1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт кибербезопасности и защиты информации

**КУРСОВАЯ РАБОТА**

1. «Современные методы анализа криптосистем RSA и решения задачи разложения на множители»
2. по дисциплине «Криптографические методы защиты информации»
3. Выполнили Берко А. С.
4. студенты гр. 4851003/90801 Кулеева А. Г.
   * + 1. Веселов Б. В.

Львов А. В.

1. Руководитель
2. профессор ИКиЗИ, Александрова Е. Б.
3. д.т.н., доцент

ассистент ИКиЗИ Ярмак А. В.

1. Санкт-Петербург
2. 2023

Оглавление

[1 Введение 3](#_Toc138706164)

[2 Теоретическая часть 4](#_Toc138706165)

[2.1 Атака методом непрерывных дробей 4](#_Toc138706166)

[2.1.1 Параметры криптосистемы 4](#_Toc138706167)

[2.1.2 Необходимая теоретическая основа 4](#_Toc138706168)

[2.1.3 Атака на основе алгоритма непрерывных дробей 7](#_Toc138706169)

[2.2 Атака Копперсмита 8](#_Toc138706170)

[2.2.1 Теорема Копперсмита 8](#_Toc138706171)

[2.2.2 Описание атаки 9](#_Toc138706172)

[2.3 Атака дешифрования на основе многоядерных графических процессоров с CUDA 10](#_Toc138706173)

[2.3.1 Факторизация составного числа методом Полларда . 10](#_Toc138706174)

[2.3.2 Алгоритм факторизации Полларда на основе модели программирования CUDA 11](#_Toc138706175)

[2.4 Разложение на множители с помощью квантового компьютера 12](#_Toc138706176)

[2.4.1 1.4.1. Основные понятия 12](#_Toc138706177)

[2.4.2 Алгоритм Шора 13](#_Toc138706178)

[2.4.3 Оптимизация вычислений на квантовых компьютерах 16](#_Toc138706179)

[2.5 Сравнительный анализ методов 18](#_Toc138706180)

[3 Практическая часть 18](#_Toc138706181)

[3.1 Атака методом непрерывных дробей 18](#_Toc138706182)

[3.1.1 Сравнение с атакой Винера 19](#_Toc138706183)

[3.2 Атака Копперсмита 20](#_Toc138706184)

[3.3 Факторизация Полларда. CUDA 24](#_Toc138706185)

[3.3.1 Тестирование 25](#_Toc138706186)

[3.3.2 Анализ применимости при атаках на современные системы 26](#_Toc138706187)

[2.1. Реализация алгоритма Шора 26](#_Toc138706188)

[4 Современные требования к параметрам RSA 30](#_Toc138706189)

[5 Заключение 31](#_Toc138706190)

[6 Список использованных источников 32](#_Toc138706191)

# Введение

В современном мире огромное количество информации передается по сети, и все чаще это информация является конфиденциальной. Для ее защиты используются различные криптографические системы, которые позволяют обеспечить конфиденциальность и целостность данных. Одной из самых популярных криптосистем является *RSA*.

*RSA* — это криптографическая система с открытым ключом, которая основана на сложности факторизации больших простых чисел. *RSA* активно используется во многих областях, таких как финансы, телекоммуникации, электронная почта и многих других. Например, *SSL*-сертификаты, которые используются для защиты веб-сайтов, основаны на криптографических алгоритмах, включая *RSA*.

Основной угрозой для *RSA* является задача разложения большого числа на простые множители, которая является вычислительно сложной и требует экспоненциального времени для ее решения. Однако существует ряд методов, позволяющих решать эту задачу не за экспоненциальное время, а за полиномиальное. Кроме того, активное развитие ведется в сфере квантовых компьютеров, которые также можно использовать в качестве механизма взлома *RSA*. Поэтому, разработка и использование безопасных параметров *RSA* становится все более актуальным вопросом в современной криптографии.

Целью данной курсовой работы является определение оптимальных параметров, позволяющих максимально защитить криптографическую систему *RSA* от современных методов факторизации больших чисел.

Для достижения этой цели поставлены следующие задачи:

1. Проанализировать и описать наиболее известные методы разложения числа на множители;
2. Составить таблицу сравнения этих методов по общим параметрам;
3. В практической части реализовать исследуемые методы;
4. Составить рекомендации по выбору наиболее безопасных параметров криптографической системы *RSA*.

# Теоретическая часть

## Атака методом непрерывных дробей

### Параметры криптосистемы

Модуль таков, что , и . Отметим, что для всегда выполняется . Однако, если то можно найти и методом Ферма или методом Копперсмита. Отправной точкой является ключевое уравнение . Точнее, для задаем , , . Затем, применяя алгоритм непрерывных дробей, покажем, что, при условии , рациональное число является сходящимся непрерывного дробного разложения .

