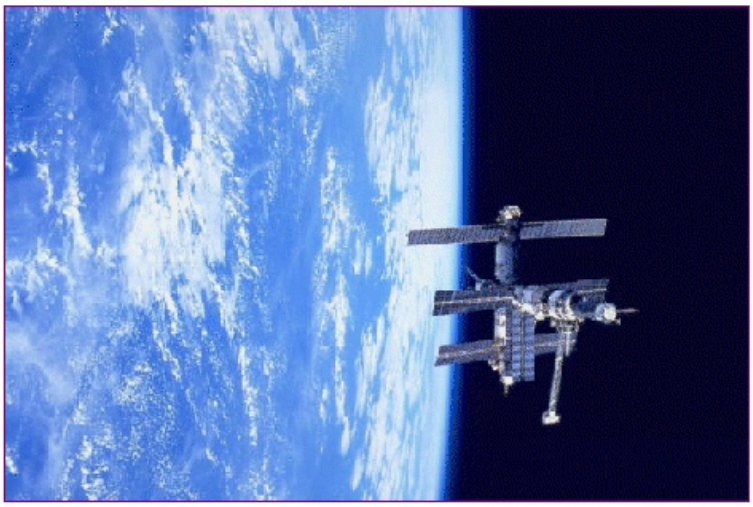
A New Stream Cipher "Mir-1"

Alexander Maximov

Dept. of Information Technology, Lund University, Sweden
P.O. Box 118, 221 00 Lund, Sweden
movax@it.lth.se

Abstract. This document describes a new design of a software oriented stream cipher for academic purposes. This cipher is 64-bit oriented and it uses a T-function to form a loop of a long period. The key size is 128 bits, and IV is 64 bits. Each round 64 bits of keystream are produced.



1 Design Criteria and Motivation of the Choice

1.1 Design Criteria

(a) This is a *software oriented* stream cipher for academic purposes. In hardware one can increase security level performing many operations in parallel. In software we consider the processor to be deterministic, and all the steps are done consecutive. These limits do not allow us to increase the security level of a cipher by parallelising the operations. However, there are some exceptions for some processors, and they could perform a few operations in one clock (like RISC processors);
(b) The cipher should require *rather small internal state memory*. Our cipher needs 256 bytes for a secret S-box table, and 48 bytes for internal registers. When the platform is a PC, then memory, most probably, is not a strict requirement. However, smart cards and other Java programmable chipsets are strict in memory and require smaller internal state;
(c) The cipher should be *efficient in software*, i.e., fast and simple;
(d) It should be *flexible for different platforms*;
(e) The cipher should *resist against different kinds of attack*, and it should have a *rather high security level*.

1.2 Claims Section

(a) This design is a 64 bit oriented stream cipher for software. The key size is 128 bits, and IV is 64 bits. Each round it produces 64 bits of the keystream;
(b) We claim that the security level is at least 2^{128}, i.e., it cannot be "broken" or distinguished faster than exhaustive search;
(c) We do not know any weaknesses in the design, and we did not introduce any trapdoors in the design. We tried to make our motivation of the choice and the description of the design as clear as possible.

1.3 Motivation for the Choice of the Design

(a) **Output word size**. The output word must be a small piece of the internal state, that is required to protect the cipher from different kinds of distinguishing attacks. In our case we output 8 bytes from the internal state;
(b) **Number of operations each round**. To make the cipher to work fast each round should contain as few number of operations as possible. However, the first trade-off is that the minimum number of operations should be compensated with larger internal state;
(c) **Internal state update**. The internal state (registers) should be updated in some way each round. Increasing the size of the internal state we increase the round evaluation time. One of the solution could be to update only a small part of the internal state. This is not good because it will leak the information between two consecutive states, such as, some variable is unchanged and could be eliminated via linear cryptanalysis, for example. This scenario makes the use of large state negligible also. Therefore, we suggest that each word of the internal state should be updated each round;

(d) **S-boxes**. We found that the implementation of an S-box application can be done efficiently, when a word (64 bits) is represented as an array of 8 bytes. In C/C++ it can easily be done by type casting of the variable address pointer to a char pointer. However, one application requires 8 look-up tables. In this design we introduce only one keyed (secret) S-box application as a good shaffler for bytes;

(e) **Why not LFSR?** The implementation of this structure can, actually, be done in a quite efficient way, and this block could work very fast. A good example of an LFSR use is the cipher SNOW 2.0 [1]. For efficient implementation we need to have several *multiplication tables*. This solution might be not very convenient when a cipher should be implemented on a smart card, or some other Java card. Otherwise, LFSRs are good for a software oriented design.

