Annals of Emerging Technologies in Computing (AETiC)
Vol. 6, No. 2, 2022

Research Article

Speeding Up Fermat's Factoring Method using Precomputation

Hatem M. Bahig

Ain Shams University, Egypt
hmbahig@sci.asu.edu.eg

Received: 19 January 2022; Accepted: 25th March 2022; Published: 1st April 2022.

Abstract: The security of many public-key cryptosystems and protocols relies on the difficulty of factoring a large positive integer n into prime factors. The Fermat factoring method is a core of some modern and important factorization methods, such as the quadratic sieve and number field sieve methods. It factors a composite integer n=pq in polynomial time if the difference between the prime factors is equal to Δ= $p - q \leq n^{0.25}$, where p>q. The execution time of the Fermat factoring method increases rapidly as Δ increases. One of the improvements to the Fermat factoring method is based on studying the possible values of (n mod 20). In this paper, we introduce an efficient algorithm to factorize a large integer based on the possible values of (n mod 20) and a precomputation strategy. The experimental results, on different sizes of n and Δ, demonstrate that our proposed algorithm is faster than the previous improvements of the Fermat factoring method by at least 48%.

Keywords: ***Fermat's Factoring Method; Integer Factorization; Precomputation; Public-key Cryptosystem; RSA***

---

1. Introduction

Let n be a positive integer. The integer factorization problem (**IFP**) is finding the prime factors of $n = p_1^{\alpha_1} p_2^{\alpha_2} \dots p_k^{\alpha_k}$, where $p_i^{\alpha_i}$ are pairwise distinct primes and each $\alpha_i \geq 1$. **IFP** is one of the fundamental problems in information security and computational number theory for the following reasons:

1. The security of many public-key cryptosystems and protocols [1-2] relies on the difficulty of **IFP**. For example, in the RSA cryptosystem [1], each user performs the following:
   - Generates two large distinct primes, p and q of the same bit-size.
   - Computes $n = p\,q$, and Euler's totient function $\varphi(n) = (p-1)(q-1)$.
   - Randomly generate an integer e with $\gcd(\varphi(n), e) = 1$, where gcd denotes the greatest common divisor.
   - Computes the multiplicative inverse d of e modulo $\varphi(n)$, $i.e.$, $ed \equiv 1 \bmod \varphi(n)$.
   - Now, the public key is the pair *(e, n),* while the private key *is d*. The prime factors p and q and the integer φ(n) are kept secret (or destroyed).
   - A message (plaintext) m is encrypted by calculating the ciphertext as follows: $c = m^e mod\ n$. An encrypted message (ciphertext) is decrypted by calculating $m = c^d mod\ n$.
2. **IFP** is an excellent example of a problem that does not currently have a polynomial time algorithm in classical computers but does in quantum computers [3-5].

There are two basic types of factoring methods for a large odd composite integer n [6-13]:

1. Special purpose factoring methods that quickly find small prime factors. The primary problem with this type of factoring method is that if n has no small factor, as in public-key cryptosystems, then factoring methods will have essentially no chance of succeeding. The Trial division, Pollard's-method, Pollard's *p-1* method, and the elliptic curve method are examples of this type of factoring method.

2. General-purpose factoring methods are exponential or subexponential time algorithms that factor n independent of the size of its prime factors. The continued fraction technique, the quadratic sieve, and the number field sieve are examples of this type of factoring methods. For factoring large n with large prime factors, the number field sieve method has proven to be the most effective method until now.

On the other hand, if additional information about some public-key cryptosystems is available, then there are some factoring techniques [14-17] that work for those cryptosystems.

In this paper, we are concerned with Fermat's factorization method [12-13] or simply Fermat's method (**FM**), which finds two integer factors p and q such that $n = pq = u^2 - v^2 = (u - v)(u + v)$. If $u - v \neq 1$, then we have found a nontrivial factor of n. The idea of **FM** is a fundamental of some modern and important factorization methods, such as the quadratic and multiple polynomial quadratic sieves, and number field sieves methods.

For security reasons, such as in public-key cryptosystems [1-2], the integer $n = pq$ is usually a product of two primes of equal bit-size. It is possible to find these factors in polynomial time if the difference between the prime factors $\triangle = \mathrm{p} - \mathrm{q} \leq n^{0.25}$ [18]. The main challenge with **FM** is when the difference $\triangle$ is greater than $n^{0.25}$.

In this paper, we are interested in speeding up **FM** by reducing the search space using a precomputation strategy, where some the preliminary computations can be made to reduce the number of necessary operations after obtaining the integer u. This paper introduces a precomputation strategy to improve Somsuk's improvement of **FM** [19], which we call **FMMod20**. We call our proposed algorithm **FMMod20Precomp**.

The experimental results show that **FMMod20Precomp** is faster than the previous improvements to **FM** by at least 48% when $\triangle > n^{0.25}$. In fact, the percentage of improvement is affected by the sizes of n and $\triangle$.

The organization of the paper is as follows. Section 2 introduces the related works. In Section 3, we give a brief description of **FMMod20.** In Section 4, we introduce our precomputation strategy to improve **FMMod20**. In Section 4.1, we present the main idea of the proposed algorithm, **FMMod20Precomp**. In Section 4.2, we give a complete description of the proposed algorithm. In Section 5, we present the experimental study and comparison with three previous algorithms and then show the performance of **FMMod20Precomp**. In Section 6, we present the conclusion of the paper and future works.

2. Related Works

In general, **FM** starts by computing $u = \lfloor \sqrt{n} \rfloor + 1$, and $v = u^2 - n$. Then, it repeatedly checks whether v is a perfect square. If v is not a perfect square, then **FM** increases u by 1 and computes $v = u^2 - n$. If v is a perfect square, then **FM** returns $p = u + \sqrt{v}$, and $q = u - \sqrt{v}$, see Algorithm **FM**, where **PS**(x) is a subroutine that returns true if the integer number x is a perfect square, and returns false otherwise. Clearly, the search space of **FM** is large when n is large.

