Cryptanalysis of Stream Cipher COS $(2, 128)$ Mode I

Hongjun Wu and Feng Bao

Kent Ridge Digital Labs
21 Heng Mui Keng Terrace, Singapore 119613
{hongjun,baofeng}@krdl.org.sg

Abstract. Filiol and Fontaine recently proposed a family of stream ciphers named COS. COS is based on nonlinear feedback shift registers and was claimed to be with high cryptographic strength. Babbage showed that COS $(2, 128)$ Mode II is extremely weak. But Babbage's attack is too expensive to break the COS $(2, 128)$ Mode I (the complexity is around 2^{52}). In this paper, we show that the COS $(2, 128)$ Mode I is too weak. With about 2^{16}-bit known plaintext, the secret information could be recovered with small amount of memory and computation time (less than one second on a Pentium IV Processor).

1 Introduction

Filiol and Fontaine recently designed a family of stream ciphers called COS [3–5]. The COS $(2, 128)$ is with two 128-bit internal registers. Two versions of COS (2,128) are available: Mode II and the more secure Mode I. In [1], Babbage showed that the COS $(2, 128)$ Mode II is too weak and the secret information could be recovered easily from a short piece of key stream. Babbage's attack also reduced the complexity of the COS $(2, 128)$ Mode I to 2^{64}. In [2], Babbage's improved attack reduced the complexity of the COS $(2, 128)$ Mode I to 2^{52}.

In this paper, we show that the COS $(2, 128)$ Mode I could be broken with small amount of plaintext and computation time. In average, only about 2^{16}-bit known plaintext is required. The time required is less than one second on a Pentium IV porcessor.

This paper is organized as follows. Section 2 introduces the COS $(2, 128)$ stream cipher. The attack against the COS $(2, 128)$ Mode I is given in Section 3. Section 4 concludes this paper.

2 COS Stream Cipher

We will only give a brief introduction to the COS $(2, 128)$. This version of COS cipher is with two 128-bit registers, L_1 and L_2, as the initial states. We will ignore the key setup of COS (the key setup has no effect on our attack) and only introduce the key stream generation process.

Let $L_1 = L_{10} \parallel L_{11} \parallel L_{12} \parallel L_{13}$, $L_2 = L_{20} \parallel L_{21} \parallel L_{22} \parallel L_{23}$, where $\parallel$ indicates concatenation and each L_{ij} is a 32-bit word. At the ith step, the output key stream is generated as:

1. Compute clocking value d.
   (a) Compute $m = 2 \times (L_{23}\&1) + (L_{13}\&1)$ where & is the binary AND operator.
   (b) $d = C_m$, where $C_0 = 64$, $C_1 = 65$, $C_2 = 66$, $C_3 = 64$.
2. If i is even, clock L_1 d times; otherwise, clock L_2 d times.
3. Let $H_i = H_{i0} \parallel H_{i1} \parallel H_{i2} \parallel H_{i3}$, then $H_{i0} = L_{20} \oplus L_{12}$, $H_{i1} = L_{21} \oplus L_{13}$, $H_{i2} = L_{22} \oplus L_{10}$, $H_{i3} = L_{23} \oplus L_{11}$.
4. For Mode II, the output for the ith step is given as H_i.
5. For Mode I, compute $j = (L_{13} \oplus L_{23})\&3$, $k = (L_{10} \oplus L_{20}) \gg 30$. If $j = k$, then let $k = j \oplus 1$. The output for the ith step is given as $H_{ij} \parallel H_{ik}$.

Two feedback boolean functions are used, $f9a$ for L_1 and $f9b$ for L_2. They use bits $2, 5, 8, 15, 26, 38, 44, 47, 57$ of L_1 and L_2. These two functions are available at [3].

3 Cryptanalysis of COS

In this section, we show that the COS $(2, 128)$ Mode I is very weak. Subsection 3.1 gives a brief introduction to our attack while the detailed attack is given in Subsection 3.2. The experiment result is given in Subsection 3.3.

3.1 The basic idea of our attack

Let us take a look at any four consequent steps starting with an odd step. L_1 is clocked at the second and forth steps; L_2 is clocked at the first and third steps.

