Белорусский государственный университет информатики и радиоэлектроники

Кафедра информатики

Лабораторная работа № 1

Симметричная криптография. Двойной и тройной DES.

Выполнила студент гр. 653502: СеребренниковВ.А.

Проверил ассистент КИ: Артемьев В. С.

Минск, 2019

Введение

Стандарт шифрования данных DES (DATA ENCRYPTION STANDARD) – блочный шифр с симметричными ключами, разработан Национальным Институтом Стандартов и Технологии (NIST – National Institute of Standards and Technology).

Для шифрования DES принимает 64-битовый открытый текст и порождает 64-битовый зашифрованный текст и наоборот, получив 64 бита зашифрованного текста, он выдает 64 бита расшифрованного. В обоих случаях для шифрования и дешифрования применяется один и тот же 56-битовый ключ.

Чтобы увеличивать криптостойкость DES, появляются несколько вариантов: double DES (2DES), triple DES (3DES).

Методы 2DES и 3DES основаны на DES, но увеличивают длину ключей (2DES — 112 бит, 3DES — 168 бит) и поэтому увеличивается криптостойкость.

Схема 3DES имеет вид $DES(k_3, DES(k_2, DES(k_1, M)))$, где k_1, k_2, k_3 ключи для каждого шифра DES. Это вариант известен как в EEE, так как три DES операции являются шифрованием. Существует 3 типа алгоритма 3DES:

- DES-EEE3: Шифруется три раза с 3 разными ключами.
- DES-EDE3: 3DES операции шифровка-расшифровка-шифровка с 3 разными ключами.
- DES-EEE2 и DES-EDE2: Как и предыдущие, за исключением того, что первая и третья операции используют одинаковый ключ.

В рамках лабораторной работы необходимо реализовать программные средства шифрования и дешифрования при помощи алгоритмов двойной и тройной DES.

