A System for Assisting Analysis of Some Block Ciphers NES/DOC/TEC/WP2/007/2 *

Arieh Bibliowicz, vainolo@vipe.technion.ac.il
Pnina Cohen pninac@cs.technion.ac.il
Eli Biham, biham@cs.technion.ac.il

February 20, 2003

1 Introduction

With the size and complication of cipher algorithms growing every day, an automatic analysis program is needed so that more time can be spent on deeper, more complicated analysis. In this document we design a system to perform automatic but restricted analysis of some block ciphers. The system is based on a formal language that can be easily used to define many existing block ciphers (e.g., DES). Given a definition of a cipher in this language, we execute an "automatic cipher analyzer" which can interpret the defined cipher, and do the routine analysis.

The paper is organized as follows: In Section 2 we describe the types of analysis that the system is capable of. In Section 3 we describe the language itself. In Section 4 we give details of some of the program limitations. In Section 5 we describe how to call the analyzing programs and in Section 6 we summarize this document. Finally, the appendix contains of an example of the definition of DES in the proposed language.

*The information in this document is provided as is, and no warranty is given or implied that the information is fit for any praticular purpose. The user thereof uses the information at its sole risk and liability.

2 Types of Analysis

Our tool treats block ciphers in several levels. In the first level statistical properties of the basic building blocks are found. In the second level properties of complete rounds are found, and in the third level properties of several rounds. In the fourth level cryptanalysis of the full cipher, which enables us to discover the key, is developed.

This Section is organized according to the components of a basic block cipher. For each component the types of analysis that can be performed on it are described.

2.1 Building Blocks

2.1.1 Selection

A *selection* is a common generalization of a permutation, an expansion, a permuted-choice, and any other operation that receives a vector of bits and rearranges them in a new vector that may be of the same size, larger or smaller.

The following analysis types are available for Selections:

1. For selections with input and output of the same size:

- **1-1**: Checks if every bit in the input vector appears in the output vector (exactly once).
- **cycles**: Checks the period of all the cycles in the permutation and how many bits enter each cycle. The period of a cycle is defined by the number of times the permutation is applied until a bit returns to its original location.
- **fix-points**: Which (if any) of the elements in the input vector remain in the same location in the output vector.

2. For all selections:

- **balancedness**: Checks how many times each input bit appears in the output vector, and analyze bits distribution.

2.1.2 S-Box

An S-box is a nonlinear transformation which is implemented as a lookup table.

1. For S-boxes with input and output of the same size:

- **1-1**: Check that the S-box is bijective.
- **cycles**: Check the period length of the cycles appearing in the S-box and how many bits enter each cycle.
- **fix-points**: Which inputs have the same value as the output.

2. For all S-boxes:
   - **balancedness**: Check how many times each possible output appears, and analyze outputs distribution.
   - **Difference Distribution Table**: Computes the difference distribution table of the S-box for differential cryptanalysis.
   - **Linear Approximation Table**: Computes the linear approximation table of the S-box for linear cryptanalysis.
   - **Rotated relations between input and output bits**
   - **Relations between the bit functions of the output**
   - **Quadratic Equations**: Finds quadratic relations between the output and the input bits of an S-box.
   - An optional extension may allow analysis with non-XOR differences (e.g., addition or subtraction), and with different types of differences in the input and the output of an S-box (e.g., XOR input difference and subtraction output difference).

2.2 One Round

Round is the basic component of a DES-like block cipher. Such a cipher can be divided into 3 parts: 1) initialization, 2) round iteration, 3) Finalization. The main part of the cryptographic strength is contributed by the round iteration, therefore an extensive analysis of its properties is essential for a complete analysis of the cipher.

The following analysis is available for Rounds:

- S-box dependency table, showing which output bits of every S-box affect which input bits of the S-boxes in the following round.
- An example for linear and differential one-round characteristics and its probability.

2.3 Several Rounds

- Most important 3-rounds characteristics and their probabilities. Available only for Feistel ciphers.
- Suggestions for good 5-rounds characteristics and their probabilities. Available only for Feistel ciphers.
- Most important $X \rightarrow 0$ iterative characteristics (similar to the iterative characteristic of DES). Available only for Feistel ciphers.
- In the future we intend to be able to find additional types of iterative characteristics.
- Best differential and linear characteristics for any number of rounds (up to the the number of the rounds iterations) built using Matsui Algorithm. Available only for Feistel ciphers and ciphers in which all round input bits enter the S-boxes each round. Currently only the differential option works for ciphers of the second type.