Algorithm 1. FM (n: integer) returns integers	
Begin	
1.	$u = \lfloor \sqrt{n} \rfloor + 1$
2.	$v = u^2 - n$
3.	While (PS(v) =False) loop
4.	$u = u + 1$
5.	$v = u^2 - n$
6.	End while
7.	return $u + \sqrt{v}$ and $u - \sqrt{v}$
End.	

Many techniques have been proposed to improve **FM.** Some of them used different **FM** formulas. Mckee [20] proposed a variant of **FM** to search for three integers u, v, w such that $w^2 = \left(u + \lceil \sqrt{n} \rceil v\right)^2 - nv^2$ and then computes $\gcd\left(u + \lceil \sqrt{n} \rceil v - w, n\right)$. Hart [21] proposed another variant of **FM** by searching for a solution to $v^2 = \left(\lceil \sqrt{n\,\mathrm{u}} \rceil\right)^2 - nu$, which was achieved by looking for squares after reduction modulo n, where u starts from 1. The main drawback of these modifications is that they require more arithmetic operations than **FM**, so the running time is large for large integers, e.g. 1024 bits.

Other techniques discarded some values of v *or* u that cannot lead to a solution, and so hence reduced some calculations such as the perfect square test for v or u. Somsuk and Kasemvilas [22] proposed to ignore the perfect square test **PS** when the least significant digits of v, denoted by LSD(v), LSD(v) $= 2, 3, 7,$ *or* 8 because if PS(v)=True, then either LSD(v) $= 0, 1, 4, 5, 6$ *or* 9. Somsuk and Kasemvilas [23] improved [22] by studying LSD(u) to not compute v. In [19], Somsuk used LSD(u) and ($n \, mod \, 20$) to study the possible values of ($v \, mod \, 20$) to be a perfect square in order to make a decision whether to compute v. We denote this method by **FMMod20.** Somsuk and Tientanopajai [24] proposed a method based on studying the last k digits of n. The main problem of this method is that it requires $4 * 10^{k-1}$ specific subroutines. Clearly, this is large, in particular for large k.

Somsuk [25] proposed to use the formula $4n = x^2 - y^2$, the Euler theorem, and the multiplication instead of **PS** to not compute **PS**, where the Euler theorem states, "Let x be a positive integer such that the greatest common divisor between x and n is 1, then $x^{\varphi(n)} \equiv 1 \bmod n$". We denote this method by **EF.** Vynnychuk *et al.* [26] improved the method in [27] which checks whether $u^2 - n$ by module of a certain set of foundations of modules b that are prime numbers is a quadratic residue. They used the relation $(v \, mod \, b)^2 \, mod \, b = ((u \, mod \, b)^2 \, mod \, b - n \, mod \, b) mod \, b$. The main disadvantage of this method is that it requires large memory storage.

Shiu [28] enhanced **FM** by ignoring all even (or odd) numbers for u or v based on writing n as $n = 4k \pm 1, n \geq 3$. If $n = 4k + 1$, then u is odd and v is even. Otherwise, $n = 4k - 1$, u is even, and v is odd. We denote this method by **OEF.** Somsuk *et al.* [29-30] proposed using the formula $n = 6k \pm 1$. If $n = 6k - 1$, then u is divisible by 3. If $n = 6k + 1$, then u is not always divisible by 3.

On the other hand, some techniques [31-33] are based on estimating the prime factors and using the continued fraction of $\frac{1}{\sqrt{n}}$ to obtain a list of convergent and initial values for u. The main drawback of the methods in [31-32] is that they do not work for balanced primes, i.e., primes of the same bit-size. Tahir *et al.* [33] studied balanced primes with a slight improvement compared to **EF** using some numerical examples.

Recently, Longhas *et al.* [34] proved theoretically that a composite (not prime) integer n of the form $4k^2 + 1$ can be factorized using **FM**.

The main lack in these studies is that there is not enough practical comparative study for large numbers. In [35], the authors presented a practical comparative study of most of these modifications when n is in the range of 100-500 bits. They showed experimentally that the fastest improvement of **FM** is **OEF**.

3. Fermat's Method using mod 20

Since we are going to improve the algorithm (**FMMod20)** proposed by Somsuk [19], we briefly describe **FMMod20**.

To avoid testing the perfect square for $u^2 - n$ for every u, Somsuk [19] proposed testing only some values of u based on the value of $n \, mod \, 20$ and the least significant digit of u, LSD(u), i.e., $u \; mod \; 10$. Somsuk observed the following:

1. Since n is odd and not divisible by 5, $n \, mod \, 20$ is either 1, 3, 7, 9, 11, 13, 17, or 19.
2. If $u^2 - n$ is a perfect square, then $u^2 - n \, mod \, 20$ is either 0, 1, 4, 5, 9, or 16.

Table 1 shows the possibility of finding perfect squares (in bold and underlined) when using the two relations ($n \, mod \, 20$) and ($u \, mod \, 10$).