	Step 1		Step 2		Step 3		Step 4	
L_1	b	a	c	b'	c	b'	d	c'
L_2	x	w	x	w	y	x'	y	x'

Fig. 1 Four Steps (starting with an odd step) of COS $(2, 128)$

In Fig. 1, a, b, b', c, c' and d are 64-bit words of L_1 at the end of a step; w, x, x', y are 64-bit words of L_2 at the end of a step. According to the key stream generation process, b' may be the same as b; b' may be obtained by right shifting b one (or two) bit position and with the most significant one (or two) bit of b being filled with unknown value. The same applies to c' and c.

The value of (c, b') could be recovered if the following two conditions are satisfied:

Condition 1. The outputs at the first, second, third and forth steps are given as $b \oplus w$, $c \oplus w$, $b' \oplus y$ and $c' \oplus y$, respectively, i.e., (j, k) is $(2, 3)$ or $(3, 2)$ at Step

1 and Step 2 and $(1,0)$ or $(0,1)$ at Step 3 and 4.

Condition 2. One of b' and c' is not the same as b and c, respectively, and b' and c' are not obtained by right shifting b and c (respectively) by the same position.

From Condition 1, we could obtain the values of $b \oplus c$ and $b' \oplus c'$ from the output key streams of these four steps. Once Condition 1 and Condition 2 are satisfied, it is trivial to compute (c, b').

In the next subsection, we will illustrate the idea above in detail and give the estimated results.

3.2 The detailed attack

Before introducing the attack in detail, we give the following two observations:

Observation 1. For the COS $(2,128)$ Mode I, at each step, the probability that (j,k) is $(2,3)$ is 2^{-12}. The same applies to $(3,2)$, $(1,0)$, $(0,1)$.

Observation 2. At the ith step, if j is 2 or 0, then the clocking value at the next step is 64. If j is 1 or 3, the clocking value at the next step is 65 or 66.

These two observations are trivial according to the specifications of the COS cipher.

We now list in Table 1 those 16 cases that satisfy Condition 1. According to Observation 1, each case appears with probability 2^{-12}.

	Step 1	Step 2	Step 3	Step 4
Case 1	(2,3)	(2,3)	(0,1)	(0,1)
Case 2	(2,3)	(2,3)	(0,1)	(1,0)
Case 3	(2,3)	(2,3)	(1,0)	(0,1)
Case 4	(2,3)	(2,3)	(1,0)	(1,0)
Case 5	(2,3)	(3,2)	(0,1)	(0,1)
Case 6	(2,3)	(3,2)	(0,1)	(1,0)
Case 7	(2,3)	(3,2)	(1,0)	(0,1)
Case 8	(2,3)	(3,2)	(1,0)	(1,0)
Case 9	(3,2)	(2,3)	(0,1)	(0,1)
Case 10	(3,2)	(2,3)	(0,1)	(1,0)
Case 11	(3,2)	(2,3)	(1,0)	(0,1)
Case 12	(3,2)	(2,3)	(1,0)	(1,0)
Case 13	(3,2)	(3,2)	(0,1)	(0,1)
Case 14	(3,2)	(3,2)	(0,1)	(1,0)
Case 15	(3,2)	(3,2)	(1,0)	(0,1)
Case 16	(3,2)	(3,2)	(1,0)	(1,0)

Table 1. The 16 cases that satisfy Condition 1

However, not all those 16 cases satisfy Condition 2. According to Observation 2, Cases 1, 2, 5, 6 do not satisfy Condition 2 since $b' = b$ and $c' = c$; Cases 11, 12, 15, 16 satisfy Condition 2 with probability 0.5; the other eight cases all satisfy Condition 2. Thus for every four steps starting with an odd step, Conditions 1 and 2 are satisfied with probability $10 \times 2^{-12} \approx 2^{-8.7}$. To determine the value of L_1, this attack requires the output of about 820 steps in average.

Now we estimate how many values of (c, b') are produced in each case, and show how to filter the wrong values of (c, b'). We illustrate Case 4 as an example: at Step 2 L_1 is clocked 64 times ($b = b'$); at Step4 L_1 is clocked 65 or 66 steps. So 6 values of (c, b') are generated for every four steps starting with an odd step. For each pair of (c, b'), the values of w and y of L_2 could be obtained. Since L_2 is clocked only 64 times at Step 3, the 7 least significant bits of y are generated from w. So the wrong (c, b') could pass this filtering process with probability 2^{-7}.

