Modification and optimization of miller – rabin simplicity test algorithm implemented by parallel computation

L Cherckesova¹, O Safaryan¹, I Trubchik¹, V Chumakov¹, V Yukhnov², and I Yengibaryan²

¹Don State Technical University, Gagarin sq., 1, Rostov on Don, 344003, Russia
²Moscow Technical University of Communications and Informatics, Serafimovicha str., 62, 344007 Rostov–on–Don, Russia

E-mail: safari_2006@mail.ru

Abstract. The project of modification and optimization Miller–Rabin’s simplicity test algorithm implemented by parallel computation in the programming language C# is presented, which works faster on comparison with standard iterative algorithm by 50%, which makes it easier to create the keys for such well-known encryption algorithms as RSA, DSA, etc. These methods are similar on their functionality, but have a different estimation of structural complexity that has been shown various tests. Proceeding from it, modified by authors Miller–Rabin's algorithm realized through parallel calculations is more reliably and faster, than the standard algorithm, that is experimentally confirmed in practice.

1. Introduction

The rapid development of information and communication technologies increases the urgency of the problem of information security. In this regard, it is required to develop a number of new methods and tools aimed at ensuring information security.

Therefore, an integrated approach is required to ensure reliable information security. In other words, there is a need for effective use of legal, organizational and engineering support for the protection of information.

In particular, cryptographic methods play important role in information protection. Today, cryptographic information security systems are widely used. All these cryptographic systems operate grounding on cryptographic algorithms. At present, algorithms RSA and El–Gamal are used as the basis for many cryptographic standards. These algorithms are based on the factorization problem and discrete logarithm in a finite field [1].

To encrypt data and create electronic digital signature, both algorithms use 1024–bit and large prime numbers. Generating and working with large prime numbers become one of main issues in cryptography. Reason for prime’s wide use in cryptography is difficulty of finding and detection of these numbers.

The task of our investigation is improving, optimization and modernization of the algorithm for checking numbers for the simplicity of Miller–Rabin, the modification of which is capable of increase operating speed of the realized standard algorithm.

2 Theoretical Foundations
Miller–Rabin’s simplicity test algorithm is a modification of Miller’s algorithm developed by Gary Miller in 1976. This Miller’s algorithm is deterministic, but its correctness is based on the unproven extended Riemann hypothesis. Michael Rabin modified it in 1980. The Miller–Rabin algorithm does not depend on the validity of hypothesis, but is probabilistic [2].

In many applications, such as cryptography, it becomes necessary to search for large random prime numbers. Large prime numbers are not rare, so it takes a little time to find a prime number by checking random integers of the appropriate size. The distribution function of primes is defined as the number of primes not exceeding a number n.

The “prime number theorem”:

\[
\lim_{n \to \infty} \frac{\pi(n)}{n/\ln(n)} = 1
\]

An approximate estimate of \( n/\ln(n) \) gives sufficiently exact and accurate estimation of the function even for small numbers n.

For example, for \( n = 10^9 \), \( n = 109 \), when \( \pi(n) = 50847534 \), and \( n/\ln(n) \approx 48254942 \), the deviation does not exceed 6% [3].

The process of random selection of a prime number n and verification of its simplicity can be regarded as Bernoulli’s tests. According to the prime number theorem, the probability that randomly chosen number n turns out to be prime is approximately equal to \( 1/\ln(n) \). Geometric distribution tells us how many attempts are required to succeed in order to succeed.

Thus, in order to find a prime number whose length is the same as the length of the number n, we need to test \( \ln(n) \) approximately integers by randomly selecting them in the neighborhood of a number n. The Miller–Rabin’s test bases and relies on the test checking of number of equalities that hold for prime numbers. If at least one such equality is not satisfied, this proves that this number is composite [3].

For the Miller–Rabin test, the following statement is used: let \( n \) Then we can represent the number in the form:

\( n = d \cdot r + s \), where \( d \) is odd number.