2.4 Cryptanalysis of the Full Cipher

At this point we do not plan to apply complete automatic cryptanalysis.

3 Language Definition

The language used to define ciphers is described in this Section. It is designed to be as simple as possible, but contains some constraints designed to facilitate the automatic analysis. The syntax of the language is as follows:
Notation:

- Words surrounded by square braces ('[' and ']') are optional.
- Words and definitions followed by * can be added 0 or more times, with a space between two appearances.
- Words and definitions followed by $^{+}$ can be added 1 or more times, with a space between two appearances.
- Words and definitions followed by $^{++}$ can be added 1 or more times, with a comma between two appearances.
- A list of words separated by '|' means that in this place one word must be used, and it must be taken from one of the options in the list.

3.1 Identifiers

An identifier is a string that begins with a letter and is composed of letters, digits and '_'s (underscore). Letters can be upper or lower case, and the interpretation is case sensitive.

Example: `foo, b_a_r, My1stIdentifier` are valid identifiers.

Example: `123go, -minus` are not valid identifiers.

3.2 Global Flags

This option affects all the other definitions. We define only one flag at this stage. This flag controls the endianess of the definition. The default is `little_endian`. It can be changed by writing `big_endian` or `inv_big_endian` at the beginning of the file.

These three types of endianess work as follows:

- **big_endian**: Count bits from 1, starting with the MSB.
  1,2,3,4,5,6,7,8, 9,10,11,12,13...
- **inv_big_endian**: Count bits from 0, starting with the LSB.
  ...12,11,10,9,8, 7,6,5,4,3,2,1,0
- **little_endian**: Count bits from 0, starting with the LSB of each byte. The first byte to be counted is the one to the left.
  7,6,5,4,3,2,1,0, ...12,11,10,9,8

3.3 Comments

Everything that follows a '#' sign in a line is considered a comment.

3.4 Constants

A numeric value that is set hard coded in the definition, and is not part of the definition of an operation or a range is considered a constant. Constants can be written in decimal (e.g. 35) or in hexadecimal (e.g. 0x23).

3.5 Variable Declaration

Variables can be declared at the beginning of some constructs, inside a **Local** block (defined below). A variable declaration consists of an identifier and a size (in bits). In the language, a specific bit inside a variable can only receive a value once (apart from a variable inside a loop, as explained below), but

many assignments containing the variable's name can appear in the definition (see Appendix A). This is done to facilitate the analysis process. In the following definitions, the definition of a variable, including a ';' at the end, is notated as `vardef`.
Syntax:

```
bitvector(num) ident;
```

Example: `bitvector(48) foo;` declares a variable named **foo** containing **48** bits.

3.6 Assignments

Assignment is the basic part from which complex operations are constructed. It is an operation that assigns the value of a constant, a variable, a group of bits from a variable, or an operation call into another variable or a group of bits from it.

We will give the definition of an assignment in three steps. First, we will show the definition of the possible left hand values in an assignment; those values will be referred to as `lvalues`. Then we will show the definition of the possible right hand values in an assignment; those values will be referred to as `rvalues`. Finally the complete definition for assignments will be given. Here we show the syntax of all possible `lvalues`. The value of a given `lvalue` is irrelevant since it is about to be changed. The size of each `lvalue` is defined as follows:

variable A bit-vector, with the same length as the variable.

variable with range A bit-vector, whose length is given by a starting and an ending bits indexes, that are included in the range as well. Note that the range is interpreted differently depending on the endianess of the definition.

subkey The next sub-key to be calculated, whose size is determined solely by the rvalue assigned to it. Sub-keys can be calculated only in the special key-scheduling function; therefore this type of `lvalue` is used only there.