As a possible alternative, one can use a T-function, which can be implemented memoryless. However, they have other minuses, such as they appeared to be much slower, and the security of these functions is not well studied yet as well;

(f) **May be T-functions as a long cycle?** This is one possible alternative to form a pseudo-nonlinear loop of a long period. These functions could substitute LFSRs, and could be implemented memoryless. However, they are not well studied, and also have some leakages. For example, if we have a T-function modulo 2^k, then it is also a T-function modulo 2^m, for $m < k$. It means, for example, that consecutive values modulo 2 will give us the sequence of zeros and ones such as $0, 1, 0, 1, 0, 1, \ldots$, which, obviously, leaks some information. In our design we ignore the first half of the bits (the least significant bits), and accept only the most significant bits for operating with. This, we hope, will prevent such obvious information leakage;

(g) **Internal state size**. The size of the internal state is another trade-off. We decided to have six 64-bit registers: *Loop State* registers x_0, x_1, x_2, x_3, which are combined with a T-function to generate a loop of a long period; and *Automata State* registers A, B also of size 64 bits bits each. They form a "modified" Feistel structure with specified operations. Each round one output word is produced. If the size of the internal state would be less than 256 bits, then it would be weak against such attacks as linear analysis, algebraic attacks, memory trade-off, etc. From the other hand, if the internal state is large, then each round would require more amount of operations to update all the words, and the speed is then undesirable decreasing;

(h) **Circular shifts**. We apply a secret S-box to a word in parallel to the word's bytes independently. To perform byte scrambling we suggest to make a circular shift of the word B to the left by 29 positions. After the second round we will have that the word is circularly shifted by 58 positions to the left, or, 6 positions to the right. It means that we can scramble the bytes of a word as well as the bits inside of each byte via this circular rotation. Since the lowest bits are more weak than the highest bits, then the rotations dismised to the right direction is preferable;

(i) **Key Setup**. The key is 128 bits. The key bits are divided into two halfs, which initialise the automata and loop states independently. Afterwards, the procedure requires to make 8 rounds before the cipher is ready for use;
(j) **IV Setup**. The IV is 64 bits. Setup procedure is done such that any small change in IV would lead to an as random state as possible. Therefore, the bytes of IV are shared between the registers. Afterwards, the procedure requires to make 2 rounds. IV setup is also faster than Key setup procedure.

1.4 Speed Measuring and Security Issues

One round evaluation speed is important for stream ciphers. We consider two ciphers which are interest to compare with. The first is AES [2] – a block cipher, and our aim is to make a stream cipher at least faster than AES. The second is SNOW 2.0 [1] – a stream cipher, which is rather fast, and we can think of this cipher as a standart to compare the evaluation time with.

We decided to give the estimate for the *plain implementation* with one operation per cycle, since we do not know the implementation platform in advance. Nowadays, a usual Pentium IV processor allows us to work with 128 bit numbers (XMM registers). That makes us to believe that this cipher can be redesigned for a 128 bits platform later, with the same number of operations, but increased speed.

Theoretical Speed Estimation: Plain Implementation The first step of speed analysis is the analysis of the plain implementation of the design. Assume our processor can effort only one operation in a time, step by step. Then we call this implementation as *plain implementation*, and we wish to cound the number of operations which are required for a particular algorithm implementation. That can be useful for estimating the algorithm speed in average.

Let us have an algorithm A_h and let us denote the number of operations to perform this algorithm by $T_p(A_h)$. Note that the expressions '$x + y$' and '$z = x + y$' take only one operation. However, just a single operation '$z = x$' also takes one operation.

The real actual implementation with *one processor* cannot take less number of clocks than the number of operations in the plain implementation. However, for the case of two or more conveyers, the work could be distributed, and the real implementation could be much faster.

Actual Speed Measuring: Practical Implementation We would like to note also that this speed measuring was given in [3]. Assume we have an algorithm implementation A_h, and we want to measure the speed of this particular implementation. Measuring procedure could be done in the following way.