Table 1. Possibility of perfect square of $u^2 - n$ [19]

LSD(u)	The values of u^2-n *mod 20* when $u^2 - n$ *mod 20* is							
	1	3	7	9	11	13	17	19
0	1	17	13	11	**9**	7	3	**1**
1	**0**	18	14	12	10	8	**4**	2
2	3	**1**	17	15	13	11	7	**5**
3	8	6	2	**0**	18	**16**	12	10
4	15	13	**9**	7	**5**	3	19	17
5	**4**	2	18	**16**	14	12	8	6
6	15	13	**9**	7	**5**	3	19	17
7	8	6	2	**0**	18	**16**	12	10
8	3	**1**	17	15	13	11	7	**5**
9	**0**	18	14	12	10	8	**4**	2

Algorithm 2. FMMod20 (n : integer): return integers

Begin

1. $u = \lceil\sqrt{n}\rceil$
2. $r = n \bmod 20$
3. u=changeU(u, r)
   /* find the first value of u such that $u^2 - n$ mod 20 =0,1,4,5,9, or 16, see Algorithm change, Lines 1-34 in [19]. */
4. $v = \sqrt{u^2 - n}$
5. While (v is not integer)
6. If (r = 1) then
7. The procedure for determining the integer v such that LSD(u) is 1, 5, or 9
8. ElseIf (r = 3) then
9. The procedure for determining the integer v such that LSD(u) is 2, or 8
10. ElseIf (r = 7) then
11. -- similar to the previous steps
12. -- the complete while loop can be found in Algorithm MFFV4, Lines 11-21 in [19].
13. End if
17. End While
18. return p=u-v, and q=u+v

End.

---

4. The Proposed Algorithm

In this section, we propose modifications to **FMMod20** using a precomputation strategy. In Section 4.1, we mention the main idea of our proposed algorithm, **FMMod20Precomp**. Section 4.2 includes the full description of **FMMod20Precomp.**

4.1 Outline of the proposed algorithm

The proposed algorithm **FMMod20Precomp** is based on two observations on **FMMod20**:

1. Given an integer n, the value of $r = n \bmod 20$ is fixed in the algorithm. Therefore, there is no need to check the value of r every iteration of the while-loop (see, for examples, Lines 6, 8, 10 of the **FMMod20** algorithm).
2. The value of $u \bmod 10$ (or $\mathrm{LSD}(u)$) is variable, but for a fixed value of $n \bmod 20$, the possible values of u, $\mathrm{PS}(u^2 - n)$ =*True*, can be determined initially. Thus, there is no need to search for u such that $\mathrm{LSD}(u)$ is one of the values that may produce a perfect square (see, for examples, Lines 6, 8, 10 in the FMMod20 algorithm). For example, suppose that $n \bmod 20 = 1$. From Table 2, the possible values for u, such that $\mathrm{PS}(u^2 - n)$ =*True*, are $\mathrm{LSD}(u) = 1, 5$, and 9. As we can see from Table 1 and Table 2, the number of possible values of u that may produce a perfect square is at most 3.

Table 2. Number of possible values of u such that PS($u^2 - n$) is true

n mod 20	1	3	7	9	11	13	17	19
Number of accepted cases	3	2	2	3	3	2	2	3

Our strategy is to compute the differences between the possible values of u every 10 consecutive integers such that $u^2 - n$ is a perfect square. To implement such strategy, we do the following steps:

First, we define a ***Cycle*** as an integer interval $[u_0, u_0 + 9]$, where $\mathrm{LSD}(u_0) = 0$, i.e., 10 consecutive integers. The integer u_0 is called the **start point** of a cycle. Since we use the relation *u mod 10*, i.e., there are 10 possible values for *u mod 10,* we choose the length of the cycle to be 10.

Second, we construct an array **dif** to hold the differences between the possible values of u such that $\mathrm{PS}(u^2 - n)$ =True based on the relations ($n \bmod 20$) and ($u \bmod 10$) . The array **dif** is a two-dimensional array that contains *4×8* elements, as shown in Table 3, where

- d_1 is the difference between the start point of a cycle and the first possible value of u in the cycle, such that $PS(u^2 - n)$ =True.
- d_2 is the difference between the first and second possible values of u in a cycle, such that $PS(u^2 - n)$ =True.

- d_3 is the difference between the second and third possible values of u in a cycle, such that $PS(u^2 - n)$ =True. If there is no third value for u, such that $PS(u^2 - n)$ =True, then d_3 has no value, denoted by "-".
- d_4 is the difference between the last possible value of u in a cycle, such that $PS(u^2 - n)$ =True, and the start point of the next cycle.

Table 3. The two-dimensional array *dif* contains the differences between the possible values of u, such that $PS(u^2 - n)$ =True, based on the relations (n *mod* 20) and (u *mod* 10).

	Index of dif							
	1	**3**	**7**	**9**	**11**	**13**	**17**	**19**
1: d_1	1	2	4	3	0	3	1	0
2: d_2	4	6	2	2	4	4	8	2
3: d_3	4	-	-	2	2	-	-	6
4: d_4	1	2	4	3	4	3	1	2

4.2 The algorithm

The proposed algorithm includes two main steps. The first step (called the **preparation step**) aims to search for a solution (prime factors) in the small interval $[u, u']$, where $u = \lfloor\sqrt{n}\rfloor + 1$ and u' is the first integer greater than or equal to u such that $(u' \; mod \; 10) = 0$. This step can be done by checking if $(u \; mod \; 10) \neq 0$. Then, the proposed algorithm terminates by calculating the prime factors if $\mathrm{PS}(u^2 - n)$ =True. Otherwise, the algorithm increases u by one. This process is repeated until the algorithm either finds u such that $u^2 - n$ is a perfect square or stops when $(u \; mod \; 10) = 0$.

The second step, called **perfect square in a cycle (PSC)**, aims to search for the prime factors in the remainder space. Based on the fixed value of ($r = n \; mod$ 20), the algorithm increases u by **dif[**1, r]. Note that, u satisfies that $u \; mod \; 10 = 0$ as described in the preparation step. Next, the proposed algorithm tests whether $u^2 - n$ is a perfect square. If $\mathrm{PS}(u^2 - n) = \mathrm{False}$, then **PSC** increases u by **dif[**2, r] and again tests u. Based on the value of r, **PSC** increases u by **dif[**3, r] or not. If there is no value of u satisfying that $\mathrm{PS}(u^2 - n) = \mathrm{True}$ in the cycle, then **PSC** increases u by **dif[**4, r] and repeats the process. If one of u's satisfies that $\mathrm{PS}(u^2 - n)$ =True, then **PSC** returns the prime factors.