The further filtering is carried out at Step 5 and Step 6. Let $d = d_0 \parallel d_1$, $c = c'_0 \parallel c'_1$, $e = e_0 \parallel e_1$, $d' = d'_0 \parallel d'_1$, $z = z_0 \parallel z_1$, $y = y'_0 \parallel y'_1$ where d_0, d_1, c'_0, c'_1, e_0, e_1, d'_0, d'_1, z_0, z_1, y'_0 and y'_1 are 32-bit words. In Fig. 2, for each (c, b'), there are 6 values for (d, c', e, d') (L_1 is clocked 65 or 66 times at Step 4 and is clocked 64 or 65 or 66 times at Step 6). The L_2 is clocked 65 or 66 times at Step 5, so there are 6 possible values for y'. Now if any one of j or k is equal to 2 or 3 in Sep 4 or 5, then for the right (c, b'), at least one of $d_0 \oplus y'_0$, $d_1 \oplus y'_1$, $e_0 \oplus y'_0$ and $e_1 \oplus y'_1$ appears in the output. Otherwise, j and k could only be 0 or 1 at Step 4 and Step 5, the output of Step 5 is $(c'_0 \oplus z_0) \parallel (c'_1 \oplus z_1)$ or $(c'_1 \oplus z_1) \parallel (c'_0 \oplus z_0)$, that of Step 6 is $(d'_0 \oplus z_0) \parallel (d'_1 \oplus z_1)$ or $(d'_1 \oplus z_1) \parallel (d'_0 \oplus z_0)$. By xoring the outputs of Step 5 and 6 (taking into the considering whether (j, k) is $(1, 0)$ or $(0, 1)$), the right (c, b') should generate $c'_0 \oplus d'_0$ and $c'_1 \oplus d'_1$. The wrong (c, b') could pass this filtering process with probability $6 \times 6 \times 8 \times 2^{-32} \approx 2^{-23.8}$.

	Step1	Step2	Step3	Step4	Step5	Step6
L_1	b | a	c | b'	c | b'	d | c'	$d_0 \parallel d_1$ | $c'_0 \parallel c'_1$	$e_0 \parallel e_1$ | $d'_0 \parallel d'_1$
L_2	x | w	x | w	y | x'	y | x'	$z_0 \parallel z_1$ | $y'_0 \parallel y'_1$	$z_0 \parallel z_1$ | $y'_0 \parallel y'_1$

Fig. 2 The 6 Steps (starting with an odd step) of COS $(2, 128)$

In Case 4, for every 4 steps starting with an odd step, a correct (c, b') is generated with probability 2^{-12} and a wrong (c, b') is generated with probability $6 \times 2^{-7} \times 2^{-23.8} \approx 2^{-28.2}$.

We list in Table 2 (in the next page) the probabilities that a right and wrong (c, b') is generated for any 4 steps starting with an odd step.

So for any 4 steps starting with an odd step, a correct (c, b') is generated with probability $2^{-8.7}$ and a wrong one is generated with probability $2^{-23.6}$. It is obvious that only the correct (c, b') could pass the filtering process. Once (c, b') is determined, it is easy to recover L_2 from the values of w and y.

3.3 Experiment Result

We implemented an attack that uses only the Case 4. In average, our program recovers L_1 in less than one second on a PC (Pentium IV processor) with the outputs of about 2^{13} steps. The computer programs are given in Appendix A and B. The COS programs provided by the COS designers [3, 4] are used in our program.

	Right L_1 Prob.	Wrong L_1 Prob.
Case 1	0	–
Case 2	0	–
Case 3	2^{-12}	$2^{-30.8}$
Case 4	2^{-12}	$2^{-28.2}$
Case 5	0	–
Case 6	0	–
Case 7	2^{-12}	$2^{-28.2}$
Case 8	2^{-12}	$2^{-25.6}$
Case 9	2^{-12}	$2^{-31.8}$
Case 10	2^{-12}	$2^{-29.2}$
Case 11	2^{-13}	$2^{-30.4}$
Case 12	2^{-13}	$2^{-26.8}$
Case 13	2^{-12}	$2^{-29.2}$
Case 14	2^{-12}	$2^{-26.6}$
Case 15	2^{-13}	$2^{-27.8}$
Case 16	2^{-13}	$2^{-25.2}$

Table 2. The probabilities that each case generates correct and wrong L_1

4 Conclusions

In this paper, we showed that the stream cipher COS $(2, 128)$ Mode I is extremely weak and should not be used.

