Ссылка на статью:
// Ученые записки УлГУ. Сер. Математика и информационные технологии. 2022, № 2, с. 90-107.
Поступила: 15.11.2022
Окончательный вариант: 26.11.2022

УДК 519.7

О реализации конечных полей характеристики два и об их применении в криптосистемах

Рацеев С. М.[*], ***Убанеева Е. Г.***

[*]ratseevsm@mail.ru
УлГУ, Ульяновск, Россия

В работе исследуется вопрос о программной реализации конечных полей характеристики два. Также рассматривается применение полученных реализаций в криптосистемах на примере схемы разделения секрета Шамира.

Ключевые слова: *конечное поле, схема разделения секрета Шамира.*

Введение

Конечные поля активно используются при построении систем защиты информации. Например, на основе конечных полей $GF(2^8)$ построены симметричные блочные шифры AES и Кузнечик [1, 2], где AES — международный стандарт блочного шифрования ISO/IEC 18033-3:2010, Кузнечик — шифр из российского стандарта ГОСТ Р 34.12-2015. На основе конечных полей строятся коды Рида—Соломона, коды Гоппы [3], перспективная постквантовая криптосистема Мак-Элиса [4] (прошла несколько этапов текущего конкурса NIST постквантовых схем). На основе конечных полей можно строить конструкции совершенных имитостойких шифров и оптимальных кодов аутентификации [5] и многое другое.

Работа носит учебно-методический характер. В данной работе приводится пример реализации конечного поля характеристики два и пример реализации схемы разделения секрета Шамира на основе конечных полей. Данная схема актуальна тем, что является совершенной (не зависит от вычислительных возможностей криптоаналитика), идеальной и т. д. [5].

1. Реализация конечного поля характеристики два

Более подробно о конечных полях и об их применении в теории корректирующих кодов и в криптосистемах можно посмотреть, например, в [3, 5, 6]. Рассмотрим задачу программной реализации конечного поля. Будем использовать язык программирования Си, так как именно на языке Си написано подавляющее число криптографических библиотек (cryptlib, LibreSSL, OpenSSL, wolfCrypt и т. д.).

Предположим, что требуется построить таблицу умножения элементов поля, таблицу степеней образующего элемента поля, таблицу дискретных логарифмов по основанию образующего элемента, таблицу порядков ненулевых элементов поля. Построение конечного поля $GF(q^m)$ начинается с построения неприводимого многочлена над $GF(q)$ степени m. Это можно сделать, например, на основе алгоритма, приведенного в работе [3]. Также можно использовать уже готовые таблицы неприводимых многочленов [6]. В одной из следующих работ будет реализован генератор неприводимых и примитивных многочленов.

В качестве примера рассмотрим конечное поле $GF(2^5)$. Будем считать, что неприводимый над $GF(2)$ многочлен $p(x)$ степени 5 мы уже построили. Пусть $p(x) = x^5 + x^3 + x^2 + x + 1$. Каждый элемент поля $GF(2^5)$ имеет вид $a_4\alpha^4 + a_3\alpha^3 + a_2\alpha^2 + a_1\alpha + a_0$, где $\alpha \in GF(2^5)$ — корень многочлена $p(x)$, все $a_i \in GF(2)$. Поэтому каждый такой элемент можно закодировать двоичной строкой $a_4a_3a_2a_1a_0$ для удобства хранения в ПК. В данном случае $\alpha^5 = \alpha^3+\alpha^2+\alpha+1$, так как $p(\alpha) = 0$. Для многочлена $p(x)$ определим константу $PX = 1111 = 0xf$, которая соответствует элементу поля $\alpha^3+\alpha^2+\alpha+1$, т. е. PX — двоичная запись элемента $\alpha^3+\alpha^2+\alpha+1$ (например, если бы многочлен $p(x)$ был равен $p(x) = x^5 + x^2 + 1$, то $\alpha^5 = \alpha^2 + 1$ и в качестве PX мы бы взяли значение $PX = 101 = 0x5$, что соответствует $\alpha^2 + 1$). Смысл константы PX раскрыт чуть ниже. Для поля $GF(2^5)$ определим следующие константы:

```
#define N 5 // поле GF(2^N)
#define NMAX 31 // число (2^N - 1) - максимальное N-битное число
#define PX 0xf  // соответствует многочлену p(x)
#define ALPHA 2 // 00010 - корень многочлена p(x)
```

