Федеральное государственное автономное образовательное
Учреждение высшего профессионального образования
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет МИЭМ
Департамент прикладной математики

ПРОЕКТНАЯ РАБОТА

для направления 01.03.04 «Прикладная математика»

«МАСКИРОВКА ПОРОЖДАЮЩЕЙ/ПРОВЕРОЧНОЙ МАТРИЦЫ КОДА»

Выполнила:
студентка группы БПМ232
Захарова Юлиана Владимировна

Руководитель:
к.ф.-м.н., доц.,
Малыгина Екатерина Сергеевна

Москва 2024

Содержание

1 Теоретическая часть

1.1 Введение

Одним из перспективных видов систем с открытым ключом, потенциально безопасных в пост-квантовом мире, являются так называемые кодовые криптосистемы, построенные на базе оригинальных схем Мак-Элиса и Нидеррайтера. В обеих этих криптосистемах секретным ключом является код, для которого известен эффективный алгоритм декодирования. Открытый ключ представляет собой замаскированную версию секретного кода, которая выглядит случайной, скрывая структуру кода, приводящую к эффективному декодированию. Таким образом, безопасность такой системы основана на том, что злоумышленнику невозможно восстановить или раскрыть базовую структуру секретного кода из открытого ключа, что не позволяет использовать эффективный алгоритм декодирования.

Код маскируется с помощью случайной перестановки столбцов его порождающей матрицы и умножения слева на случайную обратимую матрицу. Шифрование заключается в кодировании открытого текста с помощью открытого кода и добавлении случайной ошибки, вес Хэмминга которой не превышает предела исправления ошибок секретного кода. Злоумышленнику необходимо будет исправить эту ошибку, чтобы восстановить открытый текст.

По сей день система Мак-Элиса не была взломана, если ее первоначально предложенный размер скорректировать для защиты от современных компьютерных мощностей и постепенного увеличения скорости алгоритмов декодирования. Однако ее основным недостатком является большой размер открытого ключа системы. По этой причине было предпринято множество попыток найти альтернативы оригинальному предложению Мак-Элиса, используя различные базовые коды с более компактными представлениями, включая как алгебро-геометрические коды, так и современные коды, такие как коды с проверкой четности низкой плотности с квазициклической структурой.

В (почти) всех подобных системах маскировка секретного кода состоит из случайных перестановок и изменении размера столбцов образующей матрицы вместе с умножением слева на обратимую матрицу. В то время как умножение слева, очевидно, является просто изменением базиса кода, операции со столбцами приводят к так называемому мономиально эквивалентному коду. Поскольку перестановки и изменение размерности координат кодового слова оставляют вес Хэмминга неизменным, эти операции можно легко совершать при построении криптосистемы. С другой стороны, этот тип маскировки часто оставляет слишком много изначальной алгебраической структуры, которая может быть использована злоумышленником.

Действительно, в большинстве случаев алгебраический тип кода остается неизменным: например, замаскированный обобщенный код Рида-Соломона остается обобщенным кодом Рида-Соломона. Как следствие, знание класса кода дает злоумышленнику дополнительные полезные сведения.

В работе развивается идея Балди [1], рассматривается криптосистема Мак-Элиса, в которой мономиальные преобразования заменены использованием обратимых матриц, строки которых имеют вес Хэмминга 2 и 3. Данный вариант аннигилирует алгебраическую структуру секретного кода, не оставляя следов для злоумышленника, который может восстановить его, используя алгоритм Шура. Действительно, представленная модификация показывает, что квадрат Шура открытого кода, являющийся центральным инструментом в большинстве методов, ведет себя как квадрат Шура случайного кода.

1.2 Предварительные сведения

Определение 1. Блочный код C определяется как линейное пространство F_q^n над полем F_q. Элемент $x \in C$ называется кодовым словом. Количество базисных кодовых слов будет определять размерность кода как векторного пространтсва. Кодовое слово длины n над алфавитом F_q^n есть вектор длины n, имеющий вид $(x_1, x_2..., x_n)$, где $x_i \in F_q \forall 1 \leq i \leq n$. Если C является кодом длины n, размерности k, будем говорить, что C - это $[n, k]$-код.