### Необходимая теоретическая основа

***Лемма 1***

Пусть N = pq — модуль RSA с q < p < 2q. Тогда

***Непрерывные дроби***

Пусть ξ — вещественное число. Продолжение дроби ξ представляет собой выражение вида

где a0 ∈ Z, ai ∈ N∗ для i ≥ 1. Если ξ — рациональное число, то список [a0, a1, a2, ...] частичных коэффициентов конечен и может быть вычислен за полиномиальное время. Для n ≥ 0, [a0, a1, a2, ..., an] является рациональным числом и называется сходящимся расширением непрерывной дроби ξ. Существуют различные свойства разложения вещественных чисел в непрерывную дробь. Следующее полезно для того, чтобы проверить, является ли рациональное число сходящимся к вещественному числу ξ.

***Теорема 1***

Пусть ξ — положительное действительное число. Если a и b — целые числа, удовлетворяющие свойству НОД(a, b) = 1 и

тогда является сходящейся частью непрерывной дроби ξ. [1]

***Теорема 2***

Пусть (N, e) является открытым ключом в криптосистеме RSA с N = pq и q < p < 2q. Если e < (p2 – 1)(q2 – 1) и удовлетворяет уравнению ed – k (p2 – 1)(q2 – 1) = 1 при

то можно разложить число N на множители за полиномиальное время.

Доказательство.

Пусть и . Тогда точка является серединой интервала [φ1, φ2]. Поскольку (p2 – 1)(q2 – 1) ∈ [φ1, φ2], то

Используя уравнение ed – k (p2 – 1)(q2 – 1) = 1, получаем

Затем, используя и (1), получаем

Теперь используем лемму 1 и получаем

Несложный расчет показывает, что

Объединяя это с (2), получаем

Если то и по Теореме 1, является сходящимся непрерывного дробного разложения . Используя k и d, получаем

Комбинируя полученный результат с N = pq, получаем значения p и q, что приводит к факторизации N. Каждый шаг в доказательстве может быть выполнен за полиномиальное время. □

Замечание 1. Заметим, что в большинстве случаев публичный ключ e является полноразмерным, то есть e ≈ N2. Тогда данный метод Теоремы может быть применен к фактору N всякий раз, когда

Замечание 2. Поскольку e удовлетворяет ключевому уравнению ed – k (p2 – 1)(q2 – 1) = 1, то . В связи с Теоремой 2, чтобы обеспечить экспонента e должна удовлетворять

из которого мы выводим нижнюю границу для e

### Атака на основе алгоритма непрерывных дробей

***Теорема 3***

Пусть N = pq — модуль RSA с q < p < 2q и |p – q| = Nβ. Пусть e = Nα — публичный ключ, удовлетворяющий уравнению ed – k (p2 – 1)(q2 – 1) = 1 при d = Nδ. Если δ < 2 – β – ½ α, то можно найти p и q за полиномиальное время.

Доказательство.

Предположим, что N = pq при q < p < 2q и что открытая экспонента e удовлетворяет ключевому уравнению ed – k (p2 – 1)(q2 – 1) = 1. Тогда

Это приводит к

Используя ключевое уравнение, получаем k (p2 – 1)(q2 – 1) = ed – 1 < ed. Тогда

и

По Лемме 1 имеем

где последнее неравенство справедливо для N ≥ 8. Следовательно, используя e = Nα, |p – q| = Nβ, и d = Nδ получаем

Если < то есть δ < 2 – β – ½ α, тогда по Теореме 1

Отсюда следует, что можно найти среди сходимостей непрерывной дроби разложения Тогда, используя значения k и d в ключевом уравнении ed – k (p2 – 1)(q2 – 1) = 1, получаем Комбинируя это с N = pq, находим p и q. □

Заметим, что если p и q таковы, что p - q ≈ N0.5, то β ≈ ½ и ограничение на δ в Теореме 3 является . [1]

## Атака Копперсмита

Атака применяется на RSA, когда нам частично известен открытый ключ или когда открытая экспонента мала. Все это завязано на теореме Копперсмита.

### Теорема Копперсмита

Это теорема, которая позволяет найти нули приведенного (нормированного) многочлена по определенному модулю.

Теорема находит корни многочлена по модулю , которые меньше . Преимущество теоремы в том, что она дает возможность находить корни по составному модулю. Если же простое, то лучше использовать что-то другое, потому что теорема Копперсмита будет неприлично долго все считать.

***Формулировка теоремы***

Пусть – нормированный многочлен степени . Пусть . Тогда по паре атакующий эффективно найдет все целые числа , удовлетворяющие .

Время же, за которое злоумышленник сможет подобрать нужные числа, определяется выполнением LLL-алгоритма на решетке размерности .