The steps of the proposed algorithm are presented in the **FMMod20Precomp** algorithm. Line 1 represents the first value of u in the search space, while Line 2 represents the initial value of the Boolean variable "*found*". Lines 3-9 represent the preparation step, i.e., finding the first value of u such that either $(u \bmod 10) = 0$ or $\mathrm{PS}(u^2 - n)$ =True. Lines 10-14 calculate the prime factors of n when the boolean variable *found* is true, i.e., $(u^2 - n)$ is a perfect square. The remaining lines represent the second step **PSC**, i.e., searching for u such that $\mathrm{PS}(u^2 - n)$=True using the Subroutine **Cycle**(r, u, n), where $r = n \; mod$ 20. Lines 18-19 calculate the prime factors of n

Note that there is no statement in the body of the while-loop, Lines 16-17, i.e., we repeat calling the subroutine **Cycle** until it finds u such that $\mathrm{PS}(u^2 - n)$ =True. It is not difficult to put the body of the subroutine **Cycle** inside the body of the while-loop which is slightly faster.

The full pseudo code of our proposed algorithm is given in Algorithm **FMMod20Precomp**, where the subroutine Cycle performs the second step **PSC**.

Algorithm 3. Cycle(*r*: integer, *u*: in out integer, n:integer): return Boolean

```
Begin
    u ⟵ u + dif[1,r]
    If (PS(u² − n) = true) then
        return True
    Else
        u ⟵ u + dif[2,r]
        If (PS(u² − n) = true) then
            return True
        Else
            If (r ∈ {1,9,11,19}) then
                u ⟵ u + dif[3,r]
                If (PS(u² − n) = true) then
                    return True
                End if
```

```
            End if
              u ⟵ u + dif[4, r]
        End if
    End if
    return False
End.
```

Algorithm 4. FMMod20Precomp (Fermat Method using Modulus 20 and precomputation)

Input: a composite number n.

Output: two primes, p and q, s. t. $n = p\,q$.

```
Begin
    u ⟵ ⌊√n⌋+1
    Found ⟵ False
    While (Not found and (u mod 10) ≠ 0) do
        If (PS(u² − n) = true) then
            Found ⟵ True
        Else
            u ⟵ u + 1
        End if
    End while
    If (Found=True) Then
        p ⟵ u + √(u² − n)
        q ⟵ u − √(u² − n)
        return p and q
    End if
    m ⟵ n mod 20
    While ( Cycle(m, u, n) = False) do
    End while
    p ⟵ u + √(u² − n)
    q ⟵ u − √(u² − n)
    return p and q
End.
```

Now, the number of iterations executed by **FMMod20Precomp** can be calculated as follows. The total number of iterations for the search space for **FM** is $(p+q)-\left(\lfloor\sqrt{n}\rfloor+1\right)$, where the term $(p+q)$ represents the last value of u in the search space, while the term $\left(\lfloor\sqrt{n}\rfloor+1\right)$ represents the first value of u in the search space.

The algorithm **FMMod20Precomp** executes α_0 iterations to find the first u such that $(u \bmod 10) = 0$, where $0 \le \alpha_0 \le 9$. The integer α_0 takes the value 0 when the start value of u satisfies $(u \bmod 10) = 0$. On the other hand, $\alpha_0 = 9$ when the start value of u satisfies $(u \bmod 10) = 1$. In general, the worst case for the number of iterations is 9. The number of remaining iterations, based on **FM,** is

$$\alpha = (p+q)-\left(\lfloor\sqrt{n}\rfloor+1\right)-\alpha_0.$$

FMMod20Precomp excludes approximately 70% of the values in the search space since it ignores at least 7 integers out of 10 integers in each cycle. Therefore, the total number of iterations including the test of a perfect square after the preparation step is $\alpha - \alpha \times 0.7 = 0.3\,\alpha$. Hence the total number of iterations for **FMMod20Precomp** is $0.3\,\alpha + \alpha_0 = 0.3\left((p+q)-\left(\lfloor\sqrt{n}\rfloor+1\right)\right)+0.7\alpha_0$. Therefore, the **FMMod20Precomp** algorithm has a better performance compared to the previous **FM** modifications**.**

5. Experimental Results

This section presents the experimental performance of the proposed **FMMod20Precomp** algorithm. It consists of two sections. The first section describes the data set, hardware, and software used in the experimental study. The other section compares **FMMod20Precomp, FMMod20** [19], **EF** [25] and **OEF** [28-30].

5.1 Platform Specification and Data Set

The algorithms (**FMMod20Precomp**, **FMMod20, EF and OEF**) were implemented using the C++ language and executed on a computer consisting of the processor Xeon E5-2630 with a speed of 2.6 GHz and a memory of 16 GB. The computer ran the Microsoft Windows 10 operating system. We used the GMP library (GNU Multiple Precision)[1] to operate with big integers, greater than 64 bits.

We have two parameters affecting the execution time of the algorithms: (1) the size of n, which is equal to the number of bits in n and denoted by $|n|$, and (2) the value of $\triangle$. The sizes of n conducted in the experimental study were 128, 256, 512, and 1024 bits. For each value of n, the size of each prime factor is $|n|/2$. For example, if n has 1024 bits, then each prime factor has 512 bits. Now, let $\triangle_0 = \mathrm{p} - \mathrm{q} = n^{0.25}$, and therefore,

$$|\triangle_0| = \frac{|n|}{4}.$$

The sizes of $\triangle$ conducted in the experimental study were $|\triangle_0| + 10$, $|\triangle_0| + 15$, $|\triangle_0| + 20$, or $|\triangle_0| + 25$. As we mentioned in Section 1, **FM** is efficient when $\triangle \leq n^{0.25}$, while the execution time of **FM** increases as $\triangle$ increases. Therefore, there is no need to study **FM** when the size of $\triangle$ equals $|\triangle_0|$.

