Санкт-Петербургский политехнический университет Петра Великого
Институт прикладной математики и механики
Кафедра «Информационная безопасность компьютерных систем»

К У Р С О В А Я Р А Б О Т А

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

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 $\left(d < N^{1/4}\right)$[3]. В 2000 году Boneh и Durfee улучшили эту границу $(d < N^{0,292})$, 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 d_q, когда простой делитель p значительно меньше, чем q. Предложенная атака работает при $p < N^{0,468}$ [6]. Jochemsz и Мэй предложили атаку для малых d_p и d_q, когда простые делители p и q близки. Атака работает для $d_p, d_q < N^{0,073}$ [3].

В 2017 году Atsushi Takayasu, Yao Lu и Liqiang Peng [2] предложили две улучшенные атаки на систему CRT-RSA: атака на малый d_q, при $p < N^{0,5}$ (улучшение атаки от Bleichenbacher – Мэя) и атака на малые d_p и d_q, при $d_p, d_q < N^{0,122}$ (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.

1. АЛГОРИТМ LLL И ЕГО СВОЙСТВА

Алгоритм Lenstra-Lenstra-Lovasz используется для уменьшения длин векторов базиса решетки и вычисляется за полиномиальное время $O(n^5 d\ log^3 B)$, где B −максимальная длина входного вектора, n −количество векторов базиса, d − число координат. Решетка остается неизменной, но вектора нового базиса становятся короче.

Определение 1. Пусть L решетка с базисом $B = \{b_1, \dots, b_n\}$, δ −LLL алгоритм, примененный к базису B произведет новый базис решетки $L : B' = \{b'_1, \dots, b'_n\}$, удовлетворяющий следующим свойствам [3]:

1. $|\mu_{i,j}| \le \frac{1}{2},\ 1 \le j < i \le n$
2. $\delta \|b''_i\|^2 \le \left\|\mu_{i+1,i} b''_i + b''_{i+1}\right\|^2, 1 \le i < n$

$\mu_{i,j} = \frac{b''_j b'_i}{b''_j b''_j}$ и $b''_1 = b'_1$, где b''_i −элементы базиса, полученного ортогонализацией Грама-Шмидта.

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

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

Входные данные: базис $\{a_1, a_2, \dots, a_n\}$ решетки L.

Выходные данные: LLL – приведенный базис $\{b'_1, b'_2, \dots, b'_n\}$.

Шаги алгоритма:

1. Положить $\{b'_1, b'_2, \dots, b'_n\} \leftarrow \{a_1, a_2, \dots, a_n\}$.
2. Выполнить ортогонализацию Грама-Шмидта для базиса $\{b'_1, b'_2, \dots, b'_n\}$ и получить ортогональный базис $\{b''_1, b''_2, \dots, b''_n\}$, одновременно вычислив коэффициенты $\mu_{ij}, 1 \le j < i \le n$, и значения $B'_i = \|b''_i\|^2, i = 1, \dots, n$.
3. Положить $k \leftarrow 2$.
4. *При* $\left|\mu_{k,k-1}\right| > \frac{1}{2}$ выполнять:

4.1 Вычислить $r = \begin{cases} \left[\frac{1}{2} + \mu_{k,k-1}\right], & \mu_{k,k-1} > 0 \\ -\left[\frac{1}{2} - \mu_{k,k-1}\right], & \mu_{k,k-1} < 0 \end{cases}$;

4.2 Положить $b'_k \leftarrow b'_k - r b'_{k-1}$;

4.3 Для $j = 1, 2, \ldots k-2$ положить $\mu_{k,j} = \mu_{k,j} - r\mu_{k-1,j}$;

4.4 Положить $\mu_{k,k-1} \leftarrow \mu_{k,k-1} - r$.

5. При $B_k < \left(\frac{3}{4} - \mu_{k,k-1}^2\right) B_{k-1}$ выполнить:

5.1 Положить $\mu \leftarrow \mu_{k,k-1}, B \leftarrow B_k + \mu^2 B_{k-1},\ \mu_{k,k-1} \leftarrow \frac{\mu B_{k-1}}{B}$,

$B_k \leftarrow \frac{B_{k-1} B_k}{B}, B_{k-1} \leftarrow B$;

5.2 Поменять местами векторы b'_k и b'_{k-1};

5.3 *При* $k > 2$ поменять местами коэффициенты $\mu_{k,j}$ и $\mu_{k-1,j}$

для $j = 1, 2, \ldots, k-2$;

5.4 Для $s = k+1, k+2, \ldots, n$ положить:

$t \leftarrow \mu_{s,k}, \mu_{s,k} \leftarrow \mu_{s,k-1} - \mu t, \mu_{s,k-1} \leftarrow t + \mu_{k,k-1}\mu_{s,k}$;

5.5 Положить $k \leftarrow \max\{2, k-1\}$ и вернуться на шаг 4.

