Лабораторная работа №3

Потоковые шифры, функции хеширования

Цель работы: Научиться реализовывать потоковый алгоритм шифрования с использованием генераторов псевдослучайных чисел, а также получить опыт реализации собственных функций хеширования текстовых паролей.

Теоретическая часть

Хеширование или **хэширование** (англ. Hashing – мелкая нарезка и перемешивание) — преобразование массива входных данных произвольной длины в (выходную) битовую строку установленной длины, выполняемое определённым алгоритмом. Функция, воплощающая алгоритм и выполняющая преобразование, называется «хеш-функцией» или «функцией свёртки». Исходные данные называются входным массивом, «ключом» или «сообщением». Результат преобразования (выходные данные) называется «хешем», «хеш-кодом», «хеш-суммой», «сводкой сообщения».

I. Простые функции хеширования

1. MaHash7

Такую функцию очень легко реализовать как программно, так и аппаратно, скорость запроса к таблице высока в обоих случаях. У функции есть параметр - число итераций. Статистически, лучшим параметром будет одна итерация для большинства текстов. Хотя для оригинальной функции оптимальными будут три итерации. Сама таблица подстановки идентична таковой в криптоалгоритме Skipjack. Результаты не лучшие, но конструкция имеет право на жизнь.

```
static const unsigned char sTable[256] =
{
  0xa3,0xd7,0x09,0x83,0xf8,0x48,0xf6,0xf4,0xb3,0x21,0x15,0x78,0x99,0xb1,0xaf,0xf9,
  0xe7,0x2d,0x4d,0x8a,0xce,0x4c,0xca,0x2e,0x52,0x95,0xd9,0x1e,0x4e,0x38,0x44,0x28,
  0x0a,0xdf,0x02,0xa0,0x17,0xf1,0x60,0x68,0x12,0xb7,0x7a,0xc3,0xe9,0xfa,0x3d,0x53,
  0x96,0x84,0x6b,0xba,0xf2,0x63,0x9a,0x19,0x7c,0xae,0xe5,0xf5,0xf7,0x16,0x6a,0xa2,
  0x39,0xb6,0x7b,0x0f,0xc1,0x93,0x81,0x1b,0xee,0xb4,0x1a,0xea,0xd0,0x91,0x2f,0xb8,
  0x55,0xb9,0xda,0x85,0x3f,0x41,0xbf,0xe0,0x5a,0x58,0x80,0x5f,0x66,0x0b,0xd8,0x90,
  0x35,0xd5,0xc0,0xa7,0x33,0x06,0x65,0x69,0x45,0x00,0x94,0x56,0x6d,0x98,0x9b,0x76,
  0x97,0xfc,0xb2,0xc2,0xb0,0xfe,0xdb,0x20,0xe1,0xeb,0xd6,0xe4,0xdd,0x47,0x4a,0x1d,
  0x42,0xed,0x9e,0x6e,0x49,0x3c,0xcd,0x43,0x27,0xd2,0x07,0xd4,0xde,0xc7,0x67,0x18,
  0x89,0xcb,0x30,0x1f,0x8d,0xc6,0x8f,0xaa,0xc8,0x74,0xdc,0xc9,0x5d,0x5c,0x31,0xa4,
  0x70,0x88,0x61,0x2c,0x9f,0x0d,0x2b,0x87,0x50,0x82,0x54,0x64,0x26,0x7d,0x03,0x40,
  0x34,0x4b,0x1c,0x73,0xd1,0xc4,0xfd,0x3b,0xcc,0xfb,0x7f,0xab,0xe6,0x3e,0x5b,0xa5,
  0xad,0x04,0x23,0x9c,0x14,0x51,0x22,0xf0,0x29,0x79,0x71,0x7e,0xff,0x8c,0x0e,0xe2,
  0x0c,0xef,0xbc,0x72,0x75,0x6f,0x37,0xa1,0xec,0xd3,0x8e,0x62,0x8b,0x86,0x10,0xe8,
  0x08,0x77,0x11,0xbe,0x92,0x4f,0x24,0xc5,0x32,0x36,0x9d,0xcf,0xf3,0xa6,0xbb,0xac,
  0x5e,0x6c,0xa9,0x13,0x57,0x25,0xb5,0xe3,0xbd,0xa8,0x3a,0x01,0x05,0x59,0x2a,0x46
};
```

```
#define LROT14(x) (((x) << 14) | ((x) >> 18 ))
#define ITERATIONS 1
unsigned int
MaHash7 (unsigned char *str, unsigned int len)
{
  unsigned int hash = len, i;
  for (i = 0; i != len * ITERATIONS; i++)
    {
     str += (i % len);
      hash =
        LROT13 (hash + ((hash << 8) ^ (hash >> 12))) -
        sTable[(*str + i) & 255];
    }
  return hash;
}
```