Syntax:

```
ident
ident(rangeStart - rangeEnd)
subkey
```

Here we show the syntax of all possible `rvalues`. The interpretation for each `rvalue` value and size is defined as follows:

constant A bit-vector containing the hexadecimal or decimal representation of a constant. A constant size is defined to be its length in bits. At this stage the program can only parse constants whose value is less than 2^{32}.

variable A bit-vector, with the same length as the variable, containing its current value.

variable with range A bit-vector, whose length and value are determined by a sub-group of bits taken from a given variable. This sub-group, or range, is given by a starting and an ending bits indexes, that are included in the range as well. The range is interpreted differently depending on the endianess of the definition.

subkey A request for the next sub-key, whose value and size are calculated in the special key-scheduling function. Obviously this type of `rvalue` can not used in the key-scheduling function itself.

operation calls Calls to operations defined in the definition file (e.g. *low-level-functions*) or in the program (e.g., S-boxes, functions, etc...). In order to invoke an operation call, one must give an operation identifier and a list of parameters, in a comma separated list (called `argList`), inside parenthesis. Each parameter belongs to one of the five types of `rvalues`. The number of parameters to give is defined in the operation definition, or is formerly known, as in the case of *low-level-functions*, S-boxes, etc. The value of this type is the returned value of the operation call, and of the same size.

Syntax:

```
num
ident
ident(rangeStart - rangeEnd)
subkey
ident(argList)
```

Now we give the complete syntax of a statement. The size of the `lvalue` must match the size of the `rvalue`, with 3 exceptions:

- A constant assigned to a variable or to an argument of a function. If the variable/argument size is greater than the constant size, this constant is 'padded' with zeros till it reaches the variable/argument size.

- Returned value size of a *low-level-function* call. The way *low-level-functions* are handled will be discussed in details later, but as a rule the size of `lvalue` in the assignment defines the size of the returned value of the *low-level-functions* call.

- Assignment to sub-key. Here the size of the `lvalue` is determined by the `rvalue`. Problems may arise when the `rvalue` is a constant or a *low-level-function* call. In this case, a special padding function can be used. This function will be discussed in more details later.

Syntax:

```
lvalue = rvalue;
```

Example: `foo = XOR(var1, var2);` assigns `foo` the value of $var1 \oplus var2$.
Example: `temp = foo(1-4);` assigns to `temp` the value of bits 1 to 4 of foo. If we look at the bit-vector `foo` from left to right, and the notation is `inv_big_endian`, these are the 4 bits that come before the LSB. In this case `foo` must be at least 5 bits long. If the notation is `big_endian`, these are the first 4 bits beginning with the MSB, and `foo` must be at least 4 bits long. At any case `temp` must be exactly 4 bits long. *Example*: `out(1-4) = foo;` assigns `foo` to bits 1 to 4 of variable `out`. Here `foo` must be exactly 4 bits long.
Example: `bar = s1(f(left));` assigns to `bar` the result of the operation `s1` applied on the output of operation `f` called with `left` as a parameter.
In the following definitions, the assignment operation, including a ';' at the end, is notated as `assignment`.

3.7 Low-Level Functions

The language incudes a number of low-level functions that can be used in any statement. The length of the result of the operation depends on the the variable it is assigned to. In any case, the length of the parameters for low-level functions will not be longer than the length of the left variable.

1. **AND, OR, XOR**: bitwise boolean operations of two variables.

   Syntax:

   ```
   AND(rvalue, rvalue)
   OR(rvalue, rvalue)
   XOR(rvalue, rvalue)
   ```

Example: `out = XOR(left,25);` computes `left` $\oplus$ 25 and assigns the result to `out`.
Example: `out = XOR(5,8);` computes $8 \oplus 5$ and assigns it to `out`.

2. **ADD, SUBS, MULT**: mathematical functions. They interpret the first two variables as **unsigned integer** and compute the result, reducing it **modulo** 2 to the power of the given third parameter. At this stage, the maximal modulo_size is 32 or less. The `lvalue` size should be at least as big as the given modulo_size.

   Syntax:

   ```
   ADD(rvalue, rvalue, num)
   SUBS(rvalue, rvalue, num)
   MULT(rvalue, rvalue, num)
   ```

   Example: `out = ADD(4,that,32);` computes `4` + `that` mod 2^{32}.