6. При $B_k \geq \left(\frac{3}{4} - \mu_{k,k-1}^2\right) B_{k-1}$ выполнить:

6.1 Для $l = k-2, k-3, \ldots, 1$ при $\left|\mu_{k,l}\right| > \frac{1}{2}$ выполнить:

6.1.1 Вычислить $r = \begin{cases} \left[\frac{1}{2} + \mu_{k,l}\right], & \mu_{k,l} > 0 \\ -\left[\frac{1}{2} - \mu_{k,l}\right], & \mu_{k,l} < 0 \end{cases}$;

6.1.2 Положить $b'_k \leftarrow b'_k - r b'_l$;

6.1.3 Для $j = 1, 2, \ldots, l-1$ положить $\mu_{k,j} = \mu_{k,j} - r\mu_{l,j}$;

6.1.4 Положить $\mu_{k,l} \leftarrow \mu_{k,l} - r$.

6.2 Положить $k \leftarrow k+1$.

7. Если $k \leq n$, то вернуться на шаг 4.

8. Вернуть $\{b'_1, b'_2, \ldots, b'_n\}$.

Условно алгоритм делится на 3 части:

1. Добиваемся $\left|\mu_{k+1,k}\right| \leq \frac{1}{2}$.

Стоит отметить, что при это не изменяются $b'_k : i < k$.

2. Проверяем условие 2 определения (в случае приведенного алгоритма $\delta = \frac{3}{4}$).

Возможны 2 случая:

2.1 $k \geq 2$: $\frac{3}{4}\|b''_k\|^2 > \left\|\mu_{k+1,k} b''_k + b''_{k+1}\right\|^2$, тогда меняем местами b_k и b_{k+1} и возвращаемся на 1 этап, уменьшая k.

2.2 $k = 1$: $\frac{3}{4}\|b''_k\|^2 \leq \left\|\mu_{k+1,k} b''_k + b''_{k+1}\right\|^2$, если данное неравенство выполняется, то переходим на 3 этап, иначе $k = 2$ и возвращаемся на 1 этап.

3. $\left|\mu_{k,j}\right| \leq \frac{1}{2}, \forall\, j \leq k-1$

l - наибольший номер: $\left|\mu_{k,l-1}\right| > \frac{1}{2}$. Уменьшаем $\left|\mu_{k,l-1}\right|$ как на 1 этапе, при этом гарантируется $\left|\mu_{k,j}\right|,\ j > l-1$ неизменны. Продолжаем, пока все коэффициенты не удовлетворяют условию.

Данный алгоритм всегда заканчивает свою работу, поскольку шаг 5 выполняется конечное число раз. Более того, если $\|a_i\| \leq d^l$ для всех $i = 1, \dots, n,$ то сложность алгоритма равна $O(n^4 l)$ арифметических операций, а целые числа, встречающиеся в процессе работы алгоритма, имеют в системе счисления по основанию d длину $O(nl)$ [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 L решетка размерности n. За полиномиальное время LLL алгоритм выдаст сокращенные вектора $v_i,\ 1 \leq i \leq n$, такие что [3]:

$$\|v_1\| \leq \|v_2\| \leq \cdots \leq \|v_i\| \leq 2^{\frac{n(n-1)}{4(n+1-i)}} \det(L)^{\frac{1}{n+1-i}}$$

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

2. НЕЯВНОЕ РАЗЛОЖЕНИЕ ДВУХ RSA МОДУЛЕЙ

Теорема 1 (Минковского). Пусть $L \subseteq Z^{n\times n}$ целочисленная решетка. Тогда L содержит ненулевой вектор v [7]:

$\|v\| = \lambda_1(L) \leq \sqrt{n}\det(L)^{\frac{1}{n}}$, где λ_i означает i кратчайший ЛНЗ вектор.

Мэй [7] предположил следующую ситуацию: пусть даны два разных RSA модуля $N_1 = p_1q_1, N_2 = p_2q_2$, где p_1, p_2 совпадают в t значащих младших битах, то есть: $p_1 = p + 2^t p_1'$ и $p_2 = p + 2^t p_2'$. Рассмотрим следующие уравнения:

$$(p + 2^t p_1')q_1 = N_1,$$
$$(p + 2^t p_2')q_2 = N_2.$$

Эти два уравнения содержат 5 неизвестных. Взяв эти уравнения по модулю 2^t мы можем избавиться от двух из них:

$$pq_1 \equiv N_1 \ (mod\ 2^t),$$
$$pq_2 \equiv N_2 \ (mod\ 2^t).$$

Так как q_1 и q_2 нечетные, разделим на них:

$$\frac{N_1}{q_1} \equiv \frac{N_2}{q_2} \ (mod\ 2^t),$$
$$(N_1^{-1}N_2)q_1 - q_2 \equiv 0 \ (mod\ 2^t).$$

Множество решений:

$$L = \{\ (x_1, x_2) \in Z^2 |\ (N_1^{-1}N_2)x_1 - x_2 \equiv 0 \ (mod\ 2^t)\ \}.$$

Образует подгруппу Z^2. Таким образом, L – двумерная решетка, натянутая на строковые вектора базисной матрицы:

$$B_L = \begin{pmatrix} 1 & N_1^{-1}N_2 \\ 0 & 2^t \end{pmatrix}.$$

Известно, что $(q_1, q_2) \in L$. Также предполагается, что этот вектор $q = (q_1, q_2)$ кратчайший в L, к тому же длина кратчайшего вектора ограничена значением $\sqrt{2}\det(L)^{\frac{1}{2}} = \sqrt{2}\cdot 2^{\frac{t}{2}}$ по теореме 1 (Минковского).