Then if \( n \) is a prime number, then for any one \( a \) of the sequence \( Z_n \), one of the following conditions holds true:

\( a^d = 1 \pmod{n} \)

\( \exists r, 0 \leq r < s; a^{2^r d} \equiv -1 \pmod{n} \)

The Miller–Rabin’s simplicity test algorithm:

1. Choose a random number.
2. We verify that the conditions for the number
3. If conditions are fulfilled, then witness is of simplicity, otherwise \( n \) is composite.

The mathematical justification of the Miller–Rabin algorithm was presented in the work [1], [4].

If for one number one of the presented conditions is satisfied, then it is called a witness of simplicity. The idea of the algorithm is that we do not need to check all numbers from the field [5].

If the number is prime, then all numbers in the field are witnesses of simplicity. If composite, then witnesses of simplicity are less than a quarter of all field numbers, according to the Rabin theorem. Suppose we do not know a number that is simple or composite. Moreover, if we randomly choose a random number for a certain number and it will be a witness of simplicity, this will mean that the number being tested is compound with a probability of less than 0.25, and if we choose two numbers and they both will be witnesses to simplicity, the probability will be less than 1/16.

In addition, if we choose 1000, numbers and all of them will be witnesses of simplicity; we will get a very small probability that this number is compound.

Also, for large values, the probability of declaring a composite number is probably simply much smaller. Ivan Damgard, Peter Landrock and Alexander Pomerandes computed some exact errors and proposed a method for selecting the value of \( k \) to obtain the required error boundary.

Such boundaries, for example, can be used to generate probably prime numbers. However, they
should not be used to test prime numbers of unknown origin, since in cryptographic systems an attacker may try to substitute a pseudo–simple number in a situation where a prime number is required. In such cases, you can rely only on the error [6].

The program can be divided into four actions:
1) Preparation: Calculation of auxiliary values.
2) Generating a random number.
3) Verification of the first condition.
4) Check the second condition.

After testing the algorithm on known prime numbers $M_{521} = 2^{521} - 1$, $M_{607} = 2^{607} - 1$, $M_{1279} = 2^{1279} - 1$, found out how much the program spends time on each action [7].

The basis of the obtained data can be concluded: the most costly process is the verification of the first condition. This is because the first condition is checked in each round, as opposed to checking the second condition.

Change the way the random number is generated. In this application, we use standard tools to generate a random sequence of bytes, and then we turn this byte array into a decimal number. This is not the optimal solution, because the numbers in the computer are stored as a sequence of bytes. In this way, you can store the resulting byte sequence straight as a specific number, without performing any unnecessary actions [8].

Optimization of verification of the second condition. During the verification of the second condition, we consistently check for equality numbers $a^{d\cdot 2^r} \mod n \equiv n - 1$ where $r = 0, s$ [9].

You can notice that it is not necessary to recalculate this formula for each $r$, because if we know the value of this expression for $r - 1$, then from the value for $r$ we can get just raising the previous iteration to the second power. Really,

$$\left(a^{d\cdot 2^{r-1}}\right)^2 = a^{d\cdot 2^{r-1}\cdot 2} = a^{d\cdot 2^{r-1}1} = a^{d\cdot 2^r}$$

### 3 Program Realization of Modified Algorithm

In the work [4] authors presented their implementation of Miller–Rabin algorithm in the C# programming language.

In this work, we continue our investigations and improve our software implementation by optimizing the parallel computation of the algorithm rounds. We received new, improved version of the developed software.

We represent this algorithm in its standard form (Figure 1):

```csharp
public static bool MillerRabin(BigInteger n) {
    n.Get_d_s(out var d, out var s);
    var k = (int)BigInteger.Log(n, 2) + 1;
    for (var l = 0; l < k; l++) {
        var a = BigInteger.ModPow(GetRandBigInt(), d, n);
        if (a == 1 || a == n - 1) continue;
        for (var r = 1; r < s; r++)
            if (BigInteger.ModPow(a, d * BigInteger.Pow(2, r), n) == n - 1) goto l;
    }
    return true;
}
```

**Figure 1.** Standard software implementation of the algorithm.

Function Get_d_s decomposes the number to be checked for the corresponding values. Its realization is shown in Figure 2:
Further modification of the Miller–Rabin algorithm.