Определение 2. Минимальным расстоянием $d(C)$ кода C называют наименьшее расстояние межу двумя любыми ненулевыми кодовыми словами. Посредством параметра d можно определить, насколько хорошо код исправляет и выявляет ошибки. Так код способен выявить d ошибок и исправить до $t = \lfloor (d-1)/2 \rfloor$ ошибок. Таким образом, способность кода исправлять ошибки определяется его минимальным расстоянием, в таком случае мы говорим, что код исправляет t ошибок. Для генерации "хороших" кодов, очевидно, неообходимым является наличие большого числа исправляемых ошибок.

Определение 3. Для вычисления $d(C)$ используется расстояние Хэмминга, определенное над F_q^n. Пусть $x = (x_1, x_2..., x_n)$ и $y = (y_1, y_2..., y_n)$ - элементы F_q^n. Расстояние Хэмминга определяется как

$$d(x, y) := |i|x_i \neq y_i|.$$

Иными словами, расстояние между двумя кодовыми словами в F_q^n определяется как количество позиций, в которых слова отличаются.

Определение 4. Минимальным расстоянием является наименьшее возмож-

ное расстояние Хэмминга между двумя любыми словами кода:

$$d(C) := min|i|x_i \neq y_i|.$$

Определение 5. Вес элемента в F_q^n определяется как расстояние между кодовым словом и нулевым вектором, иными словами, количеством ненулевых координат. Пусть $x \in F_q^n$, тогда вес x определяется как

$$wt(x) = d(x, 0) = |i|x_i \neq 0|.$$

Определим минимальное расстояние C как наименьший ненулевой вес кодового слова:

$$d(C) = wt(C) = mind(x, y)|x \neq y, x, y \in C.$$

Определение 6. Пусть C - векторное пространство с базисом размера k. Если $C \subset F_q^n$ является $[n, k, d]$-кодом, то порождающая матрица G кода C является матрицей размера $k \times n$, строки которой являются базисом C. Порождающая матрица - это матрица, строки которой независимы и задают пространство кода.

Пусть $GL_n(F) := M \in F^{n*n} | \det(M) \neq 0$ - общая линейная группа, определим:

$$M_n := \{P \in GL_n(F) | P - \text{мономиальная матрица}, \}$$

где каждая матрица называется мономиальной, если каждая строка и каждый столбец содержат ровно один ненулевой элемент. Другими словами, мономиальные матрицы - это в точности произведения перестановочных и диагональных матриц. Очевидно, M_n является подгруппой в $GL_n(F)$. Мономиальные матрицы могут быть охарактеризованы следующим образом.
Для любой $P \in GL_n(F)$ имеем

$$P \text{ - мономиальна } \iff \left\{ \frac{\text{существует изоморфизм } F^n \to F^n, x \to xP}{\text{сохраняется вес Хэмминга}} \right\}.$$

Это свойство играет главную роль в криптосистемах, основанных на кодах, поскольку мономиальные матрицы дествуют на векторы ошибок не изменяя вес таких векторов. Это необходимо в следующем алгоритме для того, чтобы легитимный получатель действительно мог декодировать полученный шифртекст. Мы фиксируем поле F, параметры k, n и предполагаем, что эти данные публичны.

1.3 Маскировка 1. Криптосистема Мак-Элиса.

1) Выберем параметр t $(C, D) \in g_{q,n,k,t}$. Предположим, что C задается порождающей матрицей $G \in F^{k*n}$ и обладает эффективным алгоритмом декодирования D, исправляющим не менее t ошибок.

2) Выберем случайные матрицы $S \in GL_k(F)$ и $P \in M_n$ и зададим $\underline{G} := SGP^{-1}$.

3) **Открытый ключ:** $(\underline{G}, t)$.
Сектретный ключ: (S, G, P).

4) **Пространство открытого текста:** $Z = F^k$.

5) **Шифрование:** Зашифруем открытый текст $m \in Z$ в $c := m\underline{G} + e$, где $e \in F^n$ - случайно выбранный вектор с весом не более t.

6) **Расшифрование:**

- Вычислим $c' := cP = mSG + eP$
- Декодируем c' с помощью алгоритма декодирования D для C и обозначим результат через m'.
- Возвращаем $m'S^{-1}$.

Рассмотрим произведение и квадрат Шура и обсудим их важность.

