algorithm is significantly reduced compared with that of the original algorithm. Additionally, it can handle a large composite number when the difference between two prime numbers exceeds $n^{0.25}$.

The experimental results demonstrate that the proposed algorithm exhibits superior performance across a range of len and δ values. On average, it is 97% faster than the original algorithm and 85.5% faster than the best-known modified algorithm.

There are future questions for the proposed algorithm. Can we theoretically determine the range of δ such that the proposed algorithm operates efficiently? Additionally, how can we implement the proposed algorithm across various models of computation, including quantum computing, DNA computing, and Graphics Processing Unit (GPU) computing?

Author contributions

Khaled A. Fathy: Conceptualization, methodology, formal analysis, software, writing—original draft preparation, writing—review and editing. Hazem M. Bahig: Conceptualization, methodology, formal analysis, writing—review and editing, supervision, project administration; Ibrahim Alseadoon: Conceptualization, Writing—review and editing; Mohamed Hazber: Conceptualization, writing—review and editing; Yasser Kotb: Conceptualization, writing—review and editing; All authors have read and approved the final version of the manuscript for publication.

Acknowledgement

The authors extend their thanks to the Scientific Research Deanship at the University of Ha'il-Saudi Arabia through project number "RG-23 143".

Conflict of interest

The authors declare no competing financial interest.

References

[1] Shatnawi AS, Almazari MM, AlShara Z, Taqieddin E, Mustafa D. RSA cryptanalysis-Fermat factorization exact bound and the role of integer sequences in factorization problem. *Journal of Information Security and Applications*. 2023; 78: 103614. Available from: https://doi.org/10.1016/j.jisa.2023.103614.

[2] Lenstra AK. Integer factoring. *Designs, Codes and Cryptography*. 2000; 19: 101-128.

[3] Bahig HM. Speeding up Fermat's factoring method using precomputation. *Annals of Emerging Technologies in Computing*. 2022; 6(2): 50-60. Available from: https://doi.org/10.33166/aetic.2022.02.004.

[4] Bahig HM, Mahdi MA, Alutaibi KA, AlGhadhban A, Bahig HM. Performance analysis of Fermat factorization algorithms. *International Journal of Advanced Computer Science and Applications*. 2020; 11(12): 340-352. Available from: https://doi.org/10.14569/ijacsa.2020.0111242.

[5] Somsuk K. The new integer factorization algorithm based on Fermat's factorization algorithm and Euler's theorem. *International Journal of Electrical and Computer Engineering (IJECE)*. 2020; 10(2): 1469-1476. Available from: https://doi.org/10.11591/ijece.v10i2.pp1469-1476.

[6] Bahig HM, Nassr DI, Mahdi MA, Hazber MA, Al-Utaibi KA, Bahig HM. Speeding up wheel factoring method. *The Journal of Supercomputing*. 2022; 78: 15730-15748. Available from: https://doi.org/10.1007/s11227-022-04470-y.

[7] Zaki AM, Bakr ME, Alsahangiti AM, Khosa SK, Fathy KA. Acceleration of wheel factoring techniques. *Mathematics*. 2023; 11(5): 1203. Available from: https://doi.org/10.3390/math11051203.

[8] Hales J, Hiary G. A generalization of Lehman's method. *The Ramanujan Journal*. 2024; 65: 1773-1790. Available from: https://doi.org/10.1007/s11139-024-00959-7.

[9] Davis JA, Holdridge DB. Factorization using the quadratic sieve algorithm. In: *Advances in Cryptology*. Boston, MA: Springer; 1984. p.103-113. Available from: https://doi.org/10.1007/978-1-4684-4730-9_9.

[10] Stevenhagen P. The number field sieve. *Algorithmic Number Theory*. 2008; 44: 83-100.

[11] Pomykała J. Elliptic-curve factoring, witnesses and oracles. In: Dąbrowski A, Pieprzyk J, Pomykała J. (eds.) *Number-Theoretic Methods in Cryptology*. Cham: Springer; 2024. p.85-115. Available from: https://doi.org/10.1007/978-3-031-82380-0_2.

[12] Jiménez Urroz J, Pomykała J. Factoring numbers with elliptic curves. *The Ramanujan Journal*. 2024; 64: 265-273. Available from: https://doi.org/10.1007/s11139-023-00822-1.

[13] Somsuk K. An efficient variant of Pollard's $p-1$ for the case that all prime factors of the $p-1$ in B-smooth. *Symmetry*. 2022; 14(2): 312. Available from: https://doi.org/10.3390/sym14020312.

[14] Somsuk K, Kasemvilas S. MVFactor: A method to decrease processing time for factorization algorithm. In: *2013 International Computer Science and Engineering Conference (ICSEC)*. Nakhonpathom, Thailand: IEEE; 2013. p.339-342. Available from: https://doi.org/10.1109/ICSEC.2013.6694805.

[15] Fractalb. *GMP: The GNU multiple precision arithmetic library*. Available from: https://gmplib.org/ [Accessed 2nd December 2025].