### Описание атаки

***Предисловие***

Для начала надо вспомнить, что открытый ключ RSA это пара чисел *<N, e>*, где *N* — это произведение двух простых *p* и *q*, а *e* — открытая экспонента.

Число *e*, такое *(1 < e < ϕ(N)*, где *ϕ(N)=(p-1)(q-1)* — функция Эйлера от N) взаимно простое со значением этой самой функции Эйлера. Секретный ключ определяется через закрытую экспоненту . И выполняется условие . Пара публикуется в качестве открытого, а в качестве закрытого. Так вот первая пара нам и нужна.

***Формулировка атаки***

Собственно атака заключается в том, что у нас есть открытый ключ *<N, e>*, где *N* длины *n* бит. А мы возьмем множество . Пусть M ∈ ZN сообщение длины не более *n – m* бит. Определим наши сообщения различные целые числа с условием

Так вот, если злоумышленник получит *<N, e>* и шифры *C1* с *C2* от наших *M1* и *M2*, соответственно, но не получит , то он сможет восстановить сообщение .

Если же наша открытая экспонента будет маленькой, то будет следующий сценарий. Например, у нас есть сообщение “Ваш пароль: XXX”, тогда шифр-текст такого сообщения будет , где – известная часть “Ваш пароль: ”, а – собственно наш пароль. Тогда этот шифр-текст соответствует многочлену И неизвестная часть нашего сообщения – корень этого многочлена. Если мала, то алгоритм Копперсмита позволит найти наш пароль.

## Атака дешифрования на основе многоядерных графических процессоров с CUDA

### Факторизация составного числа методом Полларда .

Метод Факторизации Полларда был разработан Джонатаном Поллардом в 1974 г. Метод основан на малой теореме Ферма, которая гласит:

Если — простое число, а — целое число, не делящееся на , то

Разложить на множители большое число — значит найти простое число , такое что , а затем получить формулу: , откуда следует, что

Если предположить, что равно , число может быть увеличено от до тех пор, пока . Однако использование данного метода для нахождения точного не является эффективным, поскольку потребует выполнения операций, а временная сложность растет экспоненциально при увеличении . Это означает, что необходимо найти способ быстрого нахождения точного значения . Идея алгоритма факторизации Полларда состоит не в том, чтобы найти точное напрямую, а получить следующее значение формулы для целого числа : , :

Следовательно, для решения задачи разложения необходимо найти целое число , удовлетворяющее условию . Чтобы получить точное , возможность выполнения условий увеличивается при генерации большого количества простых чисел перед факторизацией, что может быть достигнуто с использованием алгоритма Решета Эратосфена.

### Алгоритм факторизации Полларда на основе модели программирования CUDA

*CUDA* — это расширение для языков программирования *C/C++*, с помощью которого пользователи могут писать масштабируемые многопоточные программы для вычислений на графических процессорах. Реализация программы *CUDA* делится на две части: хост и устройство. Первая часть в основном выполняется процессором, часть, названная устройством в основном, выполняется на графическом процессоре. Программа, которая выполняется на устройстве (видеопроцессоре), называется ядром. Ядро может вызываться одновременно на большом наборе *GPU* потоков, и программа. Эти потоки имеют иерархическую организацию, которая может быть объединена в системы блоков и сетей.

Предлагается использовать следующий метод для факторизации чисел на основе *GPU*:

Цель: найти делитель или составного числа .

1. Сформировать таблицу простых чисел в оперативной памяти программы. Загрузить сформированную таблицу в область глобальной памяти графического процессора.
2. Запустить значительное число ядер, в каждом из которых выполнять следующие действия:
   1. Поместить Номер текущего потока;
   2. Поместить Общее количество запущенных потоков;
   3. Поместить
   4. Выполнять для от размерности текущего блока до :

Выполнять для

* + - * Поместить
      * Поместить
      * В случае вернуть .

Поместить

## Разложение на множители с помощью квантового компьютера

### Основные понятия

Если в классическом компьютере единицей информации является бит, принимающий значения либо 0, либо 1, то в квантовом компьютере эту роль выполняет кубит.

**Кубит** – это квантовая система (по сути вектор в комплексном векторном пространстве), которая, как и бит, имеет два базисных состояния и , но в отличие от бита, кубит может находиться в любом суперпозиционном состоянии , где *a*, *b* – комплексные числа, . Таким образом, квантовый бит может принимать бесконечно много значений, но как результат измерения мы получим либо состояние с вероятностью , либо состояние с вероятностью .

**Квантовый регистр** – это набор кубит. В качестве примера рассмотрим суперпозицию для квантового регистра, состоящего из 3 кубит (сумма всех возможных тензорных произведений состояний каждого кубита в регистре):

Любая операция над регистром называется **гейт**. Гейт математически задаётся унитарной матрицей (т.е. для любой операции есть обратная, иначе говоря, после выполнения операции вектор всегда можно вернуть в исходное состояние):

### Алгоритм Шора

Алгоритм Шора был разработан еще в 1994 году. Его основное достоинство – полиномиальная сложность разложения числа на множители: .