5.2. The Results

This section compares the proposed **FMMod20Precomp** algorithm with the three previous algorithms: **FMMod20**, **OEF** and **EF**.

Figure 1-4 show the average execution times (in seconds) of the four algorithms for 50 values of n as the size of n varies 128, 256, and 512 bits, and for 20 values of n with sizes of 1024 bits. For each size 128, 256, 512, and 1024 bits, we generate n such that the size of $\triangle$ is $|\triangle| = |\triangle_0| + \delta$, where $\delta = 10, 15, 20,$ and 25.

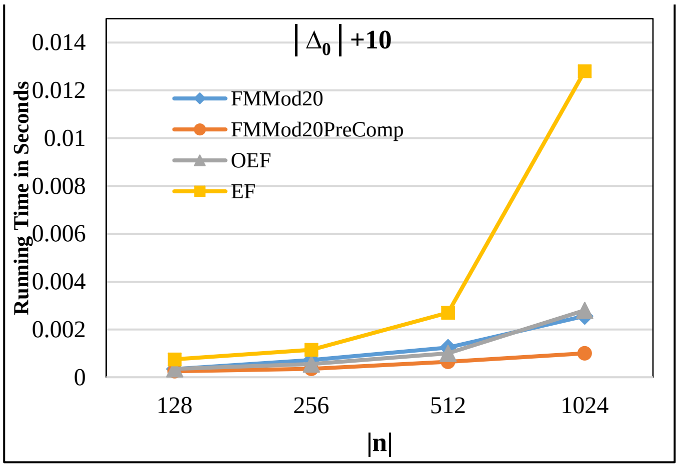

Figure 1. Comparison between the four algorithms when $|\Delta| = |\Delta_0| + 10$

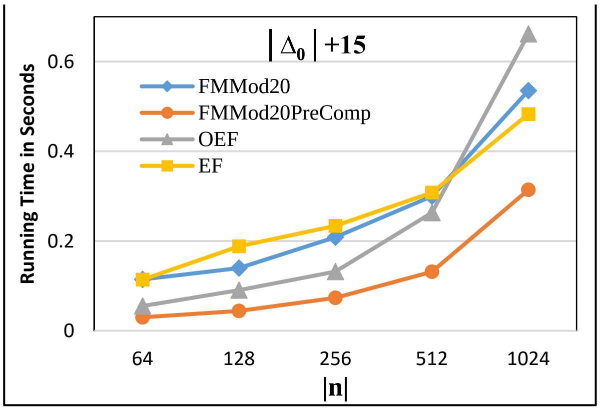

Figure 2. Comparison between the four algorithms when $|\Delta| = |\Delta_0| + 15$

[1] GMP library, "The GNU multiple precision arithmetic library", 2021. Available: https://gmplib.org.

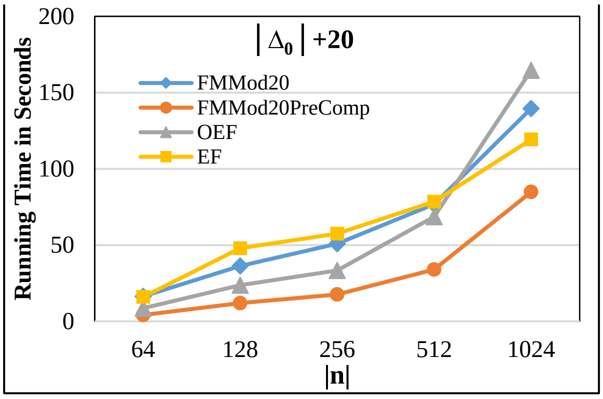

Figure 3. Comparison between the four algorithms when $|\Delta| = |\Delta_0| + 20$

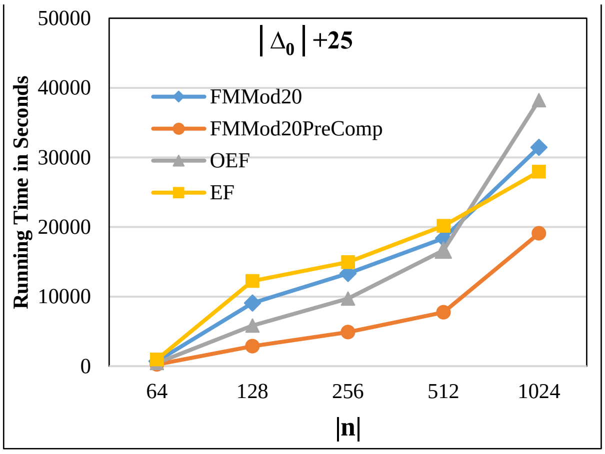

Figure 4. Comparison between the four algorithms when $|\Delta| = |\Delta_0| + 25$

Table 4. Percentage of improvement of **FMMod20Precomp** compared to **FMMod20**

δ	|n|				
	64	**128**	**256**	**512**	**1024**
10	89.3%	27.1%	51.4%	47.6%	60.9%
15	73.8%	68.3%	64.8%	56.1%	41.3%
20	74.5%	67.0%	65.4%	55.8%	39.2%
25	65.4%	68.3%	63.2%	57.9%	39.3%

Table 5. Percentage of improvement of **FMMod20Precomp** compared to **OEF**

δ	|n|				
	64	**128**	**256**	**512**	**1024**
10	66.7%	28.6%	36.4%	35.0%	64.3%
15	45.6%	51.1%	44.3%	49.9%	52.5%
20	51.5%	49.5%	47.1%	50.5%	48.5%
25	46.2%	50.8%	49.5%	53.6%	50.1%