First, choose a quite large number of runs of the algorithm for one experiment. We can set this number as a constant $N = 2^{25}$, and it should be quite large for an accurate time estimation.

Then we make two experiments:

```
Experiment 1:
       T1=get current time in clocks
       repeat N times
          empty operator ;
       T2=get current time in clocks

Experiment 2:
       T3=get current time in clocks
       repeat N times
          run algorithm implementation Ah
       T4=get current time in clocks
```

Let us denote the working time in *clocks* of the algorithm A_h as $T_c(A_h)$. Let also the accumulated time in clocks for other expenses, such as loop organization, probably the function A_h calls, etc., be denoted as σ. Then from the experiments above we get an approximate estimates of the speed in clocks.

$$\begin{aligned} \frac{T2-T1}{N} &= \sigma \\ \frac{T4-T3}{N} &= T_c(A_h)+\sigma. \end{aligned} \tag{1}$$

From above equation we can derive the expression for the speed estimate in the number of clocks that the algorithm A_h requires.

$$T_c(A_h) = \frac{(T4-T3)-(T2-T1)}{N}. \tag{2}$$

This is the time estimate of the algorithm implementation A_h in *clocks*, for a particular processor. The value must converge to an integer number as the number of runs N goes to infinity. To get the current clock state we can use the following procedure:

```
_asm
{  xor eax,eax
   cpuid
   rdtsc
   mov T1,eax
   xor eax,eax
   cpuid
}
```

1.5 The Choice of the Name

The name "Mir" is dedicated to the first space station which was also called "Mir". The space station was launched by Russians on February 20 1986, and was sinked in the Pacific ocean on March 23 2001. Actually, the word 'Mir' in

Russian has two meanings 'World' and 'Peace'. Unfortunately, we do not know exactly which one was meant when they decided to call the space station by this name, and this question is difficult for us. However, during its working period in the space a lot of scientific results were achieved. That is why we give the name to our design "Mir" in respect to science. As far as this is our the first attempt to make a stream cipher of such kind, we add "-1" to the name, as it is our the first version. The final name is "Mir-1".

2 Design Description

2.1 Notations and Definitions

In our paper $\oplus$, &, and | are a bitwise XOR, AND, and OR on two arguments, whereas $\boxplus$ and $\cdot$ denote arithmetical addition and multiplication operations with truncation by 64 bits. $x \lll t$ and $x \ggg t$ denote cyclic shift on a word x by t bits to the left and right, respectively. $x \ll t$ and $x \gg t$ denote a simple shift of a word x by t positions to the left and right, respectively; note that in this case $64 - t$ and t bits of the word x will be lost, respectively. If some variable X is represented in a binary form as $\overline{x_{63} \dots x_1 x_0}$, then the notation $X[a, b]$ denotes an integer number which has the binary representation $\overline{x_b \dots x_a}$. A 64-bit word can also be represented as a concatenation of bytes, and denoted as $X = (X.\mathtt{byte}_7 || \dots || X.\mathtt{byte}_1 || X.\mathtt{byte}_0)$; or it can also represented as concatenation of 32-bit words as $X = (X.\mathtt{half}_1 || X.\mathtt{half}_0)$. In this design we also use three constants $C_0 = 0x1248842112488421$, $C_1 = 0x1248124812481248$ and $C_3 = 0x4812481248124812$.

- A *word* is a nonnegative 64-bit integer;
- *Loop State (LS)* are 4 words registers x_0, x_1, x_2, x_3;
- *Automata State (AS)* are 2 words registers A, B;
- *Internal State (IS)* is the combination of words from LS and AS, and a secret S-box of size 256 bytes;
- *Plaintext (PT)* is a sequence of words $\mathbf{P}$=$p_0, p_1, \dots$;
- *Keystream (KS)* is a sequence of words $\mathbf{Z}$=$z_0, z_1, \dots$;
- *Ciphertext (CT)* is a sequence of words $\mathbf{C}$=$c_0, c_1, \dots$;
- *Secret Key (Key)* are 16 bytes $k_0, \dots, k_{15}$;
- *Initial value (IV)* are 8 bytes $IV_0, \dots, IV_7$;

