Правительство Российской Федерации Федеральное государственное автономное образовательное учреждение высшего образования “Национальный исследовательский институт «Высшая школа экономики»

Кафедра «Компьютерная безопасность»

ОТЧЁТ
К ИНДИВИДУАЛЬНОМУ ЗАДАНИЮ

по дисциплине

“Программирование алгоритмов защиты информации”

Выполнил
Самунин Р.О. СКБ 202

Преподаватель:

Нестеренко А.Ю.

Постановка задачи

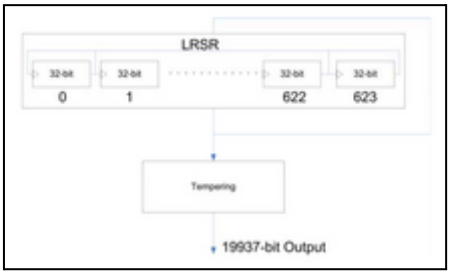
Реализовать ГПСЧ Вихрь Мерсенна.
Встроить ГСПЧ в библиотеку libokrypt.

Теоретические основы

Вихрь Мерсе́нна — генератор псевдослучайных чисел (ГПСЧ), алгоритм, разработанный в 1997 году японскими учёными Макото Мацумото и Такудзи Нисимура.

Вихрь Мерсенна генерирует псевдослучайные последовательности чисел с периодом, равным одному из простых чисел Мерсенна, отсюда этот алгоритм и получил своё название, и обеспечивает быструю генерацию высококачественных по критерию случайности псевдослучайных чисел.

Вихрь Мерсенна алгоритмически реализуется двумя основными частями: *рекурсивной* и *закалки*. Рекурсивная часть представляет собой регистр сдвига с линейной обратной связью, в котором все биты в его слове определяются рекурсивно; поток выходных битов определяются также рекурсивно функцией битов состояния.

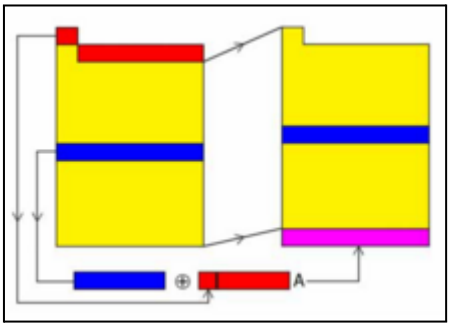


Блок-схема.

Регистр сдвига состоит из 624 элементов, и, в общей сложности, из 19937 клеток. Каждый элемент имеет длину 32 бита за исключением первого элемента, который имеет только 1 бит за счет отбрасывания бита.

Процесс генерации начинается с логического умножения на битовую маску, отбрасывающей 31 бита (кроме наиболее значащих).

Следующим шагом выполняется инициализация $(x_0, x_1, \ldots, x_{623})$ любыми беззнаковыми 32-разрядными целыми числами. Следующие шаги включают в себя объединение и переходные состояния.



Смена состояния МТ.

Пространство состояний имеет 19937 бит (624·32 — 31). Следующее состояние генерируется сдвигом одного слова вертикально вверх и вставкой нового слова в конец. Новое слово вычисляется гаммированием средней части с исключённой[14]. Выходная последовательность начинается с $x_{624}, x_{625},\ldots$

Алгоритм производится в шесть этапов:

Шаг 0. Предварительно инициализируется значение констант *u1*, *h1*, *a*

u1 ← 10…0 битовая маска старших *w-r* бит,

h1 ← 01…1 битовая маска младших *r* бит,

$a \leftarrow a_{w-1}a_{w-2}\ldots a_0$ последняя строка матрицы *A*.

Шаг 1. **x**[0], **x**[1],…,**x**[n-1] ← начальное заполнение

Шаг 2. Вычисление $(x_i^u \mid x_{i+1}^l)$

y ← (**x**[i] AND *u1*) OR (**x**[(i + 1) mod n] AND *h*1)

Шаг 3. Вычисляется значение следующего элемента последовательности по рекуррентному выражению (1)

x[i] ← **x**[(i + m) mod n] XOR (**y**>>1) XOR *a* если младший бит **y** = 1

Или

x[i] ← **x**[(i + m) mod n] XOR (**y**>>1) XOR 0 если младший бит **y** = 0

Шаг 4. Вычисление **x**[i]*T*

y ← **x**[i]

y ← **y** XOR (**y**>>*u*)

y ← **y** XOR ((**y**<<*s*) AND **b**)

y ← **y** XOR ((**y**<<*t*) AND **c**)

z ← **y** XOR (**y**>>*l*)

вывод **z**

Шаг 5. i ← (i + 1) mod n

Шаг 6. Перейти к шагу 2.