Умножение многочленов в поле можно упростить, введя произведение многочлена $b(\alpha) = \sum\limits_{i=0}^{4} b_i\alpha^i \in GF(2^5)$ на α. Так как $\alpha^5 = \alpha^3 + \alpha^2 + \alpha + 1$, то

$$c = \alpha \cdot (b_4\alpha^4 + b_3\alpha^3 + b_2\alpha^2 + b_1\alpha + b_0) =$$

$$= b_4\alpha^5 + b_3\alpha^4 + b_2\alpha^3 + b_1\alpha^2 + b_0\alpha =$$

$$= b_4(\alpha^3 + \alpha^2 + \alpha + 1) + b_3\alpha^4 + b_2\alpha^3 + b_1\alpha^2 + b_0\alpha =$$

$$= b_3\alpha^4 + (b_2 + b_4)\alpha^3 + (b_1 + b_4)\alpha^2 + (b_0 + b_4)\alpha + b_4 =$$

$$= \begin{cases} b_3\alpha^4 + b_2\alpha^3 + b_1\alpha^2 + b_0\alpha, & \text{если } b_4 = 0, \\ b_3\alpha^4 + (b_2 + 1)\alpha^3 + (b_1 + 1)\alpha^2 + (b_0 + 1)\alpha + 1, & \text{если } b_4 = 1. \end{cases}$$

Заметим, что константа PX отвечает именно за случай $b_4 = 1$, когда нужно прибавлять единицы на соответствующих позициях. Итак, умножение вектора $b = (b_4b_3b_2b_1b_0)$ на вектор (00010) (который соответствует элементу α) можно записать следующим образом:

```
if ((b >> 4) & 1)
    c = ((b << 1) ^ 0xf) & 0x1f;
else c = b << 1;
```

где NMAX=0x1f=11111, PX=0xf = 1111 — соответствует элементу $\alpha^3 + \alpha^2 + \alpha + 1 \in GF(2^5)$. Теперь этот прием можно применить при раскрытии скобок

$$a \cdot b = (a_4\alpha^4 + a_3\alpha^3 + a_2\alpha^2 + a_1\alpha + a_0) \cdot (b_4\alpha^4 + b_3\alpha^3 + b_2\alpha^2 + b_1\alpha + b_0).$$

Таким образом, произведение элемента на элемент α и произведение элементов для данного поля будут соответственно выглядеть следующим образом.

```
typedef unsigned char uint8;
uint8 Product_alpha(uint8 a)
{
    if ((a >> (N - 1)) & 1)
         return ((a << 1) ^ PX) & NMAX;
    else return a << 1;
}

uint8 Product(uint8 a, uint8 b)
{
    uint8 deg = a, rez = 0;
    int i;
    if (b & 1)
         rez = a;
    for(i = 1; i < N; i++)
    {
        if ((deg >> (N - 1)) & 1)
            deg = (deg << 1) ^ PX;
        else deg <<= 1;
        if ((b >> i) & 1)
            rez ^= deg;
    }
    return rez & NMAX;
}
```

Замечание 1. Заметим, что функции Product_alpha() и Product() можно записать более естественным образом без определения константы PX, определив при этом константу P, отвечающую за многочлен $p(x)$:

```
#define P 0x2f  // 101111 - представление многочлена p(x)
// Произведение элементов a и alpha
uint8 Product_alpha(uint8 a)
{
    uint8 c = a;
    c <<= 1;
    if ((c >> N) & 1)
         c ^= P;
    return c;
}

// Произведение элементов поля a и b
uint8 Product(uint8 a, uint8 b)
{
    uint8 buf, // для промежуточных вычислений
          rez; // результат произведения
    buf = b;
    rez = 0;
    while(a)
    {
        if (a & 1)
            rez ^= buf;
        buf <<= 1;
        if ((buf >> N) & 1)
            buf ^= P;
        a >>= 1;
    }
    return rez;
}
```

Только в этом случае нужно контролировать разрядность переменных c и buf, так как, например, проверка

```
    if ((c >> N) & 1)
```

осуществляет проверку N-го разряда. Поэтому если, например, $N = 8$, то переменные c и buf должны быть 16-разрядными.