Блок-схема алгоритма

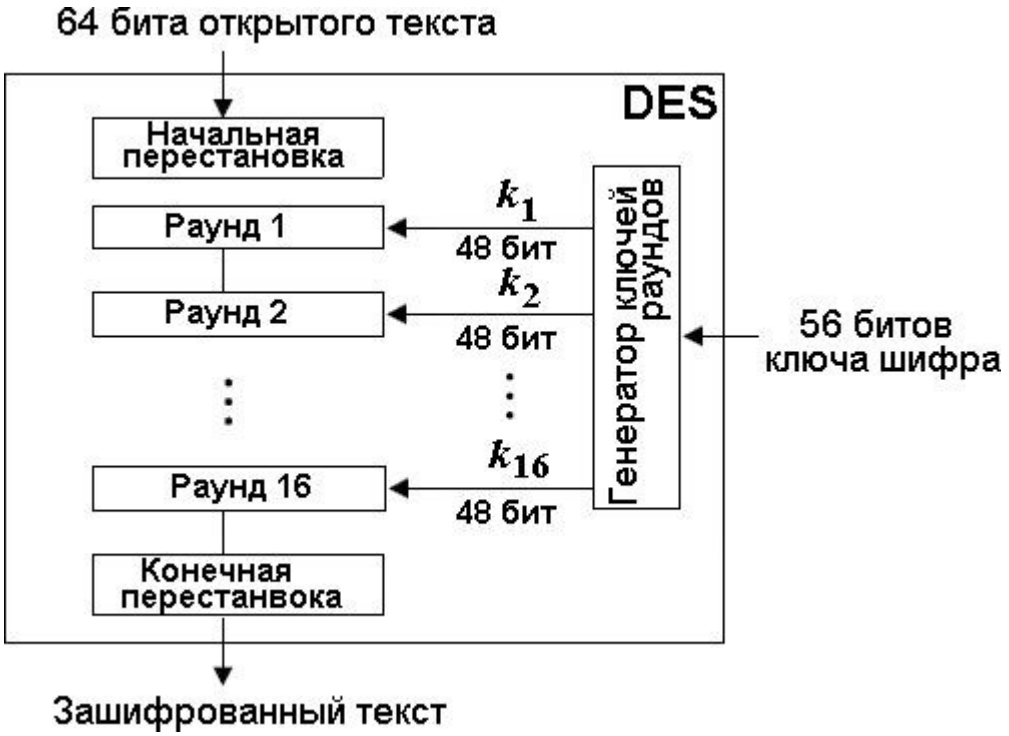

Рис.1. Блок-схема DES

Раунды DES

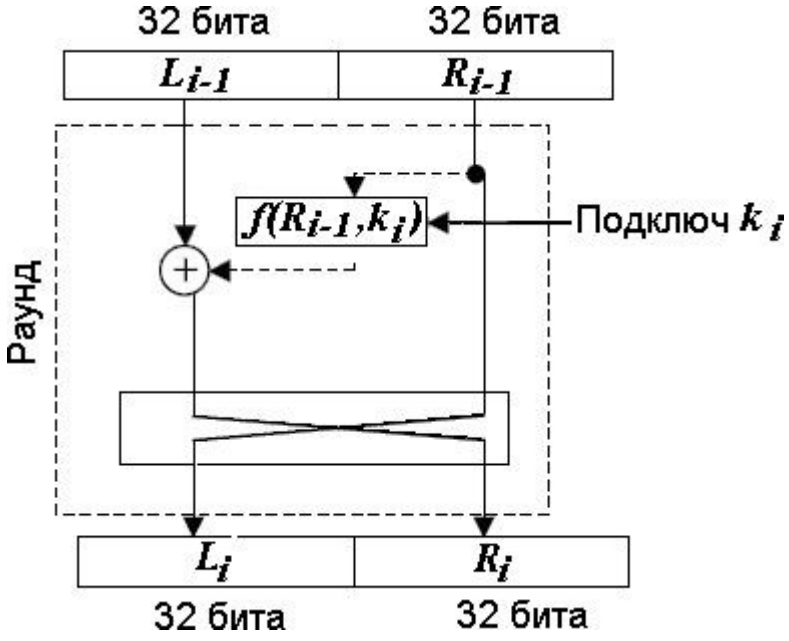

Рис.2. Раунды DES

Функция DES

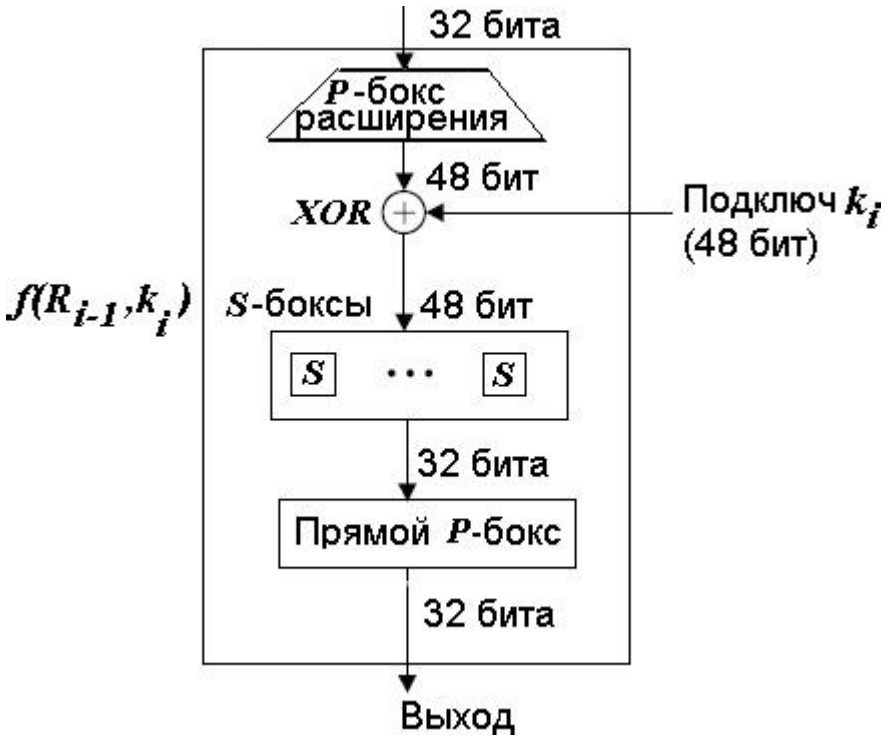

Рис.3. Функция DES

Пример работы программы

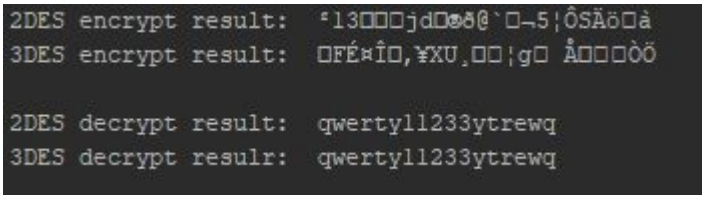

Рис.4. Пример работы

Код программы

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

typedef unsigned char ubyte;

#define KEY_LEN 8
typedef ubyte key_t[KEY_LEN];

const static ubyte PC1[] = {
  57, 49, 41, 33, 25, 17,  9,
   1, 58, 50, 42, 34, 26, 18,
  10,  2, 59, 51, 43, 35, 27,
  19, 11,  3, 60, 52, 44, 36,
  63, 55, 47, 39, 31, 23, 15,
   7, 62, 54, 46, 38, 30, 22,
  14,  6, 61, 53, 45, 37, 29,
  21, 13,  5, 28, 20, 12,  4
};

const static ubyte PC2[] = {
