

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/355479631>

Reduce the Complexity of Big Number Factoring for RSA Breaking

Article · October 2021

CITATIONS

0

READS

348

1 author:



[Kanwal Oad](#)

Southeast Missouri State University

2 PUBLICATIONS 5 CITATIONS

[SEE PROFILE](#)

Some of the authors of this publication are also working on these related projects:



Cyber Security [View project](#)

Reduce the Complexity of Big Number Factoring for RSA Breaking

By Kanwal Oad

Submitted in partial fulfillment of the requirements for the thesis,

Master of Applied Computer Science

in the "Harrison College of Business and Computing" at

Southeast Missouri State University

Under the supervision of

DR. ZHOUSHOU LI

March 2021

Acceptance Sheet

ACCEPTANCE SHEET FOR GRADUATE THESIS, NON-THESIS PAPER, OR CREATIVE PROJECT SOUTHEAST MISSOURI STATE UNIVERSITY			
PLEASE SELECT APPROPRIATE DEGREE OPTIONS:			
<input type="checkbox"/> MASTER OF ARTS	<input type="checkbox"/> MASTER OF NATURAL SCIENCE		
<input checked="" type="checkbox"/> MASTER OF SCIENCE	<input type="checkbox"/> MASTER OF PUBLIC ADMINISTRATION		
The committee in charge of the graduate work of	<u>Kanwal Oad</u>	<u>2010412</u>	
	Student Name	Student ID (S0#)	
has supervised the	<input checked="" type="checkbox"/> THESIS	<input type="checkbox"/> NON-THESIS PAPER	<input type="checkbox"/> CREATIVE PROJECT
for the master's degree selected above and reports as follows:			
Title:	<u>Reduce the complexity of big number factoring for RSA breaking</u>		
Area of Emphasis:	<u>High Performance Computing</u>		
Complementary Area(s):	<u>Computer Science</u>		
We recommend the completion of this graduate work <input checked="" type="checkbox"/> BE ACCEPTED <input type="checkbox"/> NOT ACCEPTED in partial fulfillment of the master's degree selected above.			
COMMITTEE MEMBERS			
1.	<u>Zhouzhou Li</u> Committee Chairperson Printed Name	<u>Zhouzhou Li</u> Signature <small><i>Digitally signed by Zhouzhou Li Date: 2021.03.24 18:08:17 +05'00'</i></small>	<u>03/24/2021</u> Date (mm/dd/yyyy)
2.	<u>Xiaoming Liu</u> Committee Member Printed Name	<u>Xiaoming Liu</u> Signature <small><i>Digitally signed by Xiaoming Liu Date: 2021.03.24 22:17:27 +05'00'</i></small>	<u>03/24/2021</u> Date (mm/dd/yyyy)
3.	<u>Jerzy Wojdylo</u> Committee Member Printed Name	<u>Jerzy Wojdylo</u> Signature <small><i>Digitally signed by Jerzy Wojdylo Date: 2021.03.25 17:31:34 +05'00'</i></small>	<u>03/24/2021</u> Date (mm/dd/yyyy)
APPROVAL			
	<u>Charles D. McAllister</u> Department Chairperson Printed Name	<u>Charles D. McAllister</u> Signature <small><i>Digitally signed by Charles D. McAllister Date: 2021.03.28 12:58:14 +05'00'</i></small>	<u>03/28/2021</u> Date (mm/dd/yyyy)
	<u>Doug Koch</u> Dean of Graduate Studies Printed Name	<u>Doug Koch</u> Signature <small><i>Digitally signed by Doug Koch Date: 2021.04.22 12:41:35 +05'00'</i></small>	<u>04/22/2021</u> Date (mm/dd/yyyy)
CC: Graduate Studies; Committee Chairperson; Coordinator; Department; Student			

Acknowledgments

First and foremost, I want to thank my research supervisor Dr. Zhouzhou Li for his valuable assistance, dedicated involvement, encouragement and help for completing this work. Without his assistance this paper would have never been accomplished. His support was crucial to this work and he shared a substantial part in my achievement.

I would like to extend my thanks to my committee members Dr. Xiaoming Liu and Dr. Jerzy Wojdylo for generously offering their time, support, guidance and good will during the preparation and review of this document.

Lastly, I am highly indebted to Southeast Missouri State university and the Department Chair of Computer Science Department Dr. McAllister Charles, for his continuous support throughout the process.

Kanwal Oad

Table of Contents

Acceptance Sheet.....	ii
Acknowledgments	iii
List of Tables	v
List of Figures.....	vi
Abstract.....	viii
1. INTRODUCTION.....	1
1.1. RSA Algorithm	1
1.1.1. RSA Key Generation	2
1.1.2. Encryption.....	3
1.1.3. Decryption.....	3
1.2. Large Number Factoring.....	3
1.3. Algorithm Complexity	6
2. EXISTING WORK.....	7
2.1. Basic Algorithm	7
2.2. Sieve 6K Algorithm - 6 continuous points/group	8
2.3. Sieve 30K Algorithm - 30 Continuous Points/Group	9
2.4. Fermat's Algorithm.....	9
2.5. Fermat's Sieve Algorithm.....	11
3. PROPOSAL.....	12
3.1. Discussion	14
4. EXPERIMENTS	15
4.1. Basic Algorithm	15
4.2. Sieve 6K Algorithm	16
4.3. Sieve 30K Algorithm	18
4.4. Fermat's Algorithm.....	22
4.5. Fermat's Sieve Algorithm.....	24
4.6. Comparison Between 5 Algorithms	25
5. CONCLUSION AND FUTURE WORK	26
REFERENCES.....	27
Appendix A	29

List of Tables

Table 1. Explanation of 6K Factoring18

Table 2. Explanation of 30K Factorization.....21

Table 3. Please see Table 3 representing Fermat’s Algorithm, which shows the summarized different values of n and their results.24

List of Figures

Figure 1. Example of Encryption and Decryption	2
Figure 2. The efficiency of the basic algorithm	6
Figure 3. The efficiency of the Sieve 6K algorithm	6
Figure 4. The efficiency of the Sieve 30K algorithm	7
Figure 5. The efficiency of the Fermat's algorithm	7
Figure 6. The efficiency of the Fermat's Sieve algorithm	7
Figure 7. Search direction and the range of basic algorithm	8
Figure 8. Enhancement of Sieve 6K algorithm over basic algorithm: jumping effect.....	9
Figure 9. Illustration of the relationship between i and j	11
Figure 10. Illustration of the relationship between i and j and search direction of Fermat's Sieve algorithm.....	11
Figure 11. Effect of searching for j.....	14
Figure 12. Basic Algorithm code in C language.....	15
Figure 13. Basic Algorithm Output	16
Figure 14. Sieve 6K Algorithm code in C language - Part 1	16
Figure 15. Sieve 6K Algorithm code in C language - Part 2	17
Figure 16. Sieve 6K Algorithm Output	17
Figure 17. Sieve 30K Algorithm code in C language - Part 1	18
Figure 18. Sieve 30K Algorithm code in C language - Part 2	19
Figure 19. Sieve 30K Algorithm code in C language - Part 3	20
Figure 20. Output of Sieve 30K Algorithm	20
Figure 21. Fermat's Algorithm code in C language	22
Figure 22. Output of Fermat's Algorithm.....	22
Figure 23. Mod 16 code in Python	23
Figure 24. Mod 16 Output	23
Figure 25. Mod 100 code in Python	23
Figure 26. Mod 100 Output	24

Figure 27. Output of Fermat's Sieve Algorithm.....	25
Figure 28. Comparison between 5 algorithms	25

Abstract

The Rivest-Shamir-Adleman (RSA) is a public key cryptosystem widely used in online transactions over the Internet to secure the data transmission. The security of RSA relies on the computational complexity of the large integer factorization problem, for which no efficient algorithm is known. This paper proposes an easy and efficient n factoring algorithm for RSA breaking. The first step is to reduce the loop times of the algorithm (such as from $O(n)$ to $O(\sqrt{n})$). The preliminary study is that $p \times q = n$ can be transformed into a Diophantine equation. And it has been performed by using different integer factorization algorithms in C language. The second step is to speed up the computing in each iteration. To support the second step, Field Programmable Gate Array (FPGA) it may need to use to implement a customized multiple-core processor, which can do parallel computing for the calculation. The focus of this paper is on the first step, reducing the loop times of the algorithm, and that process has been presented in a more visualized way.

Keywords: RSA, Integer factorization, Large number factoring, Cyber Security, Primality test, Fermat's Algorithm, C language.