2. MaHash9

В отличие от MaHash7 используются другие константы сдвигов и циклический сдвиг влево на 14 бит, дополнительная операция побитового НЕ. Как результат - немного лучше показатели в ряде тестов. Таблица sTable берется из MaHash7.

```
#define LROT14(x) (((x) << 14) | ((x) >> 18 ))
unsigned int MaHash9 (unsigned char *str, unsigned int len)
{
    unsigned int hash = len, i;
    for (i = 0; i != len; i++, str++)
    {
        hash = LROT14( ~hash + ((hash << 6) ^ (hash >> 11))) - sTable[(*str + i) & 255];
    }
    return hash;
}
```

3. MaHash2

По-сути, это два алгоритма MaHash для параллельного подсчета двух 32-разрядных значений. На каждом этапе результат вычисления оригинальной и видоизмененной функции с обратным циклическим сдвигом вправо взаимозаменяются. Результатом хэш-функции будет сумма подфункций по модулю 2 (операция XOR). Такая конструкция позволяет улучшить результаты алгоритма в плане числа коллизий. Достоинства алгоритма - отсутствие операций деления, умножения (наиболее медленных на современных неспециализированных процессорах) и возможность распараллеливания. Таблица sTable берется из MaHash7.

```
#define LROT(x) (((x) << 11) | ((x) >> 21 ))
#define RROT(x) (((x) << 21) | ((x) >> 11 ))
unsigned int MaHash2 (unsigned char *str, unsigned int len)
{
    unsigned int t, hash1 = len, hash2 = len, i;
    for (i = 0; i != len; i++, str++)
    {
        hash1 += sTable[(*str + i) & 255];
        hash1 = LROT(hash1 + ((hash1 << 6) ^ (hash1 >> 8)));
        hash2 += sTable[(*str + i) & 255];
        hash2 = RROT(hash2 + ((hash2 << 6) ^ (hash2 >> 8)));
        t = hash1;
        hash1 = hash2;
        hash2 = t;
    }
    return hash1 ^ hash2;
}
```

4. MaHash8

Изменены сдвиги, циклический сдвиг увеличился не несколько разрядов. На смену взаимозамены результатов подфункций теперь они “собираются” из младшего родного слова и старшего слова альтернативной функции. Таблица sTable берется из MaHash7.

```
#define LROT14(x) (((x) << 14) | ((x) >> 18 ))
#define RROT14(x) (((x) << 18) | ((x) >> 14 ))
unsigned int MaHash8 (unsigned char *str, unsigned int len)
{
    unsigned int sh1, sh2, hash1 = len, hash2 = len, i;
    for (i = 0; i != len; i++, str++)
    {
        hash1 += sTable[(*str + i) & 255];
        hash1 = LROT14(hash1 + ((hash1 << 6) ^ (hash1 >> 11)));
        hash2 += sTable[(*str + i) & 255];
        hash2 = RROT14(hash2 + ((hash2 << 6) ^ (hash2 >> 11)));
        sh1 = hash1;
        sh2 = hash2;
        hash1 = ((sh1 >> 16) & 0xffff) | (sh2 & 0xffff) << 16;
        hash2 = ((sh2 >> 16) & 0xffff) | (sh1 & 0xffff) << 16;
    }
    return hash1 ^ hash2;
}
```