Предположим, что q_1, q_2 имеют величину α бит, к тому же они простые, имеем $q_1, q_2 \leq 2^\alpha$. Если α достаточно мало, то $\|q\|$ меньше границы Минковского и, следовательно, мы можем ожидать, что q является самым

коротким вектором в L, для этого должно выполняться следующее неравенство:

$$\|q\| \leq \sqrt{2} \cdot 2^{\alpha} \leq \sqrt{2} \cdot 2^{\frac{t}{2}}$$

Если $t \geq 2\alpha$ ожидается, что q кратчайший вектор в L. Можно найти кратчайший вектор в L, используя алгоритм LLL (в оригинале статьи [7] предлагается использовать метод Гауссова исключения) над базисом B_L, получим эквивалентный базис $B = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$:

Покажем, что $b_1 = \pm q = \pm(q_1, q_2)$.

Поскольку L имеет полный ранг, по неравенству Адамара имеем:

$$\|b_1\|\|b_2\| \geq \det(L),$$

$$\|b_2\| \geq \frac{\det(L)}{\|b_1\|}.$$

Подставляя $\det(L) = 2^t$ и используя $\lambda_1(L) \leq 2^{\frac{t+1}{2}}$ ведет к:

$$\|b_2\| \geq \frac{2^t}{2^{\frac{t+1}{2}}} = 2^{\frac{t-1}{2}}.$$

Это применимо для любого вектора решетки $v = a_1 b_1 + a_2 b_2$, для которого выполняется $\|v\| < 2^{\frac{t-1}{2}}$, значит $a_2 = 0$. В противном случае, $\lambda_2(L) \leq \|v\| < \|b_2\|$, что противоречит выбору b_2. Таким образом, каждый v: $\|v\| < 2^{\frac{t-1}{2}}$ – произведение константы на b_1. Можно заметить, что для $q = (q_1, q_2) \in L$ выполняется $\|q\| = \sqrt{2} * 2^{\alpha} = 2^{\frac{2\alpha+1}{2}}$. Следовательно, $\|q\| < \|b_2\|$, если

$$2^{\frac{(2\alpha+1)}{2}} < 2^{\frac{t-1}{2}},$$

$$2(\alpha + 1) < t.$$

Следовательно, получив $q = ab_1$, для некоторого $a \in Z\backslash\{0\}$, пусть $b_1 = (b_{11}, b_{12})$, тогда $\text{НОД}(q_1, q_2) = \text{НОД}(ab_{11}, ab_{12}) \geq a$. Но $q_1, q_2 -$ простые числа и к тому же $q_1 \neq q_2$, в противном случае мы могли бы разложить N_1, N_2, вычислив $\text{НОД}(N_1, N_2)$. Следовательно, $|a| = 1$, и получаем $q = \pm b_1$, что и требовалось показать.

Время работы разложения определяется временем выполнения Гауссова исключения, которое выполняется для матрицы размерности 2 за $O(\log^2(\min\{N_1, N_2\}))$[7].

3. 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 m_0, а сообщение имеет вид $m = m_0 + x_0$. Задачу можно представить в следующем виде:

$$f(x) = (m_0 + x)^e - c, \text{ где } f(x_0) \equiv 0 \ (mod\ N).$$

Копперсмит доказал, что найти корень данного выражения можно за полиномиальное время, если x_0 и e достаточно малы [3].

Теорема 2 (Копперсмита). Пусть N – целое число, которое имеет делитель $b \geq N^{\beta}$, $0 < \beta \leq 1$. Пусть $f(x)$ – полином от одной переменной степени δ и пусть $c \geq 1$. Тогда мы можем за время $O(c\delta^5 \log^9(N))$ найти все решения x_0 уравнения:

$$f(x) = 0 \ (mod\ b), \text{ при } |x_0| \leq cN^{\frac{\beta^2}{\delta}}.$$

Поиск корней полинома над кольцом целых чисел по модулю N сложная задача, поэтому производится сведение ее к поиску корней полинома над кольцом целых чисел. Копперсмит предложил обратить внимание на такой полином [3]:

$$f(x_0) = 0 \ (mod\ N), \text{при } |x_0| < X.$$

Чтобы перейти к $g(x_0) = 0$, полиному – имеющему корень x_0 над кольцом целых чисел можно использовать следующую теорему [3].

Теорема 3 (Howgrave-Graham). Пусть $g(x)$ полином от одной переменной с n мономами, $m \in N$, при этом выполняются следующие условия:

$$g(x_0) = 0 \ (mod\ N^m), \text{где } |x_0| \leq X,$$

$$\|g(xX)\| < \frac{N^m}{\sqrt{n}}.$$

Тогда $g(x_0) = 0$ имеет 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 x_0 как корень по модулю N^m:

$$g(x_0) = \sum_{i=1}^{n} a_i f_i(x_0) = 0 \ (mod\ N^m) \ \ a_i \in Z$$

Второе свойство позволяет использовать свойство теоремы 3 (Howgrave-Graham) $\|g(xX)\| < \frac{N^m}{\sqrt{n}}$.