1. INTRODUCTION

This paper discusses five factoring algorithms such as: basic integer factoring algorithm, Sieve 6K Algorithm, Sieve 30K Algorithm, Fermat's Algorithm, and Fermat's Sieve Algorithm to check their efficiency for factorization of an integer number. This paper starts by explaining the basic integer factoring algorithm and then discusses about the Sieve 6K Algorithm and Sieve 30K Algorithm. Sieve 6K and 30K Algorithms are types of factoring algorithms, which finds all the factors less than \sqrt{n} . In Sieve 6K, it slices the searching range into equal size units, finds factors in six continuous points and skips some points to speed up the process, however in Sieve 30K, in every unit, it finds factors in 30 continuous points. It works the same as 6K because it skips the points to speed up the process. Later, the paper will explain the Fermat's Algorithm and Fermat's Sieve Algorithm. Fermat's Algorithm is another type of factoring algorithm which is based on the illustration as the difference of two squares. The Fermat's Sieve Algorithm is the advanced version of Fermat's Algorithm, which skips some points to speed up the process of finding factors. Lastly, this paper describes the proposed system for large number factoring and performs both qualitative and quantitative research methods. In the existing work section, the qualitative approach has been presented and in experiments section the quantitative approach has been presented.

1.1. RSA Algorithm

The Rivest, Shamir and Adleman (RSA) Algorithm [1] was invented by Rivest, Shamir and Adleman in 1977, which is today's most famous asymmetric public key cryptosystem in the world. The RSA is the replacement of National Bureau of Standards (NBS) algorithm. This algorithm is used for providing privacy, integrity and ensuring legitimacy of digital data, i.e., web servers, browsers, emails, login systems and credit cards, etc. While symmetric keys are more efficient than asymmetric keys, they face problems when it comes to key distribution, whereas asymmetric key does an excellent job in providing the key distribution.

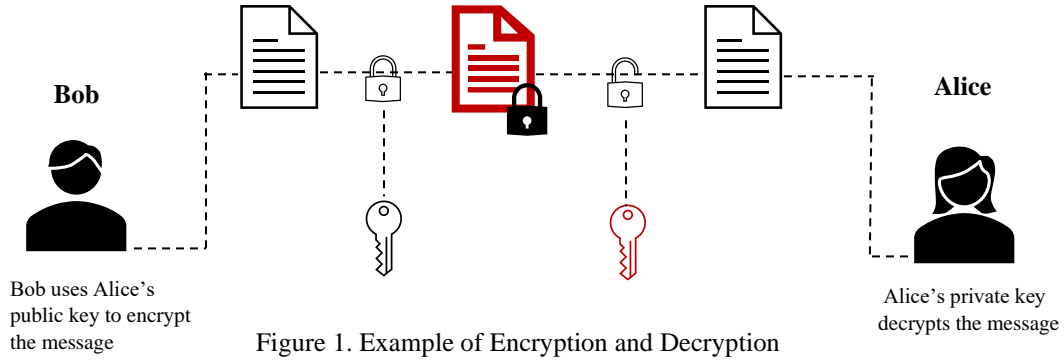


Figure 1. Example of Encryption and Decryption

The above, Figure 1, illustrates a simple example of asymmetric encryption and decryption, where Alice shares her public key with Bob first and then Bob can send an encrypted message to Alice. Bob uses Alice's public key to encrypt the message and Alice decrypts the message using her own private key, which only she knows. The RSA does not only implement the public key cryptosystem, but it also implements digital signature and can also be used to sign a message. Hence, Bob can sign a message using his private key and Alice can verify it using Bob's public key, which was shared with Alice in advance [1].

The RSA is very crucial to use in cloud environment to secure the data due to the critical characteristics of cloud computing and the large amount of complicated data it transmits, whereas a public key is known to everyone, the private key is only known to those who own the data. In the cloud environment, the encryption is completed by a cloud service provider and the decryption is completed by the consumer.

The RSA is a very important part of the Bitcoin success, as well. Bitcoin is a decentralized digital currency where each member can see the transactions and store the ledger in their computers. The technology guarantees data integrity and makes it hard to be modified. The transactions are verified by network nodes through the signatures using the public keys, and private keys are not at all disclosed [2].

This algorithm has three steps: key generation, encryption, and decryption. Where p , q represents two different big prime numbers, e represents encryption key, d represents decryption key, PU represents public key and PR represents the private key [3, 4].

1.1.1. RSA Key Generation

Step 1. Select p , q , such that should be two different but close prime numbers

Step 2. Calculate $n = p.q$

Step 3. Calculate $\phi(n) = (p - 1) (q - 1)$

Step 4. Select integer e , such that $1 < e < \phi(n)$, and such that e and $\phi(n)$ share no divisors other than 1.

Step 5. Choose d , satisfying $de \pmod{\phi(n)} = 1$

Step 6. Public key PU = $[e, n]$

Step 7. Private key PR = $[d, n]$

1.1.2. Encryption

Encryption is a method of converting the plaintext into ciphertext, where M represents the message or message block, and M should be less than n , such that ($M < n$).

Step 1. Plaintext: M

Step 2. Ciphertext: $C = (M^e \pmod n)$

1.1.3. Decryption

Decryption is the method of converting the ciphertext into plaintext, where C represents cipher text.

Step 1. Plaintext: C

Step 2. Ciphertext: $M = (C^d \pmod n)$

1.2. Large Number Factoring

RSA algorithm is used to secure electronic transmission of data. The security of the RSA public key algorithm depends on the complexity of factoring a big number n , which is the product of two alike size prime p and q . If the RSA key has a large number of bits in key, it is more difficult to crack, even though the RSA can be broken by the integer factorization. Thus, any progress in big number factorization will attract the attention of cryptographers [5].

Breaking an integer number down into a group of numbers whose product is equal to the original number is called factoring of an integer number. Factoring a small integer number is easy, i.e., factors of the number 15 can be 5 and 3. But when it comes to factoring a big integer number which is the production of two large numbers, the process is futile, due to the time it will take

utilizing even today's computers. Big integer number factorization is one of the basic problems in number theory. It has been researched for more than ten decades and has been an interesting subject for mathematicians, computer scientists and cryptographers in recent years, but no one has discovered an efficient algorithm yet [6, 7]. There are numerous factoring algorithms which have been used over the centuries to factor an integer number. Some are discussed below.

Trial division

Trial division is the simplest way for integer factorization. It was initially explained by Fibonacci in 1202 in Latin. Later, after decades it was translated by other mathematicians into English. The purpose of the Trial and Division Algorithm is to determine if the given number can be factored. To demonstrate, 5 is a prime number, divide 5 by primes less than 5 (in this example, 2 and 3). Since none of these divides 5 then 5 is a prime number [8].

Fermat's Algorithm

Fermat's Algorithm was described by a French mathematician, Pierre De Fermat (1607 - 1665). The Fermat's Factorizing Algorithm is based on the illustration of an odd integer as the difference of two squares. $n = m^2 - b^2$ where the right side of the equation will be factorized into $(m + b)(m - b)$ [9].

Example:

Input: $n = 187$

Output: [17, 11]

Explanation:

Try value of "m" as a ceil value of square root of 187 ($m > \sqrt{n}$), which will be 14.

$$n = m^2 - b^2 = (m + b)(m - b)$$

$$187 = m^2 - b^2 = (m + b)(m - b)$$

$$187 = 14^2 - b^2 = (m + b)(m - b)$$

Then,

$$b^2 = 14^2 - 187 = 9$$

So,

$$b = 3$$

$$187 = 14^2 - 3^2 = (m + b)(m - b)$$

So, the factors of 187 are:

$$187 = 14^2 - 3^2 = (14 + 3)(14 - 3)$$

$$187 = 14^2 - 3^2 = 17 * 11$$

Euler's Algorithm

Euler's algorithm was invented by another French mathematician, Leonhard Euler, in the late 17th century. Euler's algorithm looks complex, however the concept behind it is simple. It uses the differential equation, $y_{n+1} = y_n + hf(t_n, y_n)$, where the equation on the left side is the approximate solution value, y_n is the present value, h is the gap between the steps and $f(t_n, y_n)$ is the derivate value [10].