References

1. S.H. Babbage, "The COS Stream Ciphers are Extremely Weak", *http://eprint.iacr.org/2001/078/*
2. S.H. Babbage, "Cryptanalysis of the COS (2,128) Stream Ciphers", *http://eprint.iacr.org/2001/106/*
3. E. Filiol and C. Fontaine, "A New Ultrafast Stream Cipher Design: COS Ciphers", *http://www−rocq.inria.fr/codes/Eric.Filiol/English/COS/COS.html*
4. E. Filiol and C. Fontaine, "A New Ultrafast Stream Cipher Design: COS Ciphers", in *Proceedings of the 8th IMA Conference on Cryptography and Coding*, LNCS 2260, pp. 85-98.
5. E. Filiol, "COS Ciphers are not "extremely weak"! — the Design Rationale of COS Ciphers", *http://eprint.iacr.org/2001/080/*

A The Program File "cos.c"

```
/*This program breaks the stream cipher COS (2,128) Mode I
  using only the Case 4

  (The complete attack requires 1/10 amount of plaintext
  of this attack, and about the same amount of computation
  time as this attack).

  With $2^19$ bit of plaintext, L1 could be recovered with
  probability 63%
*/

#include "cos.h"

#define steps 0x8000L
/*the key stream of those steps required to retrieve L1.
  For this program, if steps = 0x8000L, then L1 could be
  recovered with probability 0.98 in less than one second
  on a Pentium IV processor
*/

void main ()
{
UINT32 i;
UINT32 L1[4],L2[4];
UINT32 block[2];
UINT32 R[steps][2];
UINT32 B0,B1,C0,C1,W0,W1,Y0,Y1;
UINT32 M0,M1,N0,N1,P0,P1;

setkey(L1,L2);

//generate a key stream and stored it in R

for (i = 0; i < steps; i++) {
coscipher(L1, L2, block, (1+i)%2,i);
R[i][0] = block[0];
R[i][1] = block[1];
}

//begin to recover L1 from the key stream R
for (i = 1; i < steps - 8; i = i + 2) {
M0 = R[i][0] ^ R[i+1][0];
M1 = R[i][1] ^ R[i+1][1];
N0 = R[i+2][1] ^ R[i+3][1];
```

```
N1 = R[i+2][0] ^ R[i+3][0];
P0 = M0 ^ N0;
P1 = M1 ^ N1;

//assume the cipher clocked 65 times at i+3
recovershiftone(&P0,&P1,&C0,&C1);

B0 = M0 ^ C0;
B1 = M1 ^ C1;
Y0 = R[i+2][1] ^ B0;
Y1 = R[i+2][0] ^ B1;
W0 = R[i+1][0] ^ C0;
W1 = R[i+1][1] ^ C1;

//the 'verify' checks whether the estimated values correct or not.
//If correct, it prints the value of L1 and the step number
verify( B0,B1,C0,C1,W0,W1,Y0,Y1,i,R[i+4][0],R[i+4][1],R[i+5][0],
        R[i+5][1]);
verify( B0^0xffffffff,B1^0xffffffff,C0^0xffffffff,C1^0xffffffff,
        W0^0xffffffff,W1^0xffffffff,Y0^0xffffffff,Y1^0xffffffff,
        i,R[i+4][0],R[i+4][1],R[i+5][0],R[i+5][1]);

//assume the cipher clocked 66 times at i+3
recovershifttwo(&P0,&P1,&C0,&C1);

B0 = M0 ^ C0;
B1 = M1 ^ C1;
Y0 = R[i+2][1] ^ B0;
Y1 = R[i+2][0] ^ B1;
W0 = R[i+1][0] ^ C0;
W1 = R[i+1][1] ^ C1;

verify( B0,B1,C0,C1,W0,W1,Y0,Y1,i,R[i+4][0],R[i+4][1],R[i+5][0],
        R[i+5][1]);
verify( B0^0xffffffff,B1^0xffffffff,C0^0xffffffff,C1^0xffffffff,
        W0^0xffffffff,W1^0xffffffff,Y0^0xffffffff,Y1^0xffffffff,
        i,R[i+4][0],R[i+4][1],R[i+5][0],R[i+5][1]);
verify( B0^0xaaaaaaaa,B1^0xaaaaaaaa,C0^0xaaaaaaaa,C1^0xaaaaaaaa,
        W0^0xaaaaaaaa,W1^0xaaaaaaaa,Y0^0xaaaaaaaa,Y1^0xaaaaaaaa,
        i,R[i+4][0],R[i+4][1],R[i+5][0],R[i+5][1]);
verify( B0^0x55555555,B1^0x55555555,C0^0x55555555,C1^0x55555555,
        W0^0x55555555,W1^0x55555555,Y0^0x55555555,Y1^0x55555555,
        i,R[i+4][0],R[i+4][1],R[i+5][0],R[i+5][1]);
}
```

```
}
```