3. **ROL, ROR**: rotate the first parameter left or right by a number of bits given in the second parameter. Before the rotation, the first parameter is padded to match the size given by the third parameter. In any case, the size of the first parameter should not be greater than size indicated by the third parameter, and the size of the third parameter should not be greater than the size of the `lvalue`.

   Syntax:

   ```
   ROL(rvalue, rvalue, num)
   ROR(rvalue, rvalue, num)
   ```

   Example: `out = ROL(in, 8, 6);` `in` is padded to the length of 6 and then rotated 8 bits left. The result is assigned to `out`.

4. **SHIFTL, SHIFTR**: shift left or shift right the first parameter a number of bits given in the second parameter.

   Syntax:

   ```
   SHIFTL(rvalue, rvalue)
   SHIFTR(rvalue, rvalue)
   ```

   Example: `loose = SHIFTL(win,16);` shifts `win` by 16 bits to the left, and assigns the result to `loose`.

3.8 The Pad Function

This function enables the user to directly increase an `rvalue`'s length in bits by padding int with zeros on its left end. This option becomes handy when a specific size of an `rvalue` is desired, for example, when a new `subkey` is created.
The `rvalue` to be padded is given as the first parameter of the `pad function`; its desired new size is given as the second parameter. Obviously the given size could not be smaller than the size of the `rvalue` to be padded, and must match the `lvalue`'s size.

Syntax:

```
PAD(rvalue, num)
```

Example: `subkey = PAD(AND(key, 7), 5);` defines a `subkey` of the size 5 by padding the result of AND function over the `key` and 7.

3.9 Definitions

The language's basic use is to define cipher components and use their definition to define ciphers. There are many components to a cipher but they were reduced to a small list to keep definitions simple. Possible definitions are:

1. **Selection**: Declaration of a Selection includes the number of input and output bits and a representation of the new order given in an array. The array is interpreted as if one is looking at the output vector, and the numbers indicate the index on the source vector where the value should be taken from (this eliminates possibly different interpretations). The input index is interpreted according to the endianess of the definition. The number of entries given must match the size of the output.

   Syntax:

   `Selection ident In num Out num { num`$^{++}$ `}`

   Example: `Selection Exp In 4 Out 8 { 0, 1, 2, 1, 2, 3, 0, 3 };` defines a selection named `Exp` taking 4 input bits and returning 8 bits in the given order. In this case, we are using `little_endian` or `inv_big_endian`.

   Example: `expanded = Exp(input);` assigns the result of the selection over `input` to the variable `expanded`. In both `little_endian` and `inv_big_endian` endianess bit 0 of `expanded` will get bit 3 of `input`, bit 1 will get bit 0 and so on.

2. **S-Box**: Declaration of an S-box includes an identifier, the number of input bits, the number of output bits and the S-box represented as an array. which is interpreted simerly to a C-style array. The arrays elements are counted from left to right and the first element is the output for input 0, the second element is the output for input 1 and so on.

   Syntax:

   ```
   SBox ident In num Out num { num++ };
   ```

   Example: `S-box S1 In 3 Out 4 { 0, 2, 4, 7, 6, 1, 5, 3 };` defines an S-box named `S1` taking 3 input bits.

   Example: `res = S1(mix);` assigns the result of the S-box over the variable `mix` into the variable `res`. Suppose `mix = 3`, then `res` the value 7 is assigned to res.

3. **Function**: a function is a transformation that can have many inputs but has only one output. Function definition is composed of local variables declarations (optional) and a series of statements. It requires an identifier, the sizes of the inputs and the size of the output. The input and output variables receive implicit names, input variables identified as `input1, input2, ...` from left to right, and the output variable is identified as `output`.

   Syntax:

   ```
   Function ident In num++ Out num
   Local [{ vardef+ }]
   Begin
     assignment+
   End
   ```

   example:

   ```
   Function F In 4, 8 Out 8
   Local { bitvector(8) temp;}
   Begin
     temp = Exp(input1);
     output = XOR(temp(1-4), input2(1-4));
   End
   ```

defines a function named F that takes two inputs, 4 and 8 bits long and returns an 8 bits long output. The function declares a local variable named `temp` that is 8 bits long, applies the selection `Exp` to the first input, XORs `temp` with the second input and returns the result.

