American Journal of Computational Mathematics, 2018, 8, 175-183
http://www.scirp.org/journal/ajcm
ISSN Online: 2161-1211
ISSN Print: 2161-1203

A Parallel Probabilistic Approach to Factorize a Semiprime

Jianhui Li[1,2]

[1]Department of Computer Science, Guangdong Neusoft Institute Foshan City, Foshan, China
[2]State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi, China
Email: lijianhui@nuit.edu.cn, joe863@163.com

How to cite this paper: Li, J.H. (2018) A Parallel Probabilistic Approach to Factorize a Semiprime. *American Journal of Computational Mathematics*, **8**, 175-183.
https://doi.org/10.4236/ajcm.2018.82013

Received: June 14, 2018
Accepted: June 26, 2018
Published: June 29, 2018

Abstract

In accordance with the distributive traits of semiprimes' divisors, the article proposes an approach that can find out the small divisor of a semiprime by parallel computing. The approach incorporates a deterministic search with a probabilistic search, requires less memory and can be implemented on ordinary multicore computers. Experiments show that certain semiprimes of 27 to 46 decimal-bits can be validly factorized with the approach on personal computer in expected time.

Keywords

Parallel, Probabilistic, Integer Factorization, Semiprime

1. Introduction

A semiprime is an odd composite number N that has exactly two distinct prime divisors, say p and q, such that $3 \le p < q$. Factorization of the semiprimes has been a difficult problem in mathematics and computer science, especially factorization of a RSA number that is a large semiprime, as introduced and overviewed in articles [1] [2] [3] [4] [5]. Let $k = q/p$ be the divisor-ratio of the semiprime $N = pq$; article [6] discovered the genetic property of the odd integers and proved that the small divisor p can be calculated by finding the greatest common divisor (GCD) with an odd integer that lies in an odd interval I_0 that is determined by N itself; article [7] further pointed out that, by taking k to be a variable quantity, algorithms could be designed to factorize big semiprimes by parallel computing and the article also demonstrated a deterministic approach to factor the RSA numbers; article [8] recently made a further investigation on k's influence to the distribution of p, showing that the interval I_0 is a cover of finite subintervals each of which is determined by a k and every two adjacent ones of which are linked with their unique common end. This means that p can be surely found out with a k-subdivision approach. Following such an idea, this paper makes an investigation on the new algorithm design for searching the odd integer that has *GCD* with *N*. As a result, an algorithm that incorporates a deterministic search with a probabilistic search is designed. This paper presents the related consequences. Section 2 lists the preliminaries for the later sections; Section 3 proves the mathematical foundations for the new algorithm; Section 4 introduces the new algorithm as well as its designing strategy and numerical experiments.

2. Preliminaries

This section lists the preliminaries that include definitions, symbols and lemmas, which are necessary for later sections.

2.1. Symbols and Notations

In this whole article, a *semiprime* $N = pq$ means p and q are both odd prime numbers and $3 \le p < q$. An *odd interval* $[a,b]$ is a set of consecutive odd numbers that take a as the lower bound and b as the upper bound; for example, $[3,11] = \{3,5,7,9,11\}$. Symbol $\lfloor x \rfloor$ is the *floor function*, an integer function of real number x that satisfies $\lfloor x \rfloor \le x < \lfloor x \rfloor + 1$; symbol *GCD* means the greatest common divisor; let $m > 0$; then e_m^b, e_m^p, e_m^q and e_m^0 are defined by

$$e_m^b = N_{\left(m+1,2^{m-1}-1\right)} = 2^m N - 1$$

$$e_m^p = 2^m N - p$$

$$e_m^q = 2^m N - q$$

$$e_m^0 = e_m^b - 2\left(\left\lfloor \frac{\sqrt{N}+1}{2} \right\rfloor - 1\right)$$

Symbol $A\Delta = \frac{B}{2}$, which was defined in [8], means A is half of B.