5. MaHash4

Здесь используются классические сдвиги и обычный циклический сдвиг. Только вместо взаимозамены опять “сборка” значений. Эта функция, как и прошлая, стабильно показывает очень хорошие результаты, не имеет “провалов”. Таблица sTable берется из MaHash7.

```
#define LROT(x) (((x) << 11) | ((x) >> 21 ))
unsigned int
MaHash4 (unsigned char *str, unsigned int len)
{
  unsigned int sh1, sh2, hash1 = len, hash2 = len, i;
  for (i = 0; i != len; i++, str++)
    {
      hash1 += sTable[(*str + i) & 255];
      hash1 = LROT (hash1 + ((hash1 << 6) ^ (hash1 >> 8)));
      hash2 += sTable[(*str + i) & 255];
      hash2 = RROT (hash2 + ((hash2 << 6) ^ (hash2 >> 8)));
      sh1 = hash1;
      sh2 = hash2;
      hash1 = ((sh1 >> 16) & 0xffff) | (sh2 & 0xffff) << 16;
      hash2 = ((sh2 >> 16) & 0xffff) | (sh1 & 0xffff) << 16;
    }
  return hash1 ^ hash2;
}
```

6. MaHash5

Добавлено дополнительное преобразование в конце шага. Таблица sTable берется из MaHash7.

```
#define LROT(x) (((x) << 11) | ((x) >> 21 ))
unsigned int
MaHash5 (unsigned char *str, unsigned int len)
{
  unsigned int sh1, sh2, hash1 = len, hash2 = len, i;
  for (i = 0; i != len; i++, str++)
    {
      hash1 += sTable[(*str + i) & 255];
      hash1 = LROT (hash1 + ((hash1 << 6) ^ (hash1 >> 8)));
      hash2 += sTable[(*str + i) & 255];
      hash2 = RROT (hash2 + ((hash2 << 6) ^ (hash2 >> 8)));
      sh1 = LROT (hash1 + ((hash1 << 6) ^ (hash1 >> 8)));
```

```
      sh2 = RROT (hash2 + ((hash2 << 6) ^ (hash2 >> 8)));
      hash1 = ((sh1 >> 16) & 0xffff) | (sh2 & 0xffff) << 16;
      hash2 = ((sh2 >> 16) & 0xffff) | (sh1 & 0xffff) << 16;
    }
  return hash1 ^ hash2;
}
```

7. MaHash11

Это MaHash2 с другими константами сдвигов. Причем, теперь для каждой подфункции теперь применяется свое значение циклического сдвига. Таблица sTable берется из MaHash7.

```
#define LROT12(x) (((x) << 12) | ((x) >> 20 ))
#define RROT13(x) (((x) << 19) | ((x) >> 13 ))
unsigned int MaHash11 (unsigned char *str, unsigned int len)
{
    unsigned int t, hash1 = len, hash2 = len, i;
    for (i = 0; i != len; i++, str++)
    {
        hash1 += sTable[(*str + i) & 255];
        hash1 = LROT12(hash1 + ((hash1 << 6) ^ (hash1 >> 14)));
        hash2 += sTable[(*str + i) & 255];
        hash2 = RROT13(hash2 + ((hash2 << 6) ^ (hash2 >> 14)));
        t = hash1;
        hash1 = hash2;
        hash2 = t;
    }
    return hash1 ^ hash2;
}
```

8. MaHash10

Здесь уже нет двух параллельных подфункций, хотя и вычисляются два 32-разядных значения на каждом шаге. Как и в прошлой функции, циклические сдвиги различны для каждого вычисления. Такая конструкция имеет небольшие всплески коллизий. Таблица sTable берется из MaHash7.

```
#define LROT14(x) (((x) << 14) | ((x) >> 18 ))
#define RROT14(x) (((x) << 18) | ((x) >> 14 ))
unsigned int MaHash10 (unsigned char *str, unsigned int len)
{
    unsigned int hash1 = len, hash2 = len, i, value;
```

```
    for (i = 0; i != len; i++, str++)
    {
        value = sTable[(*str + i) & 255];
        hash1 =  LROT14(hash2 + ((hash2 << 6) ^ (hash2 >> 11)));
        hash1 += value;
        hash2 =  RROT15(hash1 + ((hash1 << 6) ^ (hash1 >> 11)));
        hash2 += value;
    }
    return hash1 ^ hash2;
}
```