2.2 Encryption and Decryption

Encryption and decryption is a XOR with the keystream

$$\mathbf{C} = \mathbf{P} \oplus \mathbf{Z}. \tag{3}$$

One round function looks as follows:

1. `Encrypt(`p_i`)`

```
2.    Loop State Update
3.    Automata State Update
4.    return c_i = p_i ⊕ B
```

```
1. Decrypt(c_i)
2.    return Encrypt(c_i)
```

2.3 Loop State Update

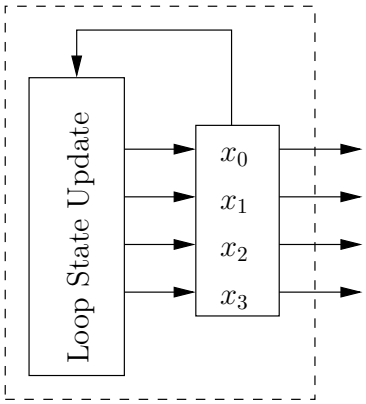


Fig. 1. Loop State update block

Update of the LS is a T-function, first studied and introduced in [4, 5]. The LS update is illustrated in Figure 1.

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \rightarrow \begin{pmatrix} x_0 \boxplus (s) & \boxplus 2 \cdot x_2 \cdot (x_1 | C_1) \\ x_1 \boxplus (s \& x_0) & \boxplus 2 \cdot x_2 \cdot (x_3 | C_3) \\ x_2 \boxplus (s \& x_0 \& x_1) & \boxplus 2 \cdot x_0 \cdot (x_3 | C_3) \\ x_3 \boxplus (s \& x_0 \& x_1 \& x_2) & \boxplus 2 \cdot x_0 \cdot (x_1 | C_1) \end{pmatrix}, \tag{4}$$

where $s = (x_0 \& x_1 \& x_2 \& x_3 \boxplus C_0) \oplus x_0 \& x_1 \& x_2 \& x_3$, and C_0, C_1, C_3 are three constants given in sub Section 2.1.

2.4 Automata State Update

The structure of the AS update function is shown in Figure 2. First we apply the secret S-box to each byte of B in parallel, and the result is xored with A, i.e., $A.\mathtt{byte}_i = A.\mathtt{byte}_i \oplus S[B.\mathtt{byte}_i]$. Then A is padded with a word $(x_2.\mathtt{half}_1 || x_0.\mathtt{half}_1)$. Register B is then rotated by 29 positions to the left, and ariphmetically added with the register A and the word $(x_3.\mathtt{half}_1 || x_1.\mathtt{half}_1)$. Finally, the words A and B are exchanged.

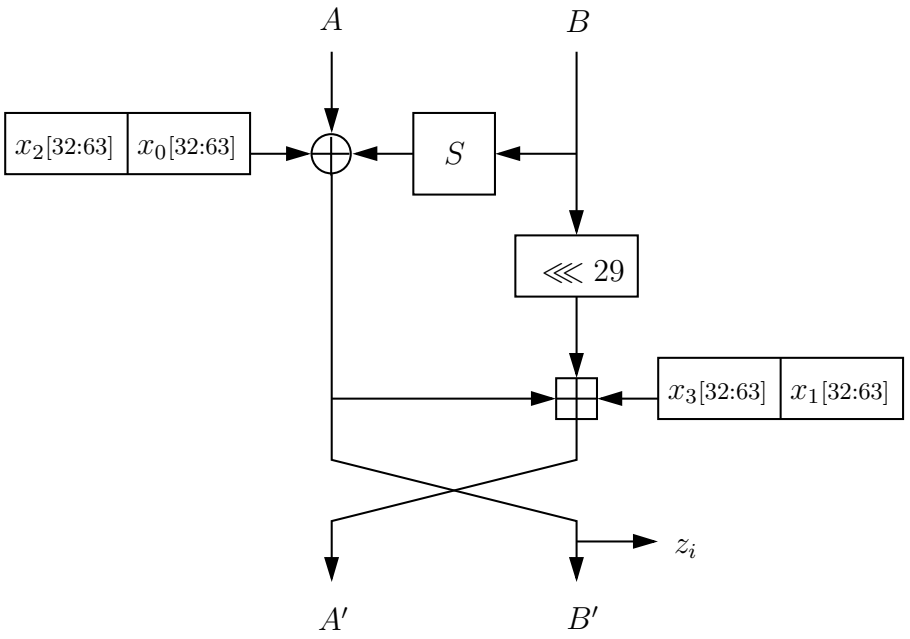


Fig. 2. Automata State Update

2.5 Key Setup

The key is of size 128 bits and given as an array of 16 bytes $Key = (k_{15}, \ldots, k_0)$. First, the secret S-box is initialized as follows.

