Санкт-Петербургский политехнический университет Петра Великого

Институт прикладной математики и механики

**Кафедра «Информационная безопасность компьютерных систем»**

**КУРСОВАЯ РАБОТА**

**Атаки на схемы подписи RSA на основе теории решеток**

по дисциплине «Криптографические методы защиты информации»

Выполнил

студент гр. 43609/1 <*подпись*> Д.А. Куликов

Руководитель

старший преподаватель <*подпись*> Е.Ю. Павленко

«\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2019 г.

Санкт-Петербург

2019

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_Toc11536828)

[1. Аалгоритм LLL и его свойства 5](#_Toc11536829)

[2. Неявное разложение двух RSA модулей 9](#_Toc11536830)

[3. Метод Копперсмита 12](#_Toc11536831)

[4. Метод Boneh И Durfee 17](#_Toc11536832)

[5. Атака на малый 19](#_Toc11536833)

[6. Реализация атаки на малый 23](#_Toc11536834)

[ЗАКЛЮЧЕНИЕ 31](#_Toc11536835)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 32](#_Toc11536836)

[ПРИЛОЖЕНИЕ А 33](#_Toc11536837)

# ВВЕДЕНИЕ

В 1995 году Копперсмит выпустил статью о том, как атаковать RSA с помощью теории решеток, используя алгоритм LLL. Howgrave-Graham пересмотрел предложенный Копперсмитом алгоритм и сделал его проще для применения. Его работа была реализована для решения различных проблем, начиная от раскрытия зашифрованного сообщения, при известной его части, до взлома криптосистемы RSA, если есть достаточно хорошее приближение одного из простых делителей [3].

Атаки, основанные на теории решеток, были рассмотрены и в других исследованиях. В 1990 году, Винер обнаружил, что можно успешно взломать RSA, если закрытый показатель достаточно мал [3]. В 2000 году Boneh и Durfee улучшили эту границу используя технику, подобную, подобной описанной Копперсмитом [3]. Их работа была позже пересмотрена и улучшена Herrmann и Мэем.

Александр Мэй в работе «Using LLL-Reduction for Solving RSA and Factorization Problems: A Survey» обнаружил уязвимость криптосистемы CRT-RSA с малыми открытыми показателями к атакам с использованием метода Копперсмита. В нескольких работах изучалась эта проблема, и было выявлено два основных метода. Bleichenbacher и Мэй в работе [6] предложили атаку для малого , когда простой делитель значительно меньше, чем . Предложенная атака работает при [6]. Jochemsz и Мэй предложили атаку для малых и , когда простые делители и близки. Атака работает для [3].

В году Atsushi Takayasu, Yao Lu и Liqiang Peng [2] предложили две улучшенные атаки на систему CRT-RSA: атака на малый , при (улучшение атаки от Bleichenbacher – Мэя) и атака на малые и , при ( улучшение атаки от Jochemsz – Мэя). В своей работе они улучшили конструкцию решетки, используя структуру генерации ключей CRT – RSA, помимо формального, приводят доказательство путем экспериментов на реальном вычислителе.

В данной работе изучаются основные методы использования теории решеток для атак на схемы подписи RSA. Помимо этого, проводится анализ постепенного улучшения этих методов и способов их достижения, в целях оценки возможности улучшения данных методов в будущем или применения в других задачах.

# АЛГОРИТМ LLL И ЕГО СВОЙСТВА

Алгоритм Lenstra-Lenstra-Lovasz используется для уменьшения длин векторов базиса решетки и вычисляется за полиномиальное время максимальная длина входного вектораколичество векторов базиса. Решетка остается неизменной, но вектора нового базиса становятся короче.

**Определение 1.** Пусть решетка с базисом , LLL алгоритм, примененный к базису произведет новый базис решетки , удовлетворяющий следующим свойствам [3]:



, где элементы базиса, полученного ортогонализацией Грама-Шмидта.

Суть алгоритма состоит в том, что на каждой итерации рассматривается подмножество из первых векторов. Если условия определения 1 для данных векторов выполнены, то число увеличивается, иначе базис изменяется. Как только станет равным , алгоритм заканчивает свою работу.

**Алгоритм 1.** Построение LLL – приведенного базиса решетки [1].

Входные данные: базис решетки .

Выходные данные: LLL – приведенный базис

Шаги алгоритма:

1. Положить
2. Выполнить ортогонализацию Грама-Шмидта для базисаи получить ортогональный базисодновременно вычислив коэффициентыи значения
3. Положить *.*
4. *При* выполнять*:*

4.1 Вычислить;

4.2 Положить;

4.3 Дляположить;

4.4 Положить *.*

1. Привыполнить*:*
   1. Положить

*;*

* 1. Поменять местами векторы
  2. *При* поменять местами коэффициентыи

для

* 1. Дляположить:
  2. Положитьи вернуться на шаг

1. Привыполнить:
   1. Дляпривыполнить:

6.1.1 Вычислить;

6.1.2 Положить

6.1.3 Дляположить *;*

6.1.4 Положить *.*

* 1. Положить

1. Еслито вернуться на шаг
2. Вернуть

Условно алгоритм делится на части:

1. Добиваемся .

Стоит отметить, что при это не изменяются .

1. Проверяем условие определения (в случае приведенного алгоритма ).

Возможны случая:

* 1. , тогда меняем местами и и возвращаемся на этап, уменьшая .
  2. : , если данное неравенство выполняется, то переходим на 3 этап, иначе и возвращаемся на этап.

- наибольший номер: . Уменьшаем как на этапе, при этом гарантируется неизменны. Продолжаем, пока все коэффициенты не удовлетворяют условию.

Данный алгоритм всегда заканчивает свою работу, поскольку шаг 5 выполняется конечное число раз. Более того, если для всех то сложность алгоритма равна арифметических операций, а целые числа, встречающиеся в процессе работы алгоритма, имеют в системе счисления по основаниюдлину[1].

LLL дает приближение решения задачи кратчайшего вектора. Это полезно, потому что, если мы рассмотрим векторы строк базиса решетки как векторы коэффициентов полиномов, мы можем найти линейную комбинацию из тех многочленов, которые имеют особенно малые коэффициенты.

**Определение 2.** Пусть решетка размерности . За полиномиальное время LLL алгоритм выдаст сокращенные вектора , такие что [3]:

Можно заметить, что, изменяя размерность и определитель базиса решетки, мы можем влиять на длины его векторов.

# НЕЯВНОЕ РАЗЛОЖЕНИЕ ДВУХ RSA МОДУЛЕЙ

**Теорема 1 (Минковского)**. Пусть целочисленная решетка. Тогда содержит ненулевой вектор [7]:

кратчайший ЛНЗ вектор

Мэй [7] предположил следующую ситуацию: пусть даны два разных RSA модулягдесовпадают взначащих младших битах, то есть*: .* Рассмотрим следующие уравнения:

Эти два уравнения содержатнеизвестных. Взяв эти уравнения по модулю мы можем избавиться от двух из них:

Так какнечетные, разделим на них:

Множество решений:

Образует подгруппу . Таким образом, – решетка, натянутая на строковые вектора базисной матрицы:

Известно, что *.* Также предполагается, что этот вектор кратчайший в *,* к тому же длина кратчайшего вектора ограничена значением *=* по теореме 1 (Минковского).

