Some Remarks on the TKIP Key Mixing Function of IEEE 802.11i

Wei Han [1,2], Dong Zheng[1,2], Ke-fei Chen[1]
(1. Dept. of Computer Sci. & Eng., Shanghai Jiaotong Univ., Shanghai 200240, P.R.China)
(2. China National Laboratory for Modern Communications, Chengdu 610041, P.R.China)

Abstract: Temporal Key Integrity Protocol (TKIP) is a sub-protocol of IEEE 802.11i. TKIP remedies some security flaws in Wired Equivalent Privacy (WEP) Protocol. TKIP adds four new algorithms to WEP: a Message Integrity Code (MIC) called Michael, an Initialization Vector (IV) sequencing discipline, a key mixing function and a re-keying mechanism. The key mixing function, also called temporal key hash, de-correlates the IVs from weak keys. Some cryptographic properties of the S-box used in the key mixing function are investigated in this paper, such as regularity, avalanche effect, differ uniform and linear structure. V.Moen, H.Raddum and K.J.Hole pointed out that there existed a temporal key recovery attack in TKIP key mixing function. In this paper a method is proposed to defend against the attack, and the resulting effect on performance is also discussed.

Key words: WLAN, 802.11i, TKIP, S-box

1 Introduction

In July 2004 IEEE published the 802.11i specification [1] for WLAN. 802.11i is a remedy for the flawed Wired Equivalent Privacy (WEP) Protocol in 802.11. In 802.11i RC4 encryption is used for 802.11 legacy devices and AES [2] is used for future 802.11 devices. A sub-protocol called Temporal Key Integrity Protocol (TKIP) is included in 802.11i to prevent the improper use of RC4 encryption in WEP, such as weak key scheduling [3], Initialization Vector (IV) collusion and packet forgery [4]. TKIP is a set of algorithms wrapping WEP. TKIP adds four new algorithms to WEP: a cryptographic Message Integrity Code (MIC) called Michael to exclude forged packets, an IV sequencing discipline to remove the replay attack, a per-packet key mixing function to de-correlate the IVs from weak keys and a re-keying mechanism to provide fresh encryption and integrity keys.

In this paper we focus on the TKIP key mixing function. The key mixing function, also called temporal key hash, produces the 128-bit RC4 per-frame encryption key. This function takes as input the 128-bit Temporal Key (TK), the 48-bit Transmitter’s Address (TA) and 48-bit IV. The 48-bit IV is often called the TKIP Sequence Counter (TSC). The 32 most significant bits of the TSC are represented by IV32 and the 16 least significant bits of the TSC are represented by IV16 here. The key mixing function outputs 128-bit WEP key, the three first bytes of which are derived from the TSC.

This work is supported by the China National Laboratory for Modern Communications under Grant No.51436050304JW0317. Email: han_wei@sjtu.edu.cn

TKIP key mixing has two phases. The input to phase 1 shall be the TK, TA and IV32. The output shall be 80-bit Phase 1 Key (P1K). The P1K will be one part of input to phase 2. P1K is the same for 2^{16} consecutive frames from the same TK, TA and IV32. So P1K is often calculated once for every 2^{16} frames and is cached for the next phase, although it can be calculated every frame in theory. In phase 2 it takes as input P1K, TK and IV16, and outputs the 128-bit WEP key for the RC4 encryption algorithm. The key mixing process can be described as follows:

P1K = Phase1 (TK, TA, IV32)

RC4Key = Phase2 (P1K, TK, IV16)

For lack of space, we only give a rough description of the key mixing function. More details can be referred to [1].

2 S-box in TKIP

In both phase 1 and phase 2 of the key mixing function an S-box is employed. The S-box is a non-linear substitution. It takes one 16-bit value as input and output one value with the same bit length. The S-box substitution can be viewed as a table look-up. The table look-up can be a single table with 2^{16} entries and a 16-bit index, or two tables with 256 entries and an 8-bit index. In [1] the latter reference implementation is demonstrated, but IEEE Task Group I (TGi) does not expound the design principle for the S-box explicitly. By analyzing the C-language reference implementation, we derive the generation method of the S-box. It can be illustrated below:

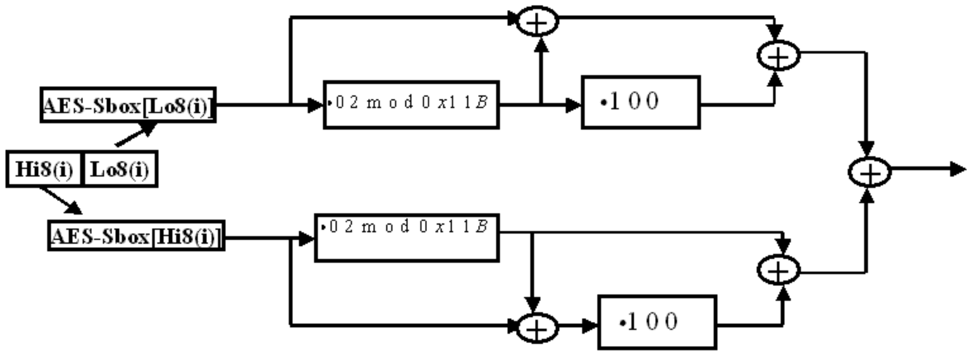

Figure 1. How to generate the S-box in TKIP

In Figure 1, the input is one 16-bit index value ‘i’. The 16-bit input is split into two 8-bit values, i.e. Lo8(i) and Hi(i). Lo8(i) represents the 8 least significant bits of the input and Hi8(i) represents the 8 most significant bits. The AES S-box substitution will be performed with Lo8(i) and Hi8(i) as index respectively. ‘$\bullet 02 \bmod 11B$’ means the ‘xtime’ operation in Rijndael, that is, the multiplication by x (hexadecimal ‘02’) modulo the irreducible binary polynomial $x^8 + x^4 + x^3 + x + 1$ (hexadecimal ‘11B’). ‘$\oplus$’ represents bit-wise XOR. ‘$\bullet 100$’ means the multiplication by x^8 (hexadecimal ‘100’). The operation ‘$\bullet 100$’ extends an 8-bit value to a 16-bit value, of which the 8 most significant bits come from the left shift by 8 of the original octet, and the 8 least significant bits are filled with 0.

S-boxes have been widely employed in private key cryptosystem. The S-box plays an important role in the symmetric cryptosystem since it is the only nonlinear portion of the algorithm. The strength of the cryptosystem depends heavily on the quality of the S-box. Much research has been concentrated on how to design a cryptographically ‘good’ S-box. A variety of criteria are presented in the S-box design.

Denote by F_2^n the vector space of n tuples of elements from $GF(2)$. In general, an $n \times m$ S-box, where $n \geq m$, $n, m \geq 1$, can be described as a function from F_2^n to F_2^m, or a vectorial Boolean function $F(x_1, ... x_n) = (f_1, ..., f_m)$ (when $m = 1$ is called Boolean function). Next we will investigate some cryptographic properties of the S-box in TKIP:

Definition 1. ([5]) An S-box F is regular, if $F(x)$ runs through all vectors in F_2^m each 2^{n-m} times while x runs through F_2^n once.

An S-box should be regular to avoid trivial statistical attacks.

Proposition 1. The S-box in TKIP is regular.

Proof: The S-box in TKIP F can be regarded as a mapping from F_2^{16} to F_2^{16}. When the input assumes the values from 0 to $2^{16} - 1$, we can compute the output $F(0), F(1), \ldots F(2^{16} - 1)$ and store them in an array. If any two elements of the array are different, the array is a permutation of $0, 1, ..., 2^{16} - 1$ and Proposition 1 holds. The comparison is made and the elements in the array are pair-wise different.

Definition 2. ([6][7]) A cryptographic function satisfies the strict avalanche criterion (SAC), if each output bit should change with a probability of one half whenever a single input bit is complemented.

Proposition 2. The S-box in TKIP does not satisfy the SAC. For the S-box, the probability that a particular output bit will change when a single input bit is complemented ranges from 0.43 to 0.56.

Proof: We compute the probability that the j-th (j=1,2,…,16) output bit will change when the i-th (i=1,2,…,16) input bit is complemented. The result is listed in a table in the appendix.