B The Program File "cos.h"

```
/*
Remarks:
This file cos.h contains:
1. the key set up table T (T, f11, f9a, f9b are not included in
2. function f11           (this appendix, they are available at
3. function f9a           ([3].
4. function f9b
5. recovershiftone and recovershifttwo (recover one value of A from
   A^(A>>j) for j=1 and j=2)
6. verify procedures (verify whether the recovered L1 is correct or not)
7. key set up
8. key stream generation (one clock cycle)
*/

#include <conio.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include <ctype.h>
#include <io.h>
#include <sys/timeb.h>

typedef unsigned long int UINT32;

/*The key value*/
#define K1  0x10101010L
#define K2  0x20202020L
#define K3  0x30303030L
#define K4  0x40404040L
#define K5  0x00000000L
#define K6  0x00000000L
#define K7  0x00000000L
#define K8  0x00000000L

/*The key size*/
#define KEYSIZE 128

/* Message key stored in a 32-bit word */
#define MK  0x12345678L
```

```
/*key setup table T
static unsigned char T[256] = ....

static UINT32 f11[2048] = ....

static UINT32 f9a[512] = ....

static UINT32 f9b[512] = ....

These four tables are not included here. they available at
http://www-rocq.inria.fr/codes/Eric.Filiol/English/COS/COS.html
*/


/* knows (C0||C1)^((C0||C1)>>j) = P0||P1, determine the value
   of C0 and C1*/
void recovershiftone(UINT32 *P0,UINT32 *P1,UINT32 *C0,UINT32 *C1)
{
int k;

*C0 = 1 << 31; //assume that the most significant bit of *C0 is 1
for (k = 30; k >=0; k--){
*C0 = *C0 | ( (*P0 ^ (*C0 >>1)) & (1 << k) );
}
*C1 = ((*C0 << 31) ^ *P1) & (1 << 31);
for (k = 30; k >=0; k--) {
*C1 = *C1 | ( (*P1 ^ (*C1 >>1)) & (1 << k) );
}
}

void recovershifttwo(UINT32 *P0,UINT32 *P1,UINT32 *C0,UINT32 *C1)
{
int k,h;
/*assume that the two most significant bits of *C0 are 11
*C0 = 3 << 30;
for (k = 14; k >=0; k--) {
h = 3 << (k<<1);
*C0 = *C0 | ( (*P0 ^ (*C0 >>2)) & h );
}
*C1 = ((*C0 << 30) ^ *P1) & (3 << 30);
for (k = 14; k >=0; k--) {
h = 3 << (k<<1);
*C1 = *C1 | ( (*P1 ^ (*C1 >>2)) & h );
}
```

```
}

/* Verify whether the L1 recovered is the correct one */
void verify(UINT32 B0,UINT32 B1,UINT32 C0,UINT32 C1,UINT32 W0,UINT32 W1,
UINT32 Y0,UINT32 Y1,UINT32 t, UINT32 R40, UINT32 R41, UINT32 R50, UINT32 R51)
{
int i,j,k;
UINT32 L1[4],L2[4];
UINT32 test,feed,G[5],C0P,C1P,D0,D1,D0P,D1P,E0,E1,Y0P,Y1P;
int r1,r2,r3,r4,r5,r6;

test = 0;

L1[0] = C0;
L1[1] = C1;
L1[2] = B0;