$$S[i] = SR[\ldots SR[SR[i \oplus k_0] \oplus k_1] \oplus \ldots \oplus k_{15}], \quad \text{for all } i = 0, \ldots, 255, \qquad (5)$$

where $SR[\cdot]$ is a fixed Rijndael S-box, which is also given in Appendix B.

Afterwards, the following procedure is performed.

```
1. Key Setup ((k15, ..., k0))
2.   Initialise secret S-box as shown above
3.   A = x1 = (k7|| ... ||k0)
4.   B = x3 = (k15|| ... ||k8)
5.   x0 = C0
6.   x2 = C1
7.   Repeat 8 times
8.     Loop State Update
9.     Automata State Update
```

The constants C_0 and C_1 are the same as in the LS update function, and also given in sub Section subsec:not.

2.6 IV Setup

The IV is of size 64 bits and given as an array of 8 bytes $IV = (IV_7, \ldots, IV_0)$. This setup function is given below.

```
1. IV Setup((IV_7, ..., IV_0))
2.    x_0.byte_4 = x_0.byte_4 ⊕ S[IV_0] ⊕ S[IV_1] ⊕ S[IV_2]
3.    x_1.byte_4 = x_1.byte_4 ⊕ S[IV_0] ⊕ S[IV_3] ⊕ S[IV_4]
4.    x_2.byte_4 = x_2.byte_4 ⊕ S[IV_2] ⊕ S[IV_5] ⊕ S[IV_7]
5.    x_3.byte_4 = x_3.byte_4 ⊕ S[IV_3] ⊕ S[IV_6] ⊕ S[IV_7]
6.    x_0.byte_0 = x_0.byte_0 ⊕ S[IV_3] ⊕ S[IV_5]
7.    x_1.byte_0 = x_1.byte_0 ⊕ S[IV_7] ⊕ S[IV_6]
8.    x_2.byte_0 = x_2.byte_0 ⊕ S[IV_0] ⊕ S[IV_1]
9.    x_3.byte_0 = x_3.byte_0 ⊕ S[IV_2] ⊕ S[IV_4]
10.   A.byte_0 = A.byte_0 ⊕ S[IV_0] ⊕ S[IV_5] ⊕ S[IV_6]
11.   A.byte_4 = A.byte_4 ⊕ S[IV_1] ⊕ S[IV_3] ⊕ S[IV_5]
12.   B.byte_0 = B.byte_0 ⊕ S[IV_1] ⊕ S[IV_4] ⊕ S[IV_7]
13.   B.byte_4 = B.byte_4 ⊕ S[IV_2] ⊕ S[IV_4] ⊕ S[IV_6]
14.   Repeat 2 times
15.     Loop State Update
16.     Automata State Update
```

3 Speed Measuring

We estimated the *plain implementation* (one operation per time is allowed), and also a *C++ implementation*. We give the detailed table with speed parameters.

Algorithm	Plain implementation (number of operations)	C++ implementation (processor's clocks)
Key Setup	around 256×16 op. for S-box 6 op. for LS and AS registers init additionally 8 rounds	11149 clocks ($2^{-30} \cdot 4972$ sec.)
IV Setup	32 op. for LS and AS registers update additionally 2 rounds	693 clocks ($2^{-30} \cdot 310$ sec.)
One Round	40 op.	314 clocks
$\cdot$ *LS Update*	24 op.	($2^{-30} \cdot 141$ sec.)
$\cdot$ *AS Update*	16 op.	(454 MBit/sec.)

The estimations were done on a usual PC computer, 32-bit operation system Windows XP, 32-bit compiler Visual C/C++ 6.0, with the processor Pentium IV, 2.40GHz and memory 1Gb.