Определение 7. Для $x = (x_1, x_2..., x_n), y = (y_1, y_2..., y_n) \in F^n$ определим произведение $(*)$ как $x * y := (x_1y_1, ..., x_ny_n)$. Для двух кодов $C_1, C_2 \subseteq F^n$ определим произведение Шура как

$$C_1 * C_2 := span_F\{x * y | x \in C_1, y \in C_2\}.$$

Зададим $C^2 := C * C$ и назовем его квадратом Шура кода C.

Очевидно, что это произведение коммутативно и билинейно. Более того, если код $C \subseteq F^n$ имеет базис $g_1, ..., g_k$, то

$$C^2 = span_F\{g_i * g_j | i = 1, ..., k, j = i, ..., k\},$$

следовательно, $\dim(C^2) \leq \left\{ \binom{k+1}{2}, n \right\}$. Имеется также следующий общий результат.

Предложение. Для двух линейных кодов C_1 и $C_2 \subseteq F_q^n$ размерности $C_1 * C_2$ и C_1^2 ограничены следующим образом:

$$\dim(C_1 * C_2) \leq \min\left\{n, \dim(C_1)\dim(C_2) - \binom{\dim(C_1 \cap C_2)}{2}\right\},$$

$$\dim(C_1^2) \leq \min\left\{n, \binom{\dim(C_1)+1}{2}\right\}. \quad (1)$$

Глядя на определение квадрата кода, отметим, что он генерируется всеми возможными произведениями двух базисных элементов кода. Таким образом, для случайно выбранного линейного кода C мы ожидаем, что неравенство 1, на самом деле, является равенством с очень большой вероятностью. Действительно, Каскудо и др.[2] показали, что при определенных условиях длина n и размеерность k у C с большой вероятностью

$$\dim(C^2) = \begin{cases} n, & \text{если } \dfrac{k(k+1)}{2} \geq n, \\ \dfrac{k(k+1)}{2}, & \text{если } \dfrac{k(k+1)}{2} < n. \end{cases} \tag{2}$$

2 Практическая часть

2.1 Маскировка 2

Рассмотрим иной вариант криптосистемы Мак-Элиса. Он отличается от классической маскировки 1 тем, что мы заменяем множество M_n мономиальных матриц на множество матриц

$$W_n := \{\text{матрицы веса 2}\}.$$

Определим вероятность того, что сумма матрицы веса 1 и единичной матрицы - матрица веса 2 ($I_n + P \in W_n, P-$ матрица веса 1).

Рассмотрим сложение матриц как сложение соответствующих строк. Если при сложении строк единицы стоят на разных позициях, то в результате получим строку с двумя единицами. Если единицы стоят на одинаковых позициях - строку с двойкой на данной позиции, остальные нули.

$$\begin{matrix} (0\ 1\ 0\ 0\ \ldots\ 0\ 0) \\ + \\ (0\ 0\ 0\ 1\ \ldots\ 0\ 0) \\ = \\ (0\ 1\ 0\ 1\ \ldots\ 0\ 0) \end{matrix} \qquad \begin{matrix} (0\ 1\ 0\ 0\ \ldots\ 0\ 0) \\ + \\ (0\ 1\ 0\ 0\ \ldots\ 0\ 0) \\ = \\ (0\ 2\ 0\ 0\ \ldots\ 0\ 0) \end{matrix}$$

В единичной матрице в i-ой строке единица стоит на i-ом месте, следовательно, чтобы получить первый случай, в матрице веса 1 единицы не должны стоять

на главной диагонали.

Аналогично для столбцов.

Вывод: чтобы получить матрицу веса 2 путем сложения единичной и матрицы веса 1, последняя не должна иметь единиц на главной диагонали.

Далее поймем, что матрица размерности n и веса 1 задается перестановкой чисел от 1 до n, первое число соответствует позиции 1 в первой строке, второе число - позиции 1 во второй строке и т.д. Таких перестановок n!.

Хорошими перестановками для нас являются беспорядки (перестановки, где для каждой позиции ее номер не равен значению). Таких перестановок !n.

$$!n = n! - \frac{n!}{1!} + \frac{n!}{2!} - ... = \sum_{k=0}^{N} (-1)^k \frac{n!}{k!}$$

Вероятность того, что при сложении получим матрицу веса 2 равна вероятности того, что сложение выполняем с матрицей веса 1, задаваемой беспорядкой. Тогда $P = \frac{!n}{n!}$. Известно, что $\sum\limits_{k=0}^{\infty} (-1)^k \frac{1}{k!} = \frac{1}{e}$. Объединим:

$$P = \frac{\sum\limits_{k=0}^{N} (-1)^k \frac{n!}{k!}}{n!} = \frac{\frac{n!}{e}}{n!} = \frac{1}{e} \approx 0.367879..., \quad n \to \infty$$

Полученное теоретически значение вероятности подтверждают экспериментальные данные, представленные в таблице 1. Результаты получены с помощью программы на языке Python (приложение 1).