В начале алгоритма необходимо сгенерировать такое число *x,* что выполняется . Квантовый компьютер решает задачу поиска периода *r* числа *x* (*mod N),* то есть выполнения равенства:

Порядок *r* также является периодом функции

Также необходимо рассчитать *q: .*

Алгоритм использует два квантовых регистра. Первоначально каждый из них состоит из совокупности кубитов в нулевом булевом состоянии Первый регистр используется для размещения *t* функции *f(t).* На первом шаге алгоритма состояние первого регистра переводится в равновероятную суперпозицию всех булевых состояний *N* за счет использования операции Уолша-Адамара. Другими словами, все кубиты первого регистра переводим в состояние суперпозиции вида . Второй регистр остается в состоянии . Получим следующее состояние для системы двух регистров:

На втором шаге система из двух регистров переводится в новое состояние с помощью гейта управляемого возведения в степень (controlled-power gate) *Uf* :

Можно заметить, что между состояниями обоих регистров образуется некоторая связь. Теперь второй регистр хранит суперпозицию всех возможных значений функции *f(t)*.

На третьем шаге над первым регистром выполняется квантовое преобразование Фурье (Quantum Fourier Transform, QFT):

Данное преобразование переводит состояние первого регистра из области кодирования числа в частотную область. Когда мы применяем преобразование Фурье к первому регистру, мы получаем распределение вероятностей амплитуд наших кубитов. В этом распределении каждой амплитуде соответствует конкретный период функции *f(t)*. Таким образом, преобразование Фурье дает возможность выбрать правильное значение периода с большей вероятностью.

На последнем шаге квантовой части алгоритма происходит измерение значения из первого регистра – потенциального периода функции *f(t).* Схема алгоритма представлена на Рисунке 1.

A picture containing diagram, sketch, font, line

Description automatically generated

Рисунок 1 — Схема алгоритма Шора

Теперь алгоритм переходит к выполнению на классическом компьютере. По полученному значению периода *r* можно вычислить разложение числа N на множители, причем время данной операции ограничено *log N.*

Если полученный период *r* нечётный ИЛИ выполняется сравнение

, то необходимо вернуться в самое начало и изменить входное число *x*.

Иначе искомый нетривиальный делитель *N* вычисляется по формуле:

**Обоснование**: так как выполняется , то получаем Так как условие не выполняется, значит множители не делятся на *N* одновременно.

Количество кубит, необходимое для выполнения алгоритма, оценивается , где *N* – число, необходимое разложить на множители.

Таким образом, согласно оценкам ученых [6], для взлома RSA-2048 потребуется около 20 миллионов кубитов. При этом время взлома будет составлять примерно 8 часов. Для сравнения, самый мощный на момент написания работы квантовый компьютер имеет 433 кубита [7].

### Оптимизация вычислений на квантовых компьютерах

23 декабря 2022 года вышла статья, в которой группа китайских ученых предложила оптимизировать вычисления на квантовых компьютерах [8].

Ученые смогли факторизовать 48-битный модуль на 10-кубитном квантовом компьютере. И они подсчитали, что можно масштабировать их алгоритм для использования с 2048-битными ключами, используя квантовый компьютер всего с 372 кубитами, что в современных реалиях не кажется чем-то далёким. Прорыв обещан путем использования алгоритма Шнорра (не путать с алгоритмом Шора) и оптимизировать его дополнительным шагом алгоритма квантовой аппроксимации (QAOA).

Схема алгоритма представлена на Рисунке 2. Как показано на схеме, алгоритм Шнорра включает два шага: нахождение достаточного количества гладких пар (sr-pairs) и решение полученной системы линейных уравнений. Как правило, нахождение sr-пар является наиболее важной и трудоемкой частью алгоритма, в то время как решение системы уравнений может быть выполнено за полиномиальное время.

A picture containing text, diagram, screenshot

Description automatically generated

Рисунок 2 — Схема предложенного алгоритма

В алгоритме Шнора задача поиска гладких пар преобразуется в задачу о ближайшем векторе на решетке (CVP) и решается алгоритмами редукции решетки, такими как алгоритм Бабая [9]. Вероятность получения гладкой пары пропорциональна качеству решения CVP. То есть, чем ближе решение CVP к идеальному, тем более эффективным будет получение гладкой пары. Исходя из этих фактов, была предложена схема, которая использует алгоритм квантовой оптимизации (QAOA) для решения CVP, полученного с помощью алгоритма Бабая.

