Ссылка на статью:
// Ученые записки УлГУ. Сер. Математика и информационные технологии. 2022, № 2, с. 75-89.
Поступила: 18.11.2022
Окончательный вариант: 18.11.2022

УДК 519.7

Эффективная реализация структур доступа для схем разделения секрета

Рацеев С. М.

ratseevsm@mail.ru
УлГУ, Ульяновск, Россия

В работе приводится эффективное построение множества правомочных коалиций, множества неправомочных коалиций, множества минимальных правомочных коалиций, множества максимальных неправомочных коалиций для структур доступа для схем разделения секрета, если хотя бы одно их перечисленных множеств известно.

Ключевые слова: *схема разделения секрета, структура доступа.*

Введение

Учитывая словарь криптографических терминов [1], пусть P — конечное множество участников разделения секрета, $\widetilde{P}$ — множество, состоящее из всех возможных непустых подмножеств множества P, R — множество, состоящее из подмножеств участников, которым разрешено восстановление секрета (правомочные коалиции), Z — множество, состоящее из подмножеств участников, которые не могут восстановить секрет (неправомочные коалиции). *Структура доступа* — разбиение $\widetilde{P} = R \cup Z$. Структура доступа обозначается как (R, Z). Далее всегда будем считать, что $R \neq \varnothing$.

Структура доступа называется *монотонной*, если все надмножества правомочных коалиций также входят в R, то есть если $A \in R$, $A \subseteq B \in \widetilde{P}$, то $B \in R$. В монотонной структуре доступа множество P всегда является правомочной коалицией. Далее будем работать именно с такими структурами доступа.

Пусть (R, Z) — структура доступа на P. $A \in R$ называют минимальной правомочной коалицией, если $B \notin R$ всегда, когда выполнено строгое включение $B \subset A$. Множество минимальных правомочных коалиций из R обозначается как $R_{\min}$ и называется базисом R. $A \in Z$ называют максимальной неправомочной коалицией, если $B \in R$ всегда, когда выполнено строгое включение $A \subset B$. Множество максимальных неправомочных коалиций из Z обозначается как $Z_{\max}$.

Замечание 1.

1. Множество $R_{\min}$ однозначно задает структуру доступа.
2. Множество $Z_{\max}$ однозначно задает структуру доступа.
3. Для любого $B \in Z$ найдется такое $A \in R$, для которого $B \subset A$.
4. Для любого $B \in Z$ найдется такое $\widetilde{B} \in Z_{\max}$, для которого $B \subseteq \widetilde{B}$.

Схема разделения секрета, в которой доли секрета любой неправомочной коалиции не позволяют получить никакой информации о значении секрета, называется *совершенной*.

Заметим, что любая криптосистема со свойством совершенности не зависит от вычислительных возможностей криптоаналитика. Такими криптосистемами, например, являются совершенные шифры и совершенные схемы разделения секрета (см., напр., [2]). Многими странами ведется поиск постквантовых схем, т. е. схем, не зависящих от квантовых вычислений [3,4]. При этом любая совершенная криптосистема, в частности, является постквантовой.

Одной из интересных схем разделения секрета с произвольной структурой доступа является схема Ито—Саито—Нишизеки [2], которая является совершенной, если составная часть этой схемы, а именно, (m, m)-схема является совершенной, где m — число максимальных неправомочных коалиций. При этом обеспечить совершенность (m, m)-схемы достаточно просто.

Работа носит учебно-методический характер. В ней приводится эффективная программная реализация на языке Си построения множеств R_{min}, Z_{max}, R, Z, если хотя бы одно из этих множеств известно (т. е. задано). В дополнение к этому строится кумулятивный массив C для схемы Ито—Саито—Нишизеки. Так как указанные множества являются подмножествами в $\widetilde{P}$, то они занимают очень много места в памяти компьютера. Поэтому эту схему можно реализовать на обычном ПК для достаточно небольшого числа участников, например, при $|P| \leq 27$.

1. Реализация структур доступа для схем разделения секрета

Пусть $P = \{P_0, P_1, \ldots, P_{n-1}\}$ — участники разделения секрета. Для реализации на ПК отождествим участников с их номерами (от 0 до $n-1$), т. е. $P = \{0, 1, \ldots, n-1\}$.