N	P
2	0.50302
4	0.37162
8	0.3666
16	0.36962
32	0.37046
64	0.36562
128	0.36608
256	0.368

Таблица 1: Экспериментальные результаты для маскировки 2.

2.2 Маскировка 3

Определим еще один вариант криптосистемы Мак-Элиса, заменив множество M_n матриц веса 1 на множество матриц

$$W'_n := \{\text{матрицы веса 3}\}.$$

Вычислим вероятность того, что сумма матрицы веса 2 и единичной матрицы - матрица веса 3, использовав для этого программный код (приложение 2). Результаты представлены в таблице 2.

N	P
2	0
4	0.07372
8	0.04624
16	0.0492
32	0.05164
64	0.05414
128	0.05372
256	0.05462

Таблица 2: Экспериментальные результаты для маскировки 3.

При $n \to \infty$ вероятность того, что $I_n + P \in W'_n, P \in W_n$, составляет $\sim 5\%$.

2.3 Поведение кода при разных маскировках

Целью маскировки кода является скрытие информации от злоумышленника. Считаем, что код ведет себя как случайный, если неравенство 2 выполняется. В противном случае, данные могут быть восстановлены с помощью алгебраической структуры кода.

Для исследования качества описанных маскировок реализуем программу (приложение 3) в системе компьютерной алгебры Sage на языке программирования Python. Будем рассматривать коды на эллиптических и гиперэллиптических кривых в разных полях и при разных значениях переменной в дивизоре. Сравнение маскировок будет произведено по вероятности, которая определяет частоту встречаемости успешного шифрования (код ведет себя как случайный). Для вычисления вероятности произведено 50 измерений. Результаты представлены в таблице 3.

q	d	m1	m2	m3
$y^2 - x^3 - x - 3$				
17	8	0	0,72	1
67	32	0	0,68	1
$y^2 - x^5 - x^3 - x - 3$				
17	8	1	0,56	0,92
67	32	0	0,6	0,96

Таблица 3: Сравнительная таблица маскировок по вероятностям.

2.4 Маскировка проколотого подкода

Рассмотрим проколотый подкод исходного кода, фиксируя элемент и удаляя из порождающей матрицы строку и столбец, которые содержат этот элемент. Для удаления выбран элемент (2, 10). К получившейся матрице подкода применяем маскировки 1-3 и считаем вероятность успешного шифрования. Результаты представлены в таблице 4.

q	d	m1	m2	m3
$y^2 - x^3 - x - 3$				
17	8	0	0,64	1
67	32	0	0,52	0,96
$y^2 - x^5 - x^3 - x - 3$				
17	8	0	0,64	0,84
67	32	0	0,52	1

Таблица 4: Сравнительная таблица маскировок по вероятностям для проколотого подкода.

3 Анализ результатов. Заключение

В теоретической части работы изучены оригинальная кодовая схема Мак-Элиса, определены расстояние Хэмминга и квадрат Шура. Рассмотрена маскировка кода, исправляющего ошибки, а также предложены ее модификации. Помимо этого исследовано поведение проколотого кода. Для оценки качества маскировок использовано неравенство 2, которое определяет, является поведение кода случайным или нет.

Результаты, полученные в ходе работы, позволяют сделать следующие выводы.

- Маскировка 1 (маскировка в оригинальной схеме Мак-Элиса) является небезопасносной, поскольку для большинства тестов вероятность "успешного" шифрования равна 0. Значение равное 1 можно отнести к выбросам и объяснить свойствами рассматриваемой кривой в выбранном поле.
- Маскировка 2 (маскировка матрицей веса 2) не рекомендуется для исполь-

зования. Она защищает данные с вероятностью $\sim 60\%$.

- Маскировка 3 (маскировка матрицей весом 3) оказалась самой качественной оказалась, (почти) всегда порождающая матрица кода ведет себя как случайная. Вероятность успешной атаки кода, замаскированного таким способом, составляет менее 5%.
- При прокалывании кода каждая из маскировок становится менее защищенной. Более чувствительной к такому преобразованию оказалась маскировка 2: вероятность "успешного" шифрования снизилась на 8%.