Для построения таблицы умножения ненулевых элементов поля $GF(2^5)$ можно использовать двойной цикл:

```
    for(a = 1; a <= NMAX; a++)
        for(b = 1; b <= NMAX; b++)
            prod = Product(a, b);
```

При этом понятно, что можно учесть коммутативность операции умножения и уменьшить число итераций.

Далее найдем хотя бы один образующий элемент поля. Для этого применим следующий критерий. Пусть $GF(q)$ — конечное поле, $q-1 = p_1^{k_1} p_2^{k_2} \dots p_n^{k_n}$ — каноническое разложение числа $q-1$. Элемент $g \in GF(q)$ является образующим тогда и только тогда, когда

$$g^{\frac{q-1}{p_i}} \neq 1, \quad i = 1, \dots, n.$$

В ходе поиска образующего элемента при возведении элемента поля в степень можно использовать алгоритм бинарного возведения в степень.

```
// Бинарное возведение в степень
uint8 Pow(uint8 a, unsigned long n)
{
   uint8 deg = a,  // степени числа a
         rez = 1;  // результат возведения в степень
   while (n)
   {
      if (n & 1)
         rez = Product(rez, deg);
      deg = Product(deg, deg);
      n >>= 1;
   }
   return rez;
}
```

Так как число $2^5 - 1$ простое, то в нашем случае все элементы поля $GF(2^5)$, отличные от нуля и единицы, будут образующими элементами. В другом случае мы бы фиксировали (например, случайным образом) элемент поля g и вычисляли $Pow(g, (q-1)/p_i)$, $i = 1, \dots, n$.

В качестве образующего элемента поля $GF(2^5)$ возьмем элемент $\alpha = 00010$. Так как α — корень многочлена $p(x)$, то этот многочлен называется примитивным многочленом.

Степени образующего элемента $g = \alpha$ запишем в массив deg_g:

```
uint8 deg_g[NMAX];
g = ALPHA;
deg_g[0] = 1;
for (i = 1; i <= NMAX; i++)
    deg_g[i] = Product(deg_g[i-1], g);
```

При этом попутно найдем таблицу дискретных логарифмов.

Для нахождения порядков элементов (хотя мы уже знаем порядки всех элементов с учетом простоты числа $2^5 - 1$) можно воспользоваться следующей формулой. Пусть $a = g^k$. Тогда

$$\operatorname{ord} a = \frac{2^5 - 1}{\text{НОД}(2^5 - 1, k)}.$$

Но это если известен дискретный логарифм элемента a по основанию g. Если дискретные логарифмы неизвестны, то можно воспользоваться быстрым алгоритмом из работы [3]. Сложение элементов поля $GF(2^5)$ является поразрядным сложением по модулю 2 двоичных векторов длины 5.

