УДК 004.056.55

## РЕАЛИЗАЦИЯ АЛГОРИТМА МИЛЛЕРА-РАБИНА НА ЯЗЫКЕ C#

***Галов К. А., Черкесова Л. В., Сафарьян О. А***
Донской государственный технический университет, Ростов-на-Дону, Российская Федерация
gkarnd@yandex.ru
chia2002@inbox.ru
safari_2006@mail.ru

Представлен проект реализации Миллера-Рабина на языке C#, который работает быстрее стандартного алгоритма на 50%, что сможет облегчить работу при создании ключей для алгоритмов шифрования типа RSA.

**Ключевые слова:** простые числа, Миллер-Рабин, алгоритмическая сложность, параллельные вычисления, оптимизация.

UDC 004.056.55

## IMPLEMENTATION OF THE MILLER–RABIN ALGORITHM IN THE C# PROGRAMMING LANGUAGE

***Galov K. A., Cherkesova L. V., Safaryan O. A.***
Don State Technical University, Rostov-on-Don, Russian Federation
gkarnd@yandex.ru
chia2002@inbox.ru
safari_2006@mail.ru

The paper presents the project for implementing Miller-Rabin in the C # language, which works faster than the standard algorithm by 50%, which makes it easier to create keys for such encryption algorithms as RSA

**Keywords**: prime numbers, Miller-Rabin, algorithmic complexity, parallel computations, optimization

**Введение**. Бурное развитие информационных и коммуникационных технологий повышает актуальность проблемы информационной безопасности. В связи с этим требуется разработать ряд новых методов и средств, направленных на обеспечение информационной безопасности. Следовательно, требуется комплексный подход для надежного обеспечения информационной безопасности. Другими словами, возникает необходимость эффективного использования правовых, организационных и инженерно-технических обеспечений защиты информации.

В частности, криптографические методы играют важную роль в защите информации. Сегодня широко используется криптографические системы защиты информации. Все эти криптографические системы работают на основе криптографического алгоритма. В настоящее время в качестве основы для многих криптографических стандартов берутся алгоритмы RSA и Эль-Гамаль. Эти алгоритмы основаны на задаче факторизации и дискретном логарифмировании в конечном поле [1].

Для шифрования данных и создания электронной цифровой подписи в обоих алгоритмах используются 1024-битные и большие простые числа. Таким образом, генерирование и работа с большими простыми числами стали одним из главных вопросов в криптографии. В общем, причиной широкого использования простых чисел в криптографии является трудность обнаружения этих чисел.

**Постановка задачи**

Задачей данного исследования являлось улучшение алгоритма проверки чисел на простоту Миллера-Рабина, модификация которого способна увеличить быстродействие реализованного стандартного алгоритма.

**Основная часть**

Алгоритм Миллера-Рабина является модификацией алгоритма Миллера, разработанного Гари Миллером в 1976 году. Алгоритм Миллера является детерминированным, но его корректность опирается на недоказанную расширенную гипотезу Римана. Майкл Рабин модифицировал его в 1980 году. Алгоритм Миллера-Рабина не зависит от справедливости гипотезы, но является вероятностным [2].

Во многих приложениях, таких, как криптография, возникает необходимость поиска больших случайных простых чисел. Большие простые числа не редки, поэтому для поиска простого числа путем проверки случайных целых чисел соответствующего размера потребуется немного времени. Функция распределения простых чисел $\pi(n)$определяется как количество простых чисел, не превышающих числа $n$.

Теорема о простых числах:

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

Приближенная оценка $n/\ln n$ дает достаточно точную оценку функции даже при малых n. Например, при от $n = 10^9$ , когда $\pi(n) = 50847534$, а $\frac{n}{\ln n} \approx 48254942$, отклонение не превышает 6% [3].

Процесс случайного выбора простого числа n и проверку его простоты можно рассматривать как испытания Бернулли. Согласно теореме о простых числах, вероятность того, что случайным образом выбранное число n окажется простым, приблизительно равна $1/\ln n$. Геометрическое распределение говорит о том, какое количество попыток требуется сделать, для того, чтобы добиться успеха. Таким образом, чтобы найти простое число, длина которого совпадает с длиной числа n, понадобится проверить приблизительно $\ln n$ целых чисел, выбрав их случайным образом в окрестности числа n.