L1[3] = B1;

L2[0] = 0x0;
L2[1] = 0x0;
L2[2] = W0;
L2[3] = W1;

/*1) use the information of L2 (w and y) to filter L1, only $2^{-7}$
wrong L1 would pass*/
for(i = 0; i <= 6; i++) {
feed = ((L2[3] & 0x4) >> 2);
feed |= ((L2[3] & 0x20L) >> 4);
feed |= ((L2[3] & 0x100L) >> 6);
feed |= ((L2[3] & 0x8000L) >> 12);
feed |= ((L2[3] & 0x4000000L) >> 22);
feed |= ((L2[2] & 0x40L) >> 1);
feed |= ((L2[2] & 0x1000L) >> 6);
feed |= ((L2[2] & 0x8000L) >> 8);
feed |= ((L2[2] & 0x2000000L) >> 17);
L2[3] = (L2[3] >> 1) | ((L2[2] & 1) << 31);
L2[2] = (L2[2] >> 1) | ((L2[1] & 1) << 31);
L2[1] = (L2[1] >> 1) | ((L2[0] & 1) << 31);
L2[0] = (L2[0] >> 1) | (f9b[feed] << 31);
}

if (L2[0] == (Y1 << 25)) {
/*2) use the information in the next two steps to filter L1
     the wrong L1 can pass this procedure with probability
     less than $2^{-21.2}$*/
```

```
for(j = 0;j < 5; j++) {
for(i = 0;i < 32;i++) {
feed = ((L1[3] & 0x4) >> 2);
feed |= ((L1[3] & 0x20L) >> 4);
feed |= ((L1[3] & 0x100L) >> 6);
feed |= ((L1[3] & 0x8000L) >> 12);
feed |= ((L1[3] & 0x4000000L) >> 22);
feed |= ((L1[2] & 0x40L) >> 1);
feed |= ((L1[2] & 0x1000L) >> 6);
feed |= ((L1[2] & 0x8000L) >> 8);
feed |= ((L1[2] & 0x2000000L) >> 17);
L1[3] = (L1[3] >> 1) | ((L1[2] & 1) << 31);
L1[2] = (L1[2] >> 1) | ((L1[1] & 1) << 31);
L1[1] = (L1[1] >> 1) | ((L1[0] & 1) << 31);
L1[0] = (L1[0] >> 1) | (f9a[feed] << 31);
}
G[j] = L1[0];
}

//d,c': 65,66  e,d':64,65,66  y':65,66
for ( i = 65; i <= 66; i++) {
for ( j = 64; j <= 66; j++) {
for ( k = 64; k <= 66; k++) {

r1 = i - 64;
r2 = 32 - r1;
r3 = j + i - 128;
r4 = 32 - r4;
r5 = k - 64;
r6 = 32 - r5;

C1P = ( C1 >> r1 ) ^ ( C0 << r2 );
C0P = ( C0 >> r1 ) ^ ( G[0] << r2 );

D1 = (G[0] >> r1) ^ (G[1] << r2);
D0 = (G[1] >> r1) ^ (G[2] << r2);

D1P = (G[0] >> r3) ^ (G[1] << r4);
D0P = (G[1] >> r3) ^ (G[2] << r4);

E1 = (G[2] >> r3) ^ (G[3] << r4);
E0 = (G[3] >> r3) ^ (G[4] << r4);

Y1P = (Y1 >> r5) ^ (Y0 << r6 );
Y0P = Y0 >> r5;
```

```
if ( (D1 ^ Y1P) == R40 || ((D0 ^ Y0P ^ R41) << 2) == 0 ||
(D1 ^ Y1P) == R41 || ((D0 ^ Y0P ^ R40) << 2) == 0)
test = 1;

if ( (E1 ^ Y1P) == R40 || ((E0 ^ Y0P ^ R41) << 2) == 0 ||
(E1 ^ Y1P) == R41 || ((E0 ^ Y0P ^ R40) << 2) == 0)
test = 1;

if ( (C0P ^ D0P) == (R40 ^ R50) || (C0P ^ D0P) == (R41 ^ R50) ||
(C0P ^ D0P) == (R40 ^ R51) || (C0P ^ D0P) == (R41 ^ R51) ||
(C1P ^ D1P) == (R40 ^ R50) || (C1P ^ D1P) == (R41 ^ R50) ||