Сведем все это в программу.

```
#include<stdio.h>

// Параметры конечного поля
#define N 5 // поле GF(2^N)
#define NMAX 31 // число (2^N - 1) - максимальное N-битное число
#define PX 0xf  // соответствует многочлену p(x)
#define ALPHA 2 // 00010 - корень многочлена p(x)

typedef unsigned char uint8;

// Произведение элемента a на alpha
uint8 Product_alpha(uint8 a)
{
    if ((a >> (N - 1)) & 1)
         return ((a << 1) ^ PX) & NMAX;
    else return a << 1;
}

// Произведение элементов поля a и b
uint8 Product(uint8 a, uint8 b)
{
    uint8 deg = a, rez = 0;
    int i;
    if (b & 1)
         rez = a;
    for(i = 1; i < N; i++)
    {
        if ((deg >> (N - 1)) & 1)
            deg = (deg << 1) ^ PX;
        else deg <<= 1;
        if ((b >> i) & 1)
            rez ^= deg;
    }
    return rez & NMAX;
}
```

```
// двоичное представление целого числа в виде строки
void Itoa2(uint8 a, char *s)
{
    int i, j, bit;
    for (i = N - 1, j = 0; i >= 0; i--, j++)
    {
       /* выделяем i-й справа бит */
       bit = (a >> i) & 1;
       s[j] = '0' + bit;
    }
    s[j] = '\0';
}
// Наибольший общий делитель
unsigned long Nod(unsigned long a, unsigned long b)
{
    while (a && b)
    {
        if (a >= b)
           a %= b;
        else b %= a;
    }
    return a | b;
}
int main()
{
    uint8 a, b, prod;
    int i, j, n;
    char s[N+1]; // для отображения числа в двоичном виде строкой
    uint8 deg_g[NMAX + 1]; // степени образующего элемента
    uint8 log[NMAX + 1];  // таблица дискретных логарифмов
    uint8 ord[NMAX + 1]; // таблица порядков ненулевых элементов поля
    uint8 g; // образующий элемент поля

    // Произведение ненулевых элементов от 0...01 до 1...1
    puts("multiplication table:");
    for(a = 1; a <= NMAX; a++)
    {
        for(b = 1; b <= NMAX; b++)
        {
            prod = Product(a, b);
            Itoa2(prod, s);
```

```
            printf("%s ", s);
        }
        puts("");
    }
    g = ALPHA; // так как ALPHA оказался образующим элементом поля
    // Вычисление степеней образующего элемента g
    // и построение таблицы дискретных логарифмов
    deg_g[0] = 1;
    for (i = 1; i <= NMAX; i++)
    {
        deg_g[i] = Product(deg_g[i-1], g);
        log[deg_g[i]] = i;
    }
    // Находим порядки ненулевых элементов поля
    for (i = 1; i <= NMAX; i++)
        ord[i] = NMAX / Nod(NMAX, log[i]);
    // Выводим степени образующего элемента
    puts("degrees:");
    for (i = 0; i < NMAX; i++)
    {
        Itoa2(deg_g[i], s);
        printf("g^%d = %s\n", i, s);
    }
    // Выводим дискретные логарифмы элементов
    puts("logarithms:");
    for (i = 1; i <= NMAX; i++)
    {
        Itoa2(i, s);
        printf("log_g (%s) = %d\n", s, log[i]);
    }
    // Выводим порядки элементов
    puts("orders:");
    for (i = 1; i <= NMAX; i++)
    {
        Itoa2(i, s);
        printf("ord (%s) = %d\n", s, ord[i]);
    }
    // Выводим образующие элементы поля
    puts("primitive elements:");
    for (i = 1; i <= NMAX; i++)
        if (ord[i] == NMAX)
```

```
        {
            Itoa2(i, s);
            printf("%s\n", s);
        }
    return 0;
}
```

В результате, в частности, получим таблицу степеней образующего элемента поля $GF(2^5)$:

$$\begin{array}{llll} g^0 = 00001, & g^1 = 00010, & g^2 = 00100, & g^3 = 01000, \\ g^4 = 10000, & g^5 = 01111, & g^6 = 11110, & g^7 = 10011, \\ g^8 = 01001, & g^9 = 10010, & g^{10} = 01011, & g^{11} = 10110, \\ g^{12} = 00011, & g^{13} = 00110, & g^{14} = 01100, & g^{15} = 11000, \\ g^{16} = 11111, & g^{17} = 10001, & g^{18} = 01101, & g^{19} = 11010, \\ g^{20} = 11011, & g^{21} = 11001, & g^{22} = 11101, & g^{23} = 10101, \\ g^{24} = 00101, & g^{25} = 01010, & g^{26} = 10100, & g^{27} = 00111, \\ g^{28} = 01110, & g^{29} = 11100, & g^{30} = 10111, & g^{31} = 00001. \end{array}$$

Таблица степеней и таблица дискретных логарифмов облегчают нахождение произведений, степеней, частных, обратных и т.д. Пусть $a, b \in GF(2^5)$, $a \neq 0$, $n \in \mathbb{N}$. Тогда для нахождения, например, значений $c = a \cdot b$, $d = a^n$, a^{-1} в поле $GF(2^5)$, учитывая равенства

$$a \cdot b = g^{\log_g a \cdot b} = g^{\log_g a + \log_g b},$$

$$a^n = g^{\log_g a^n} = g^{n \cdot \log_g a},$$

$$a^{-1} = g^{2^5 - 1 - \log_g a},$$

достаточно записать

```
c = deg_g[(log[a] + log[b]) % NMAX];
d = deg_g[(n * log[a]) % NMAX];
deg_g[NMAX - log[a]];
```

При этом, учитывая, что порядок элемента g равен 2^5-1, логарифмы вычисляются по модулю $2^5 - 1 = NMAX$.