2.2. Lemmas

Lemma 1. (See in [8]) Suppose $N = pq$ is an odd composite number and $k = \left\lfloor \frac{q}{p} \right\rfloor \ge 1$; then $e_m^p \in \left[e_m^{kl}, e_m^{kr}\right]$, where $m = \lfloor \log_2 N \rfloor - 1$,

$e_m^{kl} = e_m^b - 2\left(\left\lfloor \frac{1}{2}\left(1 + \sqrt{\frac{N}{k}}\right) \right\rfloor - 1\right)$, $e_m^{kr} = e_m^b - 2\left\lfloor \frac{1}{2}\left(1 + \sqrt{\frac{N}{k+1}}\right) \right\rfloor$ and

$e_m^b = 2^m N - 1$.

Lemma 2. (See in [8]) Suppose $N > 1$ is an odd integer and $k_1 < k_2$ are positive integers. Let $ls_{k_1} = \left\lfloor \frac{1}{2}\left(1 + \sqrt{\frac{N}{k_1+1}}\right) \right\rfloor + 1$, $ls_{k_2} = \left\lfloor \frac{1}{2}\left(1 + \sqrt{\frac{N}{k_2+1}}\right) \right\rfloor + 1$,

$lb_{k_1} = \left\lfloor \frac{1}{2}\left(1 + \sqrt{\frac{N}{k_1}}\right) \right\rfloor$ and $lb_{k_2} = \left\lfloor \frac{1}{2}\left(1 + \sqrt{\frac{N}{k_2}}\right) \right\rfloor$; then it holds

$$e_m^{kl_1} < e_m^{kr_1} \le e_m^{kl_2} < e_m^{kr_2}$$

where $e_m^{kl_1} = e_m^b - 2(lb_{k_1} - 1)$, $e_m^{kl_2} = e_m^b - 2(lb_{k_2} - 1)$, $e_m^{kr_1} = e_m^b - 2(ls_{k_1} - 1)$ and $e_m^{kr_2} = e_m^b - 2(ls_{k_2} - 1)$.

Particularly, when $k_2 = k_1 + 1$ it holds

$$e_m^{kl_1} < e_m^{kr_1} = e_m^{kl_2} < e_m^{kr_2}$$

Lemma 3. (See in [8]) For arbitrary odd number $N > 1$, let $ls_i = \left\lfloor \frac{1}{2}\left(1 + \sqrt{\frac{N}{i+1}}\right) \right\rfloor + 1$, $lb_i = \left\lfloor \frac{1}{2}\left(1 + \sqrt{\frac{N}{i}}\right) \right\rfloor$, $e_m^{il} = e_m^b - 2(lb_i - 1)$ and $e_m^{ir} = e_m^b - 2(ls_i - 1)$, where $i = 1, 2, \cdots, \omega$ are positive integers; then odd intervals $I_i = \left[e_m^{il}, e_m^{ir}\right]$ satisfy

1) $I_i \cap I_{i+1} = e_m^{ir}$;

2) $\bigcup_{i=1}^{i=\omega} I_i = \left[e_m^0, e_m^{\omega r}\right]$;

3) $\bigcup_{i=1}^{i=\infty} I_i = \left[e_m^0, e_m^b\right]$; as illustrated in Figure 1.

Lemma 4. (See in [8]) Let $\Delta_0 = \left\lfloor \frac{\sqrt{N}+1}{2} \right\rfloor$ be the length of the odd interval $I_0 = \left[e_m^0, e_m^b\right]$, Δ_i and Δ_{i+1} be respectively the lengths of the odd intervals I_i and I_{i+1} defined in Lemma 3; then when $1 < i \le \left\lfloor \frac{\sqrt[6]{N}}{16} \right\rfloor$, it holds $\frac{1}{2}\Delta_i < \Delta_{i+1} < \Delta_i$.

Lemma 5. (See in [8]) Let $\Delta_0 = \left\lfloor \frac{\sqrt{N}+1}{2} \right\rfloor$ be the length of the odd interval $I_0 = \left[e_m^0, e_m^b\right]$, Δ_1, Δ_2 and Δ_3, be respectively the lengths of I_1, I_2 and I_3; then $\Delta_1 + \Delta_2 + \Delta_3\Delta = \frac{1}{2}\Delta_0$ and when $\Delta_0 \ge 26$ it holds $\Delta_1 + \Delta_2 > \frac{1}{4}\Delta_0$.

