Application of PVM to Linear Cryptanalysis of DES

S. Bakhtiari,
R. Safavi-Naini,
University of Wollongong,
Northfield Avenue, Wollongong,
N.S.W. 2500, Australia.
E-mail: shahram@cs.uow.edu.au, rei@cs.uow.edu.au

July 25, 1994

Abstract

Data Encryption Standard (DES) is the most widely used cryptographic algorithm. Recently, Matsui [8] showed that the algorithm is not secure under a plaintext attack, called linear cryptanalysis. In this paper we show that this attack can be implemented as a concurrent program and hence the required time for breaking DES can be effectively reduced by employing a parallel computing environment.

1 Introduction

In a cryptographic system, a *cryptographic algorithm* is used to transform messages from their readable form, known as *plaintext*, into a scrambled and incomprehensible form, called *ciphertext*. A cryptographic algorithm is a collection of invertible functions, indexed by a piece of information called *key*, that maps elements of plaintext set into the set of ciphertexts. In a secure communication system, the communicants share a secret key and hence know the function used for enciphering or deciphering. However, the enemy only knows the collection of the functions but does not know the shared key and hence does not have access to the actual function used by the communicants.

In a cryptanalytic attack, the enemy tries to find the key using his/her knowledge of the algorithm and the set of ciphertexts, or plaintext/ciphertext pairs, that he/she has intercepted. These are known as ciphertext and plaintext attack respectively. A possible attack on all algorithms is the so-called *exhaustive key search* in which the enemy tries all possible keys to find the correct one used by communicants. An algorithm can foil such attack by employing to large key spaces. (A key space of size 2^{56} requires around 2179 years if 2^{10} keys are tried per microsecond.) However, having a large key space is only a necessary condition for a secure cryptographic algorithm but by no means is sufficient.

More specific attacks on algorithms exploit the weaknesses of the algorithms to find shortcuts and speed up the required search for the key. A good cryptographic algorithm prevents all such shortcuts and ensures that the required search in all cases remains infeasible.

One of the most notable cryptographic algorithms is the Data Encryption Standard (DES). This algorithm is standardized in 1977, after a lengthy process of examination by experts all around the globe. Ever since its inception, breaking DES, i.e. finding the key faster than exhaustive search, has been a

challenging problem from theoretical and practical point of view. DES could successfully withstand all cryptanalytic attacks until recently when differential cryptanalysis [4], and later linear cryptanalysis [8], were proposed. Both attacks could successfully find the key faster than exhaustive key search with the latter being the more efficient and less demanding (in terms of the information that has to be available to the enemy) of the two.

In this paper we give a parallel implementation of linear cryptanalysis using PVM. We consider a stripped off version of DES that consists of 12 rounds to present our result. This is only because of the limitations on the computing resources that we were faced with in the university environment. Implementation of the full 16 round DES is also completed, but the actual results in the present computing environment requires months to be completed. We note that, in practice, assigning dedicated workstations to this task will cut the time to a much shorter period.

The organization of the paper is as follows. In the next two sections we present an overview of DES and linear cryptanalysis. In section 4 and 5 we describe the implementation of the algorithm, and its concurrent version using PVM, and include the results of these implementations. Finally, section 6 gives ways of adopting the method to the case that real data is available.

2 Data Encryption Standard (DES)

DES is a block cipher algorithm that takes 64 bits of plaintext and produces 64 bits of ciphertext. The algorithm consists of 16 similar rounds (figure 2). The key is 56 bits and is used to generate 16 subkeys, $K_1, K_2, \ldots, K_{16}$ for rounds 1 to 16 of DES respectively. In each round a function F is applied to the right half of the block and the result is XORed with the left half. The output of the F function in round i is a complex non-linear function of the bits of the right half and subkey K_i. The F function is the heart of the security of DES. A complete description of DES can be found in [9] or [3].

3 Linear Cryptanalysis

Linear cryptanalysis is the most powerful attack known on DES-type block cipher algorithms. In this attack, the cryptographic algorithm is approximated by linear functions. Using these linear approximations on a set of plaintext/ciphertext blocks obtained using the same key, allows us to find good estimates for key bits and hence effectively reduce the size of the key search. The success of the attack in providing good guesses for key bits, depends on the size of the set of plaintext/ciphertext blocks. For 16 rounds, around 2^{45} pairs of plaintext and ciphertext blocks are required. Since similar processing is applied to all pairs of plaintext/ciphertext, we can speed up the attack by processing many pairs of plaintext/ciphertext blocks in parallel. This observation is the basis of the proposed parallel implementation of the attack.

3.1 Linear Cryptanalysis of 12 Rounds

In this section we give the linear cryptanalysis algorithm for 12 rounds. A similar approach can be used for the full 16 rounds. We need the following notations:

- P_H and P_L, the 32 high and low bits of a plaintext block.
- C_H and C_L, the 32 high and low bits of a ciphertext block.
- K_i, the ith subkey of round i.