Заметим, что в приведенном примере параметры поля задаются только в константах. Например, если необходимо построить поле $GF(2^8)$ на основе неприводимого многочлена $p(x) = x^8 + x^4 + x^3 + x^2 + 1$, то константы примут следующий вид:

```
#define N 8 // поле GF(2^N)
#define NMAX 255 // число (2^N - 1) - максимальное N-битное число
#define PX 0x1d  // соответствует многочлену p(x)
#define ALPHA 2 // корень многочлена p(x)
```

Заметим, что элемент α, являющийся корнем многочлена $p(x)$, не обязательно будет образующим элементом поля (в данном случае будет образующим элементом). Поэтому сначала нужно найти образующий элемент поля, а потом строить таблицы степеней и дискретных логарифмов. Также нужно следить, чтобы значение N было не больше разрядности используемого типа данных. Например, при использовании типа unsigned char выполнено $N \leq 8$.

2. Нахождение обратного элемента при отсутствии таблицы дискретных логарифмов

Предположим, что при реализации конечного поля нет таблицы дискретных логарифмов. Тогда для вычисления обратного к ненулевому элементу $a = (a_4 \dots a_1 a_0)$ поля можно применить обобщенный алгоритм Евклида для пары многочленов $p(x)$ и $a(x) = a_4x^4 + \dots + a_1x + a_0$ над полем $GF(2)$. Так как $p(x)$ неприводим над $GF(2)$, то найдутся такие многочлены $u(x)$ и $v(x)$ над $GF(2)$, для которых

$$p(x)u(x) + a(x)v(x) = 1, \quad \deg v(x) < \deg p(x) = 5.$$

Подставив вместо переменной x значение α, получим $a(\alpha)v(\alpha) = 1$, так как $p(\alpha) = 0$. Это значит, что $v(\alpha)$ — обратный к a. Ниже приводится реализация вычисления обратного элемента на основе обобщенного алгоритма Евклида. За это отвечает функция Inv(). Для реализации данной функции необходимо реализовать функцию умножения многочленов и функцию вычисления частного и остатка от деления многочленов над $GF(2)$.

```
// Произведение многочленов a(x) и b(x) над GF(2)
uint8 Mult(uint8 a, uint8 b)
{
    uint8 rez = 0;
    while(b)
    {
        if (b & 1)
            rez ^= a;
        a <<= 1;
        b >>= 1;
    }
    return rez;
}
// Степень многочлена a(x)
int Deg(uint8 a)
{
    int deg;
    if (a == 0)
```

```
        return -1;
    for(deg = sizeof(a)*8 - 1; !((a >> deg) & 1); deg--)
        ;
    return deg;
}

// Нахождение частного и остатка от деления многочлена a(x) на b(x),
// a(x) = b(x)q(x) + r(x), deg r(x) < deg b(x).
// Функция возвращает остаток от деления, частное записывается в *q.
uint8 Mod(uint8 a, uint8 b, uint8 *q)
{
    int n, m;
    n = Deg(a);
    m = Deg(b);
    *q = 0;
    while(n >= m)
    {
        (*q) ^= 1 << (n - m);
        a ^= b << (n - m);
        n = Deg(a);
    }
    return a;
}

#define P 0x2f // соответствует многочлену p(x)
// Нахождение обратного к a
uint8 Inv(uint8 a)
{
    uint8 y0, y1, q, r, b, buf;
    b = a; a = P;
    y0 = 0; y1 = 1;
    r = Mod(a, b, &q);
    while (r)
    {
        a = b; b = r;
        buf = y0 ^ Mult(q, y1);
        y0 = y1; y1 = buf;
        r = Mod(a, b, &q);
    }
    return y1;
}
```

Теперь $Inv(a)$ — обратный к ненулевому элементу a. Заметим, что при реализации функции Mult() нужно учитывать, что разрядность возвращаемого значения должна быть в два раза больше разрядности полученных аргументов. Здесь разрядности совпадают, так как эта функция привязана к вычислению обратных элементов.