Lemma 6. (See in [8]) If $k \ge \left\lfloor \sqrt[3\alpha]{N^\beta} \right\rfloor$ for some positive integers $\beta \ge 1$ and $\alpha > \beta$, then Δ_k is at most $\left\lfloor \frac{1}{4}\sqrt[2\alpha]{N^{\alpha-\beta}} \right\rfloor$; otherwise it is at least $\left\lfloor \frac{1}{4}\sqrt[2\alpha]{N^{\alpha-\beta}} \right\rfloor - 1$

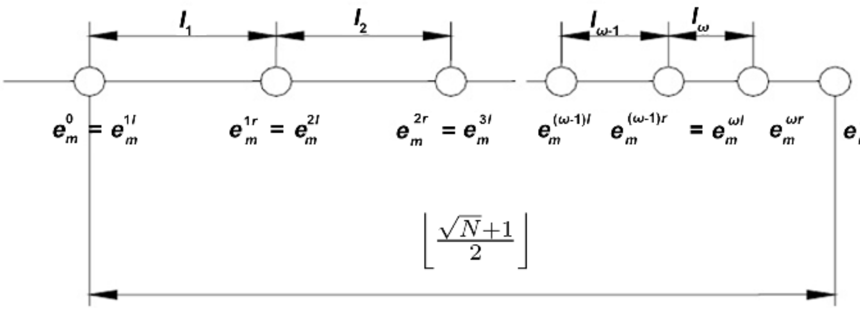

Figure 1. Odd intervals' subdivision and cover.

$\left\lfloor \frac{1}{4}\sqrt[2\alpha]{N^{\alpha-\beta}} \right\rfloor - 1$. Particularly, arbitrary $\alpha \geq 2$ yields $\Delta_k \leq \left\lfloor \frac{\sqrt[2\alpha]{N}}{4} \right\rfloor$ for $k \geq \left\lfloor \sqrt[3\alpha]{N^{\alpha-1}} \right\rfloor$ and $\Delta_k \geq \left\lfloor \frac{\sqrt[2\alpha]{N}}{4} \right\rfloor - 1$ for $k \leq \left\lfloor \sqrt[3\alpha]{N^{\alpha-1}} \right\rfloor - 1$.

Lemma 7. (See in [8]) Let $I_1, I_2, \cdots, I_\omega$ be odd intervals defined in Lemma 3; then there are intervals that contain at most $\left\lfloor \frac{\sqrt[4]{N}}{4} \right\rfloor$ nodes and there are intervals that contain at least $\left\lfloor \frac{\sqrt[4]{N}}{4} \right\rfloor - 1$ nodes.

3. Algorithm Design and Numerical Experiments

Based on the previous lemmas, theorems and corollaries, one can easily draw the following conclusions.

1) If $N = pq$ is a semiprime, then there is a term e_m^p that lies in the odd interval I_0 and satisfies $p = GCD\left(N, e_m^p\right)$;

2) If I_0 is subdivided into a series of subintervals that are defined in Lemma 3, then $e_m^p \in I_k$ with $k = \left\lfloor \frac{q}{p} \right\rfloor$, and the bigger k is the fewer nodes are contained in I_k. Among all the subintervals, I_1, I_2 and I_3 dominate half of I_0. Lemma 6 shows that, when $k \leq \left\lfloor \sqrt[3\alpha]{N^\beta} \right\rfloor - 1$ there are at least $\left\lfloor \frac{1}{4}\sqrt[2\alpha]{N^{\alpha-\beta}} \right\rfloor - 1$ nodes in I_k.

These provide a guideline for designing new algorithm for integer factorization, as the following subsections demonstrates.