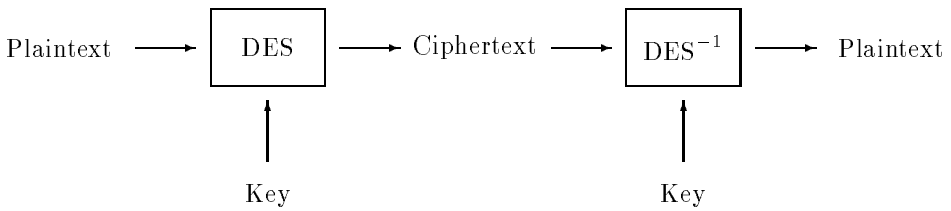

Figure 1: *DES input and output*

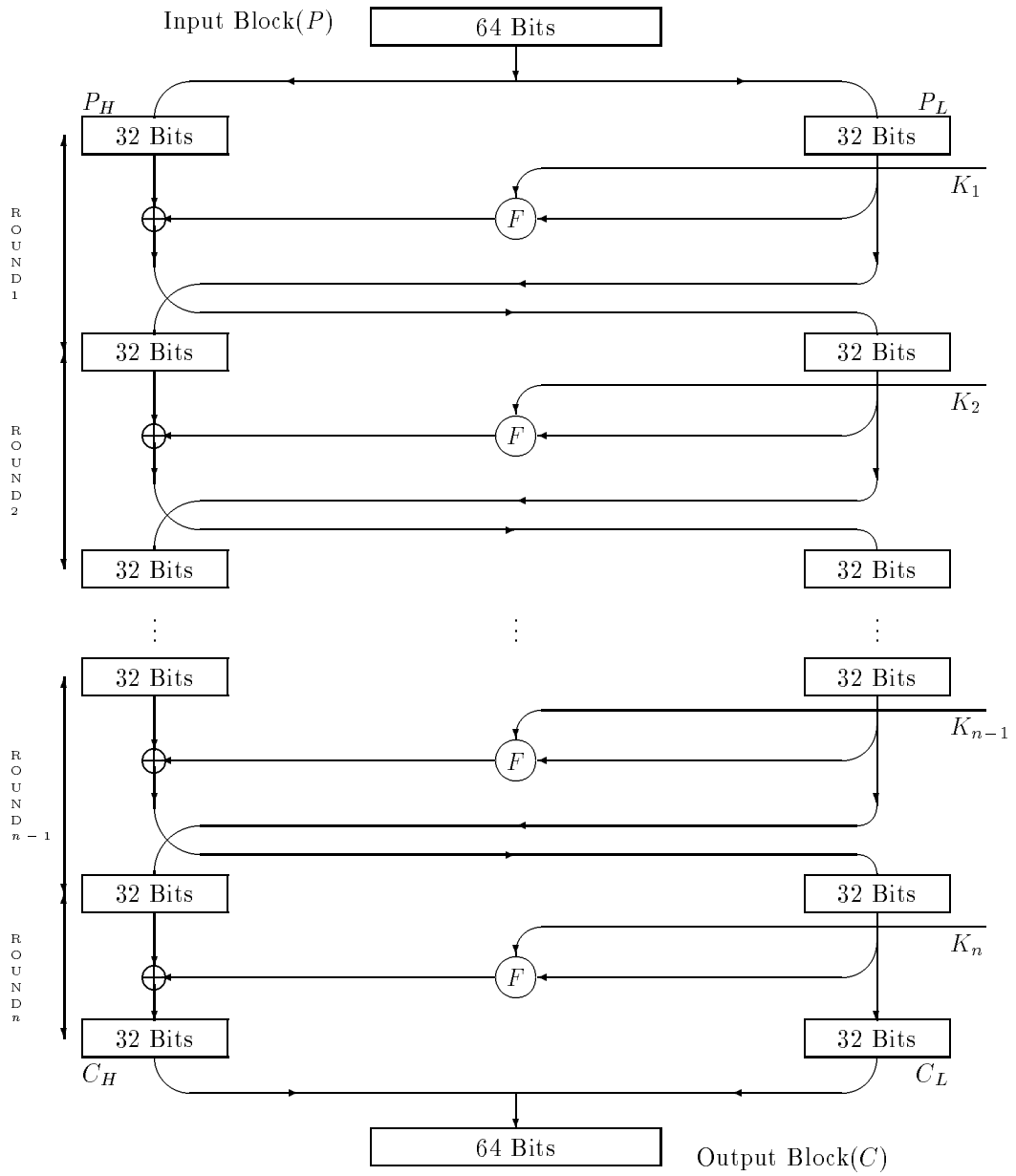

Figure 2: *DES algorithm*

- F_i, the result of the F function in round i.
- $A^{[i,j,\ldots]}$, the XOR of bits $i, j, \ldots$ of A.

For 12 rounds, we can drive two linear relations among plaintext, ciphertext and key bits:

$$P_H^{[7,18,24,29]} \oplus F_1^{[7,18,24,29]} \oplus C_L^{[7,18,24,29]} \oplus C_H^{[15]} \oplus F_{12}^{[15]} =$$
$$K_3^{[22]} \oplus K_4^{[44]} \oplus K_5^{[22]} \oplus K_7^{[22]} \oplus K_8^{[44]} \oplus K_9^{[22]} \oplus K_{11}^{[22]}, \quad (1)$$

$$C_H^{[7,18,24,29]} \oplus F_{12}^{[7,18,24,29]} \oplus P_L^{[7,18,24,29]} \oplus P_H^{[15]} \oplus F_1^{[15]} =$$
$$K_{10}^{[22]} \oplus K_9^{[44]} \oplus K_8^{[22]} \oplus K_6^{[22]} \oplus K_5^{[44]} \oplus K_4^{[22]} \oplus K_2^{[22]}. \quad (2)$$

Each of these equations holds with probability $p = \frac{1}{2} - 1.53 \times 2^{-15}$, for a randomly chosen plaintext block and its corresponding ciphertext block.

We call an *effective bit*, a bit that affects one of the linear approximations 1 or 2. Effective text bits are effective bits of plaintext and ciphertext blocks, and effective key bits are effective bits of the key. The effective bits for equation 1 are,

13 Effective Text Bits: $P_L^{[11]} \sim P_L^{[16]}$, $C_L^{[0]}$, $C_L^{[27]} \sim C_L^{[31]}$, $P_H^{[7,18,24,29]} \oplus C_L^{[7,18,24,29]} \oplus C_H^{[15]}$.

12 Effective Key Bits: $K_{12}^{[42]} \sim K_{12}^{[47]}$ and $K_1^{[18]} \sim K_1^{[23]}$.

The effective bits for equation 2 are,

13 Effective Text Bits: $C_L^{[11]} \sim C_L^{[16]}$, $P_L^{[0]}$, $P_L^{[27]} \sim P_L^{[31]}$, $C_H^{[7,18,24,29]} \oplus P_L^{[7,18,24,29]} \oplus P_H^{[15]}$.

12 Effective Key Bits: $K_1^{[42]} \sim K_1^{[47]}$ and $K_{12}^{[18]} \sim K_{12}^{[23]}$.

The following algorithm is used to guess the effective key bits from N pairs of plaintext and ciphertext blocks.

Algorithm 1

Data Counting Phase

Step 1 *Prepare 2^{13} counters U_i $(0 \le i < 2^{13})$ and initialize them to zero; a counter U_i corresponds to the value i of the 13 effective text bits of equation 1 (or 2).*

Step 2 *For each plaintext block P and the corresponding ciphertext block C, compute the value 'i' of* **Step 1** *and increment the counter U_i by one.*

Key Counting Phase

Step 3 *Prepare 2^{12} counters T_j $(0 \le j < 2^{12})$ and initialize them to zero; a counter T_j corresponds to the value j of the 12 effective key bits of equation 1 (or 2).*

Step 4 *For each 'j' of* **Step 3**, *let T_j be the sum of U_i's such that the left side of equation 1, whose value can be uniquely determined by i and j, is equal to zero.*

Step 5 *Let T_{min} be the minimal value and T_{max} be the maximal value of all T_j's.*

If $|T_{max} - \frac{N}{2}| > |T_{min} - \frac{N}{2}|$, then adopt the subkey value 'j', corresponding to T_{max}, and guess that the right side of equation 1 (or 2) is 0.