Конкретно, для рассматриваемой атаки, в работе [3] было предложено генерировать полиномы f_i (далее это $g_{i,j}$ и h_i), с помощью которых строится $g(x_0)$, следующим образом (δ здесь степень полинома f):

$$g_{i,j}(x) = x^j N^i f^{m-i}(x) \text{ для } i = 0, \dots, m-1;\ j = 0, \dots, \delta - 1,$$

$$h_i(x) = x^i f^m(x) \text{ для } i = 0, \dots, t-1.$$

Для этих полиномов справедливо следующее:

1) Они имеют такой же корень x_0, но по модулю N^m.
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 $f_i(xX)$ в качестве строк. Далее к этому базису необходимо применить алгоритм LLL, с помощью которого вычисляется вектор, удовлетворяющий условию:

$$\|b_1\| \leq 2^{\frac{n-1}{4}} \cdot \det(L)^{\frac{1}{n}}.$$

По теореме 3 (Howgrave-Graham) имеем:

$$\|b_1\| = \|g(xX)\| < \frac{N^m}{\sqrt{n}}.$$

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

$$\det(L) < 2^{-\left(\frac{n(n-1)}{4}\right)} n^{-\frac{n}{2}} \cdot N^{nm}.$$

Можно упростить оценку до:

$$\det(L) < N^{mn}.$$

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 x в своей теореме. Чтобы использовать этот алгоритм необходимо определить значения m и t, пока не получим удовлетворительные оценки. Однако даже если сделать это неверно, можно найти правильный ответ (благодаря свойствам LLL алгоритма).

Для $f(x) = (m_0 + x)^e - c = (mod\ b),$ где $b \geq N^\beta, 0 < \beta \leq 1,\ |x_0| < X.$ в работе [3] (оценивая значения диагональных элементов матрицы, полученной при помощи приведенных выше полиномов) показывается, что для атаки необходимо выполнение:

$$X^{n-1} < N^{\beta m}$$

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

Вспомним схему подписи RSA. Чтобы подписать сообщение m, сначала вычисляется хеш $\mu(m)$, далее вычисляется сама подпись, в виде $s = \mu(m)^d \ mod \ N$. Для CRT-RSA получим подпись используя соотношения:

$$s_p = \mu(m)^{d_p} \ mod \ p \ \text{ и } s_q = \mu(m)^{d_q} \ mod \ q.$$

Где $d_p = d \ (mod \ p - 1)$ и $d_q = d \ (mod \ q - 1)$.

Вычисление подписи:

$$s = s_p + p(p^{-1} \ mod \ q)\big(s_q - s_p\big) \ mod \ N.$$

Такая схема позволяет выполнить подписание примерно в 4 раза быстрее.

В работе [4] рассматривается атака на ISO/IEC 9796-2. В данном стандарте сообщение разбивается на 2 части $m = m[1]||m[2]$. Функция $\mu(m)$ в нем имеет следующий вид:

$$\mu(m) = 6A_{16} \parallel m[1] \parallel H(m) \parallel BC_{16}.$$

$H(m)$ – хеш функция.

В 2009 году, CJKNP описали атаку на случайную версию этого стандарта [4]. В данном случае $m[1] = \alpha \parallel r \parallel \alpha', m[2] = data$, где r – неизвестная часть, α, α' известны и $data$ либо известная, либо неизвестная строка. Обозначим $len\big(H(m)\big) = k_h,\ len(N) = k,\ len(r) = k_r,\ len(m[1]) = k - k_h - 16$ (в соответствии со стандартом). Тогда предыдущая функция примет вид:

$$\mu(m) = 6A_{16} \parallel \alpha \parallel r \parallel \alpha' \parallel H(\alpha \parallel r \parallel \alpha' \parallel data) \parallel BC_{16}.$$

Таким образом, общее количество неизвестных бит $\mu(m)$ равно $k_r + k_h$.

Тогда атакующий решает задачу в следующем виде:

$$s^e = v + r * 2^{n_r} + H(m) \cdot 2^8 \ (mod \ p).$$

Где v известное значение и $n_r = k_h + k_{\alpha'} + 4$.

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

$$a + bx + cy = 0 \ (mod \ p).$$

Где $a = v - s^e \ mod \ N, \ b = 2^{n_r}, \ c = 2^8$. В работе [4] приводится оценка границы для этой атаки. При $r < N^u, H(m) < N^\delta$, можно вычислить r и $H(m)$, при условии, что $u + \delta \leq \frac{\sqrt{2}-1}{2} \approx 0.207$.

4. МЕТОД BONEH И DURFEE

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

$$d < N^{0.292}.$$

Вспомним структуру ключей RSA:

$$ed = 1\big(mod\ \phi(N)\big),$$

$$ed = k\phi(N) + 1,$$

$$k\phi(N) + 1 = 0\ (mod\ e),$$

$$k(N + 1 - p - q) + 1 = 0\ (mod\ e).$$

Здесь неизвестны k и $(-p - q)$. Это можно записать в виде следующего полинома:

$$f(x, y) = x \cdot (A + y),\ \text{где}\ A = N + 1\ \text{и}\ y = -p - q,$$

$$f(x_0, y_0) = 0\ (mod\ e).$$

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 (g_1, g_2). Далее необходимо вычислить их результант. Найдя корень x_0 из результанта можно ввести его в g_1, чтобы найти y_0. Но решение может быть не найдено, например, если g_1 и g_2 не линейно независимые, результант будет равен нулю.