3.1. Strategy for Algorithm Design

Now it has gotten to know that finding e_m^p out means a successful factorization of $N = pq$. Since e_m^p hides itself in one of $I_1, I_2, \cdots, I_\omega$, it can surely be found out by searching the intervals one by one. Considering by Lemma 7 that there are intervals that contain small number of nodes that can be searched in small time and there are also intervals that contain too many nodes to be searched when N is very big, it is necessary to know which intervals contain small number of nodes and which ones contain large number of nodes and then perform a *brute-force search* on the small ones and perform other searches on the large ones. Since the brute-force search is a time-consuming process, a *Tolerable Number* (*TN*) can be defined to be an upper bound of nodes that are sure to be searched out in a *Tolerable Time* (*TM*), which was introduced in FU's article [9]. Obviously, Lemma 6 indicates that *TN* can be used to determine α by $TN = \left\lfloor \frac{\sqrt[2\alpha]{N}}{4} \right\rfloor$ and thus $\omega_0 = \left\lfloor \sqrt[3\alpha]{N^{\alpha-1}} \right\rfloor$. Since the number of nodes in I_ω is smaller than that in I_{ω_0} if $\omega > \omega_0$ by Lemma 6, *TN* is a critical number for the brute-force search. All the intervals $I_{\omega_0+1}, I_{\omega_0+2}, \cdots$ can be searched by the brute-force search.

Now it turns to the big intervals $I_1, I_2, \cdots, I_{\omega_0}$, each of which contains more than *TN* nodes. It is sure that, applying *TN* to subdivide each of these big intervals can obtain a series of new small odd intervals and then assigning each of the newly-subdivided small subintervals a process in a parallel computing system can perform the brute-force search in *TM*, as tested by FU [9]. However, this might require huge computing resources. For example, FU's approach took more than 8 days to factorize a big number of 46-decimaldigits on Chinese Tianhe Supper Computer with 392 processes. On the other hand, by Lemma 3, there is a big probability to find the objective node when applying a probabilistic approach. Thus it is worthy of trying a probabilistic search.

By now, setting a *TN*, calculating an α by $TN = \left\lfloor \frac{\sqrt[2\alpha]{N}}{4} \right\rfloor$ plus an ω by $\omega = \left\lfloor \sqrt[3\alpha]{N^{\alpha-1}} \right\rfloor$ and imagining varying $k = \left\lfloor \frac{q}{p} \right\rfloor$ result in a subdivision of the interval I_0 into two kinds of subintervals, the kind of small ones and the kind of big ones, as shown in **Figure 2**. Each of the small ones contains no more than *TN* nodes and can be searched by the brute-force search while each of the big ones contains more than *TN* nodes that need probabilistic searching approaches to search.

Now consider a big odd subinterval *I* that contains *n* terms. Suppose the objective odd number $o = ps$ lies at the *m*-th position. Referring to the analysis in [10], it knows that, choosing randomly k ($k < m$) terms on the left of *o*, say $o_{-j}, \cdots, o_{-j-1}, o_{-j-k}$, and their respectively symmetric terms on the right of *o*, namely, $o_j, o_{j+1}, \cdots, o_{j+k}$, yields $o_{-j-k} + \cdots + o_{-j-1} + o_{-j} + o_j + o_{j+1} + \ldots + o_{j+k} = 2ko$ with $j = 1, 2, \cdots, m-k-1$. Therefore, choosing in *I* randomly 2*k* terms and adding the chosen terms might obtain *o* in big probability, as proved in [10]. With the help of *multi-dimensional random-number generator*, as introduced in [11], it is easy to pick 2*k* random terms in *I*. Considering the case that *o* lies near the start-position or the end-position of the interval *I* might fail to pick the necessary number of terms, it is reasonable to make near *I*'s ends one or more small subintervals each of which contain *TN* of nodes. These small subintervals are called *TN* intervals and they can be searched in *TM* with brate-force searches. The remained part of *I* is called a probabilistic intervals, as shown in **Figure 3**. Such a subdivision that includes two *TN* intervals near the ends and a probabilistic interval in the middle is called a *TNPTN* subdivision. With the *TNPTN* subdivision, it is necessary to perform brute-force searches on the two TN intervals and a probabilistic search on the *whole interval I*.