Предположим, чтоимеют величинубит, к тому же они простые, имеем. Еслидостаточно мало, томеньше границы Минковского и, следовательно, мы можем ожидать, что является самым коротким вектором в *,* для этого должно выполняться следующее неравенство:

Еслиожидается, чтократчайший вектор в *.* Можно найти кратчайший вектор в *,* используя алгоритм LLL (в оригинале статьи [7] предлагается использовать метод Гауссова исключения) над базисом *,* получим эквивалентный базис :

Покажем, что *.*

Посколькуимеет полный ранг, по неравенству Адамара имеем:

Подставляяи используяведет к:

Это применимо для любого вектора решетки *,* для которого выполняется *,* значит *.* В противном случае, *,* что противоречитвыбору *.* Таким образом, каждый *–* произведение константы на *.* Можно заметить, что длявыполняетсяСледовательно, , если

Следовательно, получив, для некоторогопустьтогдаНопростые числа и к тому жев противном случае мы могли бы разложить *,* вычисливСледовательно*,* и получаем *,* что и требовалось показать.

Время работы разложения определяется временем выполнения Гауссова исключения, которое выполняется для матрицы размерности за[7].

# МЕТОД КОППЕРСМИТА

Данный метод является основополагающим для всего направления атак на RSA, основывающихся на теории решеток.

Рассмотрим атаку стереотипных сообщений. Представим, что мы знаем часть сообщения. К примеру, мы знаем, что Алиса всегда отправляет ее сообщение следующим образом: «Пароль: Алиса».

Положим мы знаем , а сообщение имеет вид . Задачу можно представить в следующем виде:

Копперсмит доказал, что найти корень данного выражения можно за полиномиальное время, если и достаточно малы [3].

**Теорема 2 (Копперсмита).** Пусть – целое число, которое имеет делитель Пусть – полином от одной переменной степени и пусть . Тогда мы можем за времянайти все решенияуравнения:

*,* при

Поиск корней полинома над кольцом целых чисел по модулюсложная задача*,* поэтомупроизводится сведение ее к поиску корней полинома над кольцом целых чисел.Копперсмит предложил обратить внимание на такой полином [3]:

Чтобы перейти к , полиному – имеющему корень над кольцом целых чисел можно использовать следующую теорему [3].

**Теорема 3 (Howgrave-Graham).** Пустьполином от одной переменной смономами,при этом выполняются следующие условия:

Тогдаимеет решение над кольцом целых чисел.

Howgrave-Graham доказал, что можно найти полином, который будет иметь такой же корень над кольцом целых чисел, как у исходной функции, если его коэффициенты достаточно малы.

Далее, Копперсмит предложил использовать LLL алгоритм для этой цели.

LLL алгоритм имеет два полезных для этой цели свойства:

1. Он производит только целочисленные линейные операции на базовых векторах.
2. И ограничивает кратчайший вектор выходного базиса (видно из определения 2).

Первое свойство позволяет нам построить функцию, которая все еще имеет как корень по модулю *:*

Второе свойство позволяет использовать свойство теоремы 3 (Howgrave-Graham) .

Конкретно, для рассматриваемой атаки, в работе [3] было предложено генерировать полиномы(далее это), с помощью которых строится,следующим образом здесь степень полинома:

Для этих полиномов справедливо следующее:

1. Они имеют такой же корень, но по модулю
2. Каждая итерация вводит новый моном. Это позволяет нам построить треугольную решетку (чтобы легче было вычислить определитель, и он точно не был равен нулю).

Теперь нужно вычислить базис решетки с коэффициентами в качестве строк. Далее к этому базису необходимо применить алгоритм LLL, с помощью которого вычисляется вектор, удовлетворяющий условию:

По теореме 3 (Howgrave-Graham) имеем:

Из предыдущего уравнения имеем, что для успешного использования описанных свойств, нам необходимо:

Можно упростить оценку до:

Данная оценка используется также в [2, 6], и определяет возможность применения данного метода. В реализации, которая описана в данной работе, она также будет использоваться.

Исходя из этих свойств, Копперсмит ограничил значение в своей теореме. Чтобы использовать этот алгоритм необходимо определить значения и , пока не получим удовлетворительные оценки. Однако даже если сделать это неверно, можно найти правильный ответ (благодаря свойствам LLL алгоритма).

Для

в работе [3] (оценивая значения диагональных элементов матрицы, полученной при помощи приведенных выше полиномов) показывается, что для атаки необходимо выполнение:

Данный метод нашел широкое применение. Так, например в работе [4] производится атака с использованием данного метода.

Вспомним схему подписи RSA. Чтобы подписать сообщение , сначала вычисляется хеш , далее вычисляется сама подпись, в виде . Для CRT-RSA получим подпись используя соотношения:

и

Где и .

Вычисление подписи:

Такая схема позволяет выполнить подписание примерно в раза быстрее.

В работе [4] рассматривается атака на ISO/IEC 9796-2. В данном стандарте сообщение разбивается на части . Функция в нем имеет следующий вид:

– хеш функция.

В 2009 году,CJKNPописали атаку на случайную версию этого стандарта [4]. В данном случае, где – неизвестная часть, известны и либо известная, либо неизвестная строка. Обозначим (в соответствии со стандартом). Тогда предыдущая функция примет вид:

Таким образом, общее количество неизвестных бит .

Тогда атакующий решает задачу в следующем виде:

Где известное значение и .

Тогда, можно применить метод Копперсмита для атаки стереотипных сообщений, для следующего выражения:

Где . В работе [4] приводится оценка границы для этой атаки. При , можно вычислить, при условии, что .

# МЕТОД BONEH И DURFEE

Данный метод позволяет факторизовать , если секретная экспонента достаточно мала, а именно:

Вспомним структуру ключей RSA:

Здесь неизвестны и . Это можно записать в виде следующего полинома:

где и

Здесь в отличие от случая, описанного в третьем разделе данной работы, одного полинома недостаточно для нахождения решения, поэтому Копперсмит предложил взять два первых полинома, полученных с помощью LLL алгоритма базиса *.* Далее необходимо вычислить их результант. Найдя кореньиз результанта можно ввести его в, чтобы найти. Но решение может быть не найдено, например, еслине линейно независимые, результант будет равен нулю.

Boneh и Durfee [3] предложили построить следующие полиномы, как :

Для

Используя эти полиномы для построения решетки, балансируются переменные так, чтобы определитель треугольного базиса не превышал . Boneh и Durfee показали, что LLL дает успешный результат если [3]. Пример решетки, получающейся при помощи приведенных полиномов:

Для достижения улучшенных результатов ( они используют подрешетку, исключая некоторые полиномы. Таким образом уменьшается граница кратчайшего вектора, получаемого LLL, а матрица принимает следующий вид:

Видим, что при этом матрица перестает быть треугольной. Для обработки такого базиса, Boneh и Durfee разработали понятие геометрически прогрессивных матриц.

# АТАКА НА МАЛЫЙ

Для данной атаки, рассмотрим лемму Howgrave-Graham, для полиномов от переменных.

**Лемма 1 (Howgrave-Graham).** пусть *–* полином с не более чем мономами. Пустьи справедливо следующее [2]:

Тогданад кольцом целых чисел.

Чтобы решить уравнение спеременнымидостаточно найтиновых полиномов, которые имеют такой же корень, и норма которых достаточно мала, чтобы удовлетворить лемме 1 (Howgrave-Graham).