По экспериментальным данным 48-битное число 261980999226229 было разложено с помощью этого метода на 10 кубитах. Количество кубит, требуемое для реализации алгоритма оценивается сложностью , что намного лучше, чем необходимое для алгоритма Шора. На Рисунке 3 изображен график роста количества необходимых кубит от количества бит числа N (красный – алгоритм Шора, синий – предложенная оптимизация).

A graph with a red line

Description automatically generated with low confidence

Рисунок 3 — График сравнения сложности требуемых кубит для двух алгоритмов.

Однако на практике алгоритм был встречен с долей скептицизма. Во-первых, нет никаких доказательств, что алгоритм масштабируем на практике. Кроме того, сами авторы статьи сомневаются, что квантовая оптимизация алгоритма будет уменьшать время вычисления из-за неоднозначной сходимости *QAOA*.

## Сравнительный анализ методов

В результате анализа предметной области была составлена Таблица 1, сравнивающая исследуемые методы по определенному набору параметров.

Таблица 1 — Сравнение современных методов факторизации числа

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Распараллеленный метод разложения Полларда** | **Метод непрерывных дробей** | **Атака Копперсмита** | **Квантовые компьютеры** |
| Временная сложность | ,  где *gridDim.x*, *blockDim.x* – параметры видеокарты в архитектуре *CUDA*. | Полиномиальное время |  |  |
| Условие выполнения | Применим для произвольного *N* |  | Малая экспонента, либо известный открытый ключ | В теории применим для любого N |
| Применимость | Применим только при наличии необходимого оборудования *(GPU Nvidia*, поддерживающая *CUDA)*. | Не требует особых условий | Не требует особых условий | Не применим из-за развития квантовых компьютеров |

# Практическая часть

## Атака методом непрерывных дробей

По Теореме 3 после нахождения получим следующую систему уравнений:

В первом уравнении обозначим правую часть как *х*, а во втором выразим *q* и подставим в первое уравнение:

Рассмотрим следующий пример:

Решаем квадратное уравнение (3) через дискриминант и получаем два корня: . Первый корень не является простым числом. Проверим, совпадает ли второй корень с *p* или *q*, которые сохранены в файле (Рисунок 4).

Изображение выглядит как текст, Шрифт, снимок экрана

Автоматически созданное описание

Рисунок 4 — Результат атаки

### Сравнение с атакой Винера

В Таблице 2 представлено сравнение двух атак по используемым методам, атакуемой криптосистеме, искомым параметрам.

Таблица 2 — Сравнение атаки методом непрерывных дробей с атакой Винера

|  |  |  |
| --- | --- | --- |
|  | Атака методом непрерывных дробей | Атака Винера |
| Параметры криптосистемы | φ(n) = (p2 – 1)(q2 – 1) | φ = (p – 1)(q – 1) |
| e ϵ Z/φ(n)Z, e > ½n | e ϵ Z/φ(n)Z |
| d = e-1 (mod φ(n)), | d = e-1 (mod φ(n)) |
| Цель злоумышленника | p, q | d |
| Используемый метод | Построение подходящей бесконечной дроби,  Решение квадратного уравнения | Построение подходящей бесконечной дроби |

Атака методом непрерывных дробей может быть более опасна. Пусть в системе есть несколько пользователей, каждый из которых имеет свои пары (ei, n), (di, n). Тогда, зная разложение n = pq, злоумышленник может вычислить φ(n), а отсюда вычислить все di. В атаке Винера компрометируется только один секретный показатель.

## Атака Копперсмита

Атака Копперсмита может быть применена в случае, когда часть текста неизменяема. Например, у нас есть сообщение *m* = “The secret word is ikizi”,

Известная часть *B* – “The secret word is”,

Неизвестная часть *x* – “ikizi”, чья длина меньше одной трети длины *N.*

Теперь предположим, что данное сообщение *m* было закодировано RSA с экспонентой *e* = 3. Тогда наш шифртекст *c* = *m3 (mod N)* = *(B + x)3 (mod N)*. Тогда, если мы узнаем *B, c и N*, то мы сможем использовать их в полиноме *p(x) = (B + x)3 – c* и восстановить неизвестную часть *x0*, удовлетворяющую *p(x0) = (B + x0)3 – c = 0* *(mod N),* пока *|x0| <N1/3* – длина *x0* меньше длины одной трети *N.*

Очевидно, что если B = 0, то всё сообщение состоит из секретного текста, то есть, из *x0*,*c=(x0)3*. В таком случае, при условии *x0 <N1/3* мы можем восстановить неизвестную часть x0=c1/3, если возьмем кубический корень.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 5 — Создание сообщения

Изображение выглядит как текст, Шрифт, снимок экрана, белый

Автоматически созданное описание

Рисунок 6 — Решение полинома

Если же B не равно 0, то у нас также есть возможность узнать секретную часть сообщения. Но стоит помнить, что мы не можем узнать секретную информацию любой длины с помощью метода Копперсмита, потому что граница *X*, относительно которой мы ищем наши корни *x0* зависит от *N*. Если неизвестная часть сообщения *x0* будет 250 бит, а *N* будет 512 бит, *e* = 3, то атака не увенчается успехом, потому что *x0 > N1/3*. Но стоит нам увеличить размер *N*, к примеру, до 1024 бит и оставить неизвестную часть x0 такой же с 250 бит, то *x0* снова становится уязвимой для атаки Копперсмита.