If $|T_{max} - \frac{N}{2}| < |T_{min} - \frac{N}{2}|$, then adopt the subkey value 'j', corresponding to T_{min}, and guess that the right side of equation 1 (or 2) is 1.

Figure 3 shows the flow-chart of this algorithm.

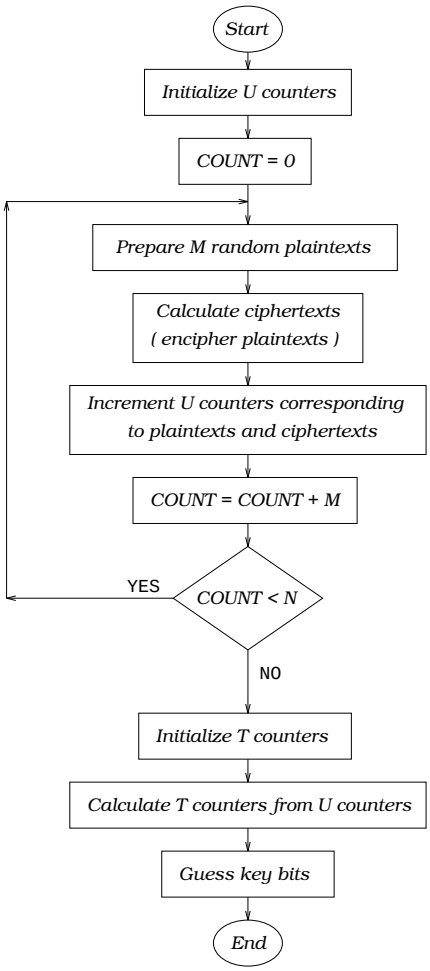

Figure 3: *Schematic flow-chart of linear cryptanalysis of DES.*

4 Implementation of linear cryptanalysis

Figure 3 shows the schematic flow-chart of linear cryptanalysis of DES. According to this chart, the required number of pairs of plaintext/ciphertext blocks (N) must be processed sequentially and a single processor performs all the computations and guesses the key bits based on algorithm 1.

For 12-round DES we need $N = 2^{32}$ pairs of plaintext and ciphertext blocks to be able to guess the key bits correctly with a high probability. The required plaintext blocks are generated using a pseudo-random number generator. The corresponding ciphertext blocks are obtained using DES. The required execution time[1] for 12 rounds is about 5 days (120 hours). Table 1 gives the results produced by the algorithm for a collection of sample data. The last row of the table contains the actual key bits for which the algorithm will provide estimates. Every other row of the table contains the estimate of the key bits at various stages of the algorithm. The equality of the last two rows shows that the algorithm has successfully found the key bits.

The program loops $2^{13} = 8192$ times and each time processes 2^{19} pairs of plaintext/ciphertext blocks. In total, 2^{32} pairs are processed. For example the first row states that the program needs 13.54 hours

[1] All of the implementation results are made in the University of Wollongong by using different types of SUN4 SPARC stations running SOLARIS 2.3.

	DATA FOR EQUATION 1					DATA FOR EQUATION 2				
Time	*Step*	*12-Keybits and Right Side*				*Step*	*12-Keybits and Right Side*			
13.54	924	1110	0000	0010	0	924	0100	0110	0000	0
28.18	1924	1110	0000	0010	0	1924	0010	0111	1110	0
42.77	2920	1100	1000	0011	0	2920	0111	1001	0010	0
57.25	3908	1111	0101	1100	0	3908	0111	1001	0010	0
73.24	5000	1100	1000	0011	0	5000	1111	0111	1001	0
84.84	5792	1100	1000	0011	0	5792	1111	0101	1110	0
97.79	6676	1100	1000	0011	0	6676	1111	0111	1001	0
110.45	7540	1100	1010	0100	0	7540	1111	0111	1001	0
119.94	8192	1100	1001	0001	1	8192	1111	0100	1100	1
	Key	1100	1001	0001	1	Key	1111	0100	1100	1

Table 1: *Results of the cryptanalysis of 12-round DES without using PVM. KEY = 8e7f65b 43a209d. Time is given in hours.*

to perform 924 loops and guess the 12-keybits plus the right hand side of the linear expressions 1 and 2. The last row states that in about 120 hours the program looped 8192 times and has correctly guessed the key bits. It can be seen that for 12-round DES, although parallel implementation will speed up the search, the search is still doable without such implementation. However, for 16-round DES, $N = 2^{45}$ pairs need to be processed and the required execution time becomes prohibitively large. The expected execution time in this case is about 122 years[2], which in practice may suggest that DES is secure. This is because the key of a system is usually changed many times during such period of time. However as will be shown in the next section, with a parallel implementation the search can be distributed among many processors and hence the required time can be lowered in proportion to the number of processors, employed.

5 Implementation using PVM

PVM (*Parallel Virtual Machine*) [6, 2, 7] is a software that integrates a collection of heterogeneous computers into a distributed computing environment. PVM environment is ideal for cryptanalytic attacks such as exhaustive key search. In this section we give an implementation of linear cryptanalysis using PVM environment. As seen in figure 4, A master program is responsible for the distribution of data and the required computations. It initializes the required number of slave programs and then assigns jobs to slaves.

Each slave executes the given job and sends a ready signal to the master program to get the next job. This process will be continued until all the data is processed and the jobs are finalized. In this case, the master program asks for the results from all slaves. The rest of the process is similar to the previous section where the key bits will be guessed using algorithm 1. Table 2 shows the result of execution at various stages of the implementation using PVM. The table is similar to table 1 and clearly shows the reduction of time in this latter case. More details about the implementation is given in appendix A.

[2]Since the process for 16 round is about the same as that of 12 rounds, to process 2^{45} pairs of plaintext/ciphertext blocks we need about $120 \times (2^{45} - 2^{32})$ hours which is about 122 years.