In this application, we use standard tools to generate a random sequence of bytes, and then we turn this byte array into a decimal number. This is not the optimal solution, because the numbers in the computer are stored as a sequence of bytes. In this way, you can store the resulting byte sequence straight as a specific number, without performing any unnecessary actions [10].

The execution of the rounds is independent of each other, so this section can be executed in parallel, as shown in Figure 3.

Improving the time to check first condition is to optimize the algorithm for rapid exponentiation in modulus.

There are a number of algorithms for rapid exponentiation in modulus. What algorithm is used in the BigInteger.ModPow() method is not known, but it can be determined using disassembler. Nevertheless, this method works relatively quickly.

To improve the result, we used the special packages in MathCad. This software product specialize in mathematical algorithms and is able to generate the appropriate program code in specific programming language – C# [11].

In addition to MathCad, you can use MatLab or Maple.

As a result, we obtain the following improved realization of the Miller–Rabin’s simplicity test algorithm in Figures 4 – 7.

### Figure 2. Implementation of an auxiliary function.

```java
private static void Get_d_s(thls BigInteger n, out BigInteger d, out int s)
{
    s = 0;
    d = n - 1;
    while (d % 2 == 1)
    {
        d /= 2;
        s++;
    }
}
```

### Figure 3. Parallel execution of rounds.

```java
public static boolean MillerRabinParallel(BigInteger n)
{
    n.Get_d_s(out var d, var s);
    var k = (int)BigInteger.Log(n, 2.0);
    var isPrime = true;
    Parallel.For(0, k, (i, pls) =>
    {
        var a = BigInteger.ModPow(GetRandBigInt(), d, n);
        if ([a == 1 || a == n - 1]) return;
        for (var r = 1; r < s; r++)
        {
            if (BigInteger.ModPow(a, d * BigInteger.Pow(2, r), n) == n - 1) return;
            isPrime = false;
            pls.Break();
        }
    });
    return isPrime;
}
```

### Figure 4. Improved implementation of Miller–Rabin.

```java
public static boolean MillerRabinParallel(BigInteger n)
{
    n.Get_d_s(out var d, var s);
    var k = (int)BigInteger.Log(n, 2.0);
    var isPrime = true;
    Parallel.For(0, k, (i, pls) =>
    {
        var a = BigInteger.ModPow(GetRandBigInt(), d, n);
        if ([a == 1 || a == n - 1]) return;
        for (var r = 1; r < s; r++)
        {
            if ((a = BigInteger.ModPow(a, 2, n)) == n - 1) return;
            isPrime = false;
            pls.Break();
        }
    });
    return isPrime;
}
```
Testing. Now we will fulfil the tests on different types of computers and ensure in the speed of algorithm operating. This testing was conducted on three computers with different processor frequencies. Test results are demonstrated in the Table I [12].

Table I shows the execution time of the high – frequency counter functions. The numerator of this table shows data on the Miller–Rabin simplicity test algorithm – method modified by the authors, and in the denominator, for the comparison, by the Pollard’s iteration method. Percentage under the counter is the efficiency of the first method before the second [13].

| Processor Type | Numbers | Average processor efficiency | Total average efficiency |
|----------------|---------|-----------------------------|-------------------------|
| Intel® Core (TM) i5 – 4200U CPU 2.3 GHz | M521 | 49% | 50% |
| Intel® Core (TM) i3 – 2330M 2.20 GHz | M607 | 43% | 49% |
| AMD Phenom (tm) II Quad–Core Processor 1.80 GHz | M2203 | 43% | 39% |
| | M2281 | 52% | 45,75% |

Table 1. The result of the algorithm on different processors.