Изображение выглядит как текст, Шрифт, снимок экрана

Автоматически созданное описание

Рисунок 7 — Создание сообщения

Изображение выглядит как текст, Шрифт, линия, снимок экрана

Автоматически созданное описание

Рисунок 8 — Решение полинома

Теперь увеличим размер N, остальные параметры оставим теми же и проверим результат.

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

Рисунок 9 — Создание сообщения

Изображение выглядит как текст, Шрифт, линия, снимок экрана

Автоматически созданное описание

Рисунок 10 — Решение полинома

Атака работает успешно и в том случае, если секретная информация находится не после, не перед, а чередуется с неизменяемой информацией. Например, есть сообщение "Login is ikizi and password is best". В данном сообщении есть неизменяемые части “login is” и “and password is”, среди которых расположены логин и пароль.

Изображение выглядит как текст, Шрифт, линия, снимок экрана

Автоматически созданное описание

Рисунок 11 — Создание сообщения

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

Рисунок 12 — Решение полинома

Сам метод Копперсмита заключается в том, что мы составляем такой полином, который связывает часть сообщения *x0*, которую нам необходимо найти, с шифртекстом *c.* У полинома *f(x) = xe – c* находим корни, зная, что они достаточно малы, потому что их значения ограничены сверху *X = N(1/d) – e.*

Для нахождения корней полинома мы также создаем матрицу *M*, которая связывает коэффициенты полинома и модуль *N*. А после применяем алгоритм *LLL*, чтобы повысить точность найденных результатов. Иными словами, использование такой матрицы после применения алгоритма *LLL* позволяет получить коэффициенты приближенного полинома, которые впоследствии используются для построения полинома *f(x)*. После того, как мы нашли приближенные коэффициенты, можно попытаться найти корни полинома.

Изображение выглядит как текст, Шрифт, снимок экрана

Автоматически созданное описание

Рисунок 13 — Создание подходящей матрицы

Теперь разберем случай, когда нам известна часть ключа, а именно то, как устроен один из множителей *N*. Например, нам известно всё до последних 160 бит у множителя *p*

Изображение выглядит как текст, Шрифт, линия, снимок экрана

Автоматически созданное описание

Рисунок 14 — Поиск множителей

Нули показывают неизвестную нам часть. Тогда мы можем попытаться найти этот множитель p, исходя из имеющейся информации. Для этого нам так же надо составить матрицу, применить LLL и найти корни полинома

Изображение выглядит как текст, Шрифт, снимок экрана, линия

Автоматически созданное описание

Рисунок 15 — Алгоритм поиска множителей

Если корни полинома будут найдены, то среди них будет единственный целочисленный, который нам и понадобится. Для того, чтобы найти полный множитель *p*, необходимо обратиться к известной части и просуммировать её с найденным целочисленным корнем. Проверим, является ли *p* реальным множителем *N*. Если разделить *n* на найденный *p*, то мы получим наш *q*.

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

Рисунок 16 — Результат разложения на множители

Атака Копперсмита показывает себя очень эффективно при небольших значениях открытой экспоненты *e* и в том случае, если известна часть шифртекста. Важным условием, определяющим успех атаки, является то, что неизвестная часть сообщения должна быть достаточно мала, а именно *|x0| <N1/3*. Стоит отметить, что несмотря на то, что существуют рекомендации для выбора *e*, многие системы всё равно используют небольшие значения, так как если экспонента будет велика, то процесс вычисления будет ресурсоемким. В случае, если *e* невелика, необходимо сократить размер неизменяемого текста в сообщении, чтобы не выполнялось правило *|x0| <N1/3.*

## Факторизация Полларда. CUDA

На языке *C++* средствами фреймворка *CUDA* была разработана программа, реализующая разложение составного числа на множители. Для обработки больших чисел использовалась библиотека *MPZ*.

Для запуска программы необходимо передать в качестве входного аргумента число, для которого ищется разложения, а также путь в файловой системе к текстовому файлу, содержащему таблицу простых чисел, используемых в процессе разложения.

Таблица простых чисел подгружается в оперативную память процесса, после чего копируется в память видеопроцессора. Далее запускается большое число потоков на видеокарте, выполняющих подбор вариантов разложения выбранного числа.