4. **Sbox in a Function Form**: Sometimes it's easier to define an S-box like a function. Instead of the an array of outputs, we're given a way to calculate them. This kind of special S-box is a function only in as way of definition. It can be analyzed as an other S-box.
   A function defining an S-box takes only one input and returns one output. Here the input is referenced as `input` and the output as `output`.

   Syntax:

   ```
   FSBox ident In num Out num
   Local [{ vardef+ }]
   Begin
     assignment+
   End
   ```

5. **Round**: Round is the main building block of a cipher. A round has only one input and a key, and the output has the same size as the input. The round definition requires an identifier and the size of the input and of the key. Here the input is referenced as `input`, the key as `key` and the output as `output`. As in Function, the user can also declare local variables inside the round. Currently only one definition of round is available.

   Syntax:

   ```
   Round ident In num Key num
   Local [{ vardef+ }]
   Begin
     assignment+
   End
   ```

   example:

   ```
   Round R In 8 Key 8
   Local { bitvector(4) temp1; }
   Begin
     temp1 = S1(key(1-3));
     output = F(temp1, input);
   End
   ```

defines a round named `R` with 8 input bits and 8 key bits. It declares a variable `temp1` of 4 bits of length, applies the S-box operation `S1` to three bits of the key and assigns the result to `temp1`, then applies the `F` operation on `temp1` and `input` and returns the result

6. **Key scheduling**: This is a special kind of function designed to calculate all the sub-keys needed in the cipher. Key scheduling definition is composed of local variables declarations (optional) and a series of statements. It requires only one input, the key size. The key is referenced as `key`. This function has no identifier or returned value and is never explicity called. The sub-keys are created by assigning `rvalue` to a 'subkey' `lvalue`. The size and value of each sub-key is defined solely by its `rvalue` in the assignment. The sub-keys will be returned at the same order they were created. Key scheduling definition adds a new keyword, `Loop`, that states looping over stated code a defined number of times. Inside a loop body is the only place where a variable can be assigned a value more than once. First we define the syntax of the loop, which becomes another type of `assignment` when used in a key scheduling definition.

   Syntax:

   ```
   Loop(num)
   Begin
     assignment+
   End
   ```

   The syntax of key scheduling is:

   Syntax:

   ```
   Keyscheduling Key num
   Local [{ vardef+ }]
   Begin
     assignment+
   End
   ```

   example:

   ```
   Keyscheduling Key 8
   Local bitvector(4) i;
   Begin
   ```

```
  subkey = key(1-4);
  i = 0;
  Loop(5)
  Begin
    subkey = F(i, key);
    i = ADD(i, 1, 4);
  End
End
```

defines a key scheduling function that takes a key that is 8 bits long. The function declares a local variable named `i` that is 4 bits long, applies to the first subkey the value of 4 bits of the `key` and initializes `i` to zero. Then it loops five times and assigns to the next five sub-keys the result of `F` function called with the parameters `i` and `key`. `i` value is increased by 1 in each iteration. Notice that the first subkey is 4 bits long and the other five are 8 bits long.

7. **Cipher**: A cipher is a repetition of the round a number of times, with some initialization and finalization. It has only one input, the plaintext. The output, the ciphertext, has the same size as the input. The cipher definition requires an identifier and the size of the plaintext. Here the input is referenced as `input` and the output as `output`. As in key scheduling definition, a loop is considered one of the possible types of `assignment` in cipher definition.

   Syntax:

   ```
   Cipher ident In num
   Local [{ vardef+ }]
   Begin
     assignment+
   End
   ```

   example:

   ```
   Cipher C In 8
   Local { bitvector(8) temp1; }
   Begin
     temp1 = XOR(input, subkey);
     Loop(5)
     Begin
       temp1 = R(temp1, subkey);
   ```

```
    End
    output = temp1;
End
```

defines a cipher named `C` with 8 input bits. It declares one local variables of 8 bits, `temp1`, XORS the first subkey with the plaintext and assigns the result to `temp1`. Then it repeats five times the round operation, `R`, with `temp1` and the next five subkeys as parameters, and then returns the last result from the round.