Table 6: Percentage of improvement of **FMMod20Precomp** compared to **EF**

δ	|n|				
	64	**128**	**256**	**512**	**1024**
10	88.9%	66.7%	69.6%	75.9%	92.2%
15	73.7%	76.5%	68.6%	57.2%	34.9%
20	74.0%	75.0%	69.3%	56.7%	28.9%
25	74.5%	76.5%	67.2%	61.7%	31.8%

From Figures 1-4 and Tables 4-6, we can conclude the following:

- **FMMod20Precomp** has execution times less than **FMMod20, EF**, and **OEF** for all different values of n and $|\triangle| = |\triangle_0| + \delta$.
- **FMMod20Precomp** reduces the average CPU time by 58% on average compared to a fast implementation of the **FMMod20** algorithm.

- **FMMod20Precomp** reduces the average CPU time by 48% on average compared to the **OEF** algorithm.
- **FMMod20Precomp** reduces the average CPU time by 66% on average compared to the **EF** algorithm.
- The percentages of the improvements depend on the size of n and Δ.
- In general, for $|n|$< 1024, the two algorithms **OEF** and **FMMod20** have better performance than the **EF** algorithm. On the other hand, the **EF** algorithm has better performance than the **OEF** and **FMMod20** algorithms in the case of $|n|$=1024.
- **EOF** and **FMMod20** perform worse when $|n|$=1024.

In general, the proposed **FMMod20Precomp** algorithm has better performance compared to **FMMod20, EF**, and **OEF** for different values of n and $|\triangle_0| + \delta$.

6. Conclusion and Future Works

A new strategy to improve the Fermat factoring method and **FMMod20** is proposed. It is based on computing the differences between the possible values of u every 10 consecutive integers such that $u^2 - n$ is a perfect square. We have used a two-dimensional array **dif** to hold these differences. Then based on the value of ($n \ mod\ 20$), and the array **dif,** the algorithm discards some values of u that cannot lead to a solution. The experimental results, on different sizes of n and Δ, show that the **FMMod20Precomp** algorithm has better performance. **FMMod20Precomp** is faster than **FMMod20** by 58% on average and faster than the previous improvements of the Fermat factoring method by 48% on average.

In future work, we will study the possibility of combining two or more methods to improve **FM.** We can also use a multicore system [36] to improve **FMMod20Precomp**.

Acknowledgment

This work was supported financially by the Academy of Scientific Research and Technology (ASRT), Egypt, Grant No 6419, ScienceUp.

References

[1] Ronald Rivest, Adi Shamir and Leonard Adleman, "A Method for Obtaining Digital Signatures and Public-key Cryptosystems", *Communication of ACM,* Print ISSN: 0001-0782, Online ISSN: 1557-7317, pp. 120-126, Vol. 21, 1978, Published by Association for Computing Machinery (ACM), DOI: 10.1145/359340.359342.

[2] Atsushi Fujioka, Koutarou Suzuki, Keita Xagawa and Kazuki Yoneyama, "Strongly Secure Authenticated Key Exchange from Factoring, Codes, and Lattices", *Designs, Codes and Cryptography,* Print ISSN: 0925-1022, Online ISSN: 1573-7586, pp. 469–504, Vol. 76, 2015, Published by Springer, DOI: 10.1007/s10623-014-9972-2.

[3] Peter Shor, "Polynomial-time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer", *SIAM Journal on Computing,* pp. 1484-1509, Vol. 26, No.5, 1997, Published by Society for Industrial and Applied Mathematics (SIAM), DOI: 10.1137/S0097539795293172, Available: https://link.springer.com/article/10.1007/s10623-014-9972-2.

[4] Nanyang Xu, Jing Zhu, Dawei Lu, Xianyi Zhou, Xinhua Peng *et al.*, "Quantum Factorization of 143 on a Dipolar-coupling Nuclear Magnetic Resonance System", *Physical Review Letter,* Print ISSN: 0031-9007, Online ISSN: 1079-7114, pp. 1–5, Vol. 108, No. 13, 2012, Published by American Physical Society (APS), Available: https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.108.130501.

[5] Baonan Wang, Feng Hu, Haonan Yao and Chao Wang, "Prime Factorization Algorithm Based on Parameter Optimization of Ising Model", *Scientific Reports,* Online ISSN: 2045-2322, Vol. 10, No. 7106, 2020, Published by Nature Portfolio, DOI: 10.1038/s41598-020-62802-5, Available: https://www.nature.com/articles/s41598-020-62802-5.

[6] Arjen K. Lenstra, "Integer Factoring", *Design Codes and Cryptography,* Print ISSN: 0925-1022, Online ISSN: 1573-7586, pp. 101–128, Vol. 19, 2000, Published by Springer, DOI: 10.1023/A:1008397921377.

[7] Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone, *Handbook of Applied Cryptography*, 1st ed., Boca Raton: CRC Press, 1997.

[8] Douglas Robert Stinson and Maura Paterson, *Cryptography Theory and Practice,* 4th ed., Boca Raton: CRC Press, 2018.

[9] Simon Rubinstein-Salzedo, "Clever Factorization Algorithms and Primality Testing", in *Springer Undergraduate Mathematics Series: Cryptography*, Cham: Springer, 2018, ch. 13, pp. 127-140, Series ISSN: 1615-2085, Series E-ISSN: 2197-4144, DOI: 10.1007/978-3-319-94818-8.

[10] Song Y. Yan, "Factoring Based Cryptography", in *Cybercryptography: Applicable Cryptography for Cyberspace Security*, ISBN: 978-3-319-72534-5, Online ISBN: 978-3-319-72536-9, Springer, Cham, pp. 217-286, 2019, DOI: 10.1007/978-3-319-72536-9.