Снова рассмотрим CRT-RSA схему, как это сделали авторы работы [2]:

Если мы сможем решить следующее выражение, корень которого *,* томожет быть факторизовано.

Будем считать, чтобольше, чем *.* Домножим обе части на *:*

Корнем следующего сравнения является *,* после его нахожденияможет быть факторизовано:

Пусть *и .* Тогда значения корняограничены сверху

Мэй в работе [5] использует генерацию полиномов следующего вида:

Используя данные полиномы, матрица у Мэя [5] принимает вид:

Здесь строки состоят из коэффициентов семи полиномов:

Все полиномы разделяют корень . К тому же к базовым полиномам он добавил дополнительный – сдвиг: .

Полиномы, полученные путем применения LLLалгоритма к этой матрице, удовлетворяют лемме 1 (Howgrave-Graham), когда:

Это выполняется тогда и только тогда, когда:

Основная идея этого подхода в том, чтобы решать уравнение вместо , потому чтозначительно меньше *.* Еслиблизкок, настолько, что атака Мэя работать не будет*.*

Чтобы улучшить предыдущую матрицу, Bleichenbacher и Мэй в работе [6] используют соотношение . Хотя значение неизвестно, это отношение позволяет уменьшить степень в диагонали. Оптимизируя возможности , матрица Bleichenbacher и Мэя всегда дает лучшие результаты чем предыдущая [2].

Здесь строки состоят из коэффициентов семи полиномов:

Граница определяется следующим образом:

В сравнение с предыдущей*,* степеньуменьшается, благодаря умножению на *.* Таким образом, атака работает для

Наконец, в работе [2] авторы, чтобы улучшить предыдущую атаку, рассматривают не только *,* но и *,* как представление для того же полинома. Это полезно, если заметить, что. В предыдущей матрице был полином *,* он был необходим, чтобы матрица была треугольной [6], по сути, он не увеличивал границу нахождения решения. Введянеобходимость в нем отпала.В работе [2] предлагается генерировать полиномы следующим образом:

Матрица принимает следующий вид [2]:

Здесь строки состоят из коэффициентов шести полиномов:

По построению эта решетка всегда лучше, чем матрица Bleichenbacher и Мэя [6], к тому же она имеет меньшие размеры. Далее вычисляется граница:

Таким образом

Также, в работе [2] авторы доказывают следующую теорему.

**Теорема 4.** Пусть , дляПусть и если достаточно большое и выполняются условия [2]:

и *.*

Тогда может быть факторизовано за полиномиальное время, с помощью описанных выше полиномов и использования LLL алгоритма.

# РЕАЛИЗАЦИЯ АТАКИ НА МАЛЫЙ

В качестве практической части данной работы было решено реализовать атаку на малый описанную в [2].

Параметры должны иметь следующий вид:

*, , ,*

*, ,*

Для генерации параметров использовалась следующая функция:

def checkNew(N1,e1,d\_q1,p1):

a = math.log(int(e1),int(N1))

d = math.log(int(d\_q1),int(N1))

b = math.log(int(p1),int(N1))

if (a > (b/(1-b))) and (d < (((1-b)\*(3+2\*b) - 2\*sqrt(b\*(1-b)\*(a\*b+3\*a+b)))/(3+b))) and (a+b+d>1) :

return 1

else:

return 0

def trygen(betta, lengthN, delta):

step = 0

while true:

print(step)

step +=1

p = random\_prime(2^int(round(betta \* length\_N))) #генерация p

q = random\_prime(2^int(round((1-betta)\*length\_N)))#генерация q

q2 = random\_prime(2^int(round(delta\*length\_N\*2)))

if gcd(p-1,q-1) != 2:

print("bad GCD")

print(gcd(p-1,q-1))

continue

N = p\*q

ZmodPhi = Zmod((p-1)\*(q-1))

ZmodQ = Zmod((q2-1))

d\_q = ZmodQ.random\_element() #Получение dq: len(dq)< delta\*lenN

ZmodP = Zmod((p-1))

try:

e = inverse\_mod(int(d\_q), q-1) #получение параметра e

ZmodE = Zmod(e);

d\_p = inverse\_mod(int(e), p-1) #получение параметра dp

if checkNew(N,e,d\_q,p): #проверка параметров

return [e, d\_q, d\_p, q, p, N]

except ZeroDivisionError:

print("ZeroDivision")

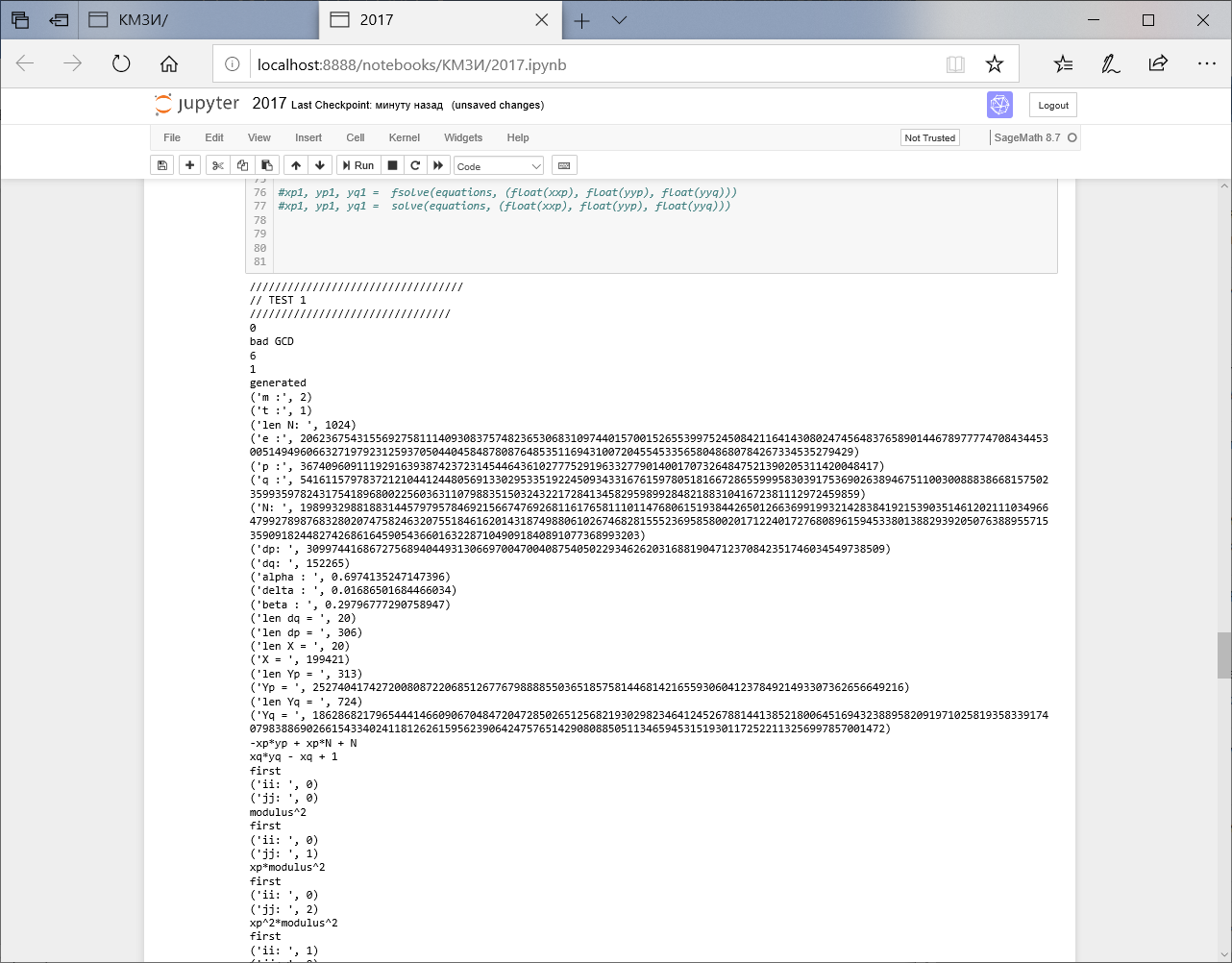


Рисунок 1 – Пример сгенерированных параметров

Далее необходимо определить полиномы, решение которых будет искаться:

P.<xp, xq, yp, yq, Xp, Xq, Yp,Yq,N,modulus>=PolynomialRing(ZZ)

fq = 1 + (xq)\*(yq-1)

fp = N + (xp)\*(N-yp)

Искомое решение определяется следующим образом:

В работе [2], в качестве сдвиговых полиномов предлагаются:

Для наглядности, использовалось .

for ii in range(mm + 1):

for jj in range(mm-ii + 1):

gg.append(P((xp)\*\*jj \* modulus\*\*(mm - ii) \* polp(xp, xq, yp, yq,0,0,0,0,N,0)\*\*ii))

pol = ((xp)\*\*jj \* modulus\*\*(mm - ii) \* polp(xp, xq, yp, yq,0,0,0,0,N,0)\*\*ii)

assert pol(xxp, xxq, yyp, yyq,0,0,0,0,N1,modulus1)%(modulus1\*\*mm) == 0

numpol+=1

for ii in range(mm + 1):

for jj in range (1, (mm)/2 + 1):

gg.append(P(((yp)\*\*jj \* modulus\*\*(mm - ii) \* polp(xp, xq, yp, yq,0,0,0,0,N,0)\*\*ii)))

pol = ((yp)\*\*jj \* modulus\*\*(mm - ii) \* polp(xp, xq, yp, yq,0,0,0,0,N,0)\*\*ii)

assert pol(xxp, xxq, yyp, yyq,0,0,0,0,N1,modulus1) %(modulus1\*\*mm) == 0

numpol+=1

print(pol)

for ii in range(1, mm + 1):

for jj in range (1, ii + 1):

gg.append(P((modulus\*\*(mm - ii) \* polp(xp, xq, yp, yq,0,0,0,0,N,0)\*\*(ii-jj) \* polq(xp,xq, yp , yq,0,0,0,0,N,0)^jj)))

pol = (modulus\*\*(mm - ii) \* polp(xp, xq, yp, yq,0,0,0,0,N,0)\*\*(ii-jj) \* polq(xp,xq, yp , yq,0,0,0,0,N,0)^jj)

assert pol(xxp, xxq, yyp, yyq,0,0,0,0,N1,modulus1) %(modulus1\*\*mm) == 0

numpol+=1

Ключевой находкой в работе 2017, как признаются авторы, является возможность использования отношений:

Они позволяют свести получившиеся полиномы в виде матрицы, к виду треугольной матрицы. Это реализуется следующим кодом:

for polynomial in gg:

newpol = 0

for monomial in polynomial.monomials():

newmonomial = monomial

coef = polynomial.monomial\_coefficient(monomial)

flag = 0

while gcd(newmonomial,yq\*yp) == yq\*yp:

newmonomial /= yq\*yp

coef \*= N

while gcd(newmonomial, xp\*yq) == xp\*yq:

newmonomial /= xp

flag = 1

coef \*= (xq - 1)

while gcd(newmonomial, xq\*xp) == xq\*xp:

newmonomial /= xq

coef \*= (xp + 1)

while gcd(newmonomial, xq\*yq) == xq and flag == 0:

newmonomial /= xq

coef \*= (xp + 1)

Рассмотрим, как это происходит на полученном мною примере, при .

Сначала, запишем полученные на шаге генерации полиномы, а рядом, полученные с их помощью, новые мономы. Необходимо помнить, что количество строк (полиномов) должно быть равным количество столбцов матрицы (коэффициентов).

Таблица 1 – Полученные полиномы и вводимые ими мономы

|  |  |  |
| --- | --- | --- |
|  | Полином | Моном |
|  |  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |  |
|  |  |
|  |  |
|  |  |  |
|  |  |
|  |  |

Видим, что при введении первого же полинома в наше множество векторов, количество мономов увеличивается более чем на .

Чтобы избавиться от них, воспользуемся свойствами, описанными выше.

Рассмотрим первый такой полином и воспользуемся следующим соотношением:

Тогда, останется лишь новый моном, а полином примет вид:

Далее, имеем целых четыре монома:

Все из получившихся мономов, кромеуже добавлены, поэтому он будет единственным новым.

С помощью этих соотношений получим полином вида:

Далее аналогично, последний полином добавит только мономи будет иметь вид:

Стоит отметить, что коэффициент, при каждом новом мономе, будет стоять на диагонали матрицы, следовательно, чем больше коэффициент, тем больше определитель, чего следует избегать т.к., где *–* размерность матрицы. В данном случае на диагональном коэффициенте появился множитель(выделен жирным), от него следует избавиться, путем умножения полинома на обратный элементпо модулю. Это реализуется следующим кодом:

for polynomial in ggnew:

polynomial = Pole(polynomial(xp,xq,yp,yq,0,0,0,0,N,modulus))

count = 0

print("i= ",i," ",polynomial)

for monomial in polynomial.monomials():

#print(monomial(xp,xq,yp,yq,1,1))

if monomial(xp,xq,yp,yq,1,1) not in monomialscheck:

newmonomial = monomial

while (gcd(newmonomial,N) == N):

print("divide", monomial)

newPol = 0

invN = inverse\_mod(N1,modulus1\*\*mm)

for monomial in polynomial.monomials():

if monomial == newmonomial:

newmonomial/=N

newPol += polynomial.monomial\_coefficient(monomial) \* newmonomial(xp,xq,yp,yq,N,modulus)

else:

newPol+= invN\*polynomial.monomial\_coefficient(monomial) \*

monomial(xp,xq,yp,yq,N,modulus)

polynomial = Pole(newPol)

for mon in polynomial.monomials():

numb = 0

for ii in range(mm + 1):

for jj in range(mm-ii + 1):

if numb == count:

ii1 =ii

jj1 =jj

break

numb+=1

print("append:",monomial(xp,xq,yp,yq,1,1))

monomialscheck.append(monomial(xp,xq,yp,yq,1,1))

count+=1

print("result pol:",polynomial)

ggnew[i] = polynomial

i+=1

В итоге, этот полином примет вид:

, где– некие коэффициенты.

Далее, записав коэффициенты полиномов в строки матрицы, в соответствии с мономами получим матрицу, вида:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |

Матрица получилась диагональной,иотмечены положительные и отрицательные числа соответственно. Оценим, границу возможности применения метода:

Сравним, с результатом из раздела , видим, что эта граница еще лучше, это связано с увеличением размерности матрицы. Таким образом, можем считать, что матрица построена верно. К сожалению, для этой атаки авторы [2] не приводят конкретного примера построенной матрицы, как не приводят и однозначного алгоритма построения и приведения полиномов, поэтому данная работа заняла значительное количество времени.