3. Реализация схемы разделения секрета Шамира над конечным полем характеристики два

Пусть имеются n участников $A_1,\ldots,A_n$, между которыми требуется разделить некоторый секрет S, и некоторый выделенный участник D, называемый *дилером*, который разделяет секрет S. Пусть также t — некоторое натуральное число, причем $1 \le t \le n$, $\{\alpha_1,\ldots,\alpha_n\}$ — некоторая информация о секрете S, и выполнены следующие условия: 1) каждый участник A_i знает некоторую информацию α_i, которая неизвестна остальным $n-1$ участникам; 2) секрет S легко может быть вычислен по произвольному t-элементному подмножеству множества $\{\alpha_1,\ldots,\alpha_n\}$; 3) секрет S нельзя вычислить ни по какому $(t-1)$-элементному подмножеству в $\{\alpha_1,\ldots,\alpha_n\}$. В этом случае множество $\{\alpha_1,\ldots,\alpha_n\}$ со свойствами 1–3 называется (n,t)-*пороговой схемой* для секрета S. Пороговую схему разделения секрета назовем *совершенной*, если наличие любых m долей секрета, $m < t$, не дает никакой информации о секрете.

Одной из самых популярных (n,t)-пороговых схем разделения секрета является схема Шамира [5]. Данная схема обладает свойством идеальности (число битов, содержащихся в каждой доле секрета, равно числу битов, содержащихся в самом секрете), совершенности, расширяемости (число владельцев долей секрета n может быть в любой момент увеличено, при этом количество частей секрета t, необходимых для восстановления секрета, останется неизменным) и т. д.

Предположим, что в роли секрета выступает некоторый (произвольный) файл (текстовый, графический и т. д.). Хорошо известно, что любой файл рассматривается как последовательность байт, оканчивающаяся маркером конца файла. В этом случае файл разбивается на последовательность S_1, S_2,..., S_l, где все $S_i \in GF(2^{8k})$ для некоторого фиксированного натурального k. Например, при $k = 1$ файл разбивается на последовательность байт, при $k = 2$ — на последовательность блоков по два байта и т. д. Очень часто при использовании схемы разделения секрета Шамира в этом случае выбирается простое число $p > \max\{2^{8k}, n\}$ и при разделении секретов S_i на части все операции проходят в поле $GF(p)$ (кольцо вычетов по простому модулю p). Поэтому при разделении секретов S_i на n участников при заданном пороге t будут получаться доли секрета, принадлежащие полю $GF(p)$. Так как $p > 2^{8k}$, то для долей секрета S_i понадобится больший объем памяти, чем для самого секрета S_i. Это противоречит свойству идеальности схемы Шамира. Выходом из данной ситуации является использование конечных полей вида $GF(2^{8k})$. В этом случае свойство идеальности будет выполнено.

Проиллюстрируем это на примере. Только в нашем случае будет использоваться ранее построенное поле $GF(2^5)$.

Предположим, что нужно разделить секрет $S = g^{10} = 01011$ на $n = 8$ участников с порогом $t = 5$.

Дилер D, генерируя (равновероятно) коэффициенты a_1, a_2, a_3, a_4 многочлена $L(x) = S + a_1x + \ldots + a_4x^4$, например, получает

$$L(x) = g^{10} + g^4x + g^{11}x^2 + g^5x^3 + g^7x^4.$$

В качестве x_i будем брать номера участников разделения секрета (от 1 до 8), представленные двоичными векторами длины 5 — элементы поля $GF(2^5)$. Далее происходит вычисление долей секрета, которые запишем в массив y. При этом в массиве a будем хранить коэффициенты многочлена Лагранжа.

```
uint8 y[8], a[5];
a[0] = S = deg_g[10];
a[1] = deg_g[4];
a[2] = deg_g[11];
a[3] = deg_g[5];
a[4] = deg_g[7];
for(i = 0; i < 8; i++)
{
    y[i] = S;
    degx = 1;
    for(j = 1; j < 5; j++)
    {
        degx = Product(degx, i+1);
        y[i] ^= Product(a[j], degx);
    }
}
```