3.2. TNPTN Parallel Probabilistic Algorithm

Based on the strategy for algorithm design stated in previous section, a parallel algorithm, which is called TNPTN MPI Algorithm, is designed to find an

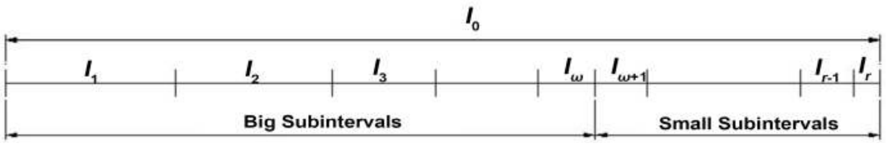

Figure 2. Subdivision of interval I_0.

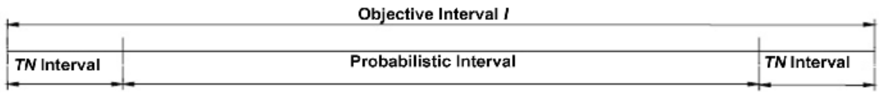

Figure 3. TNPTN subdivision.

objective node N_{obj} that has common divisor p with $N = pq$. The algorithm assumes that $k = \lfloor q/p \rfloor$ varies from 1 to an upper bound and assigns for each k a process to search N_{obj}. It requires initial input data N to be the big semi-prime, TN to be the number of the maximal steps that a brute-force search performs and a number N_{rand} with $N_{rand} < TN$ to set the number of random odd integers that are randomly picked in its searched interval with the multi-dimensional random-number generator introduced in [11]. It also requires a brute-force searching subroutine and a probabilistic searching subroutine to perform the operations.

TNPTN MPI Algorithm

```
Input: odd composite integer $N$ and $TN$
Step 1. Initial calculations for process 0.
        (1). The level $m$: $m = \lfloor \log_2 N \rfloor$;
        (2). $e_m^b$ and $e_m^0$:$e_m^b = 2^m N - 1$;$e_m^0 = e_m^b - 2(\left\lfloor \frac{\sqrt{N}+1}{2} \right\rfloor - 1)$;
        (3). $\alpha$ and $\omega_\alpha$:$\alpha = \left\lfloor \frac{\log N}{2\log(4TQ)} \right\rfloor$; $\omega_\alpha = \left\lfloor \sqrt[3\alpha]{N^{\alpha-1}} \right\rfloor$;
        (4). $\omega_b$ satisfies $\left\lfloor \frac{\sqrt{N}}{2((\omega_b+1)\sqrt{\omega_b}+\omega_b\sqrt{\omega_b+1})} \right\rfloor \le 1$;
Step 2. Parallel computing for other processes.
        For process $k = \omega_b$ to 1
          $ls = \left\lfloor \frac{1}{2}(1+\sqrt{\frac{N}{k+1}}) \right\rfloor + 1$;$lb = \left\lfloor \frac{1}{2}(1+\sqrt{\frac{N}{k}}) \right\rfloor$;
          $e_m^{kl} = e_m^b - 2(lb-1)$;$e_m^{kr} = e_m^b - 2(ls-1)$;
          If $(k > \omega_\alpha)$ res= BruteForceSearch$(N, e_m^{kl}, e_m^{kr})$;
          else res=ProbabilisticSearch$(N, e_m^{kl}, e_m^{kr}, TN, 1)$;
          If (res $> 1$)
            Begin
              BroadcasteStop; /*Call other processes to stop*/
              break; /*Break the host process*/
            Endif
        EndFor
```

SUBROUTINE *Brute-force Searching Procedure*

```
Subroutine BruteForceSearch
  Input: $N, iStart, iEnd$;
    For $el = iStart$ to $el = iEnd$ do
      Begin
        if($FindGCD(N, el)$) return $GCD$;
        $el = el + 2$;
      EndIf
    EndFor
  reture 1;
EndSubroutine
```