The speed of this design appers to be slightly less (or even similar) than the speed of AES. The reason is that the 64-bit multiplication operations are very expensive. The T-function has four multiplications, which require as much time as all the other operations in the round loop. It means that T-function are very expensive. Another reason could be that the compiler and the platform are 32-bit oriented, and the work with 64-bit variables is slow. In the nearest feauture

64-bit Windows XP will appear, and the beta-edition of Visual C/C++ has already released. We hope that on a new platform this cipher can work faster. However, this design does not need much memory and it is flexible for different platforms.

4 Conclusions

In this work we tried to use a T-function to substitute an LFSR, make the cipher to be more resistent against algebraic attacks, and make it more flexible for different platforms with a little memory requirement. Another important criteria for us was a very high security level, which we clame to be at least 2^{128}.

During our work on this design we also found several implementation issues. In particular, we found that T-functions appear to be quite expensive in time, and are not as efficient as it was stated by Shamir and Klimov before. The implementation of such functions is even much slower than LFSRs. One of the reasons why they were studied was that T-functions can perhaps efficiently resist against algebraic attacks, since the multiplication operation scrambles the bits a lot. However, the multiplication operation appeared to be a very slow operation for common PC processors. Just four multiplications require as much time as all the rest operations in the cipher. One more minus is that the security level of T-functions is also not well studied yet. However, we found them interesting from other objectives stated before, such as their resistance against algebraic attacks and other positives.

We believe that such design could be used in smart cards or other microprocessors.

References

1. P. Ekdahl and T. Johansson. A new version of the stream cipher SNOW. In K. Nyberg and H. Heys, editors, *Selected Areas in Cryptography—SAC 2002*, volume 2595 of *Lecture Notes in Computer Science*, pages 47–61. Springer-Verlag, 2002.
2. J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer-Verlag, 2002.
3. M. Matsui and S. Fukuda. How to Maximize Software Performance of Symmetric Primitives on Pentium III and 4 Processors. In *Fast Software Encryption 2005*. Springer-Verlag, 2005.
4. A. Klimov and A. Shamir. A new class of invertible mappings. In *CHES '02: Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 470–483, London, UK, 2003. Springer-Verlag.
5. A. Klimov and A. Shamir. New Applications of T-functions in Block Ciphers and Hash Functions. In *Fast Software Encryption 2005*. Springer-Verlag, 2005.

[0] The work described in this paper has been supported in part by Grant VR 621-2001-2149, in part by the Graduate School in Personal Computing and Communication PCC++, and in part by the European Commission through the IST Program under Contract IST-2002-507932 ECRYPT.