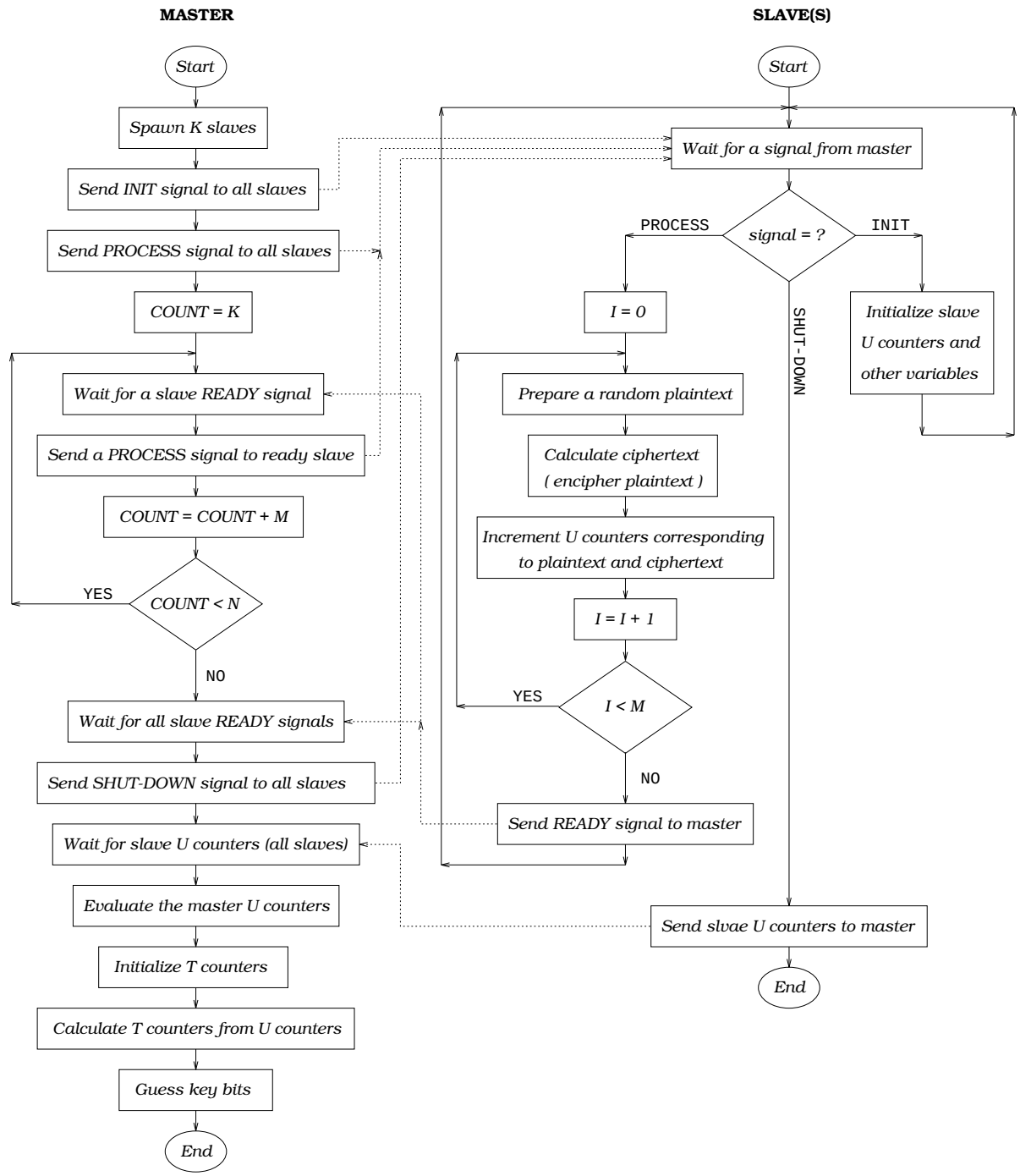

Figure 4: *Schematic flow-chart of linear cryptanalysis of DES using parallel processing.*

	DATA FOR EQUATION 1					DATA FOR EQUATION 2				
Time	*Step*	*12-Keybits and Right Side*				*Step*	*12-Keybits and Right Side*			
1.33	844	0011	1000	1110	0	844	1011	0110	1100	0
2.62	1664	0010	1001	1001	0	1664	0100	1111	1000	0
4.02	2548	0010	1011	1111	0	2548	1110	0001	0001	0
5.49	3480	0010	1011	1011	0	3480	0100	1111	1000	0
6.31	4000	0010	1011	1011	0	4000	0100	1111	1000	0
7.89	5000	1001	1011	1001	0	5000	1101	1101	0000	0
9.46	6000	1001	1011	1001	0	6000	1101	1101	0000	0
11.30	7168	1001	1011	1000	0	7168	1101	1101	0000	0
12.92	8192	0101	1101	1001	1	8192	1101	1101	0000	0
	Key	0101	1101	1001	1	Key	1101	1101	0000	0

Table 2: *Results of the cryptanalysis of 12-round DES using PVM. KEY = 8e7f65b 43a209d. Time is given in hours and the job is distributed among 11 slave programs.*

6 Real Data

In the implementations described above we have used artificial data: the required plaintext blocks are generated by a pseudo-random number generator and the corresponding ciphertexts are obtained by feeding the output of the pseudo-random number generator through DES algorithm. When real data is available, the algorithm may need adjustment to obtain the best performance. In this section we consider two possible forms of the real data and suggest an implementation for each case.

1. *Data is available from one source*
   In this case, the master program communicates with this source and then distributes the jobs to various slaves. The master program will request the result of computations when enough plaintext/ciphertext pairs are processed. [Figure 5]

2. *More than one source of data is available*
   In this case, one slave process is made responsible for processing the data from each source. Each slave process only needs to send the number of processed pairs and allow the master process to calculate the total amount of processed data. When the required amount of data is processed, the master process will request the results from all slaves and will produce the best estimates of the key bits. [Figure 6]

References

[1] S. Bakhtiari, *Linear Cryptanalysis of DES Cipher*, The report for Master of Computer Science Degree, University of Wollongong, 1994.

[2] B. E. Bauer, *Practical parallel programming*, Academic Press Inc., 1992.

[3] H. Beker and F. Piper, *Cipher Systems, the protection of communications*, pp. 273–285, 1982.

[4] E. Biham and A. Shamir, *Differential Cryptanalysis of DES-like cryptosystem, Journal of Cryptology*, pp. 3–72, vol. 4, no. 1, 1991.