SUBROUTINE *Probabilistic Searching Procedure*

```
Subroutine ProbabilisticSearch
  Input: $N, iStart, iEnd, TN, R$;
  Step 1. Do Initial Calculations.
      (1). $ll = iStart; lr = ll + 2(TN-1)$;
      (2). $rl = iEnd - 2(TN+1); rr = iEnd$;
  Step 2. Brutal Searches on $[ll, lr]$ and $[rl, rr]$ by
      res=BrutalSearch($ll, lr$);
      If (res > 1) return res;
      res=BrutalSearch($rl, rr$);
      If (res > 1) return res;
  Step 3. Random Search on $[iStart, iEnd]$ by
      Begin Loop
        $[s_1, s_2, \ldots, s_R]$ = RandomNumber($[iStart, iEnd]$);
            For ($k = 1$ to $k = R$)
            $gcd$=FindGCD($N$,$S_k$));
            If($gcd > 1$) return $gcd$;
            EndFor
        $sum = \sum_{i=1}^{R} s_i$;
        If(FindGCD($N, sum$)) return $GCD$;
      EndLoop
      return 1;
EndSubroutine
```

3.3. Numerical Experiments

Numerical experiments were made on a Sugon workstation with Xeon(R) E5-2650 V3 processor of 20 cores and 128GB memory via C++ MPI programming with gmp big number library. Several big semiprimes with 27 to 46 decimal-digits are factorized, as shown in Table 1.

4. Conclusion and Future Work

It is a convention to make a comparison of a new approach to the old ones although, sometime, there is no comparability between two things. Accordingly, this section makes comparisons and then prospects some future work.

It can see from the implementation of algorithms list in previous section that, each of the algorithms 1, 2 and 3 needs memory only for storing a few integers to be taken into the computation. They cost less memory. Since the whole procedure is a parallel one, the time it costs depends on the resources joining the computation. Theoretically, it is at most $O\left(2\sqrt[4]{N}\left(\log_2 N\right)^3\right)$ bit-operations providing that there is $\left\lfloor \frac{\sqrt[4]{N}}{2} \right\rfloor$ process joining the computation.

Now turn to the old ones. It is known that, ever since John Pollard raised in 1975 his Pollard's Rho algorithm, which is a probabilistic algorithm and is efficient in factoring small integers, many algorithms of integer factorization have developed. As stated in the introductory section, the GNFS has been regarded the fastest approach to factorize big integers under both sequential and parallel computing and almost all the factorized RSA numbers are factorized by the approach with parallel computing. So here the new approach is merely compared to the Pollard's Rho approach and the GNFS approach.

Table 1. Experiments on some big semiprimes.

Semiprime	bits	Divisors	Time(s)
N_1 = 521900076822691495534066493	27	15098125637513	20134.371596
		34567209821461	
N_2 = 63281217910257742583918406571	29	125778791843321	22487.545370
		503115167373251	
N_3 = 194920496263521028482429080527	30	289673451203483	326720.128666
		672897345109469	
N_4 = 2400000000000001550240000000000042854447	40	37678804836791	236949.403121
		63696287883753452357619017	
N_5 = l4272476927059598804393l594750096l98971949056l	46	2305843009213693951	336797.313147
		618970019642690137449562111	

First compare with the Pollard's Rho approach. From the point-view of memory cost, the new approach is like the Pollard's Rho approach that costs less memory. From the point-view of time consumption, the two are almost the same efficiency according to the experiments in articles [6] and [12]. However, as pointed out in article [13], the Pollard's Rho approach cannot be parallelised. Actually, this article is the first one that proposes a parallel probabilistic approach in factoring integers.

Next compare with the GNFS approach. The GNFS approach is a deterministic one that can be parallelised. As article [14] has pointed out, GNFS requires a large amount of memory. Hence from the point-view of memory cost, the new approach is superior to the GNFS. From the point-view of time consumption, the comparison cannot be available because the new approach has not been applied on the many computers as the GNFS used in factoring large RSA numbers. Nevertheless, it can see from the experiments of factoring the 46 decimal bits semiprime that, one computer of 20 cores can factorize it in 3 days.

There is another approach stated by Kurzweg U H [15]. Seeing from his series of Blogs, one can see that Kurzweg actually tried to find out $\phi(N)$ to factorize the semiprime N. The idea was early seen in chapter 6 of YAN's book [1]. Actually, it is quite occasional for Kurzweg's to factorize the numbers he reported because he has not brought an approach that finds out $\phi(N)$ inevitably.