9. MaHash3

Дополнительная итерация немного снизила число коллизий на самых сложных тестах. Таблица sTable берется из MaHash7.

```
#define LROT(x) (((x) << 11) | ((x) >> 21 ))
#define RROT(x) (((x) << 21) | ((x) >> 11 ))
unsigned int
MaHash3 (unsigned char *str, unsigned int len)
{
  unsigned int t, hash1 = len, hash2 = len, i;
  unsigned char index;
  for (i = 0; i != len; i++, str++)
    {
      index = (*str + i) & 255;
      hash1 += sTable[index];
      hash1 = LROT (hash1 + ((hash1 << 6) ^ (hash1 >> 8)));
      hash2 += sTable[(sTable[index] + 1) & 255];
      hash2 = RROT (hash2 + ((hash2 << 6) ^ (hash2 >> 8)));
      t = hash1;
      hash1 = hash2;
      hash2 = t;
    }

  hash1 = LROT (hash1 + ((hash1 << 6) ^ (hash1 >> 8)));
  hash1 += sTable[len];
  hash2 = RROT (hash2 + ((hash2 << 6) ^ (hash2 >> 8)));
  hash2 += sTable[len];
  return hash1 ^ hash2;
}
```

10. MaPrime

Существующие простые хэш-функции на простых числах немного деградируют в сложных случаях. Такие случаи - это наборы текстов, имеющие совпадающие участки. Более точно - это потому, что алгоритмы не предусмотрели четкой взаимосвязи между символом и его позицией в строке. Этот алгоритм очень прост, основан на операции умножения на простое число. Используется таблица подстановки sTable, так же как и в других функциях. Как вариант, может использоваться упрощенный вариант без таблицы. Преимущества - элементарная и понятная структура, простота реализации и хорошие результаты во всех тестах.

```
#define PRIME_MULT 0x1FAF
#define START_PRIME 0x3A8F05C5
#define USE_SBOX
unsigned int maPrimeHash (unsigned char *buf, unsigned int len)
{
    unsigned int hval = START_PRIME, i;
    for (i = 0; i != len; i++, buf++)
    {
        #ifdef USE_SBOX
                hval ^= sTable[(*buf +  i) & 255];
        #else
                hval += ((unsigned int)*buf)  ^  i;
        #endif
                hval *= PRIME_MULT;
    }
    return hval;
}
```

11. MaPrime2d

Функция показывает более стабильные результаты, нежели оригинальная - maPrime. Да и дополнения не так сильно отражаются на производительности. Второй момент - это возможность параметризации, т.е. тонкой настройки под определенный набор данных. Существенный параметр здесь - это константа циклического сдвига. В тесте использован сдвиг на два бита вправо, что обеспечило усредненные результаты по тестам в целом, однако использование некоторых других значений может быть более приемлемым в зависимости от набора текстов. К примеру - один или три бита. Таблица sTable берется из MaHash7.

```
unsigned int
maPrime2dHash (unsigned char *str, unsigned int len)
{
  unsigned int hash = 0, i;
```

```
  unsigned int rotate = 2;
  unsigned int seed = 0x1A4E41U;
  for (i = 0; i != len; i++, str++)
    {
      hash += sTable[(*str + i) & 255];
      hash = (hash << (32 - rotate) ) | (hash >> rotate);
      hash = (hash + i ) * seed;
    }
  return (hash + len) * seed;
}
```