Далее выполняется алгоритм LLL и оценивается значение определителя, оно должно быть меньше, чем .

Далее, в моем случае, вычисляются первые три вектора из полученной матрицы, с помощью предыдущих соотношений они сводятся к зависимости от трех переменных. Впоследствии, проверяется, что у всех этих полиномов есть корень, являющийся секретными параметрами системы. Необходимо вычислить решение системы из трех нелинейных уравнений с тремя переменными, эта задача решается за полиномиальное время, но так как это не является целью данной работы, я использовал решение методом с приближенными значениями, для наглядности того, что это можно сделать.

P.<xp, yp, yq>=PolynomialRing(ZZ)

f(xp,yp,yq) = (pol1, pol2,pol3)

print(minimize(norm(f), (xxp,yyp,yyq)))

(xp1,yp1,yq1) = minimize(norm(f), (xxp,yyp,yyq))

print (xp1)

print (yp1)

print (yq1)

print(xxp)

print(yyp)

print(yyq)

fsolve(equations,(float(xxp),float(yyp),float(yyq)))

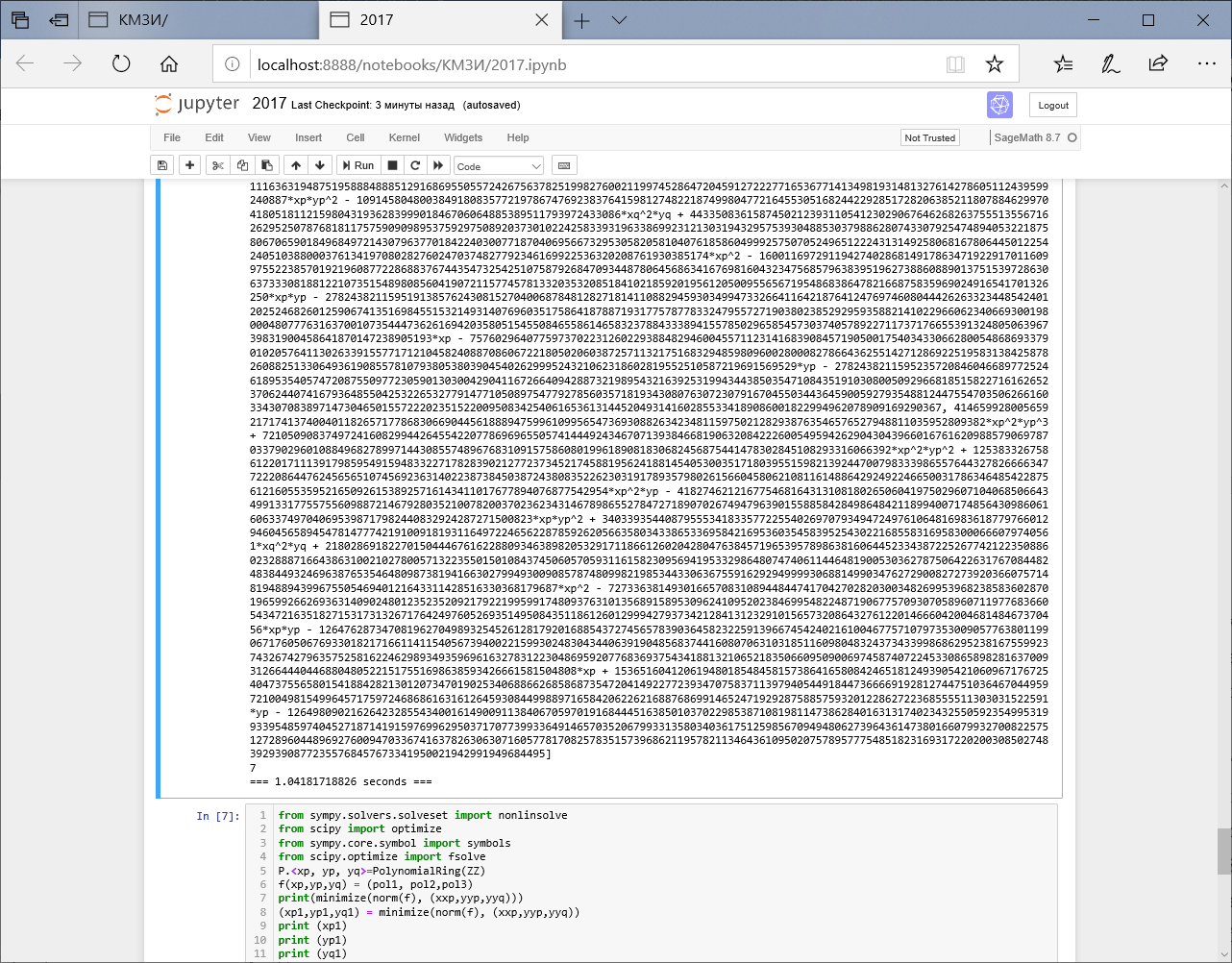


Рисунок 2 – Время выполнения и количество подходящих полиномов

Далее, – искомые секретные значения, – полученные с помощью LLL полиномы.

Результат выполнения:

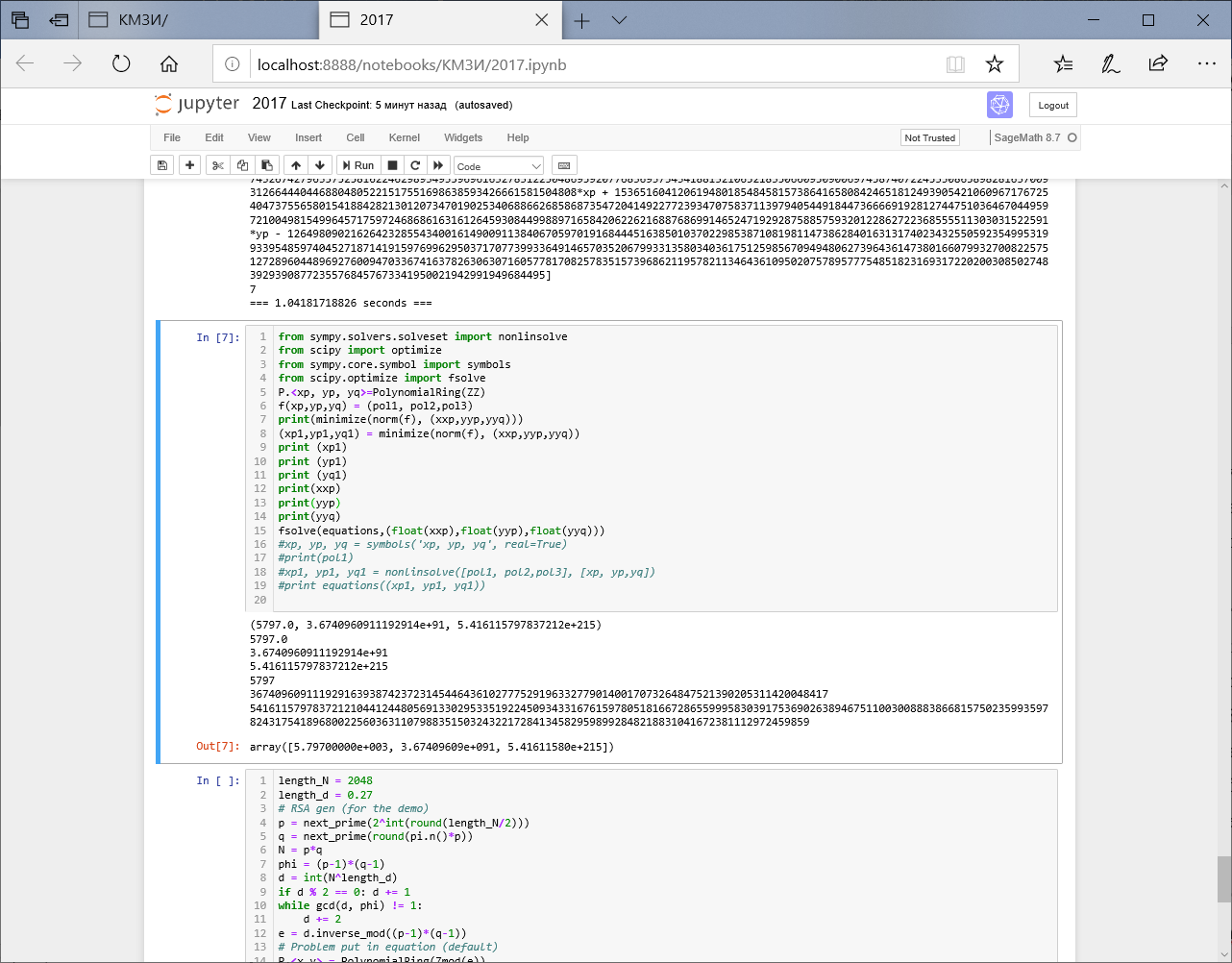


Рисунок 3 - Результат решения системы нелинейных уравнений

# ЗАКЛЮЧЕНИЕ

В данной курсовой работе были изучены методы применения теории решеток для атак на схему подписи и криптосистему RSA, изучен и реализован новый подход, к методу построения базисной матрицы, описанный в работе [2].

Существуют различные методы, использующие теорию решеток и LLL алгоритм, для атак на схему подписи RSA, большинство из них основываются на методе, предложенном Копперсмитом, описанном в работе [3]. В данной работе сделан обзор работ авторов, которые улучшили данный метод, используя различные схемы генерации сдвиговых полиномов [2,5,6], также, произведен подробный разбор метода, предложенного в работе [2], для атаки на малый показатель , который использует схему генерации ключей CRT-RSA и соотношения неизвестных параметров в ней, чтобы уменьшить определитель базисной матрицы, этот метод, позволяет атаковать систему при , еще Boneh и Durfee более чем через лет назад отметили, что их граница: , не может быть окончательным результатом, так как она слишком неестественна и предложили идею, что она может увеличена до , что и показали авторы в работе [2].

Изобретение алгоритма LLL в 1982 году послужило основой для построения Копперсмитом эффективного алгоритма для нахождения малых решений полиномиальных уравнений в 1996 году, что в свою очередь открыло новое направление исследований, направленных на решение сложных задач, таких как разложение числа, а также для атак на схемы подписи RSA.

Работы [2, 3], доказывают, что даже спустя 20 лет, данное направление все еще актуально, так, авторы [2] утверждают, что на основе данных методов можно и далее улучшать границы возможных атак, и для одной из описанных атак они отмечают, что там все еще есть полиномы, от которых можно было бы избавиться, и оставляют этот вопрос открытым.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Шенец Н.Н. Быстрые вычислительные алгоритмы в криптографии. – СПб.: Издательство политехнического университета, 2017.
2. Atsushi Takayasu, Yao Lu, Liqiang Peng. Small CRT-Exponent RSA Revisited. July 26, 2017. URL: <https://eprint.iacr.org/2017/092.pdf> (15.06.2019).
3. David Wong. Survey: Lattice Reduction Attacks on RSA. University of Bordeaux, March 2015. URL: <https://github.com/mimoo/RSA-and-LLL-attacks/blob/master/survey_final.pdf> (15.06.2019)
4. Lidong Han, Wei Wei, Mingjie Liu. On the Multiple Fault Attack on RSA Signatures with LSBs of Message Unknown. Tsinghua University, Beijing 100084, China, 2012. URL: <https://eprint.iacr.org/2012/491.pdf> (15.06.2019)
5. Alexander May. Cryptoanalysis of Unbalanced RSA with Small CRT-Exponent. University of Paderborn, Germany, 2002. URL: <https://www.researchgate.net/publication/221355597_Cryptanalysis_of_Unbalanced_RSA_with_Small_CRT-Exponent> (15.06.2019)
6. Daniel Bleichenbacher, Alexander May. New Attacks on RSA with Small Secret CRT-Exponents, 2006. URL:

<https://www.researchgate.net/publication/221010552_New_attacks_on_RSA_with_small_secret_CRT-exponents> (15.06.2019)

1. Alexander May, Maike Ritzenhofen. Implicit Factoring: On Polynomial Time Factoring Given Only an Implicit Hint. Ruhr-University of Bochum, 2009. URL: <http://cits.rub.de/imperia/md/content/may/paper/implicit_final.pdf> (15.06.2019)
2. Антиплагиат. URL: <https://www.antiplagiat.ru/> (16.06.2019)

# ПРИЛОЖЕНИЕ А

Код программы

def checkNew(N1,e1,d\_q1,p1):

a = math.log(int(e1),int(N1))

d = math.log(int(d\_q1),int(N1))

b = math.log(int(p1),int(N1))

if (a > (b/(1-b))) and (d < (((1-b)\*(3+2\*b) - 2\*sqrt(b\*(1-b)\*(a\*b+3\*a+b)))/(3+b))) and (a+b+d>1) :

return 1

else:

print("(a > (b/(1-b))) = ",(a > (b/(1-b))))

print("a=",a)

print("(b/(1-b))=", (b/(1-b)))

print("(d < (((1-b)\*(3+2\*b) - 2\*sqrt(b\*(1-b)\*(a\*b+3\*a+b)))/(3+b)))= ", (d < (((1-b)\*(3+2\*b) - 2\*sqrt(b\*(1-b)\*(a\*b+3\*a+b)))/(3+b))))

print("d = ",d )

print("right part = ",(((1-b)\*(3+2\*b) - 2\*sqrt(b\*(1-b)\*(a\*b+3\*a+b)))/(3+b)))

print("a+b+d = ", a+b+d)

return 0

def trygen(betta, lengthN, delta):

step = 0

while true:

print(step)

step +=1

p = random\_prime(2^int(round(betta \* length\_N)))

q = random\_prime(2^int(round((1-betta)\*length\_N)))

q2 = random\_prime(2^int(round(delta\*length\_N\*2)))

if gcd(p-1,q-1) != 2:

print("bad GCD")

print(gcd(p-1,q-1))

continue

N = p\*q

ZmodPhi = Zmod((p-1)\*(q-1))

ZmodQ = Zmod((q2-1))

d\_q = ZmodQ.random\_element()

ZmodP = Zmod((p-1))

try:

e = inverse\_mod(int(d\_q), q-1)