(C1P ^ D1P) == (R40 ^ R51) || (C1P ^ D1P) == (R41 ^ R51) )
test = 1;

}
}
}

}

if (test != 0) {
printf("\nThe L1 at step %8x is %8x,%8x,%8x,%8x",t+2,C0,C1,B0,B1);
}
}

/*key setup*/

void setkey(UINT32 *L1, UINT32 *L2)
{
UINT32  i,a,M[8],feed;

/* M register common part initialization */
M[0] = K1; M[1] = K2; M[2] = K3; M[3] = K4;

/* M register user's key dependent part initialization */
if (KEYSIZE == 256) {
M[4] = K5; M[5] = K6; M[6] = K7; M[7] = K8;
}

if(KEYSIZE == 192) {
M[4] = K5; M[5] = K6;
a = K1 ^ K2 ^ K3;
M[6] = T[(a & 0xFF)] | (T[((a >> 8) & 0xFF)] << 8);
M[6] |= (T[((a >> 16) & 0xFF)] << 16) | (T[a >> 24] << 24);
```

```
a = K4 ^ K5 ^ K6;
M[7] = T[(a & 0xFF)] | (T[((a >> 8) & 0xFF)] << 8);
M[7] |= (T[((a >> 16) & 0xFF)] << 16) | (T[a >> 24] << 24);
}

if(KEYSIZE == 128) {
M[4] = T[(K1 & 0xFF)] | (T[((K1 >> 8) & 0xFF)] << 8);
M[4] |= (T[((K1 >> 16) & 0xFF)] << 16) | (T[K1 >> 24] << 24);
M[5] = T[(K2 & 0xFF)] | (T[((K2 >> 8) & 0xFF)] << 8);
M[5] |= (T[((K2 >> 16) & 0xFF)] << 16) | (T[K2 >> 24] << 24);
M[6] = T[(K3 & 0xFF)] | (T[((K3 >> 8) & 0xFF)] << 8);
M[6] |= (T[((K3 >> 16) & 0xFF)] << 16) | (T[K3 >> 24] << 24);
M[7] = T[(K4 & 0xFF)] | (T[((K4 >> 8) & 0xFF)] << 8);
M[7] |= (T[((K4 >> 16) & 0xFF)] << 16) | (T[K4 >> 24] << 24);
}
/* Message key introduction */
M[0] ^= MK;

/* Clock M register 256 times */
for(i = 0;i < 256;i++)  {
feed = ((M[0] & 0x80000000L) >> 21);
feed |= ((M[0] & 0x8L) << 6);
feed |= ((M[1] & 0x200L) >> 1);
feed |= ((M[2] & 0x800L) >> 4);
feed |= ((M[2] & 0x8L) << 3);
feed |= ((M[3] & 0x400000L) >> 17);
feed |= ((M[4] & 0x80000L) >> 15);
feed |= ((M[5] & 0x800000L) >> 20);
feed |= ((M[6] & 0x2000000L) >> 23);
feed |= ((M[7] & 0x80000000L) >> 30);
feed |= ((M[7] & 0x4L) >> 2);
M[7] = (M[7] >> 1) | ((M[6] & 1) << 31);
M[6] = (M[6] >> 1) | ((M[5] & 1) << 31);
M[5] = (M[5] >> 1) | ((M[4] &1) << 31);
M[4] = (M[4] >> 1) | ((M[3] &1) << 31);
M[3] = (M[3] >> 1) | ((M[2] &1) << 31);
M[2] = (M[2] >> 1) | ((M[1] &1) << 31);
M[1] = (M[1] >> 1) | ((M[0] &1) << 31);
M[0] = (M[0] >> 1) | (f11[feed] << 31);
}

/* L1 initialization */
*L1++ = M[4]; *L1++ = M[5];
*L1++ = M[6]; *L1   = M[7];
```

```
/* Clock M register 256 times */
for(i = 0;i < 128;i++)  {
feed = ((M[0] & 0x80000000L) >> 21);
feed |= ((M[0] & 0x8L) << 6);
feed |= ((M[1] & 0x200L) >> 1);
feed |= ((M[2] & 0x800L) >> 4);
feed |= ((M[2] & 0x8L) << 3);
feed |= ((M[3] & 0x400000L) >> 17);