12. MaHash8v64

Это - двойник алгоритма MaHash8, только теперь функция возвращает 64-разрядный хэш, то есть оба 32-разрядных значения, не сжимая их в одно. Сам алгоритм полностью 32-разрядный, скорость его не уменьшилась. Такую манипуляцию можно провести со всеми алгоритмами семейства, в которых структура - это две подфункции. Достоинства алгоритма - высокая скорость, равная алгоритму с 32-разрядным хэшем, простота реализации и очень хорошие показатели по числу коллизий. Для увеличения лавинного эффекта можно использовать простую модификацию с дополнительным преобразованием финального 64-битного значения. Таблица sTable берется из MaHash7.

```
#define LROT14(x) (((x) << 14) | ((x) >> 18 ))
#define RROT14(x) (((x) << 18) | ((x) >> 14 ))
#ifdef HIAVAL
#define LROT64(x) (((x) << 29) | ((x) >> 34 ))
#endif
unsigned long long
MaHash8v64 (unsigned char *str, unsigned int len)
{
  unsigned int sh1, sh2, hash1 = len, hash2 = len, i;
  unsigned long long digest;

  for (i = 0; i != len; i++, str++)
    {
      hash1 += sTable[( *str + i) & 255];
      hash1 = LROT14 (hash1 + ((hash1 << 6) ^ (hash1 >> 11)));
      hash2 += sTable[( *str + i) & 255];
      hash2 = RROT14 (hash2 + ((hash2 << 6) ^ (hash2 >> 11)));
      sh1 = hash1;
      sh2 = hash2;
      hash1 = ((sh1 >> 16) & 0xffff) | ((sh2 & 0xffff) << 16);
```

```
      hash2 = ((sh2 >> 16) & 0xffff) | ((sh1 & 0xffff) << 16);
    }
#ifdef HIAVAL
  digest = (((unsigned long long) hash1) << 32)  | ((unsigned long long) hash2 );
  digest ^= LROT64(digest + ((digest << 13) ^ (digest >> 23)));
#else
  digest = (((unsigned long long) hash2) << 32)  | ((unsigned long long) hash1 );
#endif
  return digest;
}
```

13. MaHash4v64

Это модификация существующего алгоритма, аналогично вышерассмотренной 64-разрядной функции. Также, как и в прошлом варианте, можно усилить лавинный эффект путем добавления дополнительного преобразования. Это немного замедлит алгоритм, поскольку операции выполняются над 64-разрядными беззнаковыми числами, но функция будет выглядеть интереснее. Таблица sTable берется из MaHash7.

```
#define LROT(x) (((x) << 11) | ((x) >> 21 ))
#define RROT(x) (((x) << 21) | ((x) >> 11 ))
#ifdef HIAVAL
#define LROT64(x) (((x) << 29) | ((x) >> 34 ))
#endif
unsigned long long
MaHash4v64 (unsigned char *str, unsigned int len)
{
  unsigned int sh1, sh2, hash1 = len, hash2 = len, i;
  unsigned long long digest;
  for (i = 0; i != len; i++, str++)
    {
      hash1 += sTable[( *str + i) & 255];
      hash1 = LROT (hash1 + ((hash1 << 6) ^ (hash1 >> 8)));
      hash2 += sTable[( *str + i) & 255];
      hash2 = RROT (hash2 + ((hash2 << 6) ^ (hash2 >> 8)));
      sh1 = hash1;
      sh2 = hash2;
      hash1 = ((sh1 >> 16) & 0xffff) | ((sh2 & 0xffff) << 16);
      hash2 = ((sh2 >> 16) & 0xffff) | ((sh1 & 0xffff) << 16);
    }

#ifdef HIAVAL
```

```
  digest = (((unsigned long long) hash1) << 32)  | ((unsigned long long) hash2 );
  digest ^= LROT64(digest + ((digest << 13) ^ (digest >> 23)));
#else
  digest = (((unsigned long long) hash2) << 32)  | ((unsigned long long) hash1 );
#endif
  return digest;

}
```

14. MaHashMR1

Здесь используется несколько необычная структура: четыре “вращающихся” 8-битных регистра накопления, которые модифицируются с помощью двойной табличной замены. На каждом шаге для накопления входного текста выбирается следующий регистр, затем все биты всех регистров циклически сдвигаются на один бит. Функция может иметь любое кол-во итераций. По всей видимости, оптимальным их числом будет три. Таблица sTable берется из MaHash7.

```
#define LROT1(x) (((x) << 1) | ((x) >> 31 ))
#define MR_ITERATIONS 3
union type_regs
  {
    unsigned long d32;
    unsigned char d8[4];
  };
unsigned long
MaHashMR1 (unsigned char *str, unsigned int len)
{
  unsigned int  i, s = 0;
  unsigned char val;
  union type_regs regs;
  regs.d32 = len;
  for (i = 0; i != len * MR_ITERATIONS; i++)
    {
      val = *(str + (i % len));
      regs.d8[s] =  sTable[(sTable[(val + i) & 255] + regs.d8[s]) & 255];
      s = (s + 1) % 4;
      regs.d32 = LROT1(regs.d32);
    }
  return regs.d32;
}
```