Example:

$$y_{n+1} = y_n + hf(t_n, y_n)$$

First need to calculate $f(t_0, y_0)$

$$f(t_0, y_0) = f(0, 1) = 1$$

Multiply the above given value by the step size of h ,

$$h \cdot f(y_0) = 1 * 1 = 1$$

Then the value is added to the original y value to get the new value,

$$y_0 + hf(y_0) = y_1 = 1 + 1 * 1 = 2$$

Then above steps need to be repeated to find y_2, y_3, y_4 ,

$$y_2 = y_1 + hf(y_1) = 2 + 1 * 2 = 4$$

$$y_3 = y_2 + hf(y_2) = 4 + 1 * 4 = 8$$

$$y_4 = y_3 + hf(y_3) = 8 + 1 * 8 = 16$$

Use of Machines in Factoring Integer Number

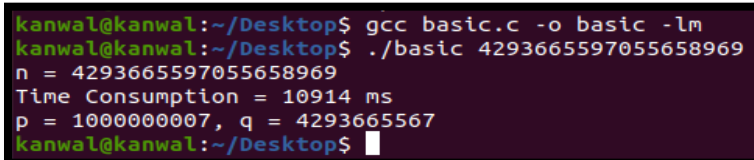
In the 20th century, numerous advancements took place which expedited the process of factoring integer numbers. In 1926, a mechanical machine named, Bicycle Chain Sieve, was introduced by an American mathematician, Derrick Henry Lehmer. This machine was built with a cylinder on which so many chains were attached of different sizes, and if the rods were at the right point in the circuit, they would shut down all the electrical switches. Thus, when all electrical switches were turned off, it created a complete electrical switch, which means the solution of one problem was found [11].

With the advancement in cryptography, research in factoring numbers has been revived. Around 1970, public key cryptography was introduced by American scientists. Public key cryptography is asymmetric encryption technique which uses two mathematically different keys: public key and private key. Cryptography is used to provide security in any kind of online transaction and the security of public key cryptography relies on how complex the factorization of the integer is [12].

1.3. Algorithm Complexity

Algorithm complexity is mathematically represented by a big O notation or $O(n)$, where n is the size of elements. It shows the performance of the algorithm by measuring the order of the consumed resources, like computer consumption time for example. In an algorithm complexity, $O(1)$ are constant. Expression $O(n)$ takes more time to run the algorithm; this expression type is known as linear, which means if the size of n grows so does the number of operations. In the quadratic expression, $O(n \wedge 2)$, one must look for each input. The expression, $O(\log n)$, looks at the fraction of the input. The expression, $O(n \log n)$, follows a divide and conquer rule. It uses a loop within a loop. The expression, $O(2 \wedge n)$, solves the problem recursively, and the expression, $O(n!)$, is not known as good as the other expressions. The algorithms with constant $O(1)$, linear $O(n)$ and logarithmic $O(\log n)$ complexity are much faster options [13]. The following are the examples of algorithm complexity which are covered in this paper:

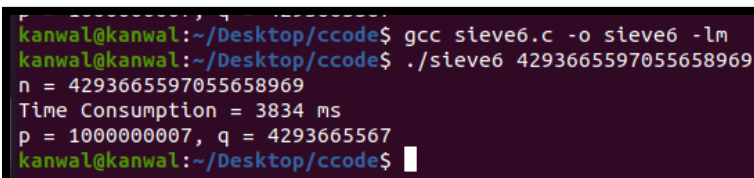
- Basic Algorithm: Can be represented by the expression $O(\sqrt{n} - p)$



```
kanwal@kanwal:~/Desktop$ gcc basic.c -o basic -lm
kanwal@kanwal:~/Desktop$ ./basic 4293665597055658969
n = 4293665597055658969
Time Consumption = 10914 ms
p = 1000000007, q = 4293665567
kanwal@kanwal:~/Desktop$
```

Figure 2. The efficiency of the basic algorithm

- Sieve 6K Algorithm: Can be represented as the expression $O((\sqrt{n} - p) * 2/6)$



```
kanwal@kanwal:~/Desktop/ccode$ gcc sieve6.c -o sieve6 -lm
kanwal@kanwal:~/Desktop/ccode$ ./sieve6 4293665597055658969
n = 4293665597055658969
Time Consumption = 3834 ms
p = 1000000007, q = 4293665567
kanwal@kanwal:~/Desktop/ccode$
```

Figure 3. The efficiency of the Sieve 6K algorithm

- Sieve 30K Algorithm: Can be represented as the expression $O((\sqrt{n} - p) * 8/30)$. Theoretically unit size can be increased to further improve the sieve efficiency. However, in practice, more judgement statements will be introduced to the code along with the unit size increasing, which will slow down the algorithm.

```
kanwal@kanwal:~/Desktop/ccode$ gcc sieve30.c -o sieve30 -lm
kanwal@kanwal:~/Desktop/ccode$ ./sieve30 4293665597055658969
n = 4293665597055658969
Time Consumption = 2974 ms
p = 1000000007, q = 4293665567
```

Figure 4. The efficiency of the Sieve 30K algorithm

- Fermat's Algorithm: Can be represented as the expression $O(m - \sqrt{n})$. Here m represents the mean value.

```
kanwal@kanwal:~/Desktop/comcode/ccode$ gcc fermat.c -o fermat -lm
kanwal@kanwal:~/Desktop/comcode/ccode$ ./fermat 17778473929466109103
n = 17778473929466109103, S0 = 4216452766, n-S0*S0 = 1557058347
Time Consumption = 12 ms
p = 4139373533, q = 4294967291
```

Figure 5. The efficiency of the Fermat's algorithm

- Fermat's Sieve Algorithm: Can be represented as the expression $O(m - \sqrt{n})$. Here m represents the mean value and K is a constant which is decided by the sieve and n .

```
Fermat Sieve Algorithm
n = 17778473929466109103, S0 = 4216452766, n-S0*S0 = 1557058347
Time Consumption = 5 ms
p = 4139373533, q = 4294967291
```

Figure 6. The efficiency of the Fermat's Sieve algorithm

2. EXISTING WORK

2.1. Basic Algorithm

According to Rivest, Shamir, and Adleman (1978), integer factoring is important for several reasons, one of them is its main role in numerous cryptographic methods. A prime number is an integer greater than 1 and divisible only by itself and 1, and a composite number is a product of several factor numbers such as $14 = 2 * 7$. In prime factorization $n = p.q$, and the value of p should be smaller than the value of q . Composite factoring is considered to be difficult, but this is not the

case with the small composites. Currently, the factoring ability threshold lies down around 130 decimal digits. The graph below illustrates the search direction and range of basic algorithm [14].

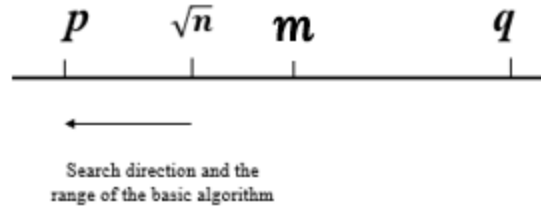


Figure 7. Search direction and the range of basic algorithm

2.2. Sieve 6K Algorithm - 6 continuous points/group

The Sieve 6K algorithm, according to a primality test example from Wikipedia, finds all the factors less than or equal to \sqrt{n} . It jumps from one integer to another and skips those numbers which possibly cannot be prime [15]. It is the simplest method to implement but not the quickest if the range is more than 6K. All prime numbers are in the form of $6K \pm 1$, however not all of the numbers which are a form of $6K \pm 1$ are primes. Since the number 6 has the factors 2 and 3, one need to divide the x with each of the factor to check whether x is prime or not. Sieve 6K loop scans $2/6$ (=33%). The Sieve 6K range can be written as $6K, 6K \pm 1, 6K \pm 2, 6K \pm 3, 6K \pm 4, 6K \pm 5$. This method checks whether the given number x is prime or not. But it does not itself tells if the number is prime, a person must verify that after the end of the process.

Example: $6K+1$, where $k = 1, 2 \cdot 3 \cdot K + 1, 7 \% 2 == 1$.

Since $x, 7$ is not divisible by 2 or 3 it can be assumed that it could be a prime.

Theoretically, more and more points can be skipped if more and more smaller primes are introduced to the algorithm. However, in practice, there are two problems. The first one is how to construct the program to implement the “customized” algorithm? The second one is, with the introduction of more and more judgement statements, the overall program efficiency will decrease. Experiments can be constructed (by using C, Java, and Python programs) to verify the upper limitation of this category of algorithms [15].