feed |= ((M[4] & 0x80000L) >> 15);
feed |= ((M[5] & 0x800000L) >> 20);
feed |= ((M[6] & 0x2000000L) >> 23);
feed |= ((M[7] & 0x80000000L) >> 30);
feed |= ((M[7] & 0x4L) >> 2);
M[7] = (M[7] >> 1) | ((M[6] & 1) << 31);
M[6] = (M[6] >> 1) | ((M[5] & 1) << 31);
M[5] = (M[5] >> 1) | ((M[4] &1) << 31);
M[4] = (M[4] >> 1) | ((M[3] &1) << 31);
M[3] = (M[3] >> 1) | ((M[2] &1) << 31);
M[2] = (M[2] >> 1) | ((M[1] &1) << 31);
M[1] = (M[1] >> 1) | ((M[0] &1) << 31);
M[0] = (M[0] >> 1) | (f11[feed] << 31);
}

/* L2 initialization */
*L2++ = M[0]; *L2++ = M[1];
*L2++ = M[2]; *L2   = M[3];
return;
}

/* encryption/decryption procedure */

/* block contains the output blocks, flag alternatively
clock either L1 (flag = 1) or L2 (flag = 0)
the index of one of the block to choose is returned */

void coscipher(UINT32 *L1, UINT32 *L2,UINT32 *block, UINT32 flag,UINT32 cont)
{
UINT32 feed,tem[4];
unsigned char clk,i,j,k;
unsigned char av[4];

av[0] = 64;
av[1] = 65;
av[2] = 66;
av[3] = 64;
```

```
clk = (L1[3] & 1) | ((L2[3] & 1) << 1);

if(flag) {
for(i = 0L;i < av[clk];i++) {
feed = ((L1[3] & 0x4) >> 2);
feed |= ((L1[3] & 0x20L) >> 4);
feed |= ((L1[3] & 0x100L) >> 6);
feed |= ((L1[3] & 0x8000L) >> 12);
feed |= ((L1[3] & 0x4000000L) >> 22);
feed |= ((L1[2] & 0x40L) >> 1);
feed |= ((L1[2] & 0x1000L) >> 6);
feed |= ((L1[2] & 0x8000L) >> 8);
feed |= ((L1[2] & 0x2000000L) >> 17);
L1[3] = (L1[3] >> 1) | ((L1[2] & 1) << 31);
L1[2] = (L1[2] >> 1) | ((L1[1] & 1) << 31);
L1[1] = (L1[1] >> 1) | ((L1[0] & 1) << 31);
L1[0] = (L1[0] >> 1) | (f9a[feed] << 31);
}
}
else {
for(i = 0L;i < av[clk];i++) {
feed = ((L2[3] & 0x4) >> 2);
feed |= ((L2[3] & 0x20L) >> 4);
feed |= ((L2[3] & 0x100L) >> 6);
feed |= ((L2[3] & 0x8000L) >> 12);
feed |= ((L2[3] & 0x4000000L) >> 22);
feed |= ((L2[2] & 0x40L) >> 1);
feed |= ((L2[2] & 0x1000L) >> 6);
feed |= ((L2[2] & 0x8000L) >> 8);
feed |= ((L2[2] & 0x2000000L) >> 17);
L2[3] = (L2[3] >> 1) | ((L2[2] & 1) << 31);
L2[2] = (L2[2] >> 1) | ((L2[1] & 1) << 31);
L2[1] = (L2[1] >> 1) | ((L2[0] & 1) << 31);
L2[0] = (L2[0] >> 1) | (f9b[feed] << 31);
}
}

//if ( cont == 0x1125L)
//printf("\n%8x,%8x,%8x,%8x",L1[0],L1[1],L1[2],L1[3]);

tem[0] = (L2[0] ^ L1[2]);
tem[1] = (L2[1] ^ L1[3]);
tem[2] = (L2[2] ^ L1[0]);
tem[3] = (L2[3] ^ L1[1]);
```

```
j = (L1[3]^L2[3]) & 3;
k = (L1[0]^L2[0]) >> 30;
if (j == k) { k = k ^ 1;}

*block++ = tem[j];
*block   = tem[k];
}
```