Коалицию участников $A = \{i_1, \ldots, i_k\}$, где $n-1 \geq i_1 > \ldots > i_k \geq 0$, отождествим с двоичным вектором $a = (a_{n-1}, \ldots, a_1, a_0)$ длины n, причем $j \in A$ тогда и только тогда, когда

j-й справа бит в a равен 1. В этом случае в векторе a единицы стоят на позициях с номерами $i_1, \ldots, i_k$, а на остальных позициях стоят нули.

Пусть $A, B \in \widetilde{P}$, двоичные векторы a и b соответствуют коалициям A и B соответственно. В этом случае очень легко проверить, является ли множество A подмножеством в B:

$$A \subseteq B \;\Leftrightarrow\; a \,|\, b = b,$$

$$A \subseteq B \;\Leftrightarrow\; a \,\&\, b = a,$$

где $|$ — поразрядная операция «ИЛИ», $\&$ — поразрядная операция «И». Также легко проверить, принадлежит ли участник $i \in P$ коалиции A: $i \in A$ тогда и только тогда, когда i-й справа бит в векторе a равен единице, что равносильно проверке

```
if ((a >> i) & 1)
```

Для нахождения множества всех максимальных неправомочных коалиций Z_{max} на основе множества неправомочных коалиций Z потребуется умение добавлять участников к очередной проверяемой коалиции $B \in Z$. Пусть $j \notin B$. Получить коалицию X при помощи добавления участника j к коалиции B можно следующим образом:

```
if (((b >> j) & 1) == 0)
    x = b | (1 << j);
```

Для нахождения множества всех минимальных правомочных коалиций R_{min} на основе множества правомочных коалиций R потребуется умение удалять участников из очередной проверяемой коалиции $A \in R$. Пусть $j \in A$. Получить коалицию X при помощи удаления участника j из коалиции A можно следующим образом:

```
if ((a >> j) & 1)
    x = a & (~(1 << j));
```

Представленная ниже программа содержит следующие функции.

Логическая функция

```
int Binary_search(const uint32 *A, const uint32 na, const uint32 x)
```

принимает на вход упорядоченное множество коалиций A размера na и с помощью бинарного поиска определяет, содержится ли коалиция x в множестве A.

Функция

```
void R_Rmin(uint32 *R, uint32 *nr, const uint32 *R_min, const uint32 nr_min)
```

принимает на вход множество всех минимальных правомочных коалиций R_{min} размера nr_{min} и на основе этого множества находит множество всех правомочных коалиций R, записывая в $*nr$ мощность множества R. При этом множество R строится таким образом, чтобы его элементы (коалиции) были упорядочены лексикографически вне зависимости от упорядоченности принятого множества R_{min}.

Функция

```
void Z_Zmax(uint32 *Z, uint32 *nz, const uint32 *Z_max, const uint32 nz_max)
```

принимает на вход множество всех максимальных неправомочных коалиций Z_{max} размера nz_{max} и на основе этого множества находит множество всех неправомочных коалиций Z, записывая в $*nz$ мощность множества Z. При этом множество Z строится таким образом, чтобы его элементы (коалиции) были упорядочены лексикографически вне зависимости от упорядоченности принятого множества Z_{max}.

Функция

```
void Z_R(uint32 *Z, uint32 *nz, const uint32 *R, const uint32 nr)
```

принимает на вход лексикографически упорядоченное множество всех правомочных коалиций R размера nr и на основе этого множества находит лексикографически упорядоченное множество всех неправомочных коалиций Z, записывая в $*nz$ мощность множества Z.

Функция

```
void R_Z(uint32 *R, uint32 *nr, const uint32 *Z, const uint32 nz)
```

принимает на вход лексикографически упорядоченное множество всех неправомочных коалиций Z размера nz и на основе этого множества находит лексикографически упорядоченное множество всех правомочных коалиций R, записывая в $*nr$ мощность множества R.

Функция

```
void Rmin_R(uint32 *R_min, uint32 *nr_min, const uint32 *R, const uint32 nr)
```

принимает на вход множество всех правомочных коалиций R размера nr и на основе этого множества находит множество всех минимальных правомочных коалиций R_{min}, записывая в $*nr_{min}$ мощность множества R_{min}. Если принимаемое на вход множество R было лексикографически упорядоченное, то множество R_{min} тоже будет лексикографически упорядоченным.