Boneh и Durfee [3] предложили построить следующие полиномы, как f_i:
Для $k = 0, \dots, m$:

$$g_{i,k}(x) = x^i \cdot f^k(x, y) \cdot e^{m-k}\ \text{для}\ i = 0, \dots, m - k,$$

$$h_{j,k}(x) = y^j \cdot f^k(x, y)\ \cdot e^{m-k}\ \text{для}\ j = 0, \dots, t.$$

Используя эти полиномы для построения решетки, балансируются переменные так, чтобы определитель треугольного базиса не превышал e^{mn}. Boneh и Durfee показали, что LLL дает успешный результат если $d < N^{0.284}$ [3]. Пример решетки, получающейся при помощи приведенных полиномов:

$$\begin{pmatrix} e^2 & & & & & & & & \\ 0 & e^2X & & & & & & & \\ e & eAX & eXY & & & & & & \\ 0 & 0 & 0 & e^2X^2 & & & & & \\ 0 & eX & 0 & eAX^2 & eX^2Y & & & & \\ 1 & 2AX & 2XY & A^2X^2 & 2AX^2Y & X^2Y^2 & & & \\ 0 & 0 & 0 & 0 & 0 & 0 & e^2Y & & \\ 0 & 0 & eAXY & 0 & 0 & 0 & eY & eXY^2 & \\ 0 & 0 & 2AXY & 0 & A^2X^2Y & 2AX^2Y^2 & Y & 2XY^2 & X^2Y^3 \end{pmatrix}$$

Для достижения улучшенных результатов ($d < N^{0.292}$), они используют подрешетку, исключая некоторые полиномы. Таким образом уменьшается граница кратчайшего вектора, получаемого LLL, а матрица принимает следующий вид:

$$\begin{pmatrix} e^2 & & & & & & & & \\ 0 & e^2X & & & & & & & \\ e & eAX & eXY & & & & & & \\ 0 & 0 & 0 & e^2X^2 & & & & & \\ 0 & eX & 0 & eAX^2 & eX^2Y & & & & \\ 1 & 2AX & 2XY & A^2X^2 & 2AX^2Y & X^2Y^2 & & & \\ 0 & 0 & 2AXY & 0 & A^2X^2Y & 2AX^2Y^2 & Y & 2XY^2 & X^2Y^3 \end{pmatrix}$$

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

5. АТАКА НА МАЛЫЙ d_q

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

Лемма 1 (Howgrave-Graham). пусть $h'(x_1, \dots, x_r) \in Z[x_1, \dots, x_r]$ – полином с не более чем n мономами. Пусть $m, W, X_1, \dots, X_r \in N$ и справедливо следующее [2]:

$$1.\, h'(x_1', \dots, x_r') = 0\ (mod\ W^m), \text{где } |x_1'| < X_1, \dots, |x_r'| < X_r,$$

$$2\,.\, \|h'(x_1X_1, \dots, x_rX_r)\| < \frac{W^m}{\sqrt{n}}.$$

Тогда $h'(x_1', \dots, x_r') = 0$ над кольцом целых чисел.

Чтобы решить уравнение с r переменными $h(x_1, \dots, x_r) = 0\ (mod\ W)$ достаточно найти r новых полиномов $h_1'(x_1, \dots, x_r), \dots, h_r'(x_1, \dots, x_r)$, 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]:

$$ed_q = 1 + k(q - 1). (1)$$

Если мы сможем решить следующее выражение, корень которого $(x_q, y_q) = (k, q)$, то N может быть факторизовано.

$$f_q(x_q, y_q) = 1 + x_q(y_q - 1) \equiv 0\ (mod\ e).$$

Будем считать, что q больше, чем p. Домножим обе части на p:

$$ed_qp = p + k(N - p) = N + (k - 1)(N - p). (2)$$

Корнем следующего сравнения является $(x_p, y_p) = (k - 1, p)$, после его нахождения N может быть факторизовано:

$$f_p(x_p, y_p) = N + x_p(N - y_p) \equiv 0\ (mod\ e).$$

Пусть $p = N^\beta, q = N^{1-\beta}$, $e = N^\alpha$ и $d_q = N^\delta$. Тогда значения корня (x_p, x_q, y_p, y_q) ограничены сверху $X_p = X_q = N^{\alpha+\beta+\delta-1}, Y_p = N^\beta, Y_q = N^{1-\beta}$.