  14, 17, 11, 24,  1,  5,
   3, 28, 15,  6, 21, 10,
  23, 19, 12,  4, 26,  8,
  16,  7, 27, 20, 13,  2,
  41, 52, 31, 37, 47, 55,
  30, 40, 51, 45, 33, 48,
  44, 49, 39, 56, 34, 53,
  46, 42, 50, 36, 29, 32
};

const static ubyte IP[] = {
  58, 50, 42, 34, 26, 18, 10,  2,
  60, 52, 44, 36, 28, 20, 12,  4,
  62, 54, 46, 38, 30, 22, 14,  6,
  64, 56, 48, 40, 32, 24, 16,  8,
  57, 49, 41, 33, 25, 17,  9,  1,
  59, 51, 43, 35, 27, 19, 11,  3,
  61, 53, 45, 37, 29, 21, 13,  5,
  63, 55, 47, 39, 31, 23, 15,  7
};

const static ubyte E[] = {
  32,  1,  2,  3,  4,  5,
   4,  5,  6,  7,  8,  9,
   8,  9, 10, 11, 12, 13,
  12, 13, 14, 15, 16, 17,
  16, 17, 18, 19, 20, 21,
  20, 21, 22, 23, 24, 25,
  24, 25, 26, 27, 28, 29,
  28, 29, 30, 31, 32,  1
};