Appendix A: Test Vectors

Key: $k_{15}, \ldots, k_0$	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
IV: $IV_7, \ldots, IV_0$	00 00 00 00 00 00 00 00
After key setup	
$x_0.\texttt{byte}_7, \ldots, x_0.\texttt{byte}_0$	47 40 4C C7 14 EA 22 3D
$x_1.\texttt{byte}_7, \ldots, x_1.\texttt{byte}_0$	CD 78 28 56 65 82 51 E8
$x_2.\texttt{byte}_7, \ldots, x_2.\texttt{byte}_0$	08 E1 F0 10 4E 88 A0 92
$x_3.\texttt{byte}_7, \ldots, x_3.\texttt{byte}_0$	91 8A A3 B0 D2 37 D3 61
$A.\texttt{byte}_7, \ldots, A.\texttt{byte}_0$	A1 70 23 49 22 4E 5F C1
$B.\texttt{byte}_7, \ldots, B.\texttt{byte}_0$	EE B1 D4 0A 14 B6 D1 C7
After IV setup	
$x_0.\texttt{byte}_7, \ldots, x_0.\texttt{byte}_0$	7A 66 37 73 33 2C D6 1F
$x_1.\texttt{byte}_7, \ldots, x_1.\texttt{byte}_0$	5C AD A9 04 E8 F2 BA 55
$x_2.\texttt{byte}_7, \ldots, x_2.\texttt{byte}_0$	6D 4F 20 0A 23 6E C3 D4
$x_3.\texttt{byte}_7, \ldots, x_3.\texttt{byte}_0$	E7 1F 73 BD 7F DE 7F FD
$A.\texttt{byte}_7, \ldots, A.\texttt{byte}_0$	49 E2 D1 B0 C9 D2 F7 99
$B.\texttt{byte}_7, \ldots, B.\texttt{byte}_0$	62 E2 39 00 0C E1 7B C1
The first keystream values	
Keystream	C5EC3D7E 3C0A7145 69050CF2 6E002DB5 ...
Key: $k_{15}, \ldots, k_0$	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
IV: $IV_7, \ldots, IV_0$	00 00 00 00 00 00 00 01
After key and IV setup	
$x_0.\texttt{byte}_7, \ldots, x_0.\texttt{byte}_0$	A6 53 3A 71 99 50 40 13
$x_1.\texttt{byte}_7, \ldots, x_1.\texttt{byte}_0$	90 65 12 F6 73 A1 D4 6D
$x_2.\texttt{byte}_7, \ldots, x_2.\texttt{byte}_0$	54 E9 C3 78 DE D1 4B CE
$x_3.\texttt{byte}_7, \ldots, x_3.\texttt{byte}_0$	D8 7C 71 C6 0F 45 CD 6D
$A.\texttt{byte}_7, \ldots, A.\texttt{byte}_0$	77 CF 78 BF D9 19 B4 B3
$B.\texttt{byte}_7, \ldots, B.\texttt{byte}_0$	26 2E EC B8 D6 78 78 49
The first keystream values	
Keystream	1DCFBAFB C0AC03B7 BFD4F59A 2BB729DB ...
Key: $k_{15}, \ldots, k_0$	0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00
IV: $IV_7, \ldots, IV_0$	07 06 05 04 03 02 01 00
After key setup	
$x_0.\texttt{byte}_7, \ldots, x_0.\texttt{byte}_0$	CD 63 A0 93 0B 20 28 3D
$x_1.\texttt{byte}_7, \ldots, x_1.\texttt{byte}_0$	A7 4F FB E8 FF 12 34 E8
$x_2.\texttt{byte}_7, \ldots, x_2.\texttt{byte}_0$	BE B6 78 2E 66 BB 40 72
$x_3.\texttt{byte}_7, \ldots, x_3.\texttt{byte}_0$	13 16 7D 0E 7B 6D 9B 29
$A.\texttt{byte}_7, \ldots, A.\texttt{byte}_0$	B2 35 0C 29 55 F7 43 52
$B.\texttt{byte}_7, \ldots, B.\texttt{byte}_0$	AE F1 8E 28 AE 75 CE EF
After IV setup	
$x_0.\texttt{byte}_7, \ldots, x_0.\texttt{byte}_0$	A0 0C 75 DE E3 72 B3 97
$x_1.\texttt{byte}_7, \ldots, x_1.\texttt{byte}_0$	1D C4 68 5D D8 BB FE 62
$x_2.\texttt{byte}_7, \ldots, x_2.\texttt{byte}_0$	E7 93 97 4A 51 1D 81 D1
$x_3.\texttt{byte}_7, \ldots, x_3.\texttt{byte}_0$	CE 9B 86 FF 18 C6 E4 B3
$A.\texttt{byte}_7, \ldots, A.\texttt{byte}_0$	4F CF E0 7C 0B 44 73 7F
$B.\texttt{byte}_7, \ldots, B.\texttt{byte}_0$	89 A1 CD 6F 1A 23 94 CF
The first keystream values	
Keystream	DBDB0F48 B40CBCA0 84EB4EE0 5683AD37 ...