Definition 3. ([5]) Let F be an $n \times m$ S-box, let δ be the largest value in differential distribution table of the S-box (not counting the first entry in the first row), namely,

$$\delta = \max |\{x \mid F(x) \oplus F(x \oplus \alpha) = \beta\}|, \text{ for } \alpha, \beta \text{ satisfy } \alpha \in F_2^n, \alpha \neq 0, \beta \in F_2^m$$

F is said to be differentially δ-uniform, and δ is called the differential uniformity of F.

The differential uniformity δ of an S-box is a nonlinearity criterion that can measure the strength of the S-box against differential cryptanalysis [8].

Proposition 3. The differential uniformity of the S-box in TKIP is 1024.

Proof: The difference distribution table of the TKIP 16×16 S-box is a $2^{16}\times2^{16}$ matrix. An entry in the table indexed by (α,β) indicates the number of input vectors which, when changed by α result in a change in the output by β (the changes both mean bit-wise XOR). The first row of the matrix is $(2^{16},0,...,0)$ and is not counted. We compute the difference distribution table of the S-box in TKIP. By an exhaustive search the maximum is found as1024.

Definition 4. ([9]) For a Boolean function $f(x_1,..,x_n)$, define

$$\| f(x_1,...,x_n)\| \triangleq |\{(x_1,...,x_n) \mid f(x_1,...,x_n)=1\}|$$

$$N_f \triangleq \min_{a_0,...a_n} \| f(x_1,...,x_n)\oplus(a_0\oplus a_1x_1\oplus\cdots\oplus a_nx_n)\|$$

N_f is called the nonliearity of f and it denotes the minimal Hamming distance between f and the set of affine functions $\{a_0\oplus a_1x_1\oplus\cdots\oplus a_nx_n\}$. For a vectorial Boolean function $F(x_1,...x_n)=(f_1,...,f_m)$, the nonlinearity N_F is defined as $N_F \triangleq \min\limits_{(c_1,...,c_m)\neq(0,...,0)} N_{c_1f_1\oplus\cdots\oplus c_mf_m}$.

N_f is a nonlinearity criterion that can measure the strength of the S-box against linear cryptanalysis [10]. It is known that $N_f \le 2^{n-1}-2^{(n/2)-1}$ and $N_F \le 2^{n-1}-2^{(n/2)-1}$, and the equal marks hold if and only if $f(x_1,...,x_n)$ is a bent function and $F(x_1,...x_n)=(f_1,...,f_m)$ is a (n,m)-bent function. Since the algebraic normal form of the S-box in TKIP is too complicated to compute, we only know the upper bound of the TKIP S-box nonlinearity is $2^{n-1}-2^{(n/2)-1}-2=32638$ from the Corollary 7 of [11]. No tighter bounds have been known yet.

Definition 5. ([12][13]) Let F be a function defined from F_2^n to F_2^m, (α,c) is an element of $F_2^n\times F_2^m$ with $\alpha\neq0$. (α,c) is a linear structure of F if $F(x)+F(x+\alpha)=c$ holds for all $x\in F_2^n$, '+' represents bit-wise XOR.

The existence of linear structures is a weakness in ciphers.

Proposition 5. The S-box in TKIP has no linear structure.

Proof: First we view the TKIP 16×16 S-box as a function F defined from F_2^{16} to F_2^{16}. We fix a nonzero vector $\alpha \in F_2^{16}$. Next we check if the values $F(x)+F(x+\alpha)$ are equal for all of the argument $x \in F_2^{16}$. If a vector α satisfies the property, $(\alpha, F(x)+F(x+\alpha))$ is a linear structure. We make an exhaustive search when the vector α ranges from 1 to $2^{16}-1$ and no linear structure is found.

3 Temporal Key Recovery Attack and Countermeasure

V.Moen, H.Raddum and K.J.Hole discovered a temporal key recovery attack in TKIP key mixing function [14]. If the attacker can collect a few RC4 WEP keys computed under the same IV32, he is able to recover the TK and MIC key. As claimed in [14], this attack is only in a theoretical sense, whereas it shows the key hash function is weaker than expected. The Leak of RC4 keys are equivalent to total loss of security.