const static ubyte S[][64] = {
```

```
  {
    14,  4, 13,  1,  2, 15, 11,  8,  3, 10,  6, 12,  5,  9,  0,  7,
     0, 15,  7,  4, 14,  2, 13,  1, 10,  6, 12, 11,  9,  5,  3,  8,
     4,  1, 14,  8, 13,  6,  2, 11, 15, 12,  9,  7,  3, 10,  5,  0,
    15, 12,  8,  2,  4,  9,  1,  7,  5, 11,  3, 14, 10,  0,  6, 13
  },
  {
    15,  1,  8, 14,  6, 11,  3,  4,  9,  7,  2, 13, 12,  0,  5, 10,
     3, 13,  4,  7, 15,  2,  8, 14, 12,  0,  1, 10,  6,  9, 11,  5,
     0, 14,  7, 11, 10,  4, 13,  1,  5,  8, 12,  6,  9,  3,  2, 15,
    13,  8, 10,  1,  3, 15,  4,  2, 11,  6,  7, 12,  0,  5, 14,  9
  },
  {
    10,  0,  9, 14,  6,  3, 15,  5,  1, 13, 12,  7, 11,  4,  2,  8,
    13,  7,  0,  9,  3,  4,  6, 10,  2,  8,  5, 14, 12, 11, 15,  1,
    13,  6,  4,  9,  8, 15,  3,  0, 11,  1,  2, 12,  5, 10, 14,  7,
     1, 10, 13,  0,  6,  9,  8,  7,  4, 15, 14,  3, 11,  5,  2, 12
  },
  {
     7, 13, 14,  3,  0,  6,  9, 10,  1,  2,  8,  5, 11, 12,  4, 15,
    13,  8, 11,  5,  6, 15,  0,  3,  4,  7,  2, 12,  1, 10, 14,  9,
    10,  6,  9,  0, 12, 11,  7, 13, 15,  1,  3, 14,  5,  2,  8,  4,
     3, 15,  0,  6, 10,  1, 13,  8,  9,  4,  5, 11, 12,  7,  2, 14
  },
  {
     2, 12,  4,  1,  7, 10, 11,  6,  8,  5,  3, 15, 13,  0, 14,  9,
    14, 11,  2, 12,  4,  7, 13,  1,  5,  0, 15, 10,  3,  9,  8,  6,
     4,  2,  1, 11, 10, 13,  7,  8, 15,  9, 12,  5,  6,  3,  0, 14,
    11,  8, 12,  7,  1, 14,  2, 13,  6, 15,  0,  9, 10,  4,  5,  3
  },
  {
    12,  1, 10, 15,  9,  2,  6,  8,  0, 13,  3,  4, 14,  7,  5, 11,
    10, 15,  4,  2,  7, 12,  9,  5,  6,  1, 13, 14,  0, 11,  3,  8,
     9, 14, 15,  5,  2,  8, 12,  3,  7,  0,  4, 10,  1, 13, 11,  6,
     4,  3,  2, 12,  9,  5, 15, 10, 11, 14,  1,  7,  6,  0,  8, 13
  },
  {
     4, 11,  2, 14, 15,  0,  8, 13,  3, 12,  9,  7,  5, 10,  6,  1,
    13,  0, 11,  7,  4,  9,  1, 10, 14,  3,  5, 12,  2, 15,  8,  6,
     1,  4, 11, 13, 12,  3,  7, 14, 10, 15,  6,  8,  0,  5,  9,  2,
     6, 11, 13,  8,  1,  4, 10,  7,  9,  5,  0, 15, 14,  2,  3, 12
  },
  {
    13,  2,  8,  4,  6, 15, 11,  1, 10,  9,  3, 14,  5,  0, 12,  7,
     1, 15, 13,  8, 10,  3,  7,  4, 12,  5,  6, 11,  0, 14,  9,  2,
     7, 11,  4,  1,  9, 12, 14,  2,  0,  6, 10, 13, 15,  3,  5,  8,
     2,  1, 14,  7,  4, 10,  8, 13, 15, 12,  9,  0,  3,  5,  6, 11
  }
};

const static ubyte P[] = {
  16,  7, 20, 21,
  29, 12, 28, 17,
   1, 15, 23, 26,
   5, 18, 31, 10,
   2,  8, 24, 14,
  32, 27,  3,  9,
```

```
  19, 13, 30,  6,
  22, 11,  4, 25
};

const static ubyte IP2[] = {
  40,  8, 48, 16, 56, 24, 64, 32,
  39,  7, 47, 15, 55, 23, 63, 31,
  38,  6, 46, 14, 54, 22, 62, 30,
  37,  5, 45, 13, 53, 21, 61, 29,
  36,  4, 44, 12, 52, 20, 60, 28,
  35,  3, 43, 11, 51, 19, 59, 27,
  34,  2, 42, 10, 50, 18, 58, 26,
  33,  1, 41,  9, 49, 17, 57, 25
};

const static ubyte SHIFTS[] = {
  1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1
};

typedef struct {
  ubyte* data;
  int len;
} String;

/*
 * Transform a single nibble into a hex character
 *
 * in: a value < 0x10
 *
 * returns: the character that represents the nibble
 */
static char toHex(ubyte in) {
  if (0x00 <= in && in < 0x0A) {
    return '0' + in;
  }
  if (0x0A <= in && in <= 0x0F) {
    return 'A' + in - 0x0A;
  }
  return 0;
}