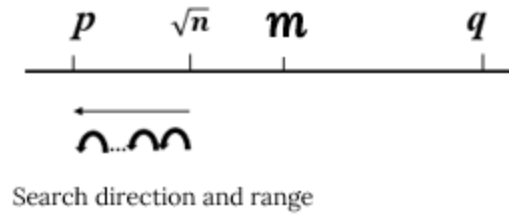


Figure 8. Enhancement of Sieve 6K algorithm over basic algorithm: jumping effect

The figure above illustrates the intuitive enhancement of the 6K Sieve Algorithm over the basic algorithm, where the search direction of both algorithms is the same but the Sieve 6K Algorithm is skipping some points, which is making Sieve 6K Algorithm more efficient.

2.3. Sieve 30K Algorithm - 30 Continuous Points/Group

The Sieve 30K Algorithm is the enhanced version of the Sieve 6K Algorithm, where the process is the same but Sieve 30K searches in 30 continuous points. Sieve 30K loop scans 8/30 (=26.6%) numbers, so 30K scans 20% less numbers than 6K Algorithm and one can expect a speed up of at best 20% as compared to 6K Algorithm. The Sieve 30K range can be written as $30K \pm 0$, $30K \pm 1$, $30K \pm 2$, $30K \pm 3$, $30K \pm 4$, $30K \pm 29$. Since number 30 has the factors 2, 3, and 5, one need to divide the x with each of the factors to check whether x is prime or not [15].

Example: $30K-12$, where $K = 1$, $2*3*5*K-12 = 18$, $18 \% 5 == 1$.

Since x , 18 is divisible by 2 and 3, one can say that it is not a prime.

2.4. Fermat's Algorithm

Fermat's algorithm, according to Barnes (2004), is one of the popular factorization methods. Its foundation is on quadratic disputes, originated by a French Mathematician, Pierre de Fermat, in 1643. It is well suited for factoring smaller composite numbers. Fermat's method of factoring takes polynomial time and is recursive [16]. Fermat's method works best when there is a factor close to the square root of n .

The equation is $n = m^2 - b^2$ where right hand side equals to $(m + b)(m - b)$. Fermat's Algorithm is fast and easy to implement and it is more efficient than the basic algorithm

of factorization, because it takes less running time and makes composite numbers of each of the prime numbers starting at the first prime number which is 2. In Fermat's algorithm, the search direction is from m mean value to q . It is based on finding m and b such that $n = m^2 - b^2$ [16].

For example: $n = 5959$ and $m = 80$.

$$n = m^2 - b^2 = (m + b)(m - b)$$

$$5959 = m^2 - b^2 = (m + b)(m - b)$$

$$5959 = 80^2 - b^2 = (m + b)(m - b)$$

$$b^2 = 80^2 - 5959 = 441$$

$$b = 21$$

$$5959 = 80^2 - 21^2 = (m + b)(m - b)$$

$$5959 = 80^2 - 21^2 = (80 + 21)(80 - 21)$$

$$5959 = 80^2 - 21^2 = 101 * 59$$

Hence, the factors of 5959 will be 101 and 59. Let us do another example with the same value of n but different value of m . In following example, the exact factors were not achieved of 5959 [16].

For example: Given $n = 5959$, $m = 78$

$$n = m^2 - b^2 = (m + b)(m - b)$$

$$5959 = m^2 - b^2 = (m + b)(m - b)$$

$$5959 = 78^2 - b^2 = (m + b)(m - b)$$

$$b^2 = 78^2 - 5959 = 125$$

$$b = 11.18$$

$$5959 = 78^2 - 11.18^2 = (m + b)(m - b)$$

$$5959 = 78^2 - 11.18^2 = (78 + 11.18)(78 - 11.18)$$

$$5959 = 78^2 - 11.18^2 = (89.18)(66.82)$$

The following figure illustrates the relationship between j and i . At the beginning, the curve is steep (the highlighted part). The m represents mean value.

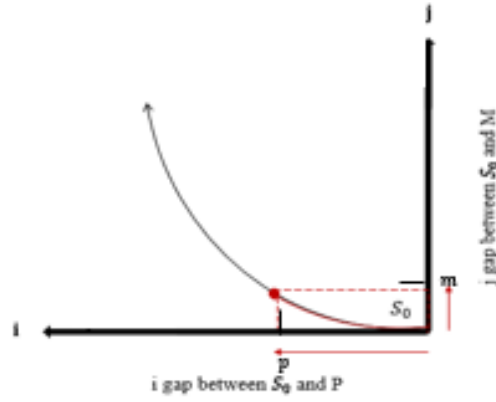


Figure 9. Illustration of the relationship between i and j

2.5. Fermat's Sieve Algorithm

Fermat's Sieve Algorithm is the improved version of Fermat's Algorithm, and skipping the points makes it different from Fermat's Algorithm. Its equation is the same as Fermat's Algorithm, $n = m^2 - b^2 = (m + b)(m - b)$, and it follows all the same steps as Fermat's Algorithm follows to find the factors, but it does not essentially calculate all the square-roots of $m^2 - n$. Also, it does not observe all the values of m . The search direction of Fermat's Sieve Algorithm is mean m value to q like Fermat's algorithm [16].

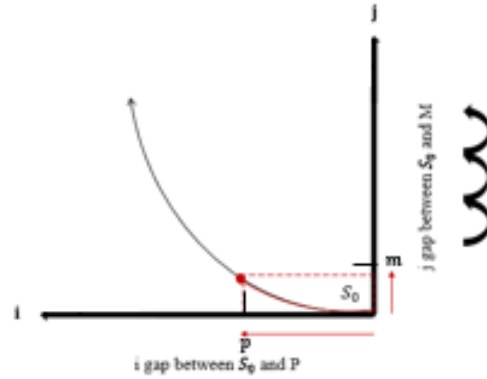


Figure 10. Illustration of the relationship between i and j and search direction of Fermat's Sieve algorithm

According to Kostopoulos (2016), integer factoring is a computationally challenging problem. It is easy for smaller numbers but if one took larger numbers the computation can take centuries to solve the problem [17]. Many mathematicians have worked on this issue, but no one has been able to invent the algorithm which can work faster for the larger numbers. The factoring algorithms like Quadratic Sieve, Fermat's and other numerous algorithms are efficient and fast on

a level, however when it comes to larger numbers, all the factoring algorithms fail. The author concludes the study by saying, “There are no well-known algorithms which can factor larger integers effectively”. Integer factorization is the main pillar of cryptosystem, therefore any new progress in the integer large number factorization produces a new research break for mathematicians and researchers.

3. PROPOSAL

The objective of this research is to provide an easy and efficient n factoring algorithm for RSA breaking. One need to consider the engineering practicality aspect of the algorithm. The draft idea of the proposal has two steps. The first step is to reduce the loop times of the algorithm, such as from $O(n^2)$ to $O(n)$. The second step is to speed up the computing in each iteration. To support the second step, Field Programmable Gate Array (FPGA) may need to use to implement a customized multiple-core processor, which can do parallel computing for the calculation. For now, the focus is on step one.

Regarding step one, the preliminary study is that $p \times q = n$ can be transformed into a Diophantine equation. Let $S_0 = \text{floor}(\sqrt{n})$, which can be calculated from n directly. Also, let $m = (p + q)/2$, which is unknown, and $S_0 < m$ can be proved True. Also, i has been used to represent the gap between p and S_0 , j to represent the gap between S_0 and m , then one can get the following:

$$p + i = S_0 \tag{2}$$

$$m = (p + q)/2 \tag{3}$$

$$S_0 + j = m \tag{4}$$

With (2)-(4), Equation (1) can be transformed as below:

$$\begin{aligned} p \times q = n &\Rightarrow (S_0 - i) \times q = n \Rightarrow (S_0 - i)(2m - p) = n \Rightarrow (S_0 - i)(2S_0 + 2j - S_0 + i) = n \\ &\Rightarrow S_0^2 - i^2 + 2S_0j - 2ij = n \Rightarrow i^2 + 2ji - 2S_0j + n - S_0^2 = 0 \Rightarrow (i + j)^2 - j^2 - 2S_0j + n - S_0^2 = 0 \end{aligned}$$

$$\Rightarrow (i + j)^2 = j^2 + 2S_0j + S_0^2 - n \quad (5)$$

Equation (5) is the Diophantine equation, where i and j are the two uncertain integer variables.

From (5) one can get

$$i + j = \sqrt{j^2 + 2S_0j + S_0^2 - n} \quad (6)$$

To continue, the idea is to increase j step by step, starting from 1 until one can find the correct m .