Figure 5. Result of testing on the processor with frequency of 2.2GHz.

Figure 6. Result of testing on processor with frequency of 2.3 GHz.
Figure 7. Result of testing on a processor with a frequency of 1.8 GHz.

The speed of the program operating was also tested on the well-known primes of Mersenne [14]: Modified software implementation of the Miller–Rabin simplicity test algorithm, which on 49% more efficient than standard iterative algorithm. These methods are similar in functionality, but have a different assessment of structural complexity, which was demonstrated by various tests. Based on this, the modified Miller–Rabin algorithm, implemented through parallel calculations, is more reliable and faster, and authors in practice experimentally confirm that.

4. Conclusion
In this work, we continue our research, started in the work [4], and improve our software implementation by optimizing the parallel computation of the algorithm rounds. We received new, improved version of the developed software.

To improve the result, we used the special packages in MathCad. This software product able to generate the appropriate program code in specific programming language – C#.

The developed modified and optimized software realization of Miller–Rabin simplicity test algorithm is more effective by 49% compared with the standard iterative algorithm. Proceeding from this, the Miller–Rabin algorithm realized through parallel computations is more reliable and faster that was experimentally confirmed by authors in practice.

References
[1] 2012 Electronic textbook for students of the Institute of Computational Mathematics and Information Technology ed H T Ishmukhametov and R G Rubtsova, Kazan, Kazan Federal University, p 138
[2] 2012 “Improved Pollard rho method for computing discrete logarithms over finite extension fields” ed P Wang and F Zhang, Journal of Computational and Applied Mathematics 236, Issue 17, pp 4336-4343
[3] “Pollard P-algorithm” [electronic resource]: access mode www / URL: http://en.wikipedia.org/wiki/Miller–Rabin algorithm.
[4] Galov K A, Cherckesova L V and Safaryan O A 2019 “Implementation of the Miller–Rabin Algorithm in the C# Programming Language” Molodoj Issledovatel Dona 1(16), pp 24 – 31
[5] Kosyakov M S 2014 Introduction to distributed computing, St. Petersburg, p.155
[6] Zheltov S A 2014 Effective Computing in the CUDA Architecture in Information Security Applications, ed PhD dis. S A Zheltov Moscow, IINTB RSUH, p. 145
[7] 2020 A factorization method for completely positive matrices ed P Groetzner and M Dür. Linear Algebra and its Applications, 591, 1-24
[8] 2019 The Semantic System Supports the Processing of Mathematical Principles, ed J Liu, L Yuan, C Chen, J Cui, H Zhang and X Zhou, Neuroscience, 404, pp 102-118
[9] Integer Factorization Algorithms Connelly Barnes Department of Physics, Oregon State University https://math.dartmouth.edu/archive/m56s14/public_html/proj/Howey0_proj.pdf

[10] 2020 Guidolin A and Vaccarino F “Computing invariants for multipersistence via spectral systems and effective homology” Journal of Symbolic Computation Available online In Press, Corrected Proof

[11] Montgomery P Speeding the Pollard and Elliptic Curve Methods of Factorization, Mathematics of Computation, 48, N.177, 243–264 http://www.connellybarnes.com/documents/factoring.pdf

[12] Howey E 2014 Primality Testing and Factorization Methods https://pdfs.semanticscholar.org/7fd4/4eb2df7b39716e984d548c28f51d9dc6bbbb.pdf

[13] Bai Sep Shi 2011 Polynomial Selection for the Number Field Sieve PhD Thesis of Australian National University http://maths–people.anu.edu.au/brent/pd/Bai–thesis.pdf

[14] Chaudhary N 2016 Metrics for Event Driven Software PhD. Scholar of Gautama Buddha University, India (IJACSA) International Journal of Advanced Computer Science and Applications, 7(1) http://thesai.org/Volume7No1/Paper_12Metrics_for_Event_Driven_Software.pdf