Функция

```
void Zmax_Z(uint32 *Z_max, uint32 *nz_max, const uint32 *Z, const uint32 nz)
```

принимает на вход множество всех неправомочных коалиций Z размера nz и на основе этого множества находит множество всех максимальных неправомочных коалиций Z_{max}, записывая в $*nz_{max}$ мощность множества Z_{max}. Если принимаемое на вход множество Z было лексикографически упорядоченное, то множество Z_{max} тоже будет лексикографически упорядоченным.

Функция

```
uint8 **Allocate(const uint32 m)
```

выделяет место в памяти для динамического кумулятивного массива C размера $N \times m$, где $m = nz_{max}$.

Функция

```
void Init_C(uint8 **C, const uint32 *Z_max, const uint32 m)
```

заполняет кумулятивный массив C на основе принятого множества максимальных неправомочных коалиций Z_{max} размера $m = nz_{max}$.

Функция

```
void Print_C(uint8 **C, const uint32 m)
```

выводит кумулятивный массив на экран.

Функция

```
void Print(const uint32 *A, const uint32 m)
```

выводит все коалиции множества коалиций A на экран.

В качестве начальных значений программы нужно задать число участников N, например,

```
#define N 7 // число участников
#define COUNT 127 // 2^7-1 - число всех непустых коалиций
```

и одно из четырех множеств R, Z, R_{min}, Z_{max}. Все остальные множества (при необходимости) можно построить на основе представленных выше функций. При этом заметим, что если на вход программе передается множество R_{min}, то на самом деле на вход можно подать произвольное множество $\widetilde{R}$ с условием $R_{min} \subseteq \widetilde{R} \subseteq R$, а уже на основе множества $\widetilde{R}$ построить с помощью функции $R\_Rmin()$ множество R, а на основе функции $Rmin\_R()$ построить множество R_{min}. Аналогично и с множеством Z_{max}, т. е. на вход можно подать произвольное множество $\widetilde{Z}$ с условием $Z_{max} \subseteq \widetilde{Z} \subseteq Z$, а уже на основе множества $\widetilde{Z}$ построить с помощью функции $Z\_Zmax()$ множество Z, а на основе функции $Zmax\_Z()$ построить множество Z_{max}.

В программе множества R, Z, R_{min}, Z_{max} определены в виде динамических массивов:

```
uint32 *R, *Z, *R_min, *Z_max;
```

2. Программная реализация структур доступа

Пусть, например, число участников равно $N = 7$ и структура доступа задается следующими минимальными правомочными коалициями:

$$\{0, 1, 5\}, \quad \{2, 3, 5\}, \quad \{2, 4\}, \quad \{4, 6\}.$$

Данные множества соответствуют двоичным векторам 0100011, 0101100, 0010100, 1010000 соответственно. В шестнадцатеричном виде эти двоичные векторы соответственно равны 0x23, 0x2c, 0x14, 0x50. Ниже представлена программа построения множеств R, Z, Z_{max} и кумулятивного массива на основе множества R_{min}.

```
#include <stdio.h>
#include <stdlib.h>

#define N 7 // число участников
#define COUNT 127 // 2^7-1 - число всех непустых коалиций

typedef unsigned long uint32;
typedef unsigned char uint8;

// Бинарный поиск коалиции x в наборе коалиций A с учетом того,
// что набор коалиций A упорядочен лексикографически
int Binary_search(const uint32 *A, const uint32 na, const uint32 x)
{
    uint32 l, r, mid;
    l = 0;
    r = na - 1;
    while(l < r)
    {
        mid = (l + r)/2;
        if (A[mid] == x)
            l = r = mid;
        else if (A[mid] < x)
            l = mid + 1;
        else r = mid - 1;
    }
    return (A[l] == x);
}

// Нахождение всех правомочных коалиций R по известным
// минимальным правомочным коалициям R_min
void R_Rmin(uint32 *R, uint32 *nr, const uint32 *R_min, const uint32 nr_min)
{
    uint32 i, j;
    // переменная flag отвечает за то, чтобы коалицию не вносить более одного раза
    int flag;

    *nr = 0;
    // пробегаем все коалиции
    for(i = 1; i <= COUNT; i++)
    {
        flag = 1;
```

```
        for(j = 0; j < nr_min && flag; j++)
            // фиксируем минимальную правомочную коалицию R_min[j]
            // и проверяем, является ли R_min[j] подмножеством в i
            if ((R_min[j] | i) == i)
            {
                 R[(*nr)++] = i;
                 flag = 0;
            }
    }
}

// Нахождение всех неправомочных коалиций Z по известным
// максимальным неправомочным коалициям Z_max
void Z_Zmax(uint32 *Z, uint32 *nz, const uint32 *Z_max, const uint32 nz_max)
{
    uint32 i, j;
    // переменная flag отвечает за то, чтобы коалицию не вносить более одного раза
    int flag;

    *nz = 0;
    // пробегаем все коалиции
    for(i = 1; i <= COUNT; i++)
    {
        flag = 1;
        for(j = 0; j < nz_max && flag; j++)
            // фиксируем максимальную неправомочную коалицию Z_max[j]
            // и проверяем, является ли i подмножеством в Z_max[j]
            if ((i | Z_max[j]) == Z_max[j])
            {
                 Z[(*nz)++] = i;
                 flag = 0;
            }
    }
}

// Нахождение всех неправомочных коалиций Z по известным
// правомочным коалициям R
void Z_R(uint32 *Z, uint32 *nz, const uint32 *R, const uint32 nr)
{
    uint32 i, j, a = 0, b;
    *nz = 0;
```

```
    // пробегаем все правомочные коалиции
    for(i = 0; i < nr; i++)
    {
        b = R[i];
        // пробегаем все коалиции от a+1 до b-1 - они неправомочные
        for(j = a + 1; j < b; j++)
            Z[(*nz)++] = j;
        a = b;
    }
    // осталось пробежать коалиции от a+1 до COUNT
    for(j = a + 1; j <= COUNT; j++)
        Z[(*nz)++] = j;
}

// Нахождение всех правомочных коалиций R по известным
// неправомочным коалициям Z
void R_Z(uint32 *R, uint32 *nr, const uint32 *Z, const uint32 nz)
{
    uint32 i, j, a = 0, b;
    *nr = 0;
    // пробегаем все неправомочные коалиции
    for(i = 0; i < nz; i++)
    {
        b = Z[i];
        // пробегаем все коалиции от a+1 до b-1 - они правомочные
        for(j = a + 1; j < b; j++)
            R[(*nr)++] = j;
        a = b;
    }
    // осталось пробежать коалиции от a+1 до COUNT
    for(j = a + 1; j <= COUNT; j++)
        R[(*nr)++] = j;
}

// Нахождение всех минимальных правомочных коалиций R_min
// по известным правомочным коалициям R
void Rmin_R(uint32 *R_min, uint32 *nr_min, const uint32 *R, const uint32 nr)
{
    uint32 i, j, x, flag;
    *nr_min = 0;
    // Пробегаем все правомочные коалиции
```

```
    for(i = 0; i < nr; i++)
    {
        flag = 1;
        // Фиксируем правомочную коалицию R[i]
        // и по очереди удаляем из нее по одному участнику
        for(j = 0; j < N && flag; j++)
            // Проверка, принадлежит ли участник j коалиции R[i]
            if ((R[i] >> j) & (uint32)1)
            {
                // Получаем коалицию x путем удаления
                // участника j из коалиции R[i]
                x = R[i] & (~((uint32)1 << j));