The estimation of m in each step is $S_0 + j$. Whether the estimation is the correct m is decided by function $Mevaluate(S_0 + j)$. The equivalent evaluation can be based on Equation (6), i.e., whether $\sqrt{j^2 + 2S_0j + S_0^2 - n}$ is an integer. The algorithm is as below, which is quite simple.

```
j = 1;
while(! Mevaluate(S0 + j))
j ++;
```

Figure 11 illustrates the effect of searching for j , the gap between S_0 and m : As one can see, one step movement at j dimension is equivalent to multiple step movement at i dimension (the gap between p and S_0). The basic algorithm is based on searching for i . One can have a draft idea about the algorithm complexity by comparing this proposal to the basic algorithm. In Figure 11, this proposal exits the loop after 174 attempts, while the basic algorithm exits the loop after 1343 attempts.

```
j = 1 i = 89.4101764184
j = 2 i = 143.797805196
j = 3 i = 182.315946427
j = 4 i = 213.782001093
j = 5 i = 241.004065007
j = 6 i = 265.309786038
j = 7 i = 287.452033445
j = 8 i = 307.906631776
j = 9 i = 326.997023796
j = 10 i = 344.954926716
j = 11 i = 361.953080159
j = 12 i = 378.124339154
j = 13 i = 393.573486593
j = 14 i = 408.384895563
j = 15 i = 422.62769565
j = 16 i = 436.359370413
j = 17 i = 449.628331759
j = 18 i = 462.475805843
```

j = 157	i = 1282.89652406
j = 158	i = 1286.54041134
j = 159	i = 1290.17010734
j = 160	i = 1293.78574763
j = 161	i = 1297.38746566
j = 162	i = 1300.97539282
j = 163	i = 1304.54965844
j = 164	i = 1308.11038988
j = 165	i = 1311.65771254
j = 166	i = 1315.19174991
j = 167	i = 1318.71262363
j = 168	i = 1322.22045349
j = 169	i = 1325.71535752
j = 170	i = 1329.19745197
j = 171	i = 1332.6668514
j = 172	i = 1336.12366867
j = 173	i = 1339.568015
j = 174	i = 1343.0
j = 175	i = 1346.4197317
j = 176	i = 1349.82731657
j = 177	i = 1353.22285959
j = 178	i = 1356.60646421
j = 179	i = 1359.97823246

Figure 11. Effect of searching for j

3.1. Discussion

This proposal is simple and efficient, however, before one can claim it is as effective, one need to address the following issues:

1. Based on the above figure, “One step movement at j dimension is equivalent to multiple step movement at i dimension”, is only true at the beginning part of the curve (highlighted). In RSA algorithm, the gap between p and q should not be much, which probably can guarantee the above statement. However, one will need to quantify the gap between p and q to see when one can apply this approach.
2. “Whether $\sqrt{j^2 + 2S_0j + S_0^2} - n$ is an integer” is a more complicated judgement than “Whether $n\%(S_0 - i)$ is 0”, one will need to figure out a way to speed up this judgement (in the second step of this research).

4. EXPERIMENTS

4.1. Basic Algorithm

Without losing the generality, one can assume $p < q$. Following is the basic algorithm in C language.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>

long long cur_ts() {
    struct timeval te;
    long long milliseconds;
    // get current time
    gettimeofday(&te, NULL);
    // calculate milliseconds
    milliseconds = te.tv_sec*1000LL +
te.tv_usec/1000;
    // printf("milliseconds: %lld\n",
milliseconds);
    return milliseconds; }

void main(int argc, char * argv[]) {
    long long start_ts, end_ts;
    unsigned long long n, q;
    unsigned long p;
    unsigned long S0;
    unsigned long i;
    char errFlag = 0;
    p = 0;
    q = 0;
    start_ts = cur_ts();
    if ( argc < 2 ) {
        errFlag = 1;
        goto TIME_CONSUMPTION; }
    n = strtoull(argv[1], NULL, 10);
    . . .

    if ( n <= 1 ) {
        errFlag = 2;
        goto TIME_CONSUMPTION; }
    printf("n = %llu\n", n);

    S0 = floor(sqrt((double)n));
    // printf("DBG 1: %lu\n", S0);
    for ( i = S0; i > 0; i -- ) {
        if ( (n%i) == 0 ) {
            p = i;
            q = n/i;
            goto TIME_CONSUMPTION; } }
    // printf("DBG 2: %lu, %lu\n", p, q);

    TIME_CONSUMPTION:
    end_ts = cur_ts();
    printf("Time Consumption = %lld ms\n",
end_ts-start_ts);
    switch ( errFlag ) {
    case 1:
        printf("Usage:\n");
        printf("$ gcc basic-u64b.c -o basic-u64b -
lm\n");
        printf("$ ./basic-u64b n\n");
        return;
    case 2:
        printf("Invalid n!\n");
        return;
    default:
        printf("p = %lu,\nq = %llu\n", p, q);
        return; } }
```

Figure 12. Basic Algorithm code in C language

The result of basic algorithm is given below. One can see that the time consumption in the example is 10914 milli seconds.


```

kanwal@kanwal:~/Desktop$ gcc basic.c -o basic -lm
kanwal@kanwal:~/Desktop$ ./basic 4293665597055658969
n = 4293665597055658969
Time Consumption = 10914 ms
p = 1000000007, q = 4293665567
kanwal@kanwal:~/Desktop$

```

Figure 13. Basic Algorithm Output

4.2. Sieve 6K Algorithm

An enhancement to the basic algorithm is to skip composite points to speed up the searching by using Sieve 6K Algorithm. The example is given below in C language.

```

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>

long long cur_ts() {
    struct timeval te;
    long long milliseconds;

    // get current time
    gettimeofday(&te, NULL);

    // calculate milliseconds
    milliseconds = te.tv_sec*1000LL +
te.tv_usec/1000;
    // printf("milliseconds: %lld\n", milliseconds);
    return milliseconds; }

void main(int argc, char * argv[]) {
    long long start_ts, end_ts;
    unsigned long long n, q;
    unsigned long p;
    unsigned long S0;
    unsigned long i;
    char errFlag = 0;

```

```

p = 0;
q = 0;

start_ts = cur_ts();
if ( argc < 2 ) {
    errFlag = 1;
    goto TIME_CONSUMPTION;
}
n = strtoull(argv[1], NULL, 10);
if ( n <= 1 ) {
    errFlag = 2;
    goto TIME_CONSUMPTION; }

printf("n = %llu\n", n);

if ( (n%2) == 0 ){
    p = 2;
    q = n/2;
    goto TIME_CONSUMPTION;
}

if ( (n%3) == 0 ){
    p = 3;
    q = n/3;
    goto TIME_CONSUMPTION;
}

```

.....

Figure 14. Sieve 6K Algorithm code in C language - Part 1

```

if ( (n%5) == 0 ) {
    p = 5;
    q = n/5;
    goto TIME_CONSUMPTION; }

```

```

S0 = floor(sqrt((double)n));
// printf("DBG 1: %lu\n", S0);

```

```

for (i = (S0/6)*6; i <= S0; i++) {
    if ( i == 0 || i == 1 ) {
        continue; }
    if ( (n%i) == 0 ) {
        p = i;
        q = n/i;
        goto TIME_CONSUMPTION;
    } }

```

```

// printf("DBG 2: %lu, %lu\n", p, q);

```

```

i = S0/6 - 1;
// printf("DBG 3: %lu\n", i);
for (; i > 0; i--)
{
    unsigned long tmp = 6*i;
    if ( (n%(tmp + 5)) == 0 ) {
        p = tmp+5;
        q = n/p;
        goto TIME_CONSUMPTION;
    }
}

```

...

```

if ( (n%(tmp + 1)) == 0 ) {
    p = tmp+1;
    q = n/p;
    goto TIME_CONSUMPTION; } }

```

```

if ( p == 0 ) {
    p = 1;
    q = n;
    goto TIME_CONSUMPTION;
}

```

```

TIME_CONSUMPTION:
end_ts = cur_ts();
printf("Time Consumption = %lld ms\n", end_ts-
start_ts);

```

```

switch ( errFlag ) {
case 1:
    printf("Usage:\n");
    printf("$ gcc sieve6-u64b.c -o sieve6-u64b -
lm\n");
    printf("$ ./sieve6-u64b n\n");
    return;
case 2:
    printf("Invalid n!\n");
    return;
default:
    printf("p = %lu,\tq = %llu\n", p, q);
    return;
} }

```

Figure 15. Sieve 6K Algorithm code in C language - Part 2

The output below in Figure 16 shows the time consumption of Sieve 6K Algorithm, which is 3834 milli seconds in this example, that means it is more efficient than the basic algorithm.