В процессе вычисления относительно номера текущего потока и количества запущенных потоков вычисляется стартовое значение и смещение показателя B, относительно которого начинается перебор числовых множителей. Для выбранного B формируется произведение всех простых чисел от 2 до B:

A picture containing text, font, screenshot

Description automatically generated

Рисунок 17 — Начало цикла. Вычисление произведения множителей в пределах до B.

Далее вычисляется значение , после чего алгоритмом Евклида находится наибольший общий делитель . Значение наибольшего общего делителя в пределах от 2 до означает нахождение одного из искомых множителей. В этом случае алгоритм завершается, а найденный множитель копируется в оперативную память процесса и выводится на экран пользователя. Иначе, значение показателя a изменяется относительно характеристик запущенного пула потоков и процесс вычисления повторяется.

A picture containing text, screenshot, font, number

Description automatically generated

Рисунок 18 — Проверка полученного значения. Завершение в случае успеха.

### Тестирование

Для реализованного алгоритма было проведено тестирование разложения чисел различной длины. Для генерации простых чисел использовался генератор *PrimeSieve* (<https://github.com/kimwalisch/primesieve>). В тестировании использовался файл, содержащий простые числа в пределах 200000000. Полученные результаты отражены в Таблице 3:

Таблица 3 — Полученные результаты при тестировании

|  |  |  |  |
| --- | --- | --- | --- |
| Число | Битовая длина | Найденный делитель | Время разложения (ms) |
| 78952d93f39b02d9 | 64 | b74b30cd | 3397 |
| 1275f09deafe4b0b5a3f | 77 | e8d495cdc1 | 235 |
| e554416c585117199d7e3 | 88 | ba7a8f | 654 |
| 13daf8d446d5f40d6070bd9 | 94 | 138c8165 | 9424 |
| 1bb01f7f532434d65b7762f496cb | 110 | 313e2911 | 33404 |
| fce99deec80e9247906656915e67b | 120 | 7a5854d | 73 |
| 187bba879ee72816e37db1bc5d80df01 | 130 | 11782cff | 216 |

### Анализ применимости при атаках на современные системы

Основной сложностью в применении данного алгоритма является необходимость формирования, а также, хранения в оперативной памяти таблицы простых чисел существенного объема. Во время тестирования, для проведения атаки использовалась таблица, содержащая простые числа в пределах и занимающая 1,02 GB дискового пространства. Очевидно, что для разложения чисел, используемых в современных криптосистемах ( бит и выше), потребуется таблица, содержащая терабайты данных.

Даже если удастся сформировать такую таблицу (например, средствами небезызвестного алгоритма Решета Эратосфена), никакое современное вычислительное устройство не позволит поместить полученные данные в оперативную память, а тем более, в память видеоадаптера.

Одним из решений данной проблемы может быть попытка организации «Радужных» таблиц, содержащих заранее преобразованные данные простых чисел меньшего объема, для использования внутри алгоритма.

Другим из возможных решений может быть организация специальной архитектуры компьютера, позволяющей обращаться изнутри ядра видеокарты к данным, находящимся в энергонезависимой памяти.

## Реализация алгоритма Шора

На языке Python с использованием библиотеки qiskit была разработана программа, моделирующая алгоритм Шора.

На выход программы подаётся число N, подлежащее факторизации, а также число *a,* меньше N и взаимно простое с ним. После этого происходит инициализация квантовых регистров и классического регистра, куда будут записываться результаты измерения квантовой схемы (см. Рисунок 16).

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 19 — Инициализация регистров

Далее происходит создание квантовой схемы с помощью функции *QuantumCircuit,* после чего начинается заполнение схемы необходимыми гейтами (см. Рисунок 17). Сначала для верхнего регистра выполняется гейт Уолша-Адамара, затем в схему добавляется гейт управляемого возведения в степень *U*. Затем с помощью функции *create\_inverse\_QFT* выполняется добавление гейта преобразования Фурье, после чего на верхний регистр добавляется измеряющее устройство, сохраняющее результат измерений в классический регистр.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Автоматически созданное описание

Рисунок 20 — Инициализация схемы работы симуляции квантового компьютера

Далее необходимо запустить симуляцию работы квантового компьютера:

simulation = execute(circuit, backend=Aer.get\_backend('qasm\_simulator'), shots=2048)

Теперь, для каждого найденного периода в цикле выполняется классическая часть алгоритма Шора (см. Рисунок 18). Здесь сначала выполняются необходимые проверки, затем выполняется нахождение нетривиального делителя N.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Автоматически созданное описание

Рисунок 21 — Классическая часть алгоритма Шора

Реализованная симуляция работы квантового компьютера и алгоритм Шора были протестированы для различных значений факторизуемого числа N. Сначала, как и в классической версии алгоритма Шора, на вход было подано число 15, а число, меньшее 15 и взаимно простое с ним – 7. Результат представлен на Рисунке 19. Для числа 247 результат тестирования представлен на Рисунке 20.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 22 — Факторизация числа 15

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 23 — Результат факторизации числа 247

В результате тестирования программы была составлена таблица (см. Таблица 4). Как и предполагалось, в рамках моделирования при увеличении факторизуемого числа N увеличивается число кубит, необходимое для работы алгоритма Шора, а также время работы алгоритма. Это означает, что бОльшие числа требуют более мощных квантовых компьютеров, которые на данный момент не достигли точки взлома современной используемой криптосистемы RSA (например, RSA-2048).