By now it can see that, the approach raised in this article is surely worthy of investigation because it is truly derived from the theorems and corollaries that are proved mathematically. In spite that the new approach is less of successful cases of factoring the RSA numbers, it does factorize many odd integers as many old approaches did in the history. As a new approach, it leaves of course quite a lot of researches to improve and perfect. For example, the probabilistic searching procedure is very rough and needs improving, and the time complexity of the algorithm has not be evaluated till now for its probabilistic trait and also for the authors' limitation of the required knowledge. This points out the study of the future work. Hope it is concerned more and successful in the future.

Acknowledgements

The research work is supported by the State Key Laboratory of Mathematical Engineering and Advanced Computing under Open Project Program No.2017A01, Department of Guangdong Science and Technology under project 2015A030401105, Foshan Bureau of Science and Technology under projects 2016AG100311, Project gg040981 from Foshan University. The authors sincerely present thanks to them all.

References

[1] Yan, S.X. (2008) Cryptanalytic Attacks on RSA. Springer US, New York.

[2] Surhone, L.M., Tennoe, M.T. and Henssonow, S.F. (2011) RSA Factoring Challenge. Springer US, New York.

[3] Wanambisi, A.W., Aywa, S., Maende, C., *et al.* (2013) Advances in Composite Integer Factorization Bibinfojournal. *Materials & Structures*, **48**, 1-12.

[4] Abubakar, A., Jabaka, S., Tijjani, B.I., *et al.* (2014) Cryptanalytic Attacks on Rivest, Shamir, and Adleman (RSA) Cryptosystem: Issues and Challenges. *Journal of Theoretical & Applied Information Technology*, **61**, 1-7.

[5] Kessler, G.C. (2017) An Overview of Cryptography (Updated Version 26 February 2017). http://commons.erau.edu/publication/412

[6] Wang, X.B. (2017) Genetic Traits of Odd Numbers with Applications in Factorization of Integers. *Global Journal of Pure and Applied Mathematics*, **13**, 493-517.

[7] Wang, X.B. (2017) Strategy for Algorithm Design in Factoring RSA Numbers. *IOSR Journal of Computer Engineering*, **19**, 1-7. https://doi.org/10.9790/0661-1903020107

[8] Wang, X.B. (2018) Influence of Divisor-Ratio to Distribution of Semiprime's Divisor. *Journal of Mathematics Research*, **10**, 54-61. https://doi.org/10.5539/jmr.v10n4p54

[9] Fu, D.B. (2017) A Parallel Algorithm for Factorization of Big Odd Numbers. *IOSR Journal of Computer Engineering*, **19**, 51-54. https://doi.org/10.9790/0661-1902055154

[10] Wang, X.B., Li, J.H., Duan, Z.H. and Wan, W. (2018) Probability to Compute Divisor of a Hidden Integer. *Journal of Mathematics Research*, **10**, 1-5. https://doi.org/10.5539/jmr.v10n1p1

[11] Hu, X.P. and Cui, H. (2010) Generating Multi-Dimensional Discrete Distribution Random Number. *Sixth International Conference on Natural Computation IEEE* 10-12, 1102-1104. https://doi.org/10.1109/ICNC.2010.5583695

[12] Li, J.H. (2017) Algorithm Design and Implementation for a Mathematical Model of Factoring Integers. *IOSR Journal of Mathematics*, **13**, 37-41. https://doi.org/10.9790/5728-1301063741

[13] Brent, R.P. (1990) Parallel Algorithms for Integer Factorisation Number Theory and Cryptography. Loxton, J.H., Ed. Cambridge University Press, Cambridge, 26-37.

[14] Wang, Q., Fan, X. and Zhang, H. (2016) The Space Complexity Analysis in the General Number Field Sieve Integer Factorization *Theoretical Computer Science*, **630**, 76-94.

[15] Kurzweg, U.H. (2012) More on Factoring Semi-Primes. http://www2.mae.ufl.edu/ũhk/MORE-ON-SEMIPRIMES.pdf