```

kanwal@kanwal:~/Desktop/ccode$ gcc sieve6.c -o sieve6 -lm
kanwal@kanwal:~/Desktop/ccode$ ./sieve6 4293665597055658969
n = 4293665597055658969
Time Consumption = 3834 ms
p = 1000000007, q = 4293665567
kanwal@kanwal:~/Desktop/ccode$

```

Figure 16. Sieve 6K Algorithm Output

The following table illustrates how $6K \pm i$ works. Where 6 is a constant number, K represents any integer > 0 . In row two, the number 6 has been broken down to its two smallest factors, 2 and 3 and then multiplying 2 and 3 with the value of K. Depending on the value of x one can have, one will add that value to the sum of the multiplication, i.e., $2*3*k+1$. This is the most efficient way to check whether x is divisible by 2 or 3, and if it is divisible by 2 or 3 then it is not a prime number. If it is not divisible by 2 or 3 then it could be a prime. Here “Candidate” is represented as a “could be” prime.

Table 1. Explanation of 6K Factoring

$6k+0$	$6k+1$	$6k+2$	$6k+3$	$6k+4$	$6k+5$
$2*3*k$	$2*3*k+1$	$2*3*k+2$	$2*3*k+3$	$2*3*k+4$	$2*3*k+5$
Where $k = 1$, $6\%2==0$	Where $k = 1$, $7\%2==1$	Where $k = 1$, $8\%2==0$	Where $k = 1$, $9\%3==0$	Where $k = 1$, $10\%2==0$	Where $k = 1$, $11\%2==1$
Divisible by 2,3	Not Divisible by 2,3	Divisible by 2	Divisible by 3	Divisible by 2	Not Divisible by 2,3
!P	Candidate	!P	!P	!P	Candidate

4.3. Sieve 30K Algorithm

An enhanced version of Sieve 6K is $30K \pm i$. It skips numbers that will be composite numbers and eliminates non-primes in a way that reduces un-necessary repetition. It takes less running time than 6K algorithm. The following code shows how the 30K algorithm works in C language.

<pre># include <stdlib.h> #include <stdio.h> #include <math.h> #include <sys/time.h> long long cur_ts() { struct timeval te; long long milliseconds; // get current time gettimeofday(&te, NULL); // calculate milliseconds milliseconds = te.tv_sec*1000LL + te.tv_usec/1000; // printf("milliseconds: %lld\n", milliseconds); return milliseconds; }</pre>	<pre>void main(int argc, char * argv[]) { long long start_ts, end_ts; unsigned long long n, q; unsigned long p; unsigned long S0; unsigned long i; char errFlag = 0; p = 0; q = 0; start_ts = cur_ts(); if (argc < 2) { errFlag = 1; goto TIME_CONSUMPTION; } n = strtoull(argv[1], NULL, 10); }</pre>
--	---

Figure 17. Sieve 30K Algorithm code in C language - Part 1

```

if ( n <= 1 ) {
    errFlag = 2;
    goto TIME_CONSUMPTION;
}

printf("n = %llu\n", n);

if ( (n%2) == 0 ){
    p = 2;
    q = n/2;
    goto TIME_CONSUMPTION;
}

if ( (n%3) == 0 ){
    p = 3;
    q = n/3;
    goto TIME_CONSUMPTION; }

if ( (n%5) == 0 ){
    p = 5;
    q = n/5;
    goto TIME_CONSUMPTION;
}

if ( (n%7) == 0 ){
    p = 7;
    q = n/7;
    goto TIME_CONSUMPTION;
}

if ( (n%11) == 0 ){
    p = 11;
    q = n/11;
    goto TIME_CONSUMPTION;
}

if ( (n%13) == 0 ){
    p = 13;
    q = n/13;
    goto TIME_CONSUMPTION; }

if ( (n%17) == 0 ){
    p = 17;
    q = n/17;

```

```

    goto TIME_CONSUMPTION;
}

if ( (n%19) == 0 ){
    p = 19;
    q = n/19;
    goto TIME_CONSUMPTION;
}

if ( (n%23) == 0 ){
    p = 23;
    q = n/23;
    goto TIME_CONSUMPTION;
}

if ( (n%29) == 0 ){
    p = 29;
    q = n/29;
    goto TIME_CONSUMPTION;
}

S0 = floor(sqrt((double)n));
// printf("DBG 1: %lu\n", S0);

for (i = (S0/30)*30; i <= S0; i++) {
    if ( i == 0 || i == 1 ) {
        continue;
    }
    if ( (n%i) == 0 ) {
        p = i;
        q = n/i;
        goto TIME_CONSUMPTION;
    }
}
// printf("DBG 2: %lu, %lu\n", p, q);

i = S0/30 - 1;
// printf("DBG 3: %lu\n", i);

for (; i > 0; i--)
{
    unsigned long tmp = 30*i;
    if ( (n%(tmp + 29)) == 0 ) {

```

Figure 18. Sieve 30K Algorithm code in C language - Part 2

<pre> p = tmp+29; q = n/p; goto TIME_CONSUMPTION; } if ((n%(tmp + 23)) == 0) { p = tmp+23; q = n/p; goto TIME_CONSUMPTION; } if ((n%(tmp + 19)) == 0) { p = tmp+19; q = n/p; goto TIME_CONSUMPTION; } if ((n%(tmp + 17)) == 0) { p = tmp+17; q = n/p; goto TIME_CONSUMPTION; } if ((n%(tmp + 13)) == 0) { p = tmp+13; q = n/p; goto TIME_CONSUMPTION; } if ((n%(tmp + 11)) == 0) { p = tmp+11; q = n/p; goto TIME_CONSUMPTION; } if ((n%(tmp + 7)) == 0) { p = tmp+7; q = n/p; goto TIME_CONSUMPTION; } </pre>	<pre> if ((n%(tmp + 1)) == 0) { p = tmp+1; q = n/p; goto TIME_CONSUMPTION; }} if (p == 0) { p = 1; q = n; goto TIME_CONSUMPTION; } TIME_CONSUMPTION: end_ts = cur_ts(); printf("Time Consumption = %lld ms\n", end_ts-start_ts); switch (errFlag) { case 1: printf("Usage:\n"); printf("\$ gcc sieve30-u64b.c -o sieve30- u64b -lm\n"); printf("\$./sieve30-u64b n\n"); return; case 2: printf("Invalid n!\n"); return; default: printf("p = %lu,\tq = %llu\n", p, q); return; } } </pre>
--	--

Figure 19. Sieve 30K Algorithm code in C language - Part 3

The following Figure 20 is the output of above Sieve 30K Algorithm, which shows the efficiency of Sieve 30K and that 30K algorithm is better than Sieve 6K algorithm, as its execution time is 2974 milli seconds in this example.