Учитывая таблицу дискретных логарифмов, вычисление долей секрета можно записать в следующем виде:

```
for(i = 0; i < 8; i++)
{
    y[i] = S;
    degx = 1;
    for(j = 1; j < 5; j++)
    {
        degx = deg_g[(log[degx] + log[i+1]) % NMAX];
        if (a[j] != 0)
            y[i] ^= deg_g[(log[a[j]] + log[degx]) % NMAX];
    }
}
```

Тогда

$$y_1 = L(00001) = g^{17} = 10001, \quad y_2 = L(00010) = g^{22} = 11101,$$
$$y_3 = L(00011) = g^{25} = 01010, \quad y_4 = L(00100) = g^{12} = 00011,$$
$$y_5 = L(00101) = g^{15} = 11000, \quad y_6 = L(00110) = g^{14} = 01100,$$
$$y_7 = L(00111) = g^{19} = 11010, \quad y_8 = L(01000) = g^{16} = 11111.$$

Видно, что размеры долей секрета совпадают с размером самого секрета, поэтому свойство идеальности выполнено. Доли секрета передаются участникам.

Предположим, что собрались вместе участники с номерами 1, 3, 4, 5, 7 и собрали свои доли секрета вместе:

$$(x_1 = 00001,\ z_1 = 10001),\ (x_2 = 00011,\ z_2 = 01010),\ (x_3 = 00100,\ z_3 = 00011),$$

$$(x_4 = 00101,\ z_4 = 11000),\ (x_5 = 00111,\ z_5 = 11010).$$

Для восстановления секрета достаточно воспользоваться формулой:

$$S = L(0) = \sum_{i=1}^{t} z_i \prod_{\substack{1 \le j \le t \\ j \ne i}} \frac{x_j}{x_j - x_i}.$$

Поэтому процесс восстановления секрета имеет следующий вид:

```
uint8 x[5], z[5], prod, b, S;
x[0] = 1; x[1] = 3; x[2] = 4; x[3] = 5; x[4] = 7;
z[0] = y[0]; z[1] = y[2]; z[2] = y[3]; z[3] = y[4]; z[4] = y[6];
S = 0;
for(i = 0; i < 5; i++)
{
    prod = 1;
    for(j = 0; j < 5; j++)
    {
        if (j != i)
        {
            // b = x_j/(x_j-x_i)
            b = Product(x[j], Inv(x[j] ^ x[i]));
            prod = Product(prod, b);
        }
    }
    S ^= Product(z[i], prod);
}
```

Так как у нас имеется таблица дискретных логарифмов, то в программе для нахождения обратного к $x_j - x_i$ достаточно написать

```
deg_g[NMAX - log[x[j] ^ x[i]]];
```

Произведение элементов тоже можно находить с использованием дискретных логарифмов. Поэтому предыдущий фрагмент кода программы можно заменить на следующий:

```
S = 0;
for(i = 0; i < 5; i++)
{
    prod = 1;
    if (z[i] != 0)
    {
        for(j = 0; j < 5; j++)
        {
            if (j != i)
            {
                // b = x_j/(x_j-x_i)
                b = deg_g[ (log[x[j]] + NMAX - log[x[j] ^ x[i]]) % NMAX];
                prod = deg_g[(log[prod] + log[b]) % NMAX];
            }
        }
        S ^= deg_g[(log[z[i]] + log[prod]) % NMAX];
    }
}
```