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

$$g_{i,j}(x, y) = e^{m-i}x^j f_p^i(x, y), i = 0, \dots, m, j = 0, \dots, m - i,$$

$$h_{i,j}(x, y) = e^{m-i}y^j f_p^i(x, y), i = 0, \dots, m, j = 1, \dots, m.$$

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

$$\begin{pmatrix} e & & & & & & \\ 0 & eX_p & & & & & \\ N & NX_p & -X_pY_p & & & & \\ 0 & 0 & 0 & eY_p & & & \\ 0 & 0 & NX_pY_p & NY_p & -X_pY_p^2 & & \\ 0 & 0 & 0 & 0 & 0 & eY_p^2 & \\ 0 & 0 & 0 & 0 & NX_pY_p^2 & NY_p^2 & -X_pY_p^3 \end{pmatrix}$$

Здесь строки состоят из коэффициентов семи полиномов:

$$e, ex_p, f_p(x_p, y_p), ey_p, y_p f_p(x_p, y_p), ey_p^2, y_p^2 f_p(x_p, y_p).$$

Все полиномы разделяют корень $f_p(x_p, y_p)$ $(mod\ e)$. К тому же к базовым полиномам $e, ex_p, f_p(x_p, y_p)$ он добавил дополнительный y_p – сдвиг: $ey_p, ey_p^2, yf_p(x_p, y_p), y^2 f_p(x_p, y_p)$.

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

$$X_p^4 Y_p^9 e^4 < e^7.$$

Это выполняется тогда и только тогда, когда:

$$4(\alpha + \beta + \delta - 1) + 9\beta < 3\alpha,$$

$$\delta < 1 - \frac{\alpha + 13\beta}{4}.$$

Основная идея этого подхода в том, чтобы решать уравнение 2 вместо 1, потому что p значительно меньше q. Если p близко к q , настолько, что $\beta \geq 0.382$, атака Мэя работать не будет.

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

$$\begin{pmatrix} e & & & & & & \\ 0 & eX_p & & & & & \\ N & NX_p & -X_pY_p & & & & \\ 0 & 0 & 0 & eY_p & & & \\ 0 & 0 & NX_pY_p & NY_p & -X_pY_p^2 & & \\ 0 & 0 & 0 & 0 & 0 & eY_q & \\ 0 & -X_p & 0 & 0 & 0 & Y_q & X_pY_q \end{pmatrix}$$

Здесь строки состоят из коэффициентов семи полиномов:

$$e, ex_p, f_p(x_p, y_p), ey_p, y_p f_p(x_p, y_p), ey_q, N^{-1} y_q f_p(x_p, y_p).$$

Граница определяется следующим образом:

$$X_p^4 Y_p^4 Y_q^2 e^4 < e^7,$$

$$4(\alpha + \beta + \delta - 1) + 4\beta + 2(1 - \beta) < 3\alpha,$$

$$\delta < \frac{1}{2} - \frac{\alpha + 6\beta}{4}.$$

В сравнение с предыдущей, степень Y_p уменьшается, благодаря умножению на Y_q. Таким образом, атака работает для $p < N^{0.468}$.

Наконец, в работе [2] авторы, чтобы улучшить предыдущую атаку, рассматривают не только $f_p(x_p, y_p)$, но и $f_q(x_q, y_q)$, как представление для того же полинома. Это полезно, если заметить, что $x_q = x_p + 1$. В предыдущей матрице был полином ey_q, он был необходим, чтобы матрица была треугольной [6], по сути, он не увеличивал границу нахождения решения. Введя $f_q(x_q, y_q)$ необходимость в нем отпала. В работе [2] предлагается генерировать полиномы следующим образом:

$$g_{[i,j]}(x_p, y_p) = x_p^j f_p^i(x_p, y_p) e^{m-i} \quad i = 0, 1, \ldots, m; j = 0, 1, \ldots m - i,$$

$$g'_{[i,j]}(x_p, y_p) = y_p^j f_p^i(x_p, y_p) e^{m-i} \quad i = 0, 1, \ldots, m; j = 1, 2, \ldots, m + 1,$$

$$g''_{[i,j]}(x_p, x_q, y_p, y_q) = f_p^{i-j}(x_p, y_p) f_q^j(x_q, y_q) e^{m-i} \quad i = 1, 2, \ldots, m;\ j = 1, 2, \ldots, i.$$

Матрица принимает следующий вид [2]:

$$\begin{pmatrix} e & & & & & \\ 0 & eX_p & & & & \\ N & NX_p & -X_pY_p & & & \\ 0 & 0 & 0 & eY_p & & \\ 0 & 0 & NX_pY_p & NY_p & -X_pY_p^2 & \\ 0 & -X_p & 0 & 0 & 0 & X_qY_q \end{pmatrix}$$

Здесь строки состоят из коэффициентов шести полиномов:

$$e, ex_p, f_p(x_p, y_p), ey_p, y_p f_p(x_p, y_p), f_q(x_q, y_q).$$

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

$$X_p^3 X_q Y_p^4 Y_q e^3 < e^6,$$

$$4(\alpha + \beta + \delta - 1) + 4\beta + (1 - \beta) < 3\alpha,$$

$$\delta < \frac{3}{4} - \frac{\alpha + 7\beta}{4}.$$

Таким образом $\beta \leq \frac{1}{2}$.