```

kanwal@kanwal:~/Desktop/ccode$ gcc sieve30.c -o sieve30 -lm
kanwal@kanwal:~/Desktop/ccode$ ./sieve30 4293665597055658969
n = 4293665597055658969
Time Consumption = 2974 ms
p = 1000000007, q = 4293665567

```

Figure 20. Output of Sieve 30K Algorithm

To further understand the Sieve 30K algorithm, please review the following table, which illustrates the process of finding could be prime numbers in 30K algorithm and it also concludes the integers which are not prime or could be prime candidates. In the “Product” column number 30 has been broken down to its smallest factors 2, 3 and 5 like it was done in the Sieve 6K algorithm. Then, those three factors are multiplied with the value of K, i.e., $2*3*5*K$. And at last, the value of x has been added to the sum of $2*3*5*K$ such as $2*3*5*K+1$. In the “Result” column, “!P” represents not prime and “Candidate” represents could be prime.

Table 2. Explanation of 30K Factorization

Range	Product	Check if Prime	Divisible or not	Result
30k+0	$2*3*5*k+0$	Where $k = 1$, $30\%2==0$	Divisible by 2,3,5	!P
30k+1	$2*3*5*k+1$	Where $k = 1$, $31\%2==1$	Not divisible by 2,3,5	Candidate
30k+2	$2*3*5*k+2$	Where $k = 1$, $32\%2==0$	Divisible by 2	!P
30k+3	$2*3*5*k+3$	Where $k = 1$, $33\%3==0$	Divisible by 3	!P
30k+4	$2*3*5*k+4$	Where $k = 1$, $34\%2==0$	Divisible by 2	!P
30k+5	$2*3*5*k+5$	Where $k = 1$, $35\%5==0$	Divisible by 5	!P
30k+6	$2*3*5*k+6$	Where $k = 1$, $36\%2==0$	Divisible by 2,3	!P
30k+7	$2*3*5*k+7$	Where $k = 1$, $37\%2==1$	Not divisible by 2,3,5	Candidate
30k+8	$2*3*5*k+8$	Where $k = 1$, $38\%2==0$	Divisible by 2,3	!P
30k+9	$2*3*5*k+9$	Where $k = 1$, $39\%3==0$	Divisible by 3	!P
30k+10	$2*3*5*k+10$	Where $k = 1$, $40\%2==0$	Divisible by 2,5	!P
30k+11	$2*3*5*k+11$	Where $k = 1$, $41\%2==1$	Not divisible by 2,3,5	Candidate
30k+12	$2*3*5*k+12$	Where $k = 1$, $42\%2==0$	Divisible by 2	!P
30k+13	$2*3*5*k+13$	Where $k = 1$, $43\%3==1$	Not divisible by 2,3,5	Candidate
30k+14	$2*3*5*k+14$	Where $k = 1$, $44\%2==0$	Divisible by 2	!P
30k+15	$2*3*5*k+15$	Where $k = 1$, $45\%5==0$	Divisible by 3,5	!P
30k+16	$2*3*5*k+16$	Where $k = 1$, $46\%2==0$	Divisible by 2,3	!P
30k+17	$2*3*5*k+17$	Where $k = 1$, $47\%2==1$	Not divisible by 2,3,5	Candidate
30k+18	$2*3*5*k+18$	Where $k = 1$, $48\%2==0$	Divisible by 2,3	!P
30k+19	$2*3*5*k+19$	Where $k = 1$, $49\%3==1$	Not divisible by 2,3,5	Candidate
30k+20	$2*3*5*k+20$	Where $k = 1$, $50\%2==0$	Divisible by 2,5	!P
30k+21	$2*3*5*k+21$	Where $k = 1$, $51\%3==0$	Divisible by 3	!P
30k+22	$2*3*5*k+22$	Where $k = 1$, $52\%2==0$	Divisible by 2	!P
30k+23	$2*3*5*k+23$	Where $k = 1$, $53\%3==1$	Not divisible by 2,3,5	Candidate
30k+24	$2*3*5*k+24$	Where $k = 1$, $54\%2==0$	Divisible by 2,3	!P
30k+25	$2*3*5*k+25$	Where $k = 1$, $55\%5==0$	Divisible by 5	!P
30k+26	$2*3*5*k+26$	Where $k = 1$, $56\%2==0$	Divisible by 2,3	!P
30k+27	$2*3*5*k+27$	Where $k = 1$, $57\%3==0$	Divisible by 3	!P
30k+28	$2*3*5*k+28$	Where $k = 1$, $58\%2==0$	Divisible by 2	!P
30k+29	$2*3*5*k+29$	Where $k = 1$, $59\%3==2$	Not divisible by 2,3,5	Candidate

4.4. Fermat's Algorithm

The following code explains the Fermat's Algorithm in C language.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>
long long cur_ts() {
    struct timeval te;
    long long milliseconds;
    // get current time
    gettimeofday(&te, NULL);
    // calculate milliseconds
    milliseconds = te.tv_sec*1000LL + te.tv_usec/1000;
    // printf("milliseconds: %lld\n", milliseconds);
    return milliseconds; }
void main(int argc, char * argv[]) {
    long long start_ts, end_ts;
    unsigned long long n, q, S0S0, sq;
    unsigned long p, i, j, ij, S0, err;
    char errFlag = 0;
    p = 0; q = 0;
    start_ts = cur_ts();
    if ( argc < 2 ) {
        errFlag = 1;
        goto TIME_CONSUMPTION; }
    n = strtoull(argv[1], NULL, 10); if ( n <= 1 ) {
        errFlag = 2;
        goto TIME_CONSUMPTION; }

    S0 = floor(sqrt((double)n));
    S0S0 = S0*2; err = n-S0*S0;
    printf("n = %llu, S0 = %lu, n-S0*S0 = %lu\n",
        n, S0, err);
    for ( j=1; j<n-S0; j++ ) {
        sq = (S0S0+j)*j-err;
        ij = floor(sqrt((double) sq));
        if (sq == ij*ij) {
            i = ij - j;
            p = S0 - i;
            q = n/p;
            break; } }
    TIME_CONSUMPTION:
    end_ts = cur_ts();
    printf("Time Consumption = %lld ms\n",
        end_ts-start_ts);
    switch ( errFlag ) {
        case 1: printf("Usage:\n");
            printf("$ gcc fermat-u64b.c -o fermat-u64b -lm\n");
            printf("$ ./fermat-u64b n\n");
            return; case 2: printf("Invalid n!\n");
            return; default: printf("p = %lu,\tq = %llu\n",
                p, q);
            return; } }
```

Figure 21. Fermat's Algorithm code in C language

The output given below in Figure 22 shows the execution time of Fermat's algorithm, which is 12 milli seconds in this example.

```
kanwal@kanwal:~/Desktop/comcode/ccode$ gcc fermat.c -o fermat -lm
kanwal@kanwal:~/Desktop/comcode/ccode$ ./fermat 17778473929466109103
n = 17778473929466109103, S0 = 4216452766, n-S0*S0 = 1557058347
Time Consumption = 12 ms
p = 4139373533, q = 4294967291
```

Figure 22. Output of Fermat's Algorithm

Example 1- Mod 16:

The following code calculates the mod of the number 16 and the range is 16 (0 to 15).

```
import random
import numpy as np
import math
print "Number|" + " + " + "Square|" + " + " + "Mod16"
for i in xrange(0, 16):
    print i, ", ", ", ", ", ", ", ", (i*i), ", ", ", ", (i*i)%16
```

Figure 23. Mod 16 code in Python

Number	Square	Mod16
0	0	0
1	1	1
2	4	4
3	9	9
4	16	0
5	25	9
6	36	4
7	49	1
8	64	0
9	81	1
10	100	4
11	121	9
12	144	0
13	169	9
14	196	4
15	225	1

Figure 24. Mod 16 Output

The results in Figure 24 shows that the valid modulus are 0, 1, 4 and 9 for number 16.

Example 2 - Mod 100:

The following code calculates the mod of the number 100 and the range is 100 (0 to 99).

```
import random
import numpy as np
import math

# main()
print "Number|" + " + " + "Square|" + " + " + "Mod100"
for i in xrange(0, 100):
    print i, ", ", ", ", ", ", ", ", (i*i), ", ", ", ", (i*i)%100
```

Figure 25. Mod 100 code in Python

[illegible]

The results in Figure 26 shows that the valid modulus are 0 and 25 for number 100.

N	5959	187	2345678917
M	80	14	48433
$(i+j)^2$	441	9	76572
$i+j$	21	3	276.7
$q=m+i+j$	101	17	48709.7
$p=m-i-j$	59	11	48156.3

In the above table, m and b are the factors of given number n .

As stated in section 2.5, Fermat's Sieve Algorithm is the improved version of Fermat's Algorithm where every step is same as Fermat's Algorithm, however it just skips points to find the factors. The following Figure 27 shows the execution time of Fermat's Sieve Algorithm, which is 5 milli seconds, in this example.

```

Fermat Sieve Algorithm
n = 17778473929466109103, S0 = 4216452766, n-S0*S0 = 1557058347
Time Consumption = 5 ms
p = 4139373533, q = 4294967291

```

Figure 27. Output of Fermat's Sieve Algorithm

4.6. Comparison Between 5 Algorithms

The following Figure 28 shows the time consumption of each algorithm and compares the results of basic, Sieve 6K, Sieve 30K, Fermat's and Fermat's Sieve algorithms given the same environment and the same 'n'.

```

Factoring 17778473929466109103

Basic Algorithm
n = 17778473929466109103
Time Consumption = 940 ms
p = 4139373533, q = 4294967291

Sieve 6K Algorithm
n = 17778473929466109103
Time Consumption = 334 ms
p = 4139373533, q = 4294967291

Sieve 30K Algorithm
n = 17778473929466109103
Time Consumption = 264 ms
p = 4139373533, q = 4294967291

Fermat Algorithm
n = 17778473929466109103, S0 = 4216452766, n-S0*S0 = 1557058347
Time Consumption = 25 ms
p = 4139373533, q = 4294967291

Fermat Sieve Algorithm
n = 17778473929466109103, S0 = 4216452766, n-S0*S0 = 1557058347
Time Consumption = 9 ms
p = 4139373533, q = 4294967291

```

Figure 28. Comparison between 5 algorithms

For better understanding see the comparison table in the Appendix A. The table compares the efficiency of basic algorithm, Sieve 6K Algorithm, Sieve 30K Algorithm, Fermat's Algorithm, and Fermat's Sieve Algorithm. The column "Test Round Number" explains how many tests rounds has been conducted, column "Algorithm" defines what type of algorithm it is, column "Big Number N" defines the big number will be used, column "p & q" defines which two prime numbers were calculated from the big number N , then the next column "Time consumption" shows how

much time each algorithm consumed for specific big number and then the last the column “Average Time Consumption” shows how much time was consumed on average by each algorithm.