Таблица 4 — Результаты тестирования программы моделирования алгоритма Шора

| **Число N** | **Размерность (бит)** | **Необходимое кол-во кубит** | **Время факторизации**  **(секунды)** |
| --- | --- | --- | --- |
| 15 | 4 | 18 | 9 |
| 21 | 5 | 22 | 26 |
| 28 | 5 | 22 | 30 |
| 39 | 6 | 26 | 744 |
| 91 | 7 | 30 | 31 |
| 195 | 8 | 34 | 60 |
| 247 | 8 | 34 | 54 |

# Заключение

В ходе выполнения данной курсовой работы были реализованы 4 различные атаки на криптосистему RSA. Они позволяют решить задачу разложения на множители. Все эти атаки возможны только потому, что в исследуемых криптосистемах использовались «слабые» параметры, то есть малые p и q, и как следствие были сгенерированы малые e и d.

Для противодействия атакам на криптосистему RSA рекомендуется использовать достаточно большие модули (1024 или 2048 бит). Также при генерации параметров стоит проверять, не удовлетворяют ли они условиям некоторых атак, например, атаке Винера или атаке на основе непрерывных дробей.

Алгоритм Полларда, а также его параллельная реализация, могут использоваться для разложения произвольных составных чисел. Основным требованием по выбору параметров криптосистемы для противодействия данной атаке остается использование ключей шифрования большой длины.

На данный момент квантовые компьютеры не получили достаточного развития, однако с каждым годом ситуация изменяется. Для защиты от алгоритма Шора рекомендуется использовать модуль RSA размера не менее 2048 бит, поскольку чем больше модуль, тем большее количество кубит необходимо задействовать, что на данный момент развития квантовых компьютеров является критическим ресурсом.

# Список использованных источников

1. «Криптоанализ вариантов RSA с простыми числами, имеющими общие старшие значащие биты» Мерьем Шеркауи-Семмуни, Абдеррахман Нитай, Вилли Сусило и Джозеф Тониен. <https://eprint.iacr.org/2021/1632.pdf>
2. «Новая атака на три варианта криптосистемы RSA» Мартин Бандер, Абдеррахман Нитадж, Вилли Сусило, Джозеф Тониен. <https://core.ac.uk/download/pdf/237331978.pdf>
3. «Алгоритмы полиномиального времени простой факторизации и дискретных логарифмов на квантовом компьютере» Питер В. Шор. <https://arxiv.org/pdf/quant-ph/9508027.pdf>
4. «Эффективный параллельный алгоритм расшифровки RSA для многоядерных графических процессоров с CUDA» Yu-Shiang Lin, Chun-Yuan Lin1, Der-Chyuan Lou. <https://www.researchgate.net/publication/284228845_Efficient_Parallel_RSA_Decryption_Algorithm_for_Many-core_GPUs_with_CUDA>
5. «Документация по набору инструментов CUDA 12.1». <https://docs.nvidia.com/cuda/>
6. «Как факторизовать 2048-битные целые числа RSA за 8 часов, используя 20 миллионов зашумленных кубитов» Крэйг Гидни, Мартин Экеро. <https://arxiv.org/abs/1905.09749>
7. «Osprey, самый мощный в мире квантовый компьютер». <https://newatlas.com/computers/ibm-osprey-worlds-most-powerful-quantum-computer/>
8. «Факторизация целых чисел с сублинейными ресурсами на сверхпроводящем квантовом процессоре» Bao Yan, Ziqi Tan, Shijie Wei, Haocong Jiang, Weilong Wang, Hong Wang, Lan Luo, Qianheng Duan, Yiting Liu, Wenhao Shi, Yangyang Fei, Xiangdong Meng, Yu Han, Zheng Shan, Jiachen Chen, Xuhao Zhu, Chuanyu Zhang, Feitong Jin, Hekang Li, Chao Song, Zhen Wang, Zhi Ma, H. Wang, Gui-Lu Long. <https://arxiv.org/abs/2212.12372>
9. «Быстрый факторинг целых чисел с помощью алгоритмов SVP, исправленный» Клаус Петер Шнорр. <https://eprint.iacr.org/2021/933.pdf>