[11] Luke Valenta, Shaanan Cohney, Alex Liao, Joshua Fried, Satya Bodduluri *et al.*, "Factoring as a Service", in *Lecture Notes in Computer Science*, *Financial Cryptography and Data Securit (FC),* Vol. 9603, Print ISBN: 978-3-662-54969-8, Online ISBN: 978-3-662-54970-4, Series Print ISSN: 0302-9743, Series Online ISSN: 1611-3349, pp. 321–338, 2017, Published by Springer, DOI: 10.1007/978-3-662-54970-4_19.

[12] Song Y. Yan, *Primality Testing and Integer Factorization in Public-Key Cryptography*, ISBN: 978-0-387-77267-7, Online ISBN: 978-0-387-77268-4, 2nd ed., Springer, Boston, MA, 2009, DOI: 10.1007/978-0-387-77268-4.

[13] Xinguo Zhang, Mohan Li, Yu Jiang and Yanbin Sun, "A Review of the Factorization Problem of Large Integers", in *Lecture Notes in Computer Science, Artificial Intelligence and Security (ICAIS)*, Vol. 11635, Print ISBN: 978-3-030-24267-1, Online ISBN: 978-3-030-24268-8, Series Print ISSN 0302-9743, Series Online ISSN 1611-3349 , pp 202-213, Springer, Cham, 2019, DOI: 10.1007/978-3-030-24268-8, Available: https://link.springer.com/book/10.1007/978-3-030-24268-8.

[14] Omar Akchiche and Omar Khadir, "Factoring RSA Moduli with Primes Sharing Bits in the Middle", *Applicable Algebra in Engineering, Communication and Computing*, Print ISSN: 0938-1279, Online ISSN: 1432-0622, pp. 245–259, Vol. 29, No. 3, 2018, DOI: 10.1007/s00200-017-0340-0.

[15] Dieaa. I. Nassr, Hatem M. Bahig, Ashraf Bhery and Sameh Daoud, "A New RSA Vulnerability using Continued Fractions", in *International Conference on Computer Systems and Applications (IEEE/ACS)*, 31 March - 4 April 2008, Doha, Qatar, Print ISSN: 2161-5322, Online ISSN: 2161-5330, pp. 694-701, 2008, DOI: 10.1109/AICCSA.2008.4493604, Available: https://ieeexplore.ieee.org/document/4493604.

[16] Hattem M. Bahig, Dieaa I. Nassr and Ashraf Bhery, "Factoring RSA Modulus with Primes not Necessarily Sharing Least Significant Bits", *Applied Mathematics and Information Sciences (AMIS)*, pp. 243-249, Vol. 11, No. 1, 2017, Published by Natural Sciences, DOI: 10.18576/amis/10130, Available: http://www.naturalspublishing.com/files/published/01cuz818m162py.pdf.

[17] Abderrahmane Nitaj, Muhammad Rezal Kamel Ariffin, Dieaa I. Nassr and Hatem M. Bahig, "New Attacks on the RSA Cryptosystem", In *Lecture Notes in Computer Science* (Progress in Cryptology – AFRICACRYPT 2014*)*, Vol. 8469, Print ISBN: 978-3-319-06733-9, Online ISBN: 978-3-319-06734-6, pp. 178-198, 2014, Springer, Cham, DOI: 10.1007/978-3-319-06734-6_12.

[18] Benne de Weger, "Cryptanalysis of RSA with Small Prime Difference", *Applicable Algebra in Engineering, Communication and Computing*, Print ISSN: 0938-1279, Online ISSN: 1432-0622, pp. 17–28, Vol. 13, 2002, DOI: 10.1007/s002000100088.

[19] Kritsanapong Somsuk, "A New Modified Integer Factorization Algorithm using Integer Modulo 20's Technique", In *2014 International Computer Science and Engineering Conference (ICSEC), 3*0 July - 1 August 2014, Khon Kaen, Thailand, pp. 312-316, Published by *IEEE*, E-ISBN**:** 978-1-4799-4963-2, DOI: 10.1109/ICSEC.2014.6978214.

[20] James Mckee, "Speeding Fermat's Factorization Method", *Mathematics of Computation*, Print ISSN 0025-5718, Online ISSN 1088-6842, pp. 1729–1737, Vol. 68, No. 228, 1999, DOI: 10.1090/S0025-5718-99-01133-3.

[21] William B. Hart, "A One Line Factoring Algorithm", *Journal of the Australian Mathematical Society*, Print ISSN 1446-7887, pp. 61-69, Vol. 92, No. 1, 2012, DOI: 10.1017/S1446788712000146.

[22] Kritsanapong Somsuk and Sumonta Kasemvilas, "MFFV2 and MNQSV2: Improved Factorization Algorithms", in *2013 International Conference on Information Science and Applications (ICISA)*, Suwon, 24-26 June 2013, pp. 1-3, Published by *IEEE*, DOI: 10.1109/ICISA.2013.6579415, Available: https://www.computer.org/csdl/proceedings-article/icisa/2013/06579415/12OmNzw8iYN.

[23] Kritsanapong Somsuk and Sumonta Kasemvilas, "MFFV3: An Improved Integer Factorization Algorithm to Increase Computation Speed", *Advanced Materials Research,* pp. 1432–1436, Vols. 931–932, 2014, DOI: 10.4028/www.scientific.net/AMR.931-932.1432.

[24] Kritsanapong Somsuk and Kitt Tientanopajai, "An Improvement of Fermat's Factorization by Considering the Last m Digits of Modulus to Decrease Computation Time", *International Journal of Network Security,* Print ISSN**:** 1816-353X, Online ISSN: 1816-3548**,** pp. 99-111, Vol. 19, No. 1, 2017, DOI: 10.6633/IJNS.201701.19, Available: http://ijns.jalaxy.com.tw/contents/ijns-v19-n1/ijns-2017-v19-n1-p99-111.pdf.