5. CONCLUSION AND FUTURE WORK

In this paper, the experiments were conducted to check the efficiency of different factoring algorithms such as basic algorithm, Sieve 6K, Sieve 30K, Fermat’s Algorithm and Fermat’s Sieve Algorithm. The basic algorithm takes high level computing power, which can take days, weeks, and months to solve the problem. The Fermat’s Sieve Algorithm is efficient (when p and q are close) as compared to the basic algorithm and other algorithms mentioned in this paper, which takes less computing power and is easier to implement.

In future, we may need to use the Field Programmable Gate Array (FPGA) to implement a customized multiple-core processor. Field Programmable Gate Array is an electronic circuit which contains programmable logic blocks and can be configured to perform a group of complex functions. Hence, FPGA can do parallel computing for our calculation to speed up the computing in each iteration.

REFERENCES

- [1] Dindayal Mahto, Dilip Kumar Yadav. "RSA and ECC: A Comparative Analysis." International Journal of Applied Engineering Research ISSN 0973-4562 Volume 12, Number 19, 2017.
- [2] Katharina Krombholz, Aljosha Judmayer, Matthias Gusenbauer, and Edgar Weippl. "The Other Side of the Coin: User Experiences with Bitcoin Security and Privacy." Financial Cryptography and Data Security 2016.
- [3] Parsi Kalpana, Sudha Singaraju. "Data Security in Cloud Computing using RSA Algorithm." International Journal of Research in Computer and Communication Technology, IJRCCT, SSN 2278-5841, Vol 1, Issue 4, September 2012.
- [4] Hung-Min Sun, Mu-En Wu, Wei-Chi Ting, and M. Jason Hinek. "Dual RSA and Its Security Analysis.", IEEE TRANSACTIONS ON INFORMATION THEORY, VOL. 53, NO. 8, AUGUST 2007.
- [5] Samuel S. Wagstaff, Jr. "The Joy of Factoring." AMS, 2013.
Website: https://books.google.com/books?hl=en&lr=&id=rowCAQAAQBAJ&oi=fnd&pg=PR9&dq=the+joy+of+factoring&ots=umrpV-Y5x_&sig=Z7-56pyeyAWgZ86ER0_QDB7Lbb4#v=onepage&q=the%20joy%20of%20factoring&f=false
- [6] Song Y. Yan. "Primality Testing and Integer Factorization in Public-Key Cryptography." 2009 Springer Science+Business Media, LLC.
- [7] George Kostopoulos. "An Original Numerical Factorization Algorithm." Journal of Information Assurance & Cyber Security, Vol. 2016, September 2016.
- [8] Nidhi Lal¹, Anurag Prakash Singh², Shishupal Kumar. "Modified Trial Division Algorithm Using KNJ-Factorization Method To Factorize RSA Public Key Encryption." 2014 International Conference on Contemporary Computing and Informatics (IC3I).
- [9] Kritsanapong Somsuk, Kitt Tientanopajai. "An Improvement of Fermat's Factorization by Considering the LastmDigits of Modulus to Decrease Computation Time." International Journal of Network Security, Vol.19, No.1, PP.99-111, Jan. 2017.
- [10] M. Z. Ahmad. "A New Approach to Incorporate Uncertainty into Euler's Method." Applied Mathematical Sciences, Vol. 4, 2010, no. 51, 2509 – 2520.
- [11] Richard F. Lukes. "A Very Fast Electronic Number Sieve." Thesis presented to the University of Manitoba, 1995.
- [12] B R Ambedkar and S S Bedi. "A New Factorization Method to Factorize RSA Public Key Encryption." IJCSI International Journal of Computer Science Issues, Vol. 8, Issue 6, No 1, November 2011.

- [13] AbdiansahAbdiansah and RetantyoWardoyo. "Time Complexity Analysis of Support Vector Machines (SVM) in LibSVM." International Journal of Computer Applications (0975 – 8887)Volume 128 –No.3,October 2015.
- [14] R. L. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and PublicKey Cryptosystems." Communications of the ACM, Volume 21, Number 2, February 1978.
- [15] David M. Bressoud. "Factorization and Primality Testing." Springer 1989.
- [16] Connelly Barnes, "Integer Factorization Algorithms." Public Domain. December 7, 2004.
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.117.1230&rep=rep1&type=pdf>
- [17] George Kostopoulos. "An Original Numerical Factorization Algorithm." Journal of Information Assurance & Cyber Security, Vol. 2016, September 2016.

Appendix A Comparison between basic, 6K, 30K, Fermat's and Fermat's Sieve Algorithms

Test Round Number	Algorithm	Big Number N	p & q	Time Consumption	Average Time Consumption
1	Basic	17778473929466109103	4139373533	773 ms	703 ms
				631 ms	
				810 ms	
				634 ms	
				667 ms	
	Sieve 6K			282 ms	335 ms
				228 ms	
				259 ms	
				239 ms	
				667 ms	
	Sieve 30K		167 ms	185.8 ms	
			165 ms		
			273 ms		
			171 ms		
			163 ms		
	Fermat's		12 ms	12.2 ms	
			12 ms		
			13 ms		
			13 ms		
			11 ms		
	Fermat's Sieve		5 ms	7.2 ms	
			9 ms		
			7 ms		
			7 ms		
			8 ms		

2	Basic	14268979894550790107	3322255777	3823 ms	4028 ms
				3728 ms	
				4216 ms	
				4326 ms	
				4047 ms	
	Sieve 6K			1343 ms	1580 ms
				1341 ms	
				1967 ms	
				1613 ms	
				1637 ms	
			Sieve 30K	1041 ms	1196 ms
				992 ms	
				1317 ms	
				1568 ms	
				1064 ms	
	Fermat's		555 ms	603 ms	
			546 ms		
			731 ms		
			642 ms		
			541 ms		
Fermat's Sieve	202 ms	198.2 ms			
	193 ms				
	209 ms				
	197 ms				
	190 ms				

3	Basic	9223372021822390277	2147483647	7411 ms	8656 ms		
				7312 ms			
				10390 ms			
				9963 ms			
				8204 ms			
	Sieve 6K			2620 ms	2888.6 ms		
				2711 ms			
				3079 ms			
				2815 ms			
				3218 ms			
			Sieve 30K	1960 ms	2146.6 ms		
				1940 ms			
				4294967291		2533 ms	3447 ms
						2054 ms	
						2246 ms	
	3170 ms						
	3212 ms						
	Fermat's		3785 ms		1050 ms		
			3652 ms				
			3417 ms				
			Fermat's Sieve			1044 ms	
						1049 ms	
	1034 ms						
	1051 ms						
	1072 ms						

4	Basic	4611967483740094469	1073807359	8804 ms	9762 ms		
				8929 ms			
				11445 ms			
				9625 ms			
				10009 ms			
	Sieve 6K			3337 ms	3775 ms		
				3122 ms			
				4305 ms			
				4183 ms			
				3929 ms			
			Sieve 30K	2409 ms	2690 ms		
				2382 ms			
				4294967291		2425 ms	10633 ms
						3649 ms	
						2585 ms	
	9224 ms						
	9357 ms						
	Fermat's		11233 ms		3949.4 ms		
			13209 ms				
			10144 ms				
			Fermat's Sieve			3935 ms	
						3947 ms	
	3954 ms						
	3948 ms						
	3963 ms						

5	Basic	3380486235539538017	787080787	8611 ms	9123 ms		
				8791 ms			
				9357 ms			
				10071 ms			
				8787 ms			
	Sieve 6K			3237 ms	3156 ms		
				3084 ms			
				3187 ms			
				3151 ms			
				3121 ms			
			Sieve 30K	2318 ms	2375 ms		
				2297 ms			
				4294967291		2615 ms	12332 ms
						2311 ms	
						2334 ms	
	12051 ms						
	12419 ms						
	Fermat's		12920 ms		3913.4 ms		
			12257 ms				
			12015 ms				
			Fermat's Sieve			3952 ms	
						3889 ms	
	3887 ms						
	3897 ms						
	3942 ms						