15. MurmurHash2AM

К предыдущей реализации добавляется счетчик блока. Ведь счетчик вообще - дело хорошее. Он помогает нам для двух одинаковых блоков вычислить разные значения. Это не совсем то, что даст хорошая табличная подстановка, но уже избавит от множества проблем. Мы не получим таких всплесков на сложных наборах, как исходный вариант. При этом общие результаты пострадать не должны. Таблица sTable берется из MaHash7.

```
#define mmix(h,k) { k *= m; k ^= k >> r; k *= m; h *= m; h ^= k; }
unsigned int MurmurHash2AM ( char * key, unsigned int len)
{
  const unsigned int m = 0x5bd1e995;
  const int r = 24;
  unsigned int l = len, h = 0, i = 1;
  const unsigned char * data = (const unsigned char *)key;
  while (len >= 4)
    {
      unsigned int k = *(unsigned int*)data + i++;
      mmix(h,k);
      data += 4;
      len -= 4;
    }
  unsigned int t = 0;
  switch (len)
    {
    case 3:
      t ^= data[2] << 16;
    case 2:
      t ^= data[1] << 8;
    case 1:
      t ^= data[0];
    };
  mmix(h,t);
  mmix(h,l);
  h ^= h >> 13;
  h *= m;
  h ^= h >> 15;
  return h;
}
```

II. Более сложные функции хеширования

1. Алгоритм SHA-1 (Secure Hash Algorithm 1)

Secure Hash Algorithm 1 — алгоритм криптографического хеширования. Описан в RFC 3174. Для входного сообщения произвольной длины (максимум $2^{64}-1$ бит, что равно 2 эксабайта) алгоритм генерирует 160-битное хеш-значение, называемое также дайджестом сообщения. Используется во многих криптографических приложениях и протоколах. Также рекомендован в качестве основного для государственных учреждений в США.

Полное описание алгоритма: https://ru.bmstu.wiki/SHA-1_(Secure_Hash_Algorithm_1)

2. Алгоритм SHA-2 (Secure Hash Algorithm 2)

SHA-2 (англ.) Secure Hash Algorithm Version 2 — безопасный алгоритм хеширования, версия 2) — собирательное название однонаправленных хеш-функций SHA-224, SHA-256, SHA-384 и SHA-512. Хеш-функции предназначены для создания «отпечатков» или «дайджестов» сообщений произвольной битовой длины. Применяются в различных приложениях или компонентах, связанных с защитой информации.

Полное описание алгоритма: https://ru.bmstu.wiki/SHA-2_(Secure_Hash_Algorithm_2)

3. Алгоритм MD4 (Message Digest 4)

MD4 (Message Digest 4) — хеш-функция, разработанная профессором Массачусетского технологического института Рональдом Ривестом в 1990 году, и впервые описанная в RFC 1186. Для произвольного входного сообщения функция генерирует 128-разрядное хеш-значение, называемое дайджестом сообщения. Этот алгоритм используется в протоколе аутентификации MS-CHAP, разработанном корпорацией Майкрософт для выполнения процедур проверки подлинности удаленных рабочих станций Windows. Является предшественником MD5.