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

Теорема 4. Пусть $N = pq,\ p = N^{\beta}, q = N^{1-\beta}$, для $\beta < \frac{1}{2}$. Пусть $e = N^{\alpha}$ и $d_q < N^{\delta} : ed_q = 1(mod(q-1))$, если N достаточно большое и выполняются условия [2]:

$$\delta < \frac{(1-\beta)(3+2\beta) - 2\sqrt{\beta(1-\beta)(\alpha\beta+3\alpha+\beta)}}{3+\beta} \text{ и } \alpha > \frac{\beta}{1-\beta}.$$

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

6. РЕАЛИЗАЦИЯ АТАКИ НА МАЛЫЙ d_q

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

Параметры должны иметь следующий вид:

$$p = N^{\beta}, q = N^{1-\beta}, \beta \leq \frac{1}{2}, e = N^{\alpha}, d_q = N^{\delta}, X_p = N^{\alpha+\beta+\delta-1}, Y_p = N^{\beta},$$

$$X_q = N^{\alpha+\beta+\delta-1}, Y_q = N^{1-\beta}, ed_q = 1\ mod(q-1), ed_p = 1\ mod(p-1).$$

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")
```

```
generated
('m :', 2)
('t :', 1)
('len N: ', 1024)
('e :', 20623675431556927581114093083757482365306831097440157001526553997524508421164143080247456483765890144678977774708434453
00514949606632719792312593705044045848780876485351169431007204554533565804868078426733453527942 9)
('p :', 36740960911192916393874237231454464361027775291963327790140017073264847521390205311420048417)
('q :', 54161157978372121044124480569133029533519224509343316761597805181667286559995830391753690263894675110030088838668157502
35993597824317541896800225603631107988351503243221728413458295899284821883104167238111297245985 9)
('N: ', 19899329881883144579795784692156674769268116176581110114768061519384426501266369919932142838419215390351461202111034966
47992789876832802074758246320755184616201431874988061026746828155523695858002017122401727680896159453380138829392050763889557 15
35909182448274268616459054366016322871049091840891077368993203)
('dp: ', 30997441686727568940449313066970047004087540502293462620316881904712370842351746034549738509)
('dq: ', 152265)
('alpha : ', 0.6974135247147396)
('delta : ', 0.01686501684466034)
('beta : ', 0.29796777290758947)
('len dq = ', 20)
('len dp = ', 306)
('len X = ', 20)
('X = ', 199421)
('len Yp = ', 313)
('Yp = ', 25274041742720080872206851267767988885503651857581446814216559306041237849214933073626566492 16)
('len Yq = ', 724)
('Yq = ', 18628682179654441466090670484720472850265125682193029823464124526788144138521800645169432388958209197102581935833917 4
07983886902661543340241181262615956239064247576514290808850511346594531519301172522113256997857001472)
```

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:

$$y_p = p$$
$$x_p = \frac{e * d_q * p - N}{N - y_p}$$
$$y_q = q$$
$$x_q = \frac{ed_q - 1}{y_q - 1}$$

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

$$g_{[i,j]}(x_p, y_p) = x_p^j f_p^i(x_p, y_p) e^{m-i} \quad i = 0, 1, \dots, m; j = 0, 1, \dots m - i,$$

$$g'_{[i,j]}(x_p, y_p) = y_p^j f_p^i(x_p, y_p) e^{m-i} \quad i = 0, 1, \dots, m; j = 1, 2, \dots, \frac{m}{2} + 1,$$

$$g''_{[i,j]}(x_p, x_q, y_p, y_q) = f_p^{i-j}(x_p, y_p) f_q^j(x_q, y_q) e^{m-i} \quad i = 1, 2, \dots, m; j = 1, 2, \dots, i.$$

Для наглядности, использовалось $m = 2$.

```
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:

$$x_q = x_p + 1, \qquad y_q y_p = N.$$

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 $m = 2$.

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
g	e^2	1
	$x_p e^2$	x_p

	$x_p^2 e^2$	x_p^2
	$-x_p y_p e + x_p N e + N e$	$x_p y_p$
	$-x_p^2 y_p e + x_p^2 N e + x_p N e$	$x_p^2 y_p$
	$x_p^2 y_p^2 - 2x_p^2 y_p N + x_p^2 N^2 - 2x_p y_p N + 2x_p N^2 + N^2$	$x_p^2 y_p^2$
g'	$y_p e^2$	y_p
	$-x_p y_p^2 e + x_p y_p N e + y_p N e$	$x_p y_p^2$
	$x_p^2 y_p^3 - 2x_p 2y_p^2 N + x_p^2 y_p N^2 - 2x_p y_p^2 N + 2x_p y_p N^2 + y_p N^2$	$x_p^2 y_p^3$
g''	$x_q y_q e - x_q e + e$	$x_q y_q, x_q$
	$-x_p x_q y_p y_q + x_p x_q y_q N + x_p x_q y_p - x_p x_q N + x_q y_q N - x_p y_q + x_p N - x_q N + N$	$x_p x_q y_p y_q, x_p x_q y_q, x_p x_q y_p, x_q x_p$
	$x_q^2 y_q^2 - 2x_q^2 y_q + x_q^2 + 2x_q y_q - 2x_q + 1$	$x_q^2 y_q^2, x_q^2 y_q, x_q^2$

Видим, что при введении первого же полинома g'' в наше множество векторов, количество мономов увеличивается более чем на 2.

Чтобы избавиться от них, воспользуемся свойствами, описанными выше.