Appendix B: Rijndael *S*-box

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

Appendix C: C++ Object "MirCipher"

```
// ================================================================
// Types and Constants
// ================================================================
typedef unsigned long long ull;
typedef unsigned long ul;
typedef unsigned char uc;

union Int
{ struct
  { uc b0, b1, b2, b3, b4, b5, b6, b7;
  } b;     // 8 -bit access
  struct
  { ul w0, w1;
  } w;     // 32-bit access
  ull v;   // 64-bit access
};
```

```
// ================================================================
// Design Block
// ================================================================
#define Mir1_STATE_UPDATE                    \
  Int r0, r1, t0, t1, p0, p1, s, m0, m1, m2, m3; \
  t0.v    = x0.v << 1;                       \
  p1.v    = x3.v | C3;                       \
  p0.v    = x1.v | C1;                       \
  t1.v    = x2.v << 1;                       \
  m2.v    = p1.v * t0.v;                     \
  m3.v    = p0.v * t0.v;                     \
  m0.v    = p0.v * t1.v;                     \
  m1.v    = p1.v * t1.v;                     \
  r0.v    = x0.v & x1.v;                     \
  a.b.b0 ^= S[b.b.b0];                       \
  a.b.b1 ^= S[b.b.b1];                       \
  r1.v    = r0.v & x2.v;                     \
  a.b.b2 ^= S[b.b.b2];                       \
  a.b.b3 ^= S[b.b.b3];                       \
  s.v     = r1.v & x3.v;                     \
  a.b.b4 ^= S[b.b.b4];                       \
  a.b.b5 ^= S[b.b.b5];                       \
  s.v     = (s.v + C0) ^ s.v;                \
  a.b.b6 ^= S[b.b.b6];                       \
  a.b.b7 ^= S[b.b.b7];                       \
  x3.v   += (s.v & r1.v) + m3.v;             \
  x2.v   += (s.v & r0.v) + m2.v;             \
  r0.v    = (b.v << 29);                     \
  x1.v   += (s.v & x0.v) + m1.v;             \
  r0.v   |= (b.v >> 35);                     \
  x0.v   +=  s.v         + m0.v;             \
  r0.v   += x1.w.w1;                         \
  r0.w.w1+= x3.w.w1;                         \
  b.w.w0  = a.w.w0 ^ x0.w.w1;                \
  b.w.w1  = a.w.w1 ^ x2.w.w1;                \
  a.v     = r0.v + b.v;

// ================================================================
// Mir-1 Cipher
// ================================================================
class MirCipher
{ private:
        const ull C0, C1, C3;          // constants
  public:
```

```
    Int a, b, x0, x1, x2, x3;      // internal state
    uc S[256];                     // secret S-box

// ----------------------------------------
// Constructor
// ----------------------------------------
MirCipher(void): C0(0x1248842112488421),
                 C1(0x1248124812481248),
                 C3(0x4812481248124812) {}

// ----------------------------------------
// Key Setup
// ----------------------------------------
inline void KeySetup(uc * key) // 16 bytes of the key
{ int i, j;
  for(i=0; i<256; ++i)
    for(S[i]=i, j=0; j<16; ++j)
      S[i] = SR[ S[i] ^ key[j] ];
  a.v  = x1.v = ((ull*)key)[0];
  b.v  = x3.v = ((ull*)key)[1];
  x0.v = C0;
  x2.v = C1;
  for(i=0; i<8; ++i) { Mir1_STATE_UPDATE }
}

// ----------------------------------------
// IV Setup
// ----------------------------------------
inline void IVSetup(uc * IV)   // IV is a 64 bit value
{ x0.b.b4 ^= S[IV[0]] ^ S[IV[1]] ^ S[IV[2]];
  x1.b.b4 ^= S[IV[0]] ^ S[IV[3]] ^ S[IV[4]];
  x2.b.b4 ^= S[IV[2]] ^ S[IV[5]] ^ S[IV[7]];
  x3.b.b4 ^= S[IV[3]] ^ S[IV[6]] ^ S[IV[7]];
  x0.b.b0 ^= S[IV[3]] ^ S[IV[5]];
  x1.b.b1 ^= S[IV[7]] ^ S[IV[6]];
  x2.b.b2 ^= S[IV[0]] ^ S[IV[1]];
  x3.b.b3 ^= S[IV[2]] ^ S[IV[4]];
  a.b.b0  ^= S[IV[0]] ^ S[IV[5]] ^ S[IV[6]];
  a.b.b4  ^= S[IV[1]] ^ S[IV[3]] ^ S[IV[5]];
  b.b.b0  ^= S[IV[1]] ^ S[IV[4]] ^ S[IV[7]];
  b.b.b4  ^= S[IV[2]] ^ S[IV[4]] ^ S[IV[6]];
  { Mir1_STATE_UPDATE }
  { Mir1_STATE_UPDATE }
}
```

```
    // ----------------------------------------
    // Encryption/Decryption
    // ----------------------------------------
    inline ull EncDec(ull z)
    { Mir1_STATE_UPDATE
      return b.v^z;
    }
};
```