ZmodE = Zmod(e);

d\_p = inverse\_mod(int(e), p-1)

if checkNew(N,e,d\_q,p):

return [e, d\_q, d\_p, q, p, N]

except ZeroDivisionError:

print("ZeroDivision")

def helpful\_vectors(BB, modulus):

nothelpful = 0

for ii in range(BB.dimensions()[0]):

if BB[ii,ii] >= modulus:

nothelpful += 1

print nothelpful, "/", BB.dimensions()[0], " vectors are not helpful"

def matrix\_overview(BB,bound):

for ii in range (BB.dimensions()[0]):

a = ('%02d ' % ii)

for jj in range (BB.dimensions()[1]):

if BB[ii,jj]==0 :

temp = '0'

else:

if BB[ii,jj]>0:

temp = '+X'

else:

temp = '-X'

a += temp

a += ' '

if BB[ii, ii] > bound:

a+='~'

print a

def may2(polp, polq, modulus1,mm,tt,XX,YYq,YYp, N1):

P.<xp, xq, yp, yq, Xp, Xq, Yp,Yq,N,modulus>=PolynomialRing(ZZ)

polp = P(polp)

polq = P(polq)

print(polp)

print(polq)

gg = []

numpol = 0

gg.sort()

for ii in range(mm + 1):

for jj in range(mm-ii + 1):

print('first')

print("ii: ",ii)

print("jj: ",jj)

gg.append(P((xp)\*\*jj \* modulus\*\*(mm - ii) \* polp(xp, xq, yp, yq,0,0,0,0,N,0)\*\*ii))

pol = ((xp)\*\*jj \* modulus\*\*(mm - ii) \* polp(xp, xq, yp, yq,0,0,0,0,N,0)\*\*ii)

assert pol(xxp, xxq, yyp, yyq,0,0,0,0,N1,modulus1)%(modulus1\*\*mm) == 0

numpol+=1

print (pol)

for ii in range(mm + 1):

for jj in range (1, (mm)/2 + 1):

print('second)')

print("ii: ",ii)

print("jj: ",jj)

gg.append(P(((yp)\*\*jj \* modulus\*\*(mm - ii) \* polp(xp, xq, yp, yq,0,0,0,0,N,0)\*\*ii)))

pol = ((yp)\*\*jj \* modulus\*\*(mm - ii) \* polp(xp, xq, yp, yq,0,0,0,0,N,0)\*\*ii)

assert pol(xxp, xxq, yyp, yyq,0,0,0,0,N1,modulus1) %(modulus1\*\*mm) == 0

numpol+=1

print(pol)

for ii in range(1, mm + 1):

for jj in range (1, ii + 1):

print("third")

print("ii: ",ii)

print("jj: ",jj)

gg.append(P((modulus\*\*(mm - ii) \* polp(xp, xq, yp, yq,0,0,0,0,N,0)\*\*(ii-jj) \* polq(xp,xq, yp , yq,0,0,0,0,N,0)^jj)))

pol = (modulus\*\*(mm - ii) \* polp(xp, xq, yp, yq,0,0,0,0,N,0)\*\*(ii-jj) \* polq(xp,xq, yp , yq,0,0,0,0,N,0)^jj)

assert pol(xxp, xxq, yyp, yyq,0,0,0,0,N1,modulus1) %(modulus1\*\*mm) == 0

#print(gg[int(numpol)])

numpol+=1

print(pol)

monomials = []

monomialsadd = []

monomialscheck=[]

ggnew = []

ggnew.sort()

for polynomial in gg:

newpol = 0

for monomial in polynomial.monomials():

newmonomial = monomial

coef = polynomial.monomial\_coefficient(monomial)

flag = 0

while gcd(newmonomial,yq\*yp) == yq\*yp:

newmonomial /= yq\*yp

coef \*= N

while gcd(newmonomial, xp\*yq) == xp\*yq:

newmonomial /= xp

flag = 1

coef \*= (xq - 1)

while gcd(newmonomial, xq\*xp) == xq\*xp:

newmonomial /= xq

coef \*= (xp + 1)

while gcd(newmonomial, xq\*yq) == xq and flag == 0:

newmonomial /= xq

coef \*= (xp + 1)

newpol += coef\*newmonomial

assert newpol(xxp, xxq, yyp, yyq,0,0,0,0,N1,modulus1) %(modulus1\*\*mm) == 0

print(newpol)

ggnew.append(P(newpol))

i=0

Pole.<xp, xq, yp, yq,N, modulus>=PolynomialRing(ZZ)

numb = 0

for polynomial in ggnew:

polynomial = Pole(polynomial(xp,xq,yp,yq,0,0,0,0,N,modulus))

count = 0

print("i= ",i," ",polynomial)

#print(monomialscheck)

for monomial in polynomial.monomials():

#print(monomial(xp,xq,yp,yq,1,1))

#print(monomial)

if monomial(xp,xq,yp,yq,1,1) not in monomialscheck:

newmonomial = monomial

#print(monomial)

#print(monomial(xp,xq,yp,yq,1,1))

while (gcd(newmonomial,N) == N):

print("divide", monomial)

#print(i)

#print("divide")

newPol = 0

invN = inverse\_mod(N1,modulus1\*\*mm)

for monomial in polynomial.monomials():

if monomial == newmonomial:

#print(newmonomial)

newmonomial/=N

#print(newmonomial)

newPol += polynomial.monomial\_coefficient(monomial) \* newmonomial(xp,xq,yp,yq,N,modulus)

else:

newPol += invN\*polynomial.monomial\_coefficient(monomial) \* monomial(xp,xq,yp,yq,N,modulus)

#newmonomial/=N

#polynomial = Pole(polynomial(xp,xq,yp,yq,N1,modulus)) \* Pole(inverse\_mod(N1, modulus1\*\*mm))

polynomial = Pole(newPol)

#polynomial = Pole(polynomial(xp,xq,yp,yq,N,modulus))/Pole(N)

#print(polynomial.monomials())

#print(newmonomial)

for mon in polynomial.monomials():

if mon == newmonomial:

#print(polynomial.monomial\_coefficient(mon))

mon = mon

numb = 0

for ii in range(mm + 1):

for jj in range(mm-ii + 1):

if numb == count:

ii1 =ii

jj1 =jj

break

numb+=1

for mon in polynomial.monomials():

if mon == newmonomial:

numb = numb

#print(mon)

#print(polynomial.monomial\_coefficient(mon))

#if (mon(XX,XX,YYp,YYq,N1,modulus1)\*polynomial.monomial\_coefficient(mon)!=( XX\*\*(ii1+jj1) \* YYp\*\*ii1 \* modulus1^(mm-ii1)) % modulus1\*\*mm):

#print("NOT EQ")

#else:

#print("EQ")

#print ("add")

print("append:",monomial(xp,xq,yp,yq,1,1))

monomialscheck.append(monomial(xp,xq,yp,yq,1,1))

#monomialsadd.append(1)

count+=1

#print("i= ",i," ",polynomial)

print("result pol:",polynomial)

ggnew[i] = polynomial

i+=1

i = 0

print("first POLYNOM :", ggnew[0])

#Pole.<xp, xq, yp, yq>=PolynomialRing(Zmod(modulus1\*\*mm))

for polynomial in ggnew:

polynomial = Pole(polynomial(xp,xq,yp,yq,N1,modulus1))

count = 0

#print(i)

for monomial in polynomial.monomials():

if monomial not in monomials:

if monomial not in monomials:

#print ("add")

print(monomial)

monomials.append(monomial)

monomialsadd.append(1)

count+=1

#print(count)

ggnew[i] = polynomial

if count>1:

print("ERROR")

#print(gg[i])

#print("i: ", i)

#print(polynomial)

i+=1

P.<xp, xq, yp, yq>=PolynomialRing(ZZ)

i=0

for pol in ggnew:

ggnew[i] = P(pol)

i+=1

nn = len(ggnew)

print("NN : ", nn)

print(len(ggnew))

print("monomials:",len(monomials))

BB = Matrix(ZZ, nn)

for ii in range(nn):

#print(ii)

#print(nn)

for jj in range(len(monomials)):

for monomial in ggnew[ii].monomials():

if monomial == monomials[jj]:

monomials[jj] = monomial

#if monomials[jj] in ggnew[ii].monomials():

BB[ii, jj] = ggnew[ii].monomial\_coefficient(monomial) \* monomial(XX,XX,YYp,YYq)

assert BB[0,0] == modulus1^mm

assert BB[1,1] == modulus1^mm \* XX

assert BB[2,2] == modulus1^mm \* XX \* XX

pols = []

for ii in range(nn):

pols.append(0)

for jj in range(nn):

#print(P(monomials[jj](XX, XX ,YYp, YYq)))

pols[-1] += P(P(monomials[jj](xp, xq, yp, yq) \* BB[ii, jj])/ monomials[jj](XX, XX ,YYp, YYq))

assert pols[-1](xxp, xxq, yyp, yyq) % modulus1\*\*mm == 0

if debug:

matrix\_overview(BB, modulus1^mm)

BB = BB.LLL()

if debug:

det = BB.det()

bound = modulus1^(mm\*nn)

if det >= bound:

print "We do not have det < bound. Solutions might not be found."

print "det(L) = ", det

diff = (log(det) - log(bound)) / log(2)

print "size det(L) − size e^(m\*n) = ", floor(diff)

else:

print "det(L) < e^(m\*n)"

pols = []

for ii in range(nn):

pols.append(0)

for jj in range(nn):

pols[-1] += P(P(monomials[jj](xp, xq, yp, yq) \* BB[ii, jj])/ P(monomials[jj](XX, XX, YYp, YYq)))

assert pols[-1](xxp, xxq, yyp, yyq) % e^m == 0

if pols[-1](xxp, xxq, yyp, yyq) != 0:

#print("Error pols: ",pols[-1])

pols.pop()

print(pols)

print(len(pols))

pol1 = pol2 = pol3 = pol4 =0

found = False

# for ii, pol in enumerate(pols):

# if found:

# break

# for jj in range(ii + 1, len(pols)):

# for kk in range(jj +1 , len(pols)):

# for gg in range (kk+1,len(pols)):

# if gcd(pol, pols[jj],pols[kk],pols[gg]) == 1:

# print "using vectors", ii, "and", jj, "and", kk, "and", gg

# pol1 = pol

# pol2 = pols[jj]

# pol3 = pols[kk]

# pol4 = pols[gg]

# found = True

# break

# if pol1==pol2==0:

# print("failure")

# return 0,0

#print(pol1)

#print(pol2)

pol1 = pols[0](xp,xp+1,yp,yq)

pol2 = pols[1](xp,xp+1,yp,yq)

pol3 = pols[2](xp,xp+1,yp,yq)

P.<xp, yp, yq>=PolynomialRing(ZZ)

pol1 = P(pol1)

pol2 = P(pol2)

pol3 = P(pol3)

return [pol1,pol2,pol3]

pol1 = pol2 =pol3 =0

def equations(p):

xp,yp,yq = p

return ( pol1(xp,yp,yq), pol2(xp,yp,yq),pol3(xp,yp,yq))

print "//////////////////////////////////"

print "// TEST 1"

print "////////////////////////////////"

# RSA gen options (for the demo)

length\_N = 1024 # size of the modulus

m = 2

t = 1

delta = 0.01

betta = 0.3

[e, d\_q, d\_p, q, p, N1] = trygen(betta,length\_N,delta)

print("generated")

a = math.log(int(e),int(N1))

d = math.log(int(d\_q),int(N1))

b = math.log(int(p),int(N1))

print("alpha : ", float(a))

print("d : ", float(d))

print("b : ",float(b))

print("len dq = ", len(bin(d\_q)))

print("len dp = ", len(bin(d\_p)))

X = floor(N1^(a+b+d -1 + 0.005))

print("len X = ", len(bin(X)))

print("X = ",X)

Y\_p = 2 \* floor(N1^(b+0.005))

print("len Yp = ", len(bin(Y\_p)))

print("Yp = ",Y\_p)

assert p < Y\_p

Y\_q = floor(N1^(1-b + 0.005))

print("len Yq = ", len(bin(Y\_q)))

print("Yq = ",Y\_q)

yyp = p

assert (int(e)\*int(d\_q))%(q-1) == 1

assert (int(e)\*int(d\_p))%(p-1) == 1

xxp = int(int(int(e)\*int(d\_q)\*int(p)) - N1) // int(N1 - yyp)

yyq = q

xxq = int(int(int(e)\*int(d\_q)) - 1) // int(yyq - 1)

assert xxq <= X

assert xxp <= X

assert yyq <= Y\_q

assert yyp <= Y\_p

assert int(int(e)\*int(d\_q)\*int(p)) == N1 + xxp \* (N1 - yyp)

assert int(int(e)\*int(d\_q)) == 1 + xxq \* (yyq - 1)

#P.<xp, xq, yp, yq, Xp, Xq, Yp,Yq,N,modulus>=PolynomialRing(Zmod(e))

P.<xp, xq, yp, yq, Xp, Xq, Yp,Yq,N,modulus>=PolynomialRing(ZZ)

fq = 1 + (xq)\*(yq-1)

fp = N + (xp)\*(N-yp)

assert X\*X\*Y\_p\*Y\_p < e^m

#assert (fp(xxp,0,yyp,0)==0)

#assert (fq(0,xxq,0,yyq)==0)

[pol1,pol2,pol3]= may2(fp, fq, int(e), int(m), int(t), int(X), int(Y\_q), int(Y\_p),int(N1))

assert pol1(xxp,yyp,yyq) == 0

assert pol2(xxp,yyp,yyq) == 0

assert pol3(xxp,yyp,yyq) == 0

#xp1, yp1, yq1 = fsolve(equations, (float(xxp), float(yyp), float(yyq)))

#xp1, yp1, yq1 = solve(equations, (float(xxp), float(yyp), float(yyq)))

from sympy.solvers.solveset import nonlinsolve

from scipy import optimize

from sympy.core.symbol import symbols

from scipy.optimize import fsolve

P.<xp, yp, yq>=PolynomialRing(ZZ)

f(xp,yp,yq) = (pol1, pol2,pol3)

print(minimize(norm(f), (xxp,yyp,yyq)))

(xp1,yp1,yq1) = minimize(norm(f), (xxp,yyp,yyq))

print (xp1)

print (yp1)

print (yq1)

print(xxp)

print(yyp)

print(yyq)

fsolve(equations,(float(xxp),float(yyp),float(yyq)))

#xp, yp, yq = symbols('xp, yp, yq', real=True)

#print(pol1)

#xp1, yp1, yq1 = nonlinsolve([pol1, pol2,pol3], [xp, yp,yq])

#print equations((xp1, yp1, yq1))