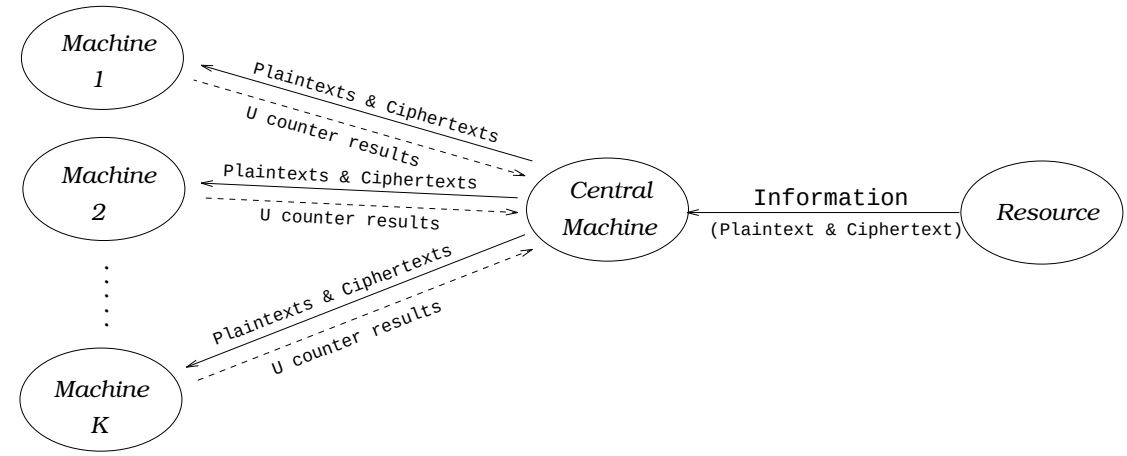

Figure 5: *One source of data (Plaintexts and Ciphertexts) is available.*

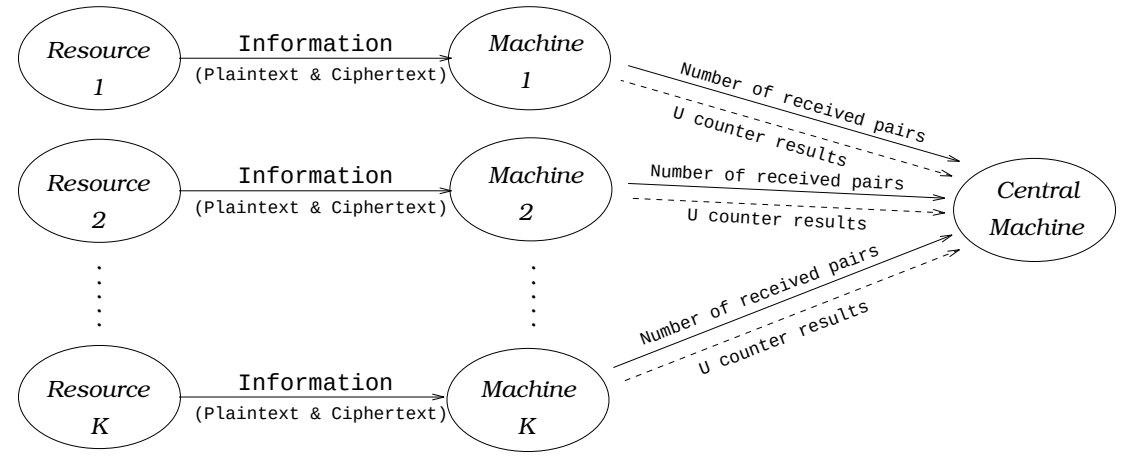

Figure 6: *Many sources of data are available.*

[5] L. Brown, *DES file encryption/decryption program*, Australian Standard AS2805.5-1985.

[6] A. L. Decegama, *The technology of parallel processing*, Volume 1, Prentice Hall, 1989.

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM 3 user's guide and reference manual*, 1994.

[8] M. Matsui, *Linear Cryptanalysis of DES Cipher (I)*, Version 1.03, 1994.

[9] J. Seberry and J. Pieprzyk, *Cryptography : an introduction to computer security*, Prentice Hall, 1989.

A Source Codes

This appendix contains the source code for the implementation of linear cryptanalysis using PVM. The master program is `master.c` which activates the required number of slaves. Each slave (task) is a copy of the `slave.c` program which processes 2^{19} pairs of plaintexts and ciphertexts in each process loop.

Before running the source codes, *PVM daemon* must be activated. It stays in the memory and provides a parallel virtual machine environment [7]. PVM daemon contains the machine codes for the PVM utility functions which have been used in our source codes and is considered as an interface between the master program and the slave program. Any master-slave communication passes through the PVM daemon and any changes in the PVM environment is saved as a PVM parameter in the PVM daemon memory space.

A.1 Execution of the Source Code

After making the PVM daemon resident in the memory (simply by running `pvmd` in the background mode, or running `pvm` and then quitting), the actual process can be activated by executing the `master.c` program. It is first connected to the PVM daemon and then reads the required information such as the start point and the end point of the process. It tries to add the requested hosts (machines) if they are not added to the system before, and spawns the requested number of slaves (tasks) on each particular host. Usually there will be more tasks on the fast machines and less tasks on the slow ones.

Next, the parameters of the algorithm are initialized and an INIT signal is sent to all slave programs. Then, a PROCESS signal together with a job number will be sent to each slave program and each slave program will initialize its variables and will start processing $M = 2^{19}$ pairs of plaintext and ciphertexts. When a process is performed by the slave program, a READY signal will be sent to the master program to get the next job and this procedure will be continued for all N jobs ($N = \frac{2^{32}}{2^{19}} = 2^{13}$ for cryptanalysis of 12-round DES).

After receiving the results from all the slave programs, the master program adds the received U counters, evaluates the T counter and guesses the key bits based on the algorithm 1.

There is a slight difference between the flow-chart in figure 3 and the corresponding source code in this appendix. Based on the flow-chart, U counters should be initialized once in each slave program and should be sent to the master program after receiving the SHUT-DOWN signal. In the source code, U counters are sent every time that the slave program processes 2^{19} plaintext/ciphertext pairs.

Since in our experiment the slave programs were running on the workstations in the university computer labs, there was a high probability of having a failure of a slave when a user was using the system. Also, there was the possibility of a workstation being turned off by a user. Therefore, we needed to send the U counter results to the master program after each process loop in the slave programs to prevent such problems.