Тест Миллера-Рабина опирается на проверку ряда равенств, которые выполняются для простых чисел. Если хотя бы одно такое равенство не выполняется, это доказывает, что число составное.

Для теста Миллера-Рабина используется следующее утверждение:

пусть $n > 2$. Представим число $n-1$ в виде $n-1 = 2^s d$, где $d$ — нечетно. Тогда, если $n$ — простое число, то для любого $a$ из $Z_n$ выполняется одно из условий:

1. $a^d = 1(mod n)$
2. $\exists r, 0 \le r < s: a^{2^r d} \equiv -1(mod n)$

**Алгоритм Миллера- Рабина:**

1. Выберем случайное число $a$ в пределах $(1, n-1)$
2. Проверим выполнимость условий для числа $a$
3. Если условия выполняются, то $a$ свидетель простоты, иначе n составное.

**Обоснование алгоритма Миллера-Рабина**

Применив индукцию по i, можно сделать вывод, что последовательность вычисленных значений $x_0, \dots x_t$ удовлетворяет условию $x_i = a^{2^i u} mod n$. Однако этот цикл может закончиться раньше, если после очередного возведения в квадрат будет обнаружен нетривиальный квадратный корень из 1. В этом случае работа алгоритма завершается, и он возвращает значение true [4].

По малой теореме Ферма: $a^{n-1} \equiv 1 mod n$. Будем извлекать квадратные корни из числа $a^{n-1}$. На каждом шаге у нас получится $-1$ или 1. Если на каком-то шаге получится $-1$, то выполняется второе из неравенств. Иначе на очередном шаге выполнится первое равенство.

Если для некоторого числа выполняется одно из представленных условий, то оно называется свидетелем простоты. Идея алгоритма заключается в том, что не нужно проверять все числа из поля.

Если число простое, то все числа в поле являются свидетелями простоты. Если число составное, то свидетелями простоты являются менее четверти всех чисел поля, согласно теореме Рабина. Пусть мы не знаем, простое число или составное. Если для некоторого числа случайным образом выберем случайное число, и оно окажется свидетелем простоты, это будет означать, что проверяемое число является составным с вероятностью менее 0,25, а если выберем два числа и они оба окажутся свидетелями простоты, вероятность будет уже менее 1/16. Если выберем 1000 чисел и все они окажутся свидетелями простоты, то получим очень маленькую вероятность того, что это число составное.

Для больших значений вероятность объявления составного числа вероятно простым существенно меньше. Дамгард, Лэндрок и Померандс вычислили некоторые точные границы ошибок и предложили метод выбора значения k для получения нужной границы ошибки. Такие границы могут, например, использоваться для генерации вероятно простых чисел. Однако, они не должны использоваться для проверки простых чисел неизвестного происхождения, поскольку в криптографических системах взломщик может попытаться подставить псевдопростое число, в той ситуации, когда требуется простое число. В таких случаях можно положиться только на ошибку [5].

Работу программы можно разбить на четыре действия:

1) Подготовка: вычисление вспомогательных значений.

2) Генерация случайного числа.

3) Проверка первого условия.

4) Проверка второго условия.

Проведя тестирования алгоритма на известных простых числах $M_{521}=2^{521}-1$, $M_{607}=2^{607}-1$, $M_{1279}=2^{1279}-1$, выяснили, сколько программа тратит времени на каждое действие