[25] Kritsanapong Somsuk, "The New Integer Factorization Algorithm Based on Fermat's Factorization Algorithm

and Euler's Theorem", *International Journal of Electrical and Computer Engineering (IJECE),* Print ISSN: 2088-8708, Online ISSN: 2722-2578, pp. 1469-1476, Vol. 10, 2020, DOI: 10.11591/ijece.v10i2, Available: http://ijece.iaescore.com/index.php/IJECE/article/view/19291/13667.

[26] Stepan Vynnychuk, Yevhen Maksymenko and Vadym Romanenko, "Application of the Basic Module's Foundation for Factorization of Big Numbers by the Fermat Method", *Eastern-European Journal of Enterprise Technologies*, Print ISSN: 1729-3774, Online ISSN: 1729-4061, pp. 14–23, Vol. *6,* No. 4 (96), 2018, DOI: 10.15587/1729-4061.2018.150870.

[27] Donald E. Knuth, *Art of Computer Programming*, Vol. 2. *Seminumerical Algorithms*, 3rd ed. Massachusetts, Addison-Wesley Professional, pp. 762, ISBN: 9780321635778.

[28] Peter Shiu, "Fermat's Method of Factorisation", *The Mathematical Gazette,* Print ISSN**:** 0025-5572, Online ISSN: 2056-6328, pp. 97–103, Vol. 99, No. 544, 2015, DOI: 10.1017/mag.2014.12, Available: https://www.jstor.org/stable/24496908?refreqid=excelsior%3A966a3fb6d9c871ff015efe2df35f0b07.

[29] Kritsanapong Somsuk and Sumonta Kasemvilas, "Possible prime modified Fermat factorization: new improved integer factorization to decrease computation time for breaking RSA", in *Advances in Intelligent Systems and Computing, International Conference on Computing and Information Technology (IC2IT2014),* Angsana Laguna, Phuket, Thailand, 8-9 May, 2014, Print ISBN: 978-3-319-06537-3, Online ISBN: 978-3-319-06538-0, Vol. 265, pp. 325-334, 2014, DOI: 10.1007/978-3-319-06538-0_32, Available: https://www.springerprofessional.de/en/possible-prime-modified-fermat-factorization-new-improved-intege/2129714.

[30] Kritsanapong Somsuk and Kitt Tientanopajai, "Improving Fermat factorization algorithm by dividing modulus into three forms", *KKU Engineering Journal,* Print ISSN 2539-6161, Online ISSN 2539-6218, pp. 350-353, Vol. 43, No. S2, 2016, DOI: 10.14456/kkuenj.2016.127, Available: https://ph01.tci-thaijo.org/index.php/easr/article/view/70248.

[31] Mu-En Wu, Raylin Tso and Hung-Min Sun, "On the improvement of Fermat factorization using a continued fraction technique", *Future Generation Computer Systems,* Print ISSN: 0167-739X, pp. 162–168, Vol. 30, January 2014, Published by ScienceDirect, DOI: 10.1016/j.future.2013.06.008, Available: https://www.sciencedirect.com/science/article/abs/pii/S0167739X13001222.

[32] Kritsanapong Somsuk, "The Improvement of Initial Value Closer to the Target for Fermat's Factorization Algorithm", *Journal of Discrete Mathematical Sciences and Cryptography,* Print ISSN: 0972-0529, Online ISSN: 2169-0065, pp. 1573-1580, Vol. 21, No. 7-8, 2018, Published by Taylor & Francis LTD, DOI: 10.1080/09720529.2018.1502737.

[33] Rasyid R.M. Tahir, Muhammad A. Asbullah, Muhammad R Ariffin and Zahari Mahad, "Determination of a Good Indicator for Estimated Prime Factor and Its Modification in Fermat's Factoring Algorithm", *Symmetry*, Online ISSN: 2073-8994, *V*ol. 13, No. 5, 2021, Published by Multidisciplinary Digital Publishing Institute (MDPI), DOI: 10.3390/sym13050735, Available: https://www.mdpi.com/2073-8994/13/5/735.

[34] Paul Ryan A. Longhas, Alsafat M. Abdul and Aurea Z. Rosal, "Factors of Composite $4n^2 + 1$ using Fermat's Factorization Method", *International Journal of Mathematics Trends and Technology*, Print ISSN: 2349-5757, Online ISSN: 2231-5373, Vol. 68, No. 1, pp 53-60, 2022, DOI: 10.14445/22315373/IJMTT-V6811P506, Available: https://www.ijmttjournal.org/archive/ijmtt-v68i1p506.

[35] Hazem M. Bahig, Mohammed A. Mahdi, Khaled A. Alutaibi, Amer AlGhadhban and Hatem M. Bahig, "Performance Analysis of Fermat Factorization Algorithms", *International Journal of Advanced Computer Science and Applications (IJACSA),* Print ISSN: 2158-107X , Online ISSN: 2156-5570, Vol. 11, No. 12, 2020, Published by Science and Information Organization (SAI), DOI: 10.14569/IJACSA.2020.0111242, Available: https://thesai.org/Publications/ViewPaper?Volume=11&Issue=12&Code=IJACSA&SerialNo=42.

[36] Hazem Bahig, Hatem Bahig and Yasser Kotb, "Fermat Factorization using a Multi-Core System", *International Journal of Advanced Computer Science and Applications (IJACSA)*, Print ISSN: 2158-107X , Online ISSN: 2156-5570, Vol. 11, No. 4, 2020, Published by SAI, DOI: 10.14569/IJACSA.2020.0110444, Available: https://thesai.org/Publications/ViewPaper?Volume=11&Issue=4&Code=IJACSA&SerialNo=44.