We ran our master program on a more reliable machine with a fast disk access and saved the required step information in a log file. Any failure in the execution is also saved for later investigation.

A.2 Constants

```
/* Constants for master.c and slave.c. */
#define DES12 1
#define ROUND 12                    /* Number of rounds in DES */
#define MAXTASKS 64                 /* Maximum number of tasks */
#define TRUE 1                      /* True value */
```

```
#define FALSE 0                                                    /* False value */
#define ARCH "SUN4"                                                /* Type of spawn for PVM */
#define NumOfCounterU 8192                                         /* 2^13, 13= Number of effective text bits */
#define NumOfPlaintext 524488                                      /* 2^19 Number of plain/cipher texts in each process */
#define INITIAL 1                                                  /* Message tag for initialization */
#define SHOUT_DOWN 2                                               /* Message tag for shouting down the slaves */
#define PROCESS 3                                                  /* Message tag for processing data */
#define RESULTS 4                                                  /* Message tag for sending the results */
#define READY 5                                                    /* Message tag for being ready */
/* The following functions are the utility functions. See descriptions in [1] */
#define L_16_11(v1r) ((v1r>>11)&0x0000003f)
#define L_31_27(v2r) ((v2r>>21)&0x000007c0)
#define L_0(v2r) ((v2r<<11)&0x00000800)
#ifdef DES12
#define H_29(v1l) ((v1l&0x20000000)>>11)
#endif
#define H_24(v1l) ((v1l&0x01000000)>>6 )
#define H_18(v1l) ((v1l&0x00040000) )
#define H_7(v1l) ((v1l&0x00000080)<<11)
#define H_15(v2l) ((v2l&0x00008000)<<3 )
#define L_29(v2r) ((v2r&0x20000000)>>11)
#define L_24(v2r) ((v2r&0x01000000)>>6 )
#define L_18(v2r) ((v2r&0x00040000) )
#define L_7(v2r) ((v2r&0x00000080)<<11)
#define I_5_0(v1r) L_16_11(v1r)
#define I_11_6(v2r) L_0(v2r) | L_31_27(v2r)
#ifdef DES12
#define I_12(v1l,v2l,v2r) (H_29(v1l)^H_24(v1l)^H_18(v1l)^H_7(v1l)^
        H_15(v2l)^L_29(v2r)^L_24(v2r)^L_18(v2r)^L_7(v2r))>>6
#else
#define I_12(v1l,v2l,v2r) (H_24(v1l)^H_18(v1l)^H_7(v1l)^H_15(v2l)^
        L_29(v2r)^L_24(v2r)^L_18(v2r)^L_7(v2r))>>6
#endif
#define eff_bits(v1l,v1r,v2l,v2r) (I_5_0(v1r))|(I_11_6(v2r))|(I_12(v1l,v2l,v2r))
/* Expansion Function E */
#define E_left(R) (((R&0x001f8000)>>15)|((R&0x01f80000)>>11)|
        ((R&0x1f800000)>>7)|(((R>>3)&0x1f000000)|((R<<29)&0x20000000)))
#define E_right(R) ((((R<<1)&0x0000003e)|((R>>31)&0x00000001))||
        ((R&0x000001f8)<<5)|((R&0x00001f80)<<9)|((R&0x0001f800)<<13))
/* F Function ( Note: left and right, are left and right parts of the XOR result */
#define F_result(left,right) (S_P[7][(B8)right]|S_P[6][(B8)(right>>8)]|
        S_P[5][(B8)(right>>16)]|S_P[4][(B8)(right>>24)]|S_P[3][(B8)left]|
        S_P[2][(B8)(left>>8)]|S_P[1][(B8)(left>>16)]|S_P[0][(B8)(left>>24)])
typedef int BOOLEAN;                                               /* True/False variable */
typedef unsigned long int LONG;                                    /* 32-bit variable */
typedef unsigned char B8;                                          /* 8-bit variable */
```

A.3 Variables

```
/* Variables for master.c and slave.c */
LONG U_counter_1[NumOfCounterU],
        U_counter_2[NumOfCounterU];                    /* Counters for effective text bits */
LONG subkey_result[ROUND][2];                          /* Space for subkey results */
LONG tempU_counter_1[NumOfCounterU],                   /* Temporary space for U_counter_1 */
        tempU_counter_2[NumOfCounterU];                /* Temporary space for U_counter_2 */
LONG key[2];                                           /* Space for the master key (56 bits) */
int tid, parentTid, who;                  /* Task idendifier of current process, parent, and sender */
int encoding = PvmDataRaw;                             /* Encoding form (No encoding) */
int msgtag;                                            /* Message tag */
int bufid, bytes;                  /* Buffer idendification and Number of bytes in receiving buffer */
BOOLEAN encrypt_mode = TRUE;                           /* Flag for encryption/decryption mode */
FILE *fp;                                              /* Log file pointer */
char logfile[60] = "logfile";                          /* Log file path and name */
int seed, start, end;                          /* Current seed value , Start seed, End seed */
BOOLEAN break_loop = FALSE;
int slaveTid[MAXTASKS];                                /* List of slaves (tids) */
int slaveLastSeed[MAXTASKS];                           /* List of last seeds sent to slaves */
int seedBuffer[MAXTASKS];                              /* Buffer for failed seeds */
int seedBufferPointer=0;                               /* Pointer for above buffer */
int slaves, workingslaves;                 /* Number of slaves and Number of working slaves */
int hosts, hostslaves[MAXTASKS];           /* Number of hosts and Number of tasks per host */
char hostname[MAXTASKS][50];                           /* List of host names */
int f1, f2;                                            /* file pointers for U counters */
int shift[] = { 1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1};      /* Shifts for 16 steps of key schedule */
/* Combination of S-boxes and P permutation. See [1] for complete declration */
LONG S_P[8][64]={ 0x00808200, ... ,0x08020800 };
void initialization();                     /* Initializes the program. U counters, DES, ... */
void finalization();                                   /* Finalizes the program. U counters */
void error();                                          /* Writes error message and exits */
void keysch();                                         /* DES key scheduling without PC1 */
void get_key();                                        /* Gets 56-bit key from command line */
void shift_bits();                                     /* Shifts bits in key schedule */
/* Complete description of the last three functions can be found in [1] */
```