4 Список литературы

[1] Baldi M. et al. Enhanced public key security for the mceliece cryptosystem. *Journal of Cryptology*, 29:1–27, 2016.

[2] Cascudo I. et al. Squares of random linear codes. *IEEE Transactions on Information Theory*, 61(3):1159–1173, 2015.

Приложение 1

Программа для вычисления вероятности того, что сумма матрицы веса 1 и единичной матрицы - матрица веса 2.

```
from random import shuffle


def weight_2(matrix):  # checking that weight is 2
    for i in range(N):
        if matrix[i].count(1) != 2 or matrix[i].count(0) != (N - 2):  # checking i string
            return False
        col = [j[i] for j in matrix]  # array of elements i column
        if col.count(1) != 2 or col.count(0) != (N - 2):  # checking i column
            return False
    return True


def do(p, count):
    matrix = [([0] * N) for i in range(N)]  # making zero matrix
    for i in range(N):
        matrix[i][p[i]] = 1  # putting 1 (by permutation)
        matrix[i][i] += 1  # adding E matrix
    if weight_2(matrix):  # checking that weight is 2
        count += 1
    return count

```

```
23
24 N = 2  # dimension of matrix
25 permutation = list(range(0, N))
26 shuffle(permutation)
27 good = 0
28 for i in range(10_000):
29     good = do(permutation, good)
30     shuffle(permutation)
31
32 print(good / 10_000)
```

Приложение 2

Программа для вычисления вероятности того, что сумма матрицы веса 2 и единичной матрицы - матрица веса 3.

```
1  from random import shuffle
2  import numpy as np
3
4
5  def weight_3(matrix):  # checking that weight is 2
6      for i in range(N):
7          if matrix[i].count(1) != 3 or matrix[i].count(0) != (N - 3):  #
       checking i string
8              return False
9          col = [j[i] for j in matrix]  # array of elements i column
10         if col.count(1) != 3 or col.count(0) != (N - 3):  # checking i column
11             return False
12     return True
13
14
15 def make_weight2(n):
16     trace = 1
17     permutation = list(range(0, n))
18     while trace != 0:
19         shuffle(permutation)
20         matrix = np.zeros((n, n))  # making zero matrix
21         for i in range(n):
22             matrix[i][permutation[i]] = 1  # putting 1 (by permutation)
23         trace = np.trace(matrix)
24     matrix = np.add(matrix, np.eye(n))  # matrix weight 2 with 1 on the
       diagonal
25     permutation_rows = list(range(0, n))
26     shuffle(permutation_rows)
27     for i in range(n):
28         matrix[[i, permutation_rows[i]]] = matrix[[permutation_rows[i], i]]
29     return matrix
30
31
32 def do(count):
33     matrix = make_weight2(N)  # making matrix weight 2
34     matrix += np.eye(N)  # adding E matrix
35     if weight_3(matrix.tolist()):  # checking that weight is 3
```

```
        count += 1
    return count


N = 2  # dimension of matrix
good = 0
for i in range(10_000):
    good = do(good)

print(good / 10_000)
```