Работа программы

Разберем код по функциям:

Структура mersenne_data:

```
typedef struct random_mersenne {
  ak_uint32 state[624];
```

```
    ak_uint32 current_index;
} mersenne_data;
```
Эта структура хранит внутреннее состояние генератора. state — массив из 624 32-битных целых чисел, используемых для вычисления следующего случайного числа. current_index — индекс текущего элемента в массиве state.

ak_random_mersenne_randomize_ptr:

```
static int ak_random_mersenne_randomize_ptr(ak_random rnd, const ak_pointer ptr, const ssize_t size) {
    // ... (проверка на NULL и правильный размер)
    ctx->state[0] = (ak_uint32)ptr;
    ctx->current_index = 1;
    // ... (инициализация остальных элементов массива state)
}
```

Эта функция инициализирует генератор. Она получает указатель на 32-битное целое число (подразумевается, что это начальное значение) и устанавливает его как начальное состояние генератора. Затем инициализирует остальные элементы массива state с помощью псевдослучайной формулы, гарантируя, что следующее число в последовательности действительно отличается от предыдущего.

ak_random_mersenne_next:

```
static int ak_random_mersenne_next(ak_random rnd) {
    // ... (вычисление следующего случайного числа в массиве state)
}
```

Эта функция генерирует следующее случайное число в последовательности. Она использует формулу для пересчета элементов массива state, чтобы получить новое случайное значение, основанное на предыдущем состоянии. Важно, что метод Вихря Мерсенна гарантирует, что последовательность случайных чисел будет очень длинной и почти случайной.

ak_random_mersenne_random:

```
static int ak_random_mersenne_random(ak_random rnd, const ak_pointer buffer, ssize_t size) {
    // ... (проверка на NULL и правильный размер)
    // ... (получение данных для генерации)
    while (size-- > 0) {
        buf++ = output[ctx->current_index++];
        if (ctx->current_index == 624)
            ak_random_mersenne_next(rnd);
    }
}
```

Эта функция генерирует последовательность случайных байтов и записывает их в указанный буфер. Она использует метод *ak_random_mersenne_next* для получения очередного случайного

числа из генератора. Если индекс достигает конца массива state, она вызывает *ak_random_mersenne_next*, чтобы обновить состояние генератора.

ak_random_mersenne_free:

static int ak_random_mersenne_free(ak_random rnd) {
*   // ... (освобождение памяти, выделенной для генератора)*
}

Освобождает выделенную память для генератора, предотвращая утечки памяти.

ak_random_create_mersenne:

int ak_random_create_mersenne(ak_random generator) {
*   // ... (инициализация генератора и выделение памяти)*
*   // ... (инициализация с помощью ak_random_value())*
}

Эта функция создает и инициализирует новый генератор случайных чисел Вихря Мерсенна. Она получает указатель на структуру *ak_random*, инициализирует все необходимые поля и возвращает *ak_error_ok* при успехе.

Тестирование

```
root@LAPTOP-IP3M7OOS:~/test# gcc main.c -o test -lakrypt
root@LAPTOP-IP3M7OOS:~/test# ./test
9ae36879a3ea8589526cc3756aae8bf7af0a90ae8814b8e2f7f1e9cc34457de9892a639190ed8809

bf20e72cd9f85b79b1968b05507323f2cfc66eb2cf5045e1041995f63f4739f30a5195aaa40e70ad
```

Результат выполнения - первые и последние 40 бит.