A.4 Master Program

```
/* master.c program distributes the job among the slaves */
#include <stdio.h>
#include "pvm3.h"                                      /* PVM utility function and constants */
int main(){
   int register i, j, k;
   if( (tid = pvm_mytid()) == PvmSysErr ){             /* Register master as a PVM program */
      fprintf(stderr,"Error - DES_master :  PVM daemon not responding.\n");
      exit(1);
   }
   initialization();                                   /* Read input data and initialize variables */
```

```
fprintf(fp,"Spawned %d slaves, broadcast sibling info.\n", slaves);
if( pvm_initsend(encoding) < 0 )        error("pvm_initsend");
if( pvm_pkstr(logfile) < 0 )        error("pvm_pkstr");
if( pvm_pkint((int *)subkey_result, ROUND*2, 1) < 0 )        error("pvm_pkint");
if( pvm_mcast(slaveTid, slaves, INITIAL) < 0 )        error("pvm_mcast");
for( i=0; i< slaves; i++){
  if( pvm_initsend(encoding) < 0 )        error("pvm_initsend");
  if( pvm_pkint(&start, 1, 1) < 0 ) /*seed*/        error("pvm_pkint");
  if( pvm_send(slaveTid[i], PROCESS) < 0 )        error("pvm_send");
  slaveLastSeed[i] = start++;
}
while( end > start ){
  for(;;){
    bufid = pvm_nrecv(-1, -1);
    if( bufid < 0 )        error("pvm_recv");
    if( bufid > 0 )        break;
    for( j=0; j<slaves; j++)
      if(( slaveTid[j] != 0 ) && ( pvm_pstat(slaveTid[j]) != PvmOk )){
        fprintf(fp, "tid=%x is removed.\n", slaveTid[j]);
        seedBuffer[seedBufferPointer++] = slaveLastSeed[j];
        slaveTid[j] = 0;        workingslaves--;
      }
    if( workingslaves == 0 )        error("pvm_pstat.  No slave left");
    (void)sleep(1);
  }
  if( pvm_bufinfo(bufid, &bytes, &msgtag, &who) < 0 )        error("pvm_bufinfo");
  if(msgtag != READY)        error("recv.  Unexpected message from slaves");
  if( pvm_upkint((int *)tempU_counter_1,NumOfCounterU,1) < 0 )  error("pvm_upkint");
  if( pvm_upkint((int *)tempU_counter_2,NumOfCounterU,1) < 0 )  error("pvm_upkint");
  for( i=0; i<NumOfCounterU ; i++){
    U_counter_1[i] += tempU_counter_1[i];        U_counter_2[i] += tempU_counter_2[i];
  }
  if( seedBufferPointer != 0 )        seed = seedBuffer[--seedBufferPointer];
  else        seed = start++;
  if( pvm_initsend(encoding) < 0 )        error("pvm_initsend");
  if( pvm_pkint(&seed, 1, 1) < 0 )        error("pvm_pkint");
  if( pvm_send(who, PROCESS) < 0 )        error("pvm_send");
  for( i=0; i<slaves ; i++)
    if( who == slaveTid[i] ){        slaveLastSeed[i] = seed;        break;        }
}
for( i=0; i<workingslaves; i++){
  for(;;){
    bufid = pvm_nrecv(-1, -1);
    if( bufid < 0 )        error("pvm_recv");
    if( bufid > 0 )        break;
    for( j=0; j<slaves; j++)
      if(( slaveTid[j] != 0 ) && ( pvm_pstat(slaveTid[j]) != PvmOk )){
        fprintf(fp,"%x removed.  %d not processed\n",slaveTid[j],slaveLastSeed[j]);
```

```
            slaveTid[j] = 0;          workingslaves--;
          }
        if( workingslaves == 0 ){
          fprintf(fp, "No slave left.  Above seeds not processed.  Break\n");
          break_loop = TRUE;          break;
        }
        (void)sleep(1);
      }
      if( break_loop )          break;
      if( pvm_bufinfo(bufid, &bytes, &msgtag, &who) < 0 )          error("pvm_bufinfo");
      if(msgtag != READY)          error("recv.  Unexpected message in final steps");
      if( pvm_upkint((int *)tempU_counter_1,NumOfCounterU,1) < 0 )  error("pvm_upkint");
      if( pvm_upkint((int *)tempU_counter_2,NumOfCounterU,1) < 0 )  error("pvm_upkint");
      for( j=0; j<NumOfCounterU ; j++){
        U_counter_1[j] += tempU_counter_1[j];          U_counter_2[j] += tempU_counter_2[j];
      }
    }
    if( pvm_initsend(encoding) < 0 )          error("pvm_initsend");
    if( pvm_mcast(slaveTid, slaves, SHOUT_DOWN) < 0 )          error("pvm_mcast");
    finalization();                                          /* Write U counters in U counter files */
    fprintf(fp,"Monte Carlo master closing down\n");
    fclose(fp);          pvm_exit();          return(0);
}

void error(char *message){
    int i;
    fprintf(fp, "Error - DES_master :  error after %s.\n", message);
    fprintf(stderr, "Error - DES_master :  error after %s.\n", message);
    (void) pvm_initsend(encoding);
    for( i=0; i<slaves; i++)
      (void) pvm_send(slaveTid[i], SHOUT_DOWN);
    pvm_exit();          fclose(fp);          exit(1);
}

void initialization(){
   char U_1_BAK[50], U_2_BAK[50];                                  /* Old U counter files */
   char U_1_DAT[50], U_2_DAT[50];                                  /* New U counter files */
   int infos[MAXTASKS], i;                                         /* Temporary variables */
   char *host_address, **hosts_address;
    printf("Enter the Path for the log-file :");          scanf("%s", logfile)
    sprintf(logfile, "%s%x", logfile, tid);
    if( (fp = fopen(logfile, "w")) == NULL ){
      fprintf(stderr, "Error - DES_master :  cannot open log-file.");
      pvm_exit();          exit(1);
    }
    printf("Enter the host information.\nEnter Q for the last host.\n");
    hosts = slaves = 0;
    for(;;){
```

```
      printf("Name of host # %d ?  ", hosts + 1);          scanf("%s", hostname[hosts])
      if( (hostname[hosts][0] == 'Q') && (hostname[hosts][1] == '\0') )          break;
      printf("# of tasks on %s?", hostname[hosts]);          scanf("%d", &hostslaves[hosts])
      if( pvm_mstat( hostname[hosts] ) != PvmOk ){
        host_address = hostname[hosts];          hosts_address = &host_address;
        if( pvm_addhosts( hosts_address, 1, infos ) < 0 ){
          fprintf(fp, "%s cannot be added.  Ignored.\n", hostname[hosts]);
          continue;
        }
      }
      /* Spawn the slave processes */
      if( pvm_spawn( "slave", NULL, PvmTaskHost, hostname[hosts],
                          hostslaves[hosts], &slaveTid[slaves]) != hostslaves[hosts] ){
        fprintf(fp, "Unable to spawn the required slaves.\n");
        continue;
      }
      slaves += hostslaves[hosts];          hosts++;
    }
    if( slaves < 1 )          error("spawn.  No slave was spawned");
    workingslaves = slaves;
    printf("Start point:");          scanf("%d",&start)
    printf("Final point:");          scanf("%d",&end)
    if( (end - start) < slaves )          error("scanf.  (start - end) >= slaves");
    printf("U_1_BAK? ");          scanf("%s",U_1_BAK)
    printf("U_2_BAK? ");          scanf("%s",U_2_BAK)
    printf("U_1_DAT? ");          scanf("%s",U_1_DAT)
    printf("U_2_DAT? ");          scanf("%s",U_2_DAT)
    if( (f1=open(U_1_BAK,0))==-1 )          error("open");                    /* Open first old U file */
    if( read( f1, (char *)U_counter_1, 4*NumOfCounterU ) != 4*NumOfCounterU )
      error("read");                                    /* Read old U counters for the first equation */
    close(f1);
    if( (f1=creat(U_1_DAT,0600))==-1 )          error("creat");                    /* Creat first new U file */
    if( (f2=open(U_2_BAK,0))==-1 )          error("open");                    /* Open second old U file */
    if( read( f2, (char *)U_counter_2, 4*NumOfCounterU ) != 4*NumOfCounterU )
      error("read");                                    /* Read old U counters for the second equation */
    close(f2);
    if( (f2=creat(U_2_DAT,0600))==-1 )          error("creat");                    /* Creat second new U file */
    get_key();                                                                    /* Get the master key */
    keysch();                                                                     /* Calculate the subkeys */
}

void finalization(){
    if( write( f1, (char *)U_counter_1, 4*NumOfCounterU ) != 4*NumOfCounterU )
      error("write");                                    /* Write new U counters for the first equation */
    if( write( f2, (char *)U_counter_2, 4*NumOfCounterU ) != 4*NumOfCounterU )
      error("write");                                    /* Write new U counters for the second equation */
    close(f1); close(f2);
}
```