/*
 * Convert an array of bytes into a string
 *
 * ptr: the array of bytes
 * len: the number of bytes
 * out: a buffer allocated by the caller with enough space for 2*len+1 characters
 */
static void printBytes(const ubyte* ptr, int len, char* out) {
  while (len-- > 0) {
    *out++ = toHex(*ptr >> 4);
    *out++ = toHex(*ptr & 0x0F);

    ptr++;
  }
  *out = 0;
}
```

```
/*
 * Gets the value of a bit in an array of bytes
 *
 * src: the array of bytes to index
 * index: the desired bit to test the value of
 *
 * returns: the bit at the specified position in the array
 */
static int peekBit(const ubyte* src, int index) {
  int cell = index / 8;
  int bit = 7 - index % 8;
  return (src[cell] & (1 << bit)) != 0;
}

/*
 * Sets the value of a bit in an array of bytes
 *
 * dst: the array of bits to set a bit in
 * index: the position of the bit to set
 * value: the value for the bit to set
 */
static void pokeBit(ubyte* dst, int index, int value) {
  int cell = index / 8;
  int bit = 7 - index % 8;
  if (value == 0) {
    dst[cell] &= ~(1 << bit);
  }
  else {
    dst[cell] |= (1 << bit);
  }
}

/*
 * Transforms one array of bytes by shifting the bits the specified number of positions
 *
 * src: the array to shift bits from
 * len: the length of the src array
 * times: the number of positions that the bits should be shifted
 * dst: a bytes array allocated by the caller to store the shifted values
 */
static void shiftLeft(const ubyte* src, int len, int times, ubyte* dst) {
  int i, t;
  for (i = 0; i <= len; ++i) {
    pokeBit(dst, i, peekBit(src, i));
  }
  for (t = 1; t <= times; ++t) {
    int temp = peekBit(dst, 0);
    for (i = 1; i <= len; ++i) {
      pokeBit(dst, i - 1, peekBit(dst, i));
    }
    pokeBit(dst, len - 1, temp);
  }
}

/*
 * Calculates the sub keys to be used in processing the messages
```

```
 *
 * key: the array of bytes representing the key
 * ks: the subkeys that have been allocated by the caller
 */
typedef ubyte subkey_t[17][6]; /* 17 sets of 48 bits */
static void getSubKeys(const key_t key, subkey_t ks) {
  ubyte c[17][7];  /* 56 bits */
  ubyte d[17][4];  /* 28 bits */
  ubyte kp[7];
  int i, j;

  /* intialize */
  memset(c, 0, sizeof(c));
  memset(d, 0, sizeof(d));
  memset(ks, 0, sizeof(subkey_t));

  /* permute 'key' using table PC1 */
  for (i = 0; i < 56; ++i) {
    pokeBit(kp, i, peekBit(key, PC1[i] - 1));
  }

  /* split 'kp' in half and process the resulting series of 'c' and 'd' */
  for (i = 0; i < 28; ++i) {
    pokeBit(c[0], i, peekBit(kp, i));
    pokeBit(d[0], i, peekBit(kp, i + 28));
  }

  /* shift the components of c and d */
  for (i = 1; i < 17; ++i) {
    shiftLeft(c[i - 1], 28, SHIFTS[i - 1], c[i]);
    shiftLeft(d[i - 1], 28, SHIFTS[i - 1], d[i]);
  }

  /* merge 'd' into 'c' */
  for (i = 1; i < 17; ++i) {
    for (j = 28; j < 56; ++j) {
      pokeBit(c[i], j, peekBit(d[i], j - 28));
    }
  }

  /* form the sub-keys and store them in 'ks'
   * permute 'c' using table PC2 */
  for (i = 1; i < 17; ++i) {
    for (j = 0; j < 48; ++j) {
      pokeBit(ks[i], j, peekBit(c[i], PC2[j] - 1));
    }
  }
}

/*
 * Function used in processing the messages
 *
 * r: an array of bytes to be processed
 * ks: one of the subkeys to be used for processing
 * sp: output from the processing
 */
static void f(ubyte* r, ubyte* ks, ubyte* sp) {
```

```
  ubyte er[6]; /* 48 bits */
  ubyte sr[4]; /* 32 bits */
  int i;

  /* initialize */
  memset(er, 0, sizeof(er));
  memset(sr, 0, sizeof(sr));

  /* permute 'r' using table E */
  for (i = 0; i < 48; ++i) {
    pokeBit(er, i, peekBit(r, E[i] - 1));
  }