Рассмотрим первый такой полином и воспользуемся следующим соотношением:

$$x_q \to x_p + 1,$$

Тогда, останется лишь 1 новый моном $x_q y_q$, а полином примет вид:

$$x_q y_q e - x_q e + e = x_q y_q e - x_p e.$$

Далее, имеем целых четыре монома:

$$x_p x_q y_p y_q \to x_p (x_p + 1) N,$$

$$x_p x_q y_q \to (x_q - 1) x_q y_q,$$

$$x_p x_q y_p \to x_p (x_p + 1) y_p,$$

$$x_q x_p \to x_p (x_p + 1).$$

Все из получившихся мономов, кроме $x_q^2 y_q$ уже добавлены, поэтому он будет единственным новым.

С помощью этих соотношений получим полином вида:

$$-x_p x_q y_p y_q + x_p x_q y_q N + x_p x_q y_p - x_p x_q N + x_q y_q N - x_p y_q + x_p N - x_q N + N = x_q^2 y_q \boldsymbol{N} + x_p^2 y_P - 2x_p^2 N - 2x_p N.$$

Далее аналогично, последний полином добавит только моном $x_q^2 y_q^2$ и будет иметь вид:

$$x_q^2 y_q^2 - 2x_q^2 y_q + x_q^2 + 2x_q y_q - 2x_q + 1 = x_q^2 y_q^2 - 2x_q^2 y_q + x_p^2 + 2x_q y_q.$$

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. $\det(B) < e^{mn}$, где n – размерность матрицы. В данном случае на диагональном коэффициенте появился множитель N (выделен жирным), от него следует избавиться, путем умножения полинома на обратный элемент N по модулю e^m. 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:

$a_1 * x_p^2 * y_p + x_q^2 * y_q - a_2 * x_p^2 - a_3 * x_p$, где a_1, a_2, a_3 – некие коэффициенты.

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

$$1 \quad x_p \quad x_p^2 \quad x_p y_p \quad x_p^2 y_p \quad x_p^2 y_p^2 \quad y_p \quad x_p y_p^2 \quad x_p^2 y_p^3 \quad x_q y_q \quad x_q^2 y_q \quad x_q^2 y_q^2$$

$$\begin{pmatrix}
e^2 & & & & & & & & & & & \\
0 & Xe^2 & & & & & & & & & & \\
0 & 0 & X^2e^2 & & & & & & & & & \\
+ & + & 0 & -XY_pe & & & & & & & & \\
0 & + & + & 0 & -X^2Y_pe & & & & & & & \\
+ & + & + & - & - & X^2Y_p^2 & & & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & Y_pe^2 & & & & & \\
0 & 0 & 0 & + & 0 & 0 & + & -XY_p^2e & & & & \\
0 & 0 & 0 & + & + & - & + & - & X^2Y_p^3 & & & \\
0 & - & 0 & 0 & 0 & 0 & 0 & 0 & 0 & XY_qe & & \\
0 & - & - & 0 & + & 0 & 0 & 0 & 0 & 0 & X^2Y_q & \\
0 & 0 & + & 0 & 0 & 0 & 0 & 0 & 0 & + & - & X^2Y_q^2
\end{pmatrix}$$

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

$$X^{16}Y_p^{10}Y_q^4e^{12} < e^{12\cdot 2},$$

$$X^{16}Y_p^{10}Y_q^4 < e^{12},$$

$$16(\alpha+\beta+\delta-1)+10\beta+4(1-\beta) < 12\alpha,$$

$$8(\alpha+\beta+\delta-1)+5\beta+2-2\beta < 6\alpha,$$

$$8\alpha+11\beta+8\delta-6 < 6\alpha,$$

$$2\alpha < 6-11\beta-8\delta,$$

$$\delta < \frac{3}{4}-\frac{\alpha}{4}-\frac{11\beta}{8}.$$

Сравним, с результатом из раздела 5, 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 e^{mn}.

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)))
```

```
7
=== 1.04181718826 seconds ===
```

Рисунок 2 – Время выполнения и количество подходящих полиномов

Далее, xxp, yyp, yyq – искомые секретные значения, $pol1, pol2, pol3$ – полученные с помощью LLL полиномы.

Результат выполнения:

```
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))

(5797.0, 3.6740960911192914e+91, 5.416115797837212e+215)
5797.0
3.6740960911192914e+91
5.416115797837212e+215
5797
3674096091119291639387423723145446436102777529196332779014001707326484752139020531142004841 7
5416115797837212104412448056913302953351922450934331676159780518166728655999583039175369026389467511003008883866815750235993597
8243175418968002256036311079883515032432217284134582959899284821883104167238111297245985 9

: array([5.79700000e+003, 3.67409609e+091, 5.41611580e+215])
```

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 d_q, который использует схему генерации ключей CRT-RSA и соотношения неизвестных параметров в ней, чтобы уменьшить определитель базисной матрицы, этот метод, позволяет атаковать систему при $\beta \leq \frac{1}{2}$, еще Boneh и Durfee более чем через 19 лет назад отметили, что их граница: $d < N^{1-\frac{1}{\sqrt{2}}}$, не может быть окончательным результатом, так как она слишком неестественна и предложили идею, что она может увеличена до $d < N^{\frac{1}{2}}$, 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)

7 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)

8 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.mono
mial_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))
```