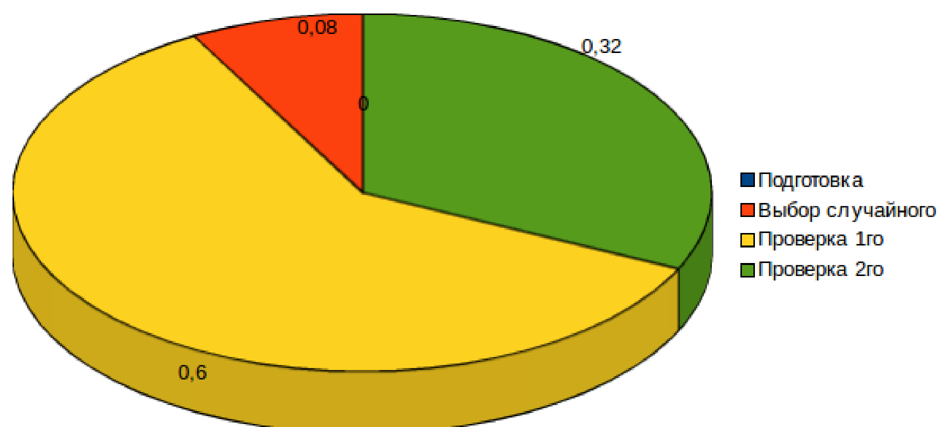



Рис. 1. Время, затраченное на каждое из действий

Нас основание полученных данных можно сделать вывод: самым затратным процессом является проверка первого условия. Это объясняется тем, что проверка первого условия происходит в каждом раунде, в отличии от проверки второго условия.

Изменение способа генерации случайного числа. В данном приложении с помощью стандартных средств генерируем случайную последовательность байт, затем сворачиваем этот массив байт в десятеричное число. Это не оптимальное решение, потому что числа в компьютере и так хранятся в виде последовательности байт. Таким образом можно сохранять полученную последовательность байт прямиком в виде конкретного числа, не выполняя лишних действий.

Оптимизация проверки второго условия. Во время проверки второго условия последовательно проверяем на равенства числа $a^{d*2^r} \mod n \equiv n-1$ где $r=0,s$. Можно заметить, что не обязательно каждый раз для каждого r заново вычислять эту формулу, потому что если знаем значение этого выражения при r -1, то от значения при r можно получить, просто возведя предыдущей итерации во вторую степень. Действительно, $(a^{d*2^{r-1}})^2 = a^{d*2^{r-1}*2} = a^{d*2^{r-1+1}} = a^{d*2^r}$.

**Программная реализация алгоритма**

Представим данный алгоритм в его стандартном виде

```
public static bool MillerRabin(BigInteger n)
{
    n.Get_d_s(out var d, out var s);
    var k = (int)BigInteger.Log(n, 2.0);
    for (var i = 0; i < k; i++)
    {
        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 false;
        l:;
    }
    return true;
}
```

Рис. 2. Стандартная программная реализация алгоритма

Функция Get_d_s раскладывает проверяемое число на соответствующие значения. Ее реализация представлена на рис. 3.

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

Рис. 3. Реализация вспомогательной функции

**Модификация алгоритма Миллера-Рабина**

В данном приложении с помощью стандартных средств генерируем случайную последовательность байт, затем сворачиваем этот массив байт в десятеричное число. Это не оптимальное решение, потому что числа в компьютере и так хранятся в виде последовательности байт. Таким образом можно сохранять полученную последовательность байт прямиком в виде конкретного числа, не выполняя лишних действий.

Выполнение раундов не зависит друг от друга, поэтому этот участок можно выполнять параллельно, как показано на рис. 2.

```
public static bool MillerRabinParallel(BigInteger n)
{
    n.Get_d_s(out var d, out 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;
}
```

Рис. 4. Параллельное выполнение раундов

Во время проверки второго условия последовательно проверяем на равенства числа $a^{d*2^r} \bmod n \equiv n-1$ где $r=0,s$. Можно заметить, что не обязательно каждый раз для каждого r заново вычислять эту формулу, потому что если знаем значение этого выражения при r -1, то от значения при r можно получить возведя предыдущей итерации во вторую степень. Действительно, $\left(a^{d*2^{r-1}}\right)^2 = a^{d*2^{r-1}*2} = a^{d*2^{r-1+1}} = a^{d*2^r}$.

Улучшение времени проверки первого условия заключается в оптимизации алгоритма быстрого возведения в степень по модулю. Существует целый ряд алгоритмов быстрого возведения в степень по модулю. Какой алгоритм используется в методе BigInteger.ModPow() неизвестно, однако это можно выяснить при помощи дизасемблера. Тем не менее, этот метод работает относительно быстро. Для улучшения результата можно использовать такие специальные пакеты как MathCad, Matlab или Maple. Эти программные продукты специализируются на математических алгоритмах и имеют возможность сгенерировать соответствующий код на определенном языке программирования.