  /* xor 'er' with 'ks' and store back into 'er' */
  for (i = 0; i < 6; ++i) {
    er[i] ^= ks[i];
  }

  /* process 'er' six bits at a time and store resulting four bits in 'sr' */
  for (i = 0; i < 8; ++i) {
    int j = i * 6;
    int b[6];
    int k, row, col, m, n;

    for (k = 0; k < 6; ++k) {
      b[k] = peekBit(er, j + k) != 0 ? 1 : 0;
    }

    row = 2 * b[0] + b[5];
    col = 8 * b[1] + 4 * b[2] + 2 * b[3] + b[4];
    m = S[i][row * 16 + col]; /* apply table s */
    n = 1;

    while (m > 0) {
      int p = m % 2;
      pokeBit(sr, (i + 1) * 4 - n, p == 1);
      m /= 2;
      n++;
    }
  }

  /* permute sr using table P */
  for (i = 0; i < 32; ++i) {
    pokeBit(sp, i, peekBit(sr, P[i] - 1));
  }
}

/*
 * Processing of block of the message
 *
 * message: an 8 byte block from the message
 * ks: the subkeys to use in processing
 * ep: space for an encoded 8 byte block allocated by the caller
 */
static void processMessage(const ubyte* message, subkey_t ks, ubyte* ep) {
  ubyte left[17][4];  /* 32 bits */
  ubyte right[17][4]; /* 32 bits */
```

```
  ubyte mp[8];        /* 64 bits */
  ubyte e[8];         /* 64 bits */
  int i, j;

  /* permute 'message' using table IP */
  for (i = 0; i < 64; ++i) {
    pokeBit(mp, i, peekBit(message, IP[i] - 1));
  }

  /* split 'mp' in half and process the resulting series of 'l' and 'r */
  for (i = 0; i < 32; ++i) {
    pokeBit(left[0], i, peekBit(mp, i));
    pokeBit(right[0], i, peekBit(mp, i + 32));
  }
  for (i = 1; i < 17; ++i) {
    ubyte fs[4]; /* 32 bits */

    memcpy(left[i], right[i - 1], 4);
    f(right[i - 1], ks[i], fs);
    for (j = 0; j < 4; ++j) {
      left[i - 1][j] ^= fs[j];
    }
    memcpy(right[i], left[i - 1], 4);
  }

  /* amalgamate r[16] and l[16] (in that order) into 'e' */
  for (i = 0; i < 32; ++i) {
    pokeBit(e, i, peekBit(right[16], i));
  }
  for (i = 32; i < 64; ++i) {
    pokeBit(e, i, peekBit(left[16], i - 32));
  }

  /* permute 'e' using table IP2 ad return result as a hex string */
  for (i = 0; i < 64; ++i) {
    pokeBit(ep, i, peekBit(e, IP2[i] - 1));
  }
}

/*
 * Encrypts a message using DES
 *
 * key: the key to use to encrypt the message
 * message: the message to be encrypted
 * len: the length of the message
 *
 * returns: a paring of dynamically allocated memory for the encoded message,
 *          and the length of the encoded message.
 *          the caller will need to free the memory after use.
 */
String encrypt(const key_t key, const ubyte* message, int len) {
  String result = { 0, 0 };
  subkey_t ks;
  ubyte padByte;
  int i;

  getSubKeys(key, ks);
```

```
  padByte = 8 - len % 8;
  result.len = len + padByte;
  result.data = (ubyte*)malloc(result.len);
  memcpy(result.data, message, len);
  memset(&result.data[len], padByte, padByte);

  for (i = 0; i < result.len; i += 8) {
    processMessage(&result.data[i], ks, &result.data[i]);
  }

  return result;
}

/*
 * Decrypts a message using DES
 *
 * key: the key to use to decrypt the message
 * message: the message to be decrypted
 * len: the length of the message
 *
 * returns: a paring of dynamically allocated memory for the decoded message,
 *          and the length of the decoded message.
 *          the caller will need to free the memory after use.
 */
String decrypt(const key_t key, const ubyte* message, int len) {
  String result = { 0, 0 };
  subkey_t ks;
  int i, j;
  ubyte padByte;