Код

```
/*
-------------------------------------------------------------------------------
------------  */

/*                            реализация Вихря Мерсенна
*/

/*
-------------------------------------------------------------------------------
------------- */

/** @brief Класс с параметрами для Вихря Мерсенна */

typedef struct random_mersenne {

    /** @brief Внутреннее состояние генератора. */

    ak_uint32 state[624];

    /** @brief Количество использованных байт в векторе */

    ak_uint32 current_index;

} mersenne_data;



/*
-------------------------------------------------------------------------------
------------ */

 /**

  * @brief  Инициализирует NLFSR генератор специального вида переданными
параметрами.

  *

  * @param rnd NLFSR генератор.

  * @param ptr Указатель на данные для инициализации NLFSR генератора.

  * @param size Количество параметров для инициализации.

  * @return int В случае успеха, функция возвращает \ref ak_error_ok. В противном
случае

            возвращается код ошибки.

  */
```

```
static int ak_random_mersenne_randomize_ptr(ak_random rnd, const ak_pointer ptr,
const ssize_t size)

{

    mersenne_data* ctx = NULL;

    if (rnd == NULL) return ak_error_message(ak_error_null_pointer, __func__,

        "use a null pointer to a random generator");

    if (ptr == NULL) return ak_error_message(ak_error_null_pointer, __func__,

        "use a null pointer to initial vector");

    if (size != 4) return ak_error_message(ak_error_wrong_length, __func__,

        "use initial vector with wrong length");



    ctx = (mersenne_data*)rnd->data.ctx;

    ctx->state[0] = *(ak_uint32*)ptr;

    ctx->current_index = 1;



    ak_uint32 i;

    ak_uint32 N = 624;



    for (i = 1; i < N; ++i) {

        ctx->state[i] = (1812433253UL * (ctx->state[i - 1] ^ (ctx->state[i - 1] >>
30)) + i);

    }



    return ak_error_ok;

}
```

```
/*
------------------------------------------------------------------------------
------------ */

static int ak_random_mersenne_next(ak_random rnd)

{

    ak_uint32 kk;

    ak_uint32 y;

    static const ak_uint32 mag01[2] = { 0x0, 0x9908b0df };

    mersenne_data* ctx = (mersenne_data*)rnd->data.ctx;



    ak_uint32 N = 624;

    ak_uint32 M = 397;

    ak_uint32 UPPER_MASK = 0x7fffffff;

    ak_uint32 LOWER_MASK = 0x80000000;



    for (kk = 0; kk < N - M; kk++) {

        y = (ctx->state[kk] & UPPER_MASK) | (ctx->state[kk + 1] & LOWER_MASK);

        ctx->state[kk] = ctx->state[kk + M] ^ (y >> 1) ^ mag01[y & 0x1];

    }

    for (; kk < N - 1; kk++) {

        y = (ctx->state[kk] & UPPER_MASK) | (ctx->state[kk + 1] & LOWER_MASK);

        ctx->state[kk] = ctx->state[kk + (M - N)] ^ (y >> 1) ^ mag01[y & 0x1];

    }

    y = (ctx->state[N - 1] & UPPER_MASK) | (ctx->state[0] & LOWER_MASK);

    ctx->state[N - 1] = ctx->state[M - 1] ^ (y >> 1) ^ mag01[y & 0x1];



    ctx->current_index = 0;
```

```
    return ak_error_ok;
}

/*
------------------------------------------------------------------------------
------------ */
/**
 */
static int ak_random_mersenne_random(ak_random rnd, const ak_pointer buffer,
ssize_t size)
{
    if (rnd == NULL) return ak_error_message(ak_error_null_pointer, __func__,
        "use a null pointer to a random generator");
    if (buffer == NULL) return ak_error_message(ak_error_null_pointer, __func__,
        "use a null pointer to data");
    if (size <= 0) return ak_error_message(ak_error_wrong_length, __func__,
        "use a data vector with wrong length");

    ak_uint8* buf = (ak_uint8*)buffer;
    mersenne_data* ctx = (mersenne_data*)rnd->data.ctx;
    ak_uint8* output = (ak_uint8*)ctx->state;

    while (size-- > 0)
    {
        *buf++ = output[ctx->current_index++];
        if (ctx->current_index == 624)
```

```
            ak_random_mersenne_next(rnd);

    }


    return ak_error_ok;

}


/*
------------------------------------------------------------------------------
------------ */

/**

static int ak_random_mersenne_free(ak_random rnd)

{

    if (rnd == NULL) return ak_error_message(ak_error_null_pointer, __func__,

        "use a null pointer to a random generator");

    if (rnd->data.ctx != NULL)

        free(rnd->data.ctx);


    return ak_error_ok;

}


/*
------------------------------------------------------------------------------
------------ */

/**                                                              */

int ak_random_create_mersenne(ak_random generator)

{

    int error = ak_error_ok;
```

```
    if ((error = ak_random_create(generator)) != ak_error_ok)
        return ak_error_message(error, __func__, "wrong initialization of random
generator");

    generator->oid = 0;
    generator->next = ak_random_mersenne_next;
    generator->randomize_ptr = ak_random_mersenne_randomize_ptr;
    generator->random = ak_random_mersenne_random;
    generator->free = ak_random_mersenne_free;

    if ((generator->data.ctx = calloc(1, sizeof(struct random_mersenne))) == NULL)
{
        ak_random_destroy(generator);
        return ak_error_message(ak_error_null_pointer, __func__, "incorrect memory
allocation ");
    }

    ak_uint64 seed = ak_random_value();
    generator->randomize_ptr(generator, &seed, 4);

    return error;
}

/*
-------------------------------------------------------------------------------
------------ */
```

```
/*
ak_random.c  */

/*
-------------------------------------------------------------------------------
------------ */
```

Список литературы:

- *M. Matsumoto, T. Nishimura.* Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator (англ.) // ACM Trans. on Modeling and Computer Simulations : journal. — 1998. — Vol. 8, no. 1. — P. 3—30. — doi:10.1145/272991.272995.
- *Matsumoto, M.; Kurita, Y.* Twisted GFSR generators (неопр.) // ACM Trans. on Modeling and Computer Simulations. — 1992. — Т. 2, № 3. — С. 179—194. — doi:10.1145/146382.146383.
- *Matsumoto, Makoto; Nishimura, Takuji; Hagita, Mariko; Saito, Mutsuo.* Cryptographic Mersenne Twister and Fubuki Stream/Block Cipher (англ.) : journal. — 2005.
- *T. Nishimura.* Tables of 64-bit Mersenne twisters (неопр.) // ACM Trans. on Modeling and Computer Simulations. — 2000. — Т. 10, № 4. — С. 248—357. — doi:10.1145/369534.369540.
- *Matsumoto M., Saito M., Nishimura T., Hagita M.* CryptMT Stream Cipher Ver. 3.The eSTREAM Project (неопр.).