                // Проверяем, принадлежит ли коалиция x
                // правомочным коалициям R
                if (Binary_search(R, nr, x))
                    flag = 0;
            }
        if (flag)
            R_min[(*nr_min)++] = R[i];
    }
}

// Нахождение всех максимальных неправомочных коалиций Z_max
// по известным неправомочным коалициям Z
void Zmax_Z(uint32 *Z_max, uint32 *nz_max, const uint32 *Z, const uint32 nz)
{
    uint32 i, j, x, flag;
    *nz_max = 0;
    // Пробегаем все неправомочные коалиции
    for(i = 0; i < nz; i++)
    {
        flag = 1;
        // Фиксируем неправомочную коалицию Z[i]
        // и по очереди добавляем к ней по одному участнику
        for(j = 0; j < N && flag; j++)
            // Проверка, отсутствует ли участник j в коалиции Z[i]
            if (((Z[i] >> j) & (uint32)1) == 0)
            {
                // Получаем коалицию x путем добавления
                // участника j в коалицию Z[i]
                x = Z[i] | ((uint32)1 << j);
```

```
                // Проверяем, принадлежит ли коалиция x
                // неправомочным коалициям Z
                if (Binary_search(Z, nz, x))
                    flag = 0;
            }
        if (flag)
            Z_max[(*nz_max)++] = Z[i];
    }
}

// Выделение памяти для двумерного динамического массива
uint8 **Allocate(const uint32 m)
{
    uint8 **C;
    uint32 i;
    C = (uint8 **)malloc(N*m*sizeof(uint8) + N*sizeof(uint8 *));
    if (C != NULL)
        for(i = 0; i < N; i++)
             C[i] = (uint8 *)(C + N) + i*m;
    return C;
}

// Заполнение кумулятивного массива
void Init_C(uint8 **C, const uint32 *Z_max, const uint32 m)
{
    uint32 i, j;
    for(i = 0; i < N; i++)
        for(j = 0; j < m; j++)
            // C[i,j] = 0, если участник i принадлежит
            // j-й  максимальной неправомочной коалиции
            if ((Z_max[j] >> i) & (uint32)1)
                C[i][j] = 0;
            else C[i][j] = 1;
}

// Вывод массива C на экран
void Print_C(uint8 **C, const uint32 m)
{
    uint32 i, j;
    for(i = 0; i < N; i++)
    {
```

```
        for(j = 0; j < m; j++)
            printf("%u ", C[i][j]);
        puts("");
    }
}

// вывод коалиций на экран
void Print(const uint32 *A, const uint32 m)
{
    uint32 i, j;
    for(i = 0; i < m; i++)
    {
        for(j = 0; j < N; j++)
            if ((A[i] >> j) & (uint32)1)
                printf("%u ", j);
        puts("");
    }
}

int main()
{
    uint32 nr, // число правомочных коалиций
        nz, // число неправомочных коалиций
        nr_min, // число минимальных правомочных коалиций
        nz_max, // число максимальных неправомочных коалиций
        i, m;
    // коалиции R и Z будут в лексикографическом порядке
    uint32 *R, // правомочные коалиции
           *Z, // неправомочные коалиции
           *R_min, // минимальные правомочные коалиции
           *Z_max; // максимальные неправомочные коалиции
    // Кумулятивный массив C размера N на m, где
    // N - число участников, m - число максимальных неправомочных коалиций
    uint8 **C;

    // Выделяем память для массивов R, Z, R_min, Z_max
    R = (uint32*)malloc(COUNT * sizeof(*R));
    Z = (uint32*)malloc(COUNT * sizeof(*Z));
    R_min = (uint32*)malloc(COUNT * sizeof(*R_min));
    Z_max = (uint32*)malloc(COUNT * sizeof(*Z_max));
    if (R == NULL || Z == NULL || R_min == NULL || Z_max == NULL)
```

```
{
    puts("not enough memory");
    return 1;
}
// Заполнение массива R_min
nr_min = 4;
R_min[0] = 0x23; // 0100011
R_min[1] = 0x2c; // 0101100
R_min[2] = 0x14; // 0010100
R_min[3] = 0x50; // 1010000
puts("R_min = ");
Print(R_min, nr_min); // вывод R_min на экран

// Построение R на основе R_min
R_Rmin(R, &nr, R_min, nr_min);
puts("R = ");
Print(R, nr); // вывод R на экран

// Построение Z на основе R
Z_R(Z, &nz, R, nr);
puts("Z = ");
Print(Z, nz); // вывод Z на экран

// Построение Z_max на основе Z
Zmax_Z(Z_max, &nz_max, Z, nz);
puts("Z_max = ");
Print(Z_max, nz_max); // вывод Z_max на экран

m = nz_max;
C = Allocate(m); // выделяем память под массив C
if(C != NULL)
{
    Init_C(C, Z_max, m); // строим массив C
    puts("C = ");
    Print_C(C, m);
}
else puts("not enough memory");
free(R);
free(Z);
free(R_min);
free(Z_max);
```

```
    return 0;
}
```