Полное описание алгоритма: https://ru.bmstu.wiki/MD4_(Message_Digest_4)

4. Алгоритм MD5 (Message Digest 5)

MD5 (Message Digest 5) — 128-битный алгоритм хеширования, разработанный профессором Рональдом Л. Ривестом из Массачусетского технологического института (Massachusetts Institute of Technology, MIT) в 1991 году. Предназначен для создания «отпечатков» или «дайджестов» сообщений произвольной длины. Является улучшенной в плане безопасности версией MD4. Зная MD5-образ (называемый также MD5-хеш или MD5-дайджест), невозможно восстановить входное сообщение, так как одному MD5-образу могут соответствовать разные сообщения. Используется для проверки подлинности опубликованных сообщений путём сравнения дайджеста сообщения с опубликованным. Эту операцию называют «проверка хеша» (hashcheck). Описан в RFC 1321

Полное описание алгоритма: https://ru.bmstu.wiki/MD5_(Message_Digest_5)

5. Алгоритм ГОСТ Р 34.11-94

ГОСТ Р 34.11-94 — устаревший российский криптографический стандарт вычисления хеш-функции. Стандарт определяет алгоритм и процедуру вычисления хеш-функции для последовательности символов. Этот стандарт является обязательным для применения в качестве алгоритма хеширования в государственных организациях РФ и ряде коммерческих организаций.

До 2013 г. ЦБ РФ требовал использовать ГОСТ Р 34.11-94 для электронной подписи предоставляемых ему документов. С 1 января 2013 года заменён на ГОСТ Р 34.11-2012 «Стрибог».

Полное описание алгоритма:

https://ru.wikipedia.org/wiki/%D0%93%D0%9E%D0%A1%D0%A2_%D0%A0_34.11-94

Задание

Реализовать приложение с графическим интерфейсом, позволяющее выполнять следующие действия:

1. Шифровать и дешифровать текстовые и двоичные файлы с помощью потокового шифрования и генератора псевдослучайных чисел, разработанного в предыдущей лабораторной работе
2. Сохранять зашифрованные/дешифрованные данные в файл
3. Хешировать текстовый пароль, который используется при шифровании для инициализации генератора псевдослучайных чисел с помощью функции хеширования, указанной в варианте

Дополнительные требования к приложению

- Программа должна быть оформлена в виде удобной утилиты
- Программа должна обрабатывать файлы любого размера и содержания
- Должна быть предусмотрена возможность просмотра сгенерированного хеш-значения по введенному паролю
- Текст программы оформляется прилично (удобочитаемо, с описанием ВСЕХ функций, переменных и критических мест).
- В процессе работы программа ОБЯЗАТЕЛЬНО выдает информацию о состоянии процесса шифрования / дешифрования.
- Интерфейс программы может быть произвольным, но удобным и понятным (разрешается использование библиотек GUI)
- Среда разработки и язык программирования могут быть произвольными.

Примечание: Задание является дифференцированным по сложности. На оценку «Удовлетворительно» достаточно использовать готовую реализацию хеш-функции, на оценку «хорошо» и «отлично» необходимо реализовать свою функцию хеширования текстового пароля согласно варианту

Требования для сдачи лабораторной работы:

- Демонстрация работы реализованной вами системы.
- АВТОРСТВО
- Теория (ориентирование по алгоритмам и теоретическим аспектам методов тестирования)
- Оформление и представление письменного отчета по лабораторной работе, который содержит:
    - Титульный лист
    - Задание на лабораторную работу
    - Описание используемых алгоритмов шифрования
    - Листинг программы

Варианты задания

№	На оценку «Удовлетворительно»	На оценку «Хорошо»	На оценку «Отлично»
1	Любая готовая функция хеширования. Разрешается использовать готовые реализации хеш-функций, такие как MD 2/3/4/5/6, SHA-1, SHA-2 и т.п. для хеширования текстового пароля	MaHash7	SHA-1
2		MaHash9	SHA-2
3		MaHash2	MD-4
4		MaHash8	MD-5
5		MaHash4	ГОСТ Р 34.11-94
6		MaHash5	SHA-1
7		MaHash11	SHA-2
8		MaHash10	MD-4
9		MaHash3	MD-5
10		MaPrime	ГОСТ Р 34.11-94
11		MaPrime2d	SHA-1
12		MaHash8v64	SHA-2
13		MaHash4v64	MD-4
14		MaHashMR1	MD-5
15		MurmurHash2AM	ГОСТ Р 34.11-94