В результате получаем следующую улучшенную реализацию алгоритма Миллера-Рабина на рис. 5.

```
public static bool MillerRabinParallel(BigInteger n)
{
    n.Get_d_s(out var d, out 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;
}
```

Рис. 5. Улучшенная реализация Миллера-Рабина

**Тестирование**

Теперь сделаем тесты на разных компьютерах и убедимся в скорости работы алгоритма. Данное тестирование проводилось на 3-х компьютерах с разной частотой процессора. Результаты тестирования представлены на таблице 1.

Таблица 1

Результат работы алгоритма на различных процессорах

| Тип процесса | Числа | | | | Средняя эффективность по процессору | Общая средняя эффективность |
|---|---|---|---|---|---|---|
| | M521 | M607 | M2203 | M2281 | | |
| Intel® Core(TM) i5-4200U CPU 2,3 GHz | 49% | 51% | 52% | 50% | 50% | 49% |
| Intel ® Core(TM) i3-2330M 2,20 GHz | 43% | 49% | 39% | 52% | 45.75% | |
| AMD Phenom(tm) II Quad-Core Processor 1,80 HGz | 46% | 47% | 49% | 48% | 47% | |

В таблице 1 показано время выполнения функций высокочастотного счетчика. В числителе данной таблицы показаны данные по модифицированному авторами рекурсивному методу, а в знаменателе — по итерационному методу Полларда. Процент под счетчиком вычислениями показывает эффективность первого метода перед вторым.

Графически данный тест представлен на рис. 6, 7, 8.

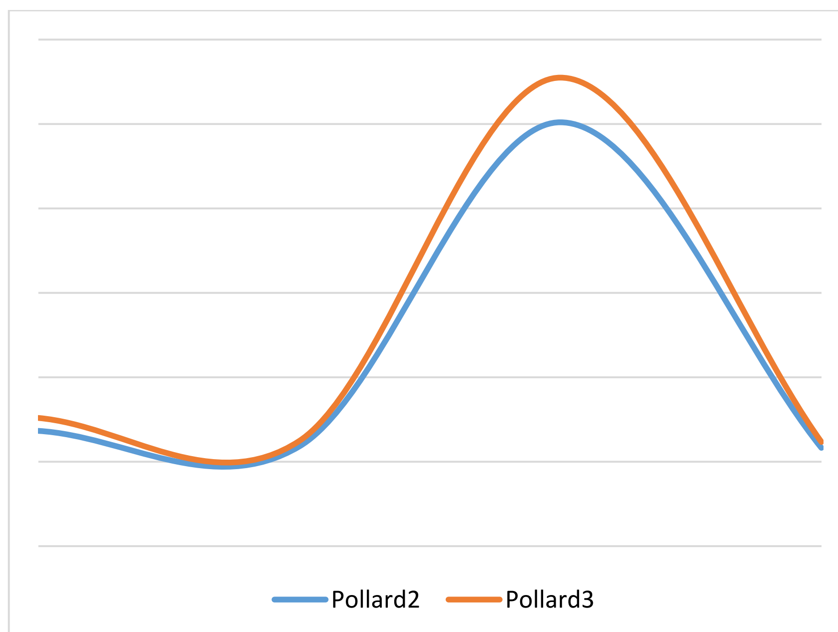
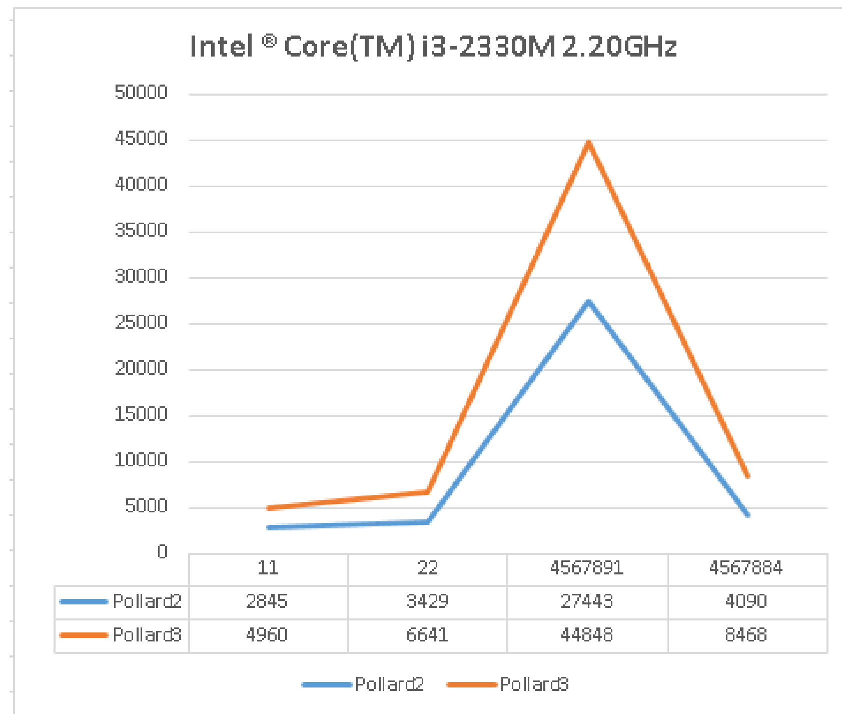
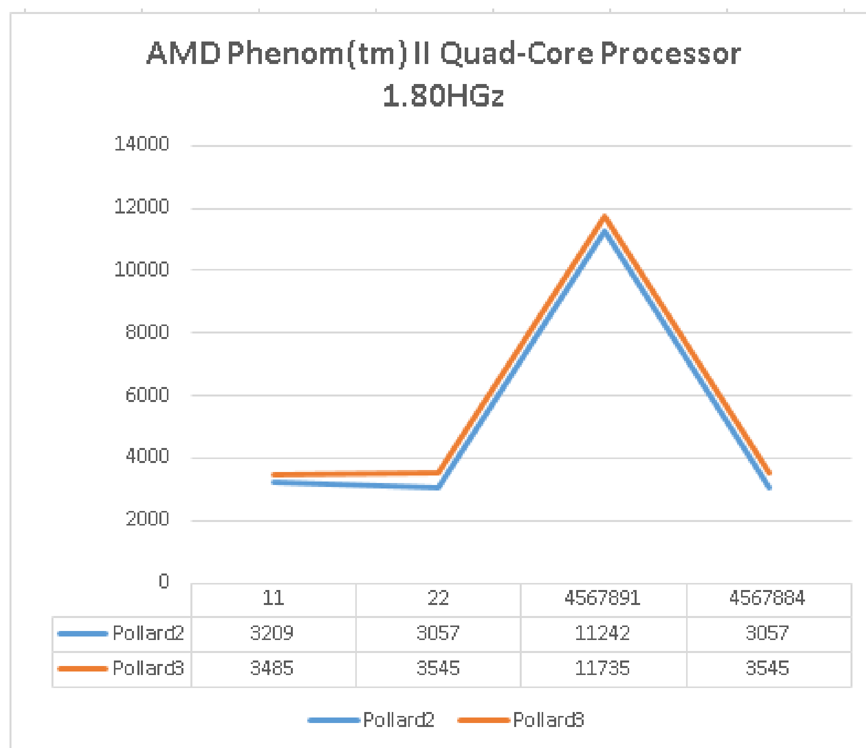