В результате, в частности, множество максимальных неправомочных коалиций будет состоять из следующих коалиций:

$$\{0,1,3,4\},\{0,3,4,5\},\{1,3,4,5\},\{0,1,2,3,6\},\{0,2,5,6\},\{1,2,5,6\},\{0,3,5,6\},\{1,3,5,6\}.$$

Заметим, что можно и вовсе обойтись без бинарного поиска в функциях Rmin_R() и Zmax_Z(). Рассмотрим это на примере функции Rmin_R(). Определим массив r размером COUNT+1, индексы которого в диапазоне от 1 до COUNT в точности соответствуют двоичным векторам, которые несут информацию о коалициях. Поэтому в массиве r достаточно напротив индексов, которые соответствуют правомочным коалициям, поставить единицы, а все остальные элементы обнулить. В этом случае коалиция X, которая соответствует двоичному вектору x, является правомочной тогда и только тогда, когда $r[x] = 1$. Тогда функция Rmin_R() примет следующий вид.

```
int Rmin_R(uint32 *R_min, uint32 *nr_min, const uint32 *R, const uint32 nr)
{
    uint32 i, j, x, flag;
    uint8 *r;
    r = (uint8 *)calloc(COUNT+1, sizeof(*r));
    if(r == NULL)
        return 0;
    *nr_min = 0;
    // Пробегаем все правомочные коалиции
    for(i = 0; i < nr; i++)
        r[R[i]] = 1;
    // Пробегаем все правомочные коалиции
    for(i = 0; i < nr; i++)
    {
        flag = 1;
        // Фиксируем правомочную коалицию R[i]
        // и по очереди удаляем из нее по одному участнику
        for(j = 0; j < N && flag; j++)
            // Проверка, принадлежит ли участник j коалиции R[i]
            if ((R[i] >> j) & (uint32)1)
            {
                // Получаем коалицию x путем удаления
                // участника j из коалиции R[i]
                x = R[i] & (~((uint32)1 << j));
                // Проверяем, принадлежит ли коалиция x
                // правомочным коалициям R
```

```
                if (r[x])
                    flag = 0;
            }
        if (flag)
            R_min[(*nr_min)++] = R[i];
    }
    free(r);
    return 1;
}
```

Аналогично можно поступить и с функцией Zmax_Z().

```
int Zmax_Z(uint32 *Z_max, uint32 *nz_max, const uint32 *Z, const uint32 nz)
{
    uint32 i, j, x, flag;
    uint8 *z;
    z = (uint8 *)calloc(COUNT+1, sizeof(*z));
    if (z == NULL)
        return 0;
    *nz_max = 0;
    // Пробегаем все неправомочные коалиции
    for(i = 0; i < nz; i++)
        z[Z[i]] = 1;
    // Пробегаем все неправомочные коалиции
    for(i = 0; i < nz; i++)
    {
        flag = 1;
        // Фиксируем неправомочную коалицию Z[i]
        // и по очереди добавляем к ней по одному участнику
        for(j = 0; j < N && flag; j++)
            // Проверка, отсутствует ли участник j в коалиции Z[i]
            if (((Z[i] >> j) & (uint32)1) == 0)
            {
                // Получаем коалицию x путем добавления
                // участника j в коалицию Z[i]
                x = Z[i] | ((uint32)1 << j);
                // Проверяем, принадлежит ли коалиция x
                // неправомочным коалициям Z
                if (z[x])
                    flag = 0;
            }
        if (flag)
```

```
            Z_max[(*nz_max)++] = Z[i];
    }
    free(z);
    return 1;
}
```

Заключение

В работе приведен пример реализации структур доступа для схем разделения секрета. Основной упор делается на использовании операций работы с битами памяти. В этом случае, в частности, удалось очень эффективно проверять, является ли одно множество подмножеством другого.

Список литературы

1. Погорелов Б. А., Сачков В. Н. *Словарь криптографических терминов*. М.: МЦНМО, 2006. 91 с.
2. Рацеев С. М. *Математические методы защиты информации*: учебное пособие для вузов. СПб. : Лань, 2022. 544 с.
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.

Efficient implementation of access structures for secret sharing schemes

Ratseev, S. M.

ratseevsm@mail.ru

Ulyanovsk State University, Ulyanovsk, Russia

In the paper the efficient construction of a set of eligible coalitions, a set of not eligible coalitions, a set of minimum eligible coalitions, a set of maximum not eligible coalitions for access structures for secret sharing schemes is investigated, if at least one of the listed sets is known.

Keywords: *secret sharing scheme, access structure.*