4 Further Instructions

Analyzer program's grammar allows a relatively liberated programming style. Most of the analyzer options will run smoothly as long as the cipher-file is written as instructed. Only with the round analysis option things get more complicated. Using this option may cause analyzer program to return warnings and requests for a simpler cipher code. Also during round analysis the analyzer considers only the declared round function and the functions that are called by it. And so if, for example, some of the S-Boxes are called from the cipher itself instead of from the round function, these calls are ignored during round analysis.

Advanced S-Boxes analysis and round analysis may take long time to complete.

5 Calling the Program

The program is called from the command line, and requires a file with the cipher definition. This file is referred to as 'cipher-file'. The syntax of all possible calls to analyzer is as follows:

`analyzer -sbox analysisType` *`cipher-file`*

`analyzer -sel` *`cipher-file`*

`analyzer -round [num | itr | matsui[diff | linear]]`
`cipher-file`

`analyzer -single [-with-rounds] input-file output-file`
`cipher-file`

`analyzer -test-vectors [-with-rounds] output-file` *`cipher-file`*

`analyzer` *`cipher-file`*
The last option preforms a short version of all of the analysis options.

- `-sbox analysisType`: analyze all S-boxes in the definitions. The types of analysis available are:
  - `basic`: performs the 1-1, cycles, fix-points and balancedness analysis.
  - `advanced`: returns the different distribution and the linear approximation tables, and computes relations equations between input and output bits and in between the functions of the output bits.
  - `quad`: calculates the S-boxes quadratic equations.
  - `complete`: performs all of the above.
- `-sel`: analyze all Selections in the definitions. The Types of analyzes performed are: 1-1, cycles, fix-points, balancedness.
- `-round`: analyze a round. The types of analysis available are the S-boxes dependencies table and 1-5 rounds characteristics, iterative characteristics and Matsui Algorithm.
- `num`: number of rounds of the linear and differential characteristics, when not using Matsui Algorithm.
- `itr`: linear and differential iterative characteristics.
- `matsui`: runs Matsui Algorithm.
- `diff`: runs Matsui Algorithm to find all best diffrential characteristics.
- `linear`: runs Matsui Algorithm to find all best linear characteristics.
- `-single`: run analyzer once to compute the ciphertext of given plaintext and key.
- `-test-vectors`: create test-vectors, with the same data as in the NESSIE test suite.
- `-with-rounds`: show intermediate values for each round.
- `input-file`: name of a file that includes specific plaintext and key for the cipher. The form the file should be written is:
  key: key in hex
  plain: plaintext in hex

- `output-file`: name of a file to which the results (ciphertext with or without rounds intermediate values) will be written.

6 Summary

This document describes the structure of an automatic cipher analyzing program. A cipher definition language is described to define the internal representations of ciphers. The program can analyze components of ciphers or compute test-vectors with or without intermediate values for the cipher. This program makes the design and analysis of ciphers easier, faster and more reliable, thus helping the achievement of more secure ciphers.

A Definition of DES

```
big_endian

  Selection P In 32 Out 32
     {  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  };

  Selection E In 32 Out 48
     {  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  };

  Selection IP In 64 Out 64
     {  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  };

  Selection FP In 64 Out 64
```

```
 {  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  };

SBox s1 In 6 Out 4
 {  14,   0,   4,  15,  13,   7,   1,    4
     2,  14,  15,   2,  11,  13,   8,    1
     3,  10,  10,   6,   6,  12,  12,  11,
     5,   9,   9,   5,   0,   3,   7,   8,
     4,  15,   1,  12,  14,   8,   8,   2,
    13,   4,   6,   9,   2,   1,  11,   7,
    15,   5,  12,  11,   9,   3,   7,  14,
     3,  10,  10,   0,   5,   6,   0,   13  };

SBox s2 In 6 Out 4
 {  15,   3,   1,  13,   8,   4,  14,   7,
     6,  15,  11,   2,   3,   8,   4,  14,
     9,  12,   7,   0,   2,   1,  13,  10,
    12,   6,   0,   9,   5,  11,  10,   5,
     0,  13,  14,   8,   7,  10,  11,   1,
    10,   3,   4,  15,  13,   4,   1,   2,
     5,  11,   8,   6,  12,   7,   6,  12,
     9,   0,   3,   5,   2,  14,  15,    9  };

SBox s3 In 6 Out 4
 {  10,  13,   0,   7,   9,   0,  14,   9,
     6,   3,   3,   4,  15,   6,   5,  10,
     1,   2,  13,   8,  12,   5,   7,  14,
    11,  12,   4,  11,   2,  15,   8,   1,
    13,   1,   6,  10,   4,  13,   9,   0,
     8,   6,  15,   9,   3,   8,   0,   7,
    11,   4,   1,  15,   2,  14,  12,   3,
     5,  11,  10,   5,  14,   2,   7,   12  };

SBox s4 In 6 Out 4
```