Создадим библиотечный файл "field.h" и запишем туда функции Product_alpha(), Product(), Itoa2() из реализации первого параграфа, а также константы N, NMAX, PX, ALPHA, включая определение типа typedef. Тогда программа разделения и восстановления секрета с использованием дискретных логарифмов примет следующий вид.

```
#include<stdio.h>
#include"field.h"
int main()
{
    uint8 b, prod;
    int i, j;
    char s[N+1], t[N+1]; // для отображения числа в двоичном виде строкой
    uint8 deg_g[NMAX + 1]; // степени образующего элемента
    uint8 log[NMAX + 1];  // таблица дискретных логарифмов
    uint8 degx;
    uint8 S; // секрет
    uint8 y[8], // доли всех участников
          x[5], // номера собравшихся участников
          z[5], // доли собравшихся участников
          a[5]; // коэффициенты многочлена Лагранжа
    uint8 g; // образующий элемент поля
```

```
g = ALPHA;
// Вычисление степеней образующего элемента g и
// построение таблицы дискретных логарифмов
deg_g[0] = 1;
for (i = 1; i <= NMAX; i++)
{
    deg_g[i] = Product(deg_g[i-1], g);
    log[deg_g[i]] = i;
}

// Пусть значение секрета S равно следующему значению:
S = deg_g[10];

// Дилер генерирует случайным (равновероятным) образом
// коэффициенты a[1],...,a[4] многочлена Лагранжа, например,
a[0] = S;
a[1] = deg_g[4];
a[2] = deg_g[11];
a[3] = deg_g[5];
a[4] = deg_g[7];

// Нахождение долей секрета с использованием логарифмов
for(i = 0; i < 8; i++)
{
    y[i] = S;
    degx = 1;
    for(j = 1; j < 5; j++)
    {
        degx = deg_g[(log[degx] + log[i+1]) % NMAX];
        if (a[j] != 0)
            y[i] ^= deg_g[(log[a[j]] + log[degx]) % NMAX];
    }
    Itoa2(i+1, t);
    Itoa2(y[i], s);
    printf("L(%s) = %s\n", t, s);
}

// Предположим, что собрались вместе участники с номерами 1, 3, 4, 5, 7
x[0] = 1; x[1] = 3; x[2] = 4; x[3] = 5; x[4] = 7;
z[0] = y[0]; z[1] = y[2]; z[2] = y[3]; z[3] = y[4]; z[4] = y[6];
// Восстановление секрета с использованием логарифмов
S = 0;
for(i = 0; i < 5; i++)
```

```
    {
        prod = 1;
        if (z[i] != 0)
        {
            for(j = 0; j < 5; j++)
            {
                if (j != i)
                {
                    // b = x_j/(x_j-x_i)
                    b = deg_g[ (log[x[j]] + NMAX - log[x[j] ^ x[i]]) % NMAX];
                    prod = deg_g[(log[prod] + log[b]) % NMAX];
                }
            }
            S ^= deg_g[(log[z[i]] + log[prod]) % NMAX];
        }
    }
    Itoa2(S, t);
    printf("S = %s\n", t);
    return 0;
}
```

Заключение

Конечные поля характеристики два часто применяются в системах защиты информации. В работе приведен пример реализации конечного поля, показано, что таблица дискретных логарифмов играет важную роль при вычислении произведений, частных, степеней и т.д. элементов поля. Также приводится пример реализации схемы разделения секрета на основе поля характеристики два с сохранением свойства идеальности данной схемы при реализации на ПК.

Список литературы

1. *ГОСТ Р 34.12-2015. Информационная технология. Криптографическая защита информации. Блочные шифры.* М.: Стандартинформ, 2016.
2. *ГОСТ Р 34.12-2018. Информационная технология. Криптографическая защита информации. Блочные шифры.* М.: Стандартинформ, 2018.
3. Рацеев С. М. *Элементы высшей алгебры и теории кодирования*: учебное пособие для вузов. СПб.: Лань, 2022. 656 с.
4. Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process. National Institute of Standards and Technology

Interagency or Internal Report. NIST IR 8413-upd1. July 2022, 102 pages. https://csrc.nist.gov/publications/detail/nistir/8413/final.

5. Рацеев С. М. *Математические методы защиты информации*: учебное пособие для вузов. СПб.: Лань, 2022. 544 с.

6. Лидл Р., Нидеррайтер Г. *Конечные поля*: в 2-х томах / пер. с англ. М.: Мир, 1988. Т. 1. 430 с.

On the implementation of finite fields of characteristic two and their application in cryptosystems

Ratseev, S. M.**, *Ubaneeva, E. G.

*ratseevsm@mail.ru

Ulyanovsk State University, Ulyanovsk, Russia

In the paper the issue of software implementation of finite fields of characteristic two is investigated. Implementations in cryptosystems is also investigated using the example of Shamir's secret sharing scheme.

Keywords: *finite field, Shamir secret sharing scheme.*