Рис. 6. Результат тестирования на процессоре с частотой 2,3 GHz




| | 11 | 22 | 4567891 | 4567884 |
|---|---|---|---|---|
| Pollard2 | 2845 | 3429 | 27443 | 4090 |
| Pollard3 | 4960 | 6641 | 44848 | 8468 |

Рис. 7. Результат тестирования на процессоре с частотой 2,2 GHz




| | 11 | 22 | 4567891 | 4567884 |
|---|---|---|---|---|
| Pollard2 | 3209 | 3057 | 11242 | 3057 |
| Pollard3 | 3485 | 3545 | 11735 | 3545 |

Рис. 8. Результат тестирования на процессоре с частотой 1,8 GHz

Скорость работы программы была также протестирована на известных простых числах Мерсенна.

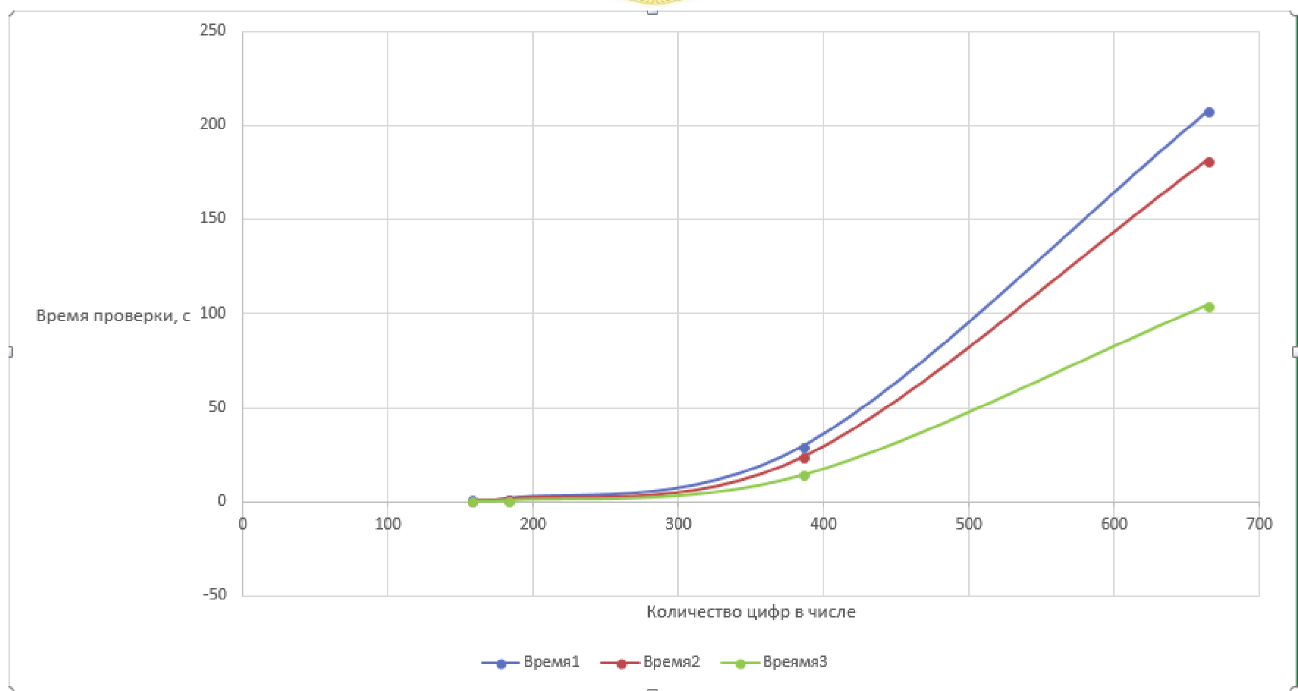



Рис. 9. Тестирование скорости на числах Мерсенна.

**Заключение**. Разработана программная реализация алгоритма Миллера-Рабина, которая эффективнее на 49% по сравнению со стандартным итерационным алгоритмом. Эти методы схожи по функционалу, но имеют разную оценку структурной сложности, что было продемонстрировано различными тестами.

Исходя из этого, алгоритм Миллера-Рабина, реализованный через параллельные вычисления, надежнее и быстрее, что экспериментально подтверждено на практике.

**Библиографический список**

1. Ишмухаметов, Ш. Т. Математические основы защиты информации. Электронное учебное пособие для студентов института вычислительной математики и информационных технологий [Электронный ресурс] / Ш. Т. Ишмухаметов, Р. Г. Рубцова. — Казань : Казанский федеральный университет, 2012. — 138 с. — Режим доступа : http://doc.knigi-x.ru/22informatika/42064-1-sht-ishmuhametov-rubcova-matematicheskie-osnovi-zaschiti-informacii-elektronnoe-uchebnoe-posobie-dlya-studentov-i.php

2. Оценка структурной сложности программ [Электронный ресурс] / Studfiles. — Режим доступа : https://studfiles.net/preview/5850014/page:12/ (дата обращения : 23.03.2018).

3. P-алгоритм Полларда [Электронный ресурс] / Википедия. — Режим доступа : www/URL:http://ru.wikipedia.org/wiki/Алгоритм_Миллера-Рабина (дата обращения : 23.03.2018).

4. Косяков, М. С. Введение в распределенные вычисления / М. С. Косяков. — Санкт-Петербург :НИУ ИТМО. — 2014. — 155 с.

5. Желтов, С. А. Эффективные вычисления в архитектуре CUDA в приложениях информационной безопасности : Дис. …канд. техн. наук / С. А. Желтов. — Москва, 2014. — 145 с.