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

SBox s5 In 6 Out 4
 {   2,  14,  12,  11,   4,   2,   1,  12,
     7,   4,  10,   7,  11,  13,   6,   1,
     8,   5,   5,   0,   3,  15,  15,  10,
    13,   3,   0,   9,  14,   8,   9,   6,
     4,  11,   2,   8,   1,  12,  11,   7,
    10,   1,  13,  14,   7,   2,   8,  13,
    15,   6,   9,  15,  12,   0,   5,   9,
     6,  10,   3,   4,   0,   5,  14,    3  };

SBox s6 In 6 Out 4
 {  12,  10,   1,  15,  10,   4,  15,   2,
     9,   7,   2,  12,   6,   9,   8,   5,
     0,   6,  13,   1,   3,  13,   4,  14,
    14,   0,   7,  11,   5,   3,  11,   8,
     9,   4,  14,   3,  15,   2,   5,  12,
     2,   9,   8,   5,  12,  15,   3,  10,
     7,  11,   0,  14,   4,   1,  10,   7,
     1,   6,  13,   0,  11,   8,   6,   13  };

SBox s7 In 6 Out 4
 {   4,  13,  11,   0,   2,  11,  14,   7,
    15,   4,   0,   9,   8,   1,  13,  10,
     3,  14,  12,   3,   9,   5,   7,  12,
     5,   2,  10,  15,   6,   8,   1,   6,
     1,   6,   4,  11,  11,  13,  13,   8,
    12,   1,   3,   4,   7,  10,  14,   7,
    10,   9,  15,   5,   6,   0,   8,  15,
     0,  14,   5,   2,   9,   3,   2,   12  };

SBox s8 In 6 Out 4
```

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

SBox shifts In 4 Out 4
 {  1,  1,  2,  2,  2,  2,  2,  2,
    1,  2,  2,  2,  2,  2,  2,   1  };

Selection PC1 In 64 Out 56
 {  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  };

Selection PC2 In 56 Out 48
 {  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  };
Keyscheduling Key 64
Local{bitvector(56) k;
      bitvector(4) i;}
Begin
  i = 0;
  k = PC1(key);
  Loop(16)
  Begin
      k(1-28) = ROL(k(1-28), shifts(i), 28);
```

```
      k(29-56) = ROL(k(29-56), shifts(i), 28);
      subkey = PC2(k);
      i = ADD(i, 1, 4);
  End
End

Function F In 32, 48 Out 32
Local { bitvector(48) expinput;
      bitvector(48) tmp;
      bitvector(32) tempout; }
Begin
  expinput = E(input1);
  tmp = XOR(expinput1,input2);
  tmpout(1-4) = S1(tmp(1-6));
  tmpout(5-8) = S2(tmp(7-12));
  tmpout(9-12) = S3(tmp(13-18));
  tmpout(13-16) = S4(tmp(19-24));
  tmpout(17-20) = S5(tmp(25-30));
  tmpout(21-24) = S6(tmp(31-36));
  tmpout(25-28) = S7(tmp(37-42));
  tmpout(28-32) = S8(tmp(43-48));
  output = P(tmpout);
End

Round DESRound In 64 Key 48
Local { bitvector(32) temp; }
Begin
  output(1-32) = input(33-64);
  temp = F(input(33-64),key);
  output(33-64) = XOR(temp,input(1-32));
End

Cipher DES In 64 Key 56
Local { bitvector(64) temp1;
      bitvector(64) temp2;}
Begin
  temp1 = IP(input);
  Loop(16)
  Begin
    temp1 = DESRound(temp1, subkey);
  End
```

```
  temp2(1-32) = temp1(33-64);
  temp2(33-64) = temp1(1-32);
  output = FP(temp2);
End
```