A.5 Slave Program

```
/* slave.c program must be activated by master.c */
#include <stdio.h>
#include "pvm3.h"                                        /* PVM utility function and constants */
int main(){
   BOOLEAN initialized = FALSE;
   LONG register R, L, r, l;                             /* Fast variables for plaintext & ciphertext */
   LONG register XOR_result_left, XOR_result_right;      /* Left & right parts of XOR result */
   LONG register Left_side, plain_left, plain_right, i, j;   /* Temporary variables */
   if( (tid = pvm_mytid()) == PvmSysErr )                /* Find my tid and my parent's */
     error("pvm_mytid.  Cannot talk to the PVM daemon");
   if( (parentTid = pvm_parent()) == PvmNoParent )
     error("pvm_parent.  Slave program must be spawned");
   for( ;; ){                                            /* Wait for the first message to arrive */
     if( pvm_pstat(parentTid) != PvmOk )      error("pvm_pstat");
     if( (bufid = pvm_recv(-1, -1)) < 0 )     error("pvm_recv");
     if( pvm_bufinfo(bufid, &bytes, &msgtag, &who) < 0 )     error("pvm_bufinfo");
     if( who != parentTid )      error("pvm_recv.  Message not from parent");
     switch( msgtag ){
       case INITIAL:                                                    /* initialization */
         if( pvm_upkstr(logfile) < 0 )      error("pvm_upkstr");
         if( pvm_upkint((int *)subkey_result,ROUND*2,1) < 0 )      error("pvm_upkint");
         initialized = TRUE;
         break;
       case SHOUT_DOWN:                                                 /* shouting down */
         pvm_exit();      exit(3);
         break;
       case PROCESS:                                                    /* actual message */
         if( ! initialized ) error("IF. Slave not initialized");
         if( pvm_upkint(&seed,1,1) < 0 ) error("pvm_upkint");
         for(i=0; i< NumOfCounterU; i++)                    /* Initialize U counters to zero */
           U_counter_1[i] = U_counter_2[i] = 0;
         srand48((long)seed + 1);                  /* Initializing the reandom number generator */
         for(i=0; i< NumOfPlaintext; i++){
           plain_left = L = mrand48();                      /* Get a 32-bit random number */
           plain_right = R = mrand48();                     /* Get a 32-bit random number */
           for(j=0; j<ROUND; j++){                  /* DES algorithm for ROUND rounds */
             /* Evaluates XOR result from subkey and F function */
             XOR_result_left = subkey_result[j][0] ^ E_left(R);
             XOR_result_right = subkey_result[j][1] ^ E_right(R);
             /* Evaluates XOR of previous left part and F function result */
             Left_side =L ^ F_result(XOR_result_left,XOR_result_right);
             L = R;      R = Left_side;                    /* Swaps left and right parts */
           }
           l = R;      r = L;                                          /* Ciphertext */
           L = plain_left;      R = plain_right;                       /* Plaintext */
           /* Increment U conters for both equations based on effective text bits */
```

```
            U_counter_1[ eff_bits(L, R, l, r) ]++;          U_counter_2[ eff_bits(l, r, L, R) ]++;
          }
          if( pvm_initsend(encoding) < 0 ) error("pvm_initsend");
          if( pvm_pkint((int *)U_counter_1,NumOfCounterU,1) < 0 )   error("pvm_pkint");
          if( pvm_pkint((int *)U_counter_2,NumOfCounterU,1) < 0 )   error("pvm_pkint");
          if( pvm_send(parentTid, READY) < 0 ) error("pvm_send");
          break;
        default:
          error("pvm_recv.  Unexpected message tag from parent");
          break;
      }
    }
}

void error(char *message){
    sprintf(logfile, "%s-%x", logfile, tid);
    fp = fopen(logfile,"w");
    fprintf(fp, "Error - DES_slave :  error after %s.\n", message);
    fprintf(fp,"tid=%d\nmsgtag=%d\nseed=%d\nwho=%d\n", tid,msgtag,seed,who);
    fclose(fp);          pvm_exit();          exit(1);
}
```