Приложение 3

Программа для сравнения поведения кода и проколотого подкода при разных маскировках.

```
import random
import numpy as np
from random import shuffle


def Pmatrix(n):  # генерирует матрицу веса 1
    slots = [i for i in range(n)]
    P = matrix(F, n, [0 for i in range(n ^ 2)])
    for i in range(n):
        index = random.choice(slots)
        slots.remove(index)
        P[i, index] = 1
    return P


def PmatrixW2(n):  # генерирует матрицу веса 2
    slots1 = [i for i in range(n)]
    slots2 = [i for i in range(n)]
    P = matrix(F, n, [0 for i in range(n ^ 2)])

    for i in range(n):
        index1 = random.choice(slots1)
        slots1.remove(index1)
        index2 = index1
        while index2 == index1:
            index2 = random.choice(slots2)
        slots2.remove(index2)
        P[i, index1] = 1
        P[i, index2] = 1
    return P


def check_inequality(n, k, r):
    if k * (k + 1) * 0.5 >= n:
        if r == n:
            return True
        else:
```

```
            return False
    else:
        if r == (k * (k + 1) * 0.5):
            return True
        else:
            return False


def make_weight2_numpy(n):
    trace = 1
    permutation = list(range(0, n))
    while trace != 0:
        shuffle(permutation)
        matrix = np.zeros((n, n))  # making zero matrix
        for i in range(n):
            matrix[i][permutation[i]] = 1  # putting 1 (by permutation)
        trace = np.trace(matrix)
    matrix = np.add(matrix, np.eye(n))  # matrix weight 2 with 1 on the
   diagonal
    permutation_rows = list(range(0, n))
    shuffle(permutation_rows)
    for i in range(n):
        matrix[[i, permutation_rows[i]]] = matrix[[permutation_rows[i], i]]
    return matrix


def make_weight1(n, q):
    MS = MatrixSpace(QQ, n)
    permutation = list(range(0, n))
    shuffle(permutation)
    matrix = np.zeros((n, n))  # making zero matrix
    for i in range(n):
        matrix[i][permutation[i]] = 1  # putting 1 (by permutation)
    matrix_new = MS.matrix(matrix)
    return matrix_new.change_ring(GF(q))


def make_weight2(n, q):
    MS = MatrixSpace(QQ, n)
    trace = 1
    permutation = list(range(0, n))
    while trace != 0:
        shuffle(permutation)
        matrix = np.zeros((n, n))  # making zero matrix
        for i in range(n):
            matrix[i][permutation[i]] = 1  # putting 1 (by permutation)
        trace = np.trace(matrix)
    matrix = np.add(matrix, np.eye(n))  # matrix weight 2 with 1 on the
   diagonal
    permutation_rows = list(range(0, n))
    shuffle(permutation_rows)
    for i in range(n):
        matrix[[i, permutation_rows[i]]] = matrix[[permutation_rows[i], i]]
    matrix_new = MS.matrix(matrix)
    return matrix_new.change_ring(GF(q))