  getSubKeys(key, ks);
  /* reverse the subkeys */
  for (i = 1; i < 9; ++i) {
    for (j = 0; j < 6; ++j) {
      ubyte temp = ks[i][j];
      ks[i][j] = ks[17 - i][j];
      ks[17 - i][j] = temp;
    }
  }

  result.data = (ubyte*)malloc(len);
  memcpy(result.data, message, len);
  result.len = len;
  for (i = 0; i < result.len; i += 8) {
    processMessage(&result.data[i], ks, &result.data[i]);
  }

  padByte = result.data[len - 1];
  result.len -= padByte;
  return result;
}

/*
 * Convienience method for showing the round trip processing of a message
 */
void driver(const key_t keys[], const ubyte* message, int len) {
```

```
  String encoded, decoded;
  char buffer[128];

  printBytes(keys[0], KEY_LEN, buffer);
  printf("Key     : %s\n", buffer);

  printBytes(message, len, buffer);
  printf("Message : %s\n", buffer);

  encoded = encrypt(keys[0], message, len);
  printBytes(encoded.data, encoded.len, buffer);
  printf("Encoded : %s\n", buffer);

  printBytes(keys[1], KEY_LEN, buffer);
  printf("Key     : %s\n", buffer);

  encoded = encrypt(keys[1], encoded.data, encoded.len);
  printBytes(encoded.data, encoded.len, buffer);
  printf("Encoded : %s\n", buffer);

  printBytes(keys[2], KEY_LEN, buffer);
  printf("Key     : %s\n", buffer);

  encoded = encrypt(keys[2], encoded.data, encoded.len);
  printBytes(encoded.data, encoded.len, buffer);
  printf("Encoded : %s\n", buffer);

  decoded = decrypt(keys[2], encoded.data, encoded.len);
  printBytes(decoded.data, decoded.len, buffer);
  printf("Decoded : %s\n\n", buffer);

  decoded = decrypt(keys[1], decoded.data, decoded.len);
  printBytes(decoded.data, decoded.len, buffer);
  printf("Decoded : %s\n\n", buffer);

  decoded = decrypt(keys[0], decoded.data, decoded.len);
  printBytes(decoded.data, decoded.len, buffer);
  printf("Decoded : %s\n\n", buffer);

  /* release allocated memory */
  if (encoded.len > 0) {
    free(encoded.data);
    encoded.data = 0;
  }
  if (decoded.len > 0) {
    free(decoded.data);
    decoded.data = 0;
  }
}

int main() {
  const key_t keys[] = {
    {0x13, 0x34, 0x57, 0x79, 0x9B, 0xBC, 0xDF, 0xF1},
    {0x0E, 0x32, 0x92, 0x32, 0xEA, 0x6D, 0x0D, 0x73},
    {0x0E, 0x32, 0x92, 0x32, 0xEA, 0x6D, 0x0D, 0x73}
  };
```

```
  const ubyte text[] = { 0x42, 0x65, 0x73, 0x74, 0x20, 0x6C, 0x61, 0x62, 0x6F, 0x72, 0x61,
0x74, 0x6F, 0x72, 0x79, 0x20, 0x6F, 0x6F, 0x74, 0x68, 0x65, 0x72, 0x20, 0x74, 0x68, 0x61, 0x6E,
0x20, 0x76, 0x61, 0x73, 0x65, 0x6C, 0x69, 0x6E, 0x65, 0x0D, 0x0A };
  int len;

  len = sizeof(text) / sizeof(ubyte);
  driver(keys, text, len);

  return 0;
}
```

Вывод

Сам по себе алгоритм DES уже не является криптостойким, т.к. силами современной вычислительной техники его вполне можно взломать. С попыткой увеличения криптостойкости, используя двойной DES, была выявлена слабость, называемая “встреча посередине”, что тоже делает алгоритм уязвимым. Что касается тройного DES, то на данный момент его можно считать криптостойким. Для успешной атаки на 3DES потребуется около 2^{32} бит известного открытого текста, 2^{113} шагов, 2^{90} циклов DES-шифрования и 2^{88} бит памяти. На данный момент это непрактично, и, по оценкам НИСТ, алгоритм с выбором трех различных ключей должен остаться надежным до 2030-х.