```

```
def make_weight3(n, q):
    MS = MatrixSpace(QQ, n)
    trace = 1
    while trace != 0:
        matrix = make_weight2_numpy(n)
        trace = np.trace(matrix)
    matrix = np.add(matrix, np.eye(n))  # matrix weight 3 with 1 on the diagonal
    permutation_rows = list(range(0, n))
    shuffle(permutation_rows)
    for i in range(n):
        matrix[[i, permutation_rows[i]]] = matrix[[permutation_rows[i], i]]
    matrix_new = MS.matrix(matrix)
    return matrix_new.change_ring(GF(q))


def Schur_product(M, k1):
    Schur = []
    for i in range(k1):
        a = np.array(list(M[i]))
        for j in range(i, k1):
            b = np.array(list(M[j]))
            Schur.append(list(np.multiply(a, b)))
    return Schur


def dimrandomcode(n, k):
    if k * (k + 1) / 2 >= n:
        return n
    else:
        return (k * (k + 1)) / 2


def Prob(n, k):
    c1 = 1
    c2 = 1
    c = 1
    if ((k * (k + 1)) / 2 >= n and n >= k):
        return '1 Pr(' + str(n) + ')>=' + str(round(1 - 2 ** ((-c1) * k) - 2 ** ((-c2) * ((k * (k - 1)) / 2 - n)), 10))
    if n >= (k * (k + 1) / 2):
        return '2 Pr(' + str((k * (k + 1)) / 2) + ')>=' + str(round(1 - 2 ** ((-c) * (n - (k * (k - 1)) / 2)), 10))
    else:
        return 'условие не вписывается в теорему ' + 'n=' + str(n) + ' k=' + str(k) + ' (k*(k+1))/2=' + str(
            (k * (k + 1)) / 2)


def codegen(k, q, d, codetype,
            curv=0):  # возвращает матрицу кода, функциональное поле, размерность кодаи, случайную матрицу
    if codetype in [1, 2, 3]:
        if codetype == 1:
            k. < a > = GF(q ^ 2)
            A. < x, y > = AffineSpace(k, 2)
            C = Curve(y ^ q + y - x ^ (q + 1))
```

```
            Q, = C.places_at_infinity()
        if codetype == 2:
            if curv == 0:
                raise Exception('не введена кривая!')
            C = Curve(curv)
            Q, = C.places_at_infinity()
        if codetype == 3:
            A. < x, y > = AffineSpace(k, 2)
            C = Curve(x ^ 3 * y + y ^ 3 + x)
            p = C([0, 0])
            Q, = p.places()

        F = C.function_field()
        pls = F.places()
        pls.remove(Q)
        G = d * Q
        code = codes.EvaluationAGCode(pls, G)
        M = code.generator_matrix()
        k1 = M.rank()
        S = random_matrix(k, k1, k1)  # рандомная матрица
        n = len(M[0])
        while S.is_singular():  # проверка на обратимость
            S = random_matrix(k, k1, k1)
        return M, F, n, k1, S
    else:
        raise Exception('не верный тип кода!')


def SchurCalc(n, k1, M, S):  # вычисление квадратов шура при маскировке весом 1 и 2 и 3
    # Вычисление размерности случайного кода
    random_Schur = dimrandomcode(n, k1)
    print('Размерность квадрата шура случайного кода той же размерности = ', random_Schur)
    print(f'n = {n}, k = {k1}')

    # маскировка 1

    P1 = make_weight1(n, q)  # случайная матрица
    G_cap_hidden1 = Matrix(S * M * P1)  # замаскированная матрица вида 1
    G_cap_Schur1 = Matrix(F, Schur_product(G_cap_hidden1, k1))  # квадрат шура при маскировке 1
    print("\Выводn матрицы шура при маскировке вида 1")
    # print(G_cap_Schur1)
    G1k = G_cap_Schur1.rank()
    print('Размерность квадрата шура маскировка весом 1 -', G1k)
    print(f'Проверка неравенства - {check_inequality(n, k1, G1k)}')

    # маскировка 2

    P = make_weight2(n, q)  # маскировочная матрица веса 2
    G_cap_hidden2 = Matrix(S * M * P)  # замаскированная матрица веса 2
    G_cap_Schur2 = Matrix(F, Schur_product(G_cap_hidden2, k1))  # квадрат шура при маскировке 2
    print("\Выводn матрицы шура при маскировке вида 2")
    # print(G_cap_Schur2)
    G2k = G_cap_Schur2.rank()
```

```
    print('Размерность квадрата шура маскировка весом 2 -', G2k)
    print(f'Проверка неравенства - {check_inequality(n, k1, G2k)}')

    # маскировка 3

    P3 = make_weight3(n, q)  # маскировочная матрица веса 3
    G_cap_hidden3 = Matrix(S * M * P3)  # замаскированная матрица веса 3
    G_cap_Schur3 = Matrix(F, Schur_product(G_cap_hidden3, k1))  # квадрат шура
   при маскировке 3
    print("\Выводn матрицы шура при маскировке вида 3")
    # print(G_cap_Schur3)
    G3k = G_cap_Schur3.rank()
    print('Размерность квадрата шура маскировка весом 2 -', G3k)
    print(f'Проверка неравенства - {check_inequality(n, k1, G3k)}')


# Работа с АГ кодами
# задание поля, в случае эрмитовой учитывать что оно будет возведено в квадрат!
'''Типы
 кодов:
    1 - эрмитов код, не требует задачи кривой
    2 - гиперэллиптический-/ код, требует задания кривой
    3 - код на квартике клейна, не требует задания кривой
    M,F,k1,S=codegenAG(k,q,d,codetype) эрмитова криваяквартика/
    M,F,n,k1,S=codegenAG(k,q,d,codetype,curv) эллиптическая кривая
'''
q = 17
d = 8  # переменная в дивизоре G
k. < a > = GF(q)
A. < x, y > = AffineSpace(k, 2)
codetype = 2
curv = y ^ 2 - x ^ 5 - x ^ 3 - x - 3  # кривая
M, F, n, k1, S = codegen(k, q, d, codetype, curv)
print('Порождающая матрица')
# print(M)
print('Случайная обратимая матрица')
# print(S)
SchurCalc(n, k1, M, S)
print('-------------------------------------------------')
i = 1  # <k+1
j = 9  # <n+1
print(f'Преобразуем порождающую матрицу - вычтем {i + 1} строку, {j + 1} столбец')
M = M.delete_rows([i]).delete_columns([j])
# print(M)
k1 = M.rank()
S = random_matrix(k, k1, k1)  # рандомная матрица
while S.is_singular():  # проверка на обратимость
    S = random_matrix(k, k1, k1)
print('Новая случайна матрица')
# print(S)
SchurCalc(n - 1, k1, M, S)
```