The inputs to phase 1 of the key mixing function are TK, TA and IV32. The outputs of phase 1 are 80-bit P1K. The inputs to phase 2 are TK, P1K and IV16. The RC4 key is the output of phase 2 and changes when the value of IV16 varies. When RC4 keys are computed under the same IV32 for one mobile station, The P1K-values of 2^{16} consecutive RC4 keys are the same. The algorithm of phase 2 is as follows:

```
Input: intermediate key P1K[0], ..., P1K[4], TK and IV16
Output: RC4Key[0], ..., RC4Key[15]

Phase 2 Step 1:
      PPK[0] = P1K[0]
      PPK[1 ]= P1K[1]
      PPK[2] = P1K[2]
      PPK[3] = P1K[3]
      PPK[4] = P1K[4]
      PPK[5] = P1K[4] + IV16
Phase 2 Step 2:
      PPK[0] = PPK[0] + S[PPK5 ⊕ Mk16(TK[1],TK[0])]
      PPK[1] = PPK[1] + S[PPK0 ⊕ Mk16(TK[3],TK[2])]
      PPK[2] = PPK[2] + S[PPK1 ⊕ Mk16(TK[5],TK[4])]
      PPK[3] = PPK[3] + S[PPK2 ⊕ Mk16(TK[7],TK[6])]
      PPK[4] = PPK[4] + S[PPK3 ⊕ Mk16(TK[9],TK[8])]
      PPK[5] = PPK[5] + S[PPK4 ⊕ Mk16(TK[11],TK[10])]

      PPK[0] = PPK[0] + RotR1(PPK[5] ⊕ Mk16(TK[13],TK[12]))
      PPK[1] = PPK[1] + RotR1(PPK[0] ⊕ Mk16(TK[15],TK[14]))
      PPK[2] = PPK[2] + RotR1(PPK[1])
      PPK[3] = PPK[3] + RotR1(PPK[2])
      PPK[4] = PPK[4] + RotR1(PPK[3])
      PPK[5] = PPK[5] + RotR1(PPK[4])
Phase 2 Step 3:
      RC4Key[0] = Hi8(IV16)
      RC4Key[1] = (Hi8(IV16) | 0x20) & 0x7F
      RC4Key[2] = Lo8(IV16)
      RC4Key[3] = Lo8( (PPK[5] ⊕ Mk16(TK[1],TK[0])) >>1 )
      for i = 0 to 5
      {
        RC4Key[4+2*i] = Lo8(PPK[i])
        RC4Key[5+2*i] = Hi8(PPK[i])
      }
```

Figure.2 The Algorithm of Key Mixing Phase 2

In Figure 2, ‘&’, ‘|’, ‘$\oplus$’ denote bit-wise AND, OR, XOR respectively. ‘+’ denotes addition mod 2^{16}. RotR1 denotes right circular shift by 1 bit. ‘>>1’ denotes right 1 bit shift (The authors of [14] mistake ‘>>1’ for right circular shift by 1). Mk16(X, Y)=256*X+Y=X||Y. ‘||’ denotes concatenation. P1K is treated as arrays of P1K[0…4] of 16-bit word, and TK and RC4key are viewed as arrays TK[0…15] and RC4Key[0…15] of 8-bit byte respectively.

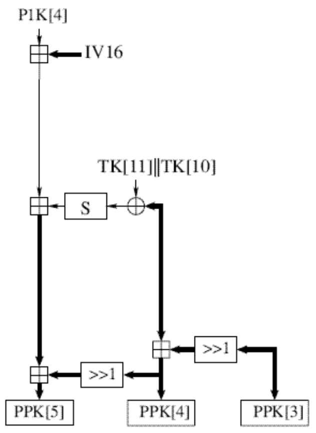

Figure 3. Part of Phase 2 needed to compute TK[10] and TK[11], cited from Refs.[14]

Figure 3 is an illustration on how to recover the unknown TK[10] and TK[11] . PPK[3], PPK[4] and PPK[5] is known from an RC4 key. We can guess the value of TK[11]||TK[10]. Then we can compute P1K[4] backward. The guess is right if we derive the same value of P1K[4] by using different WEP keys, otherwise the guess is wrong. Other bytes of TK can be derived in a similar way.

The TK recovery attack is effective from the fact that in phase 2 individual bytes of the WEP key are only correlated with certain bytes of TK. For example, PPK[5] is only correlated with TK[11] and TK[10], other byte of TK are not involved in the derivation of PPK[5]. To enhance the security a modification in the step 2 of phase 2 is shown:

```
Phase 2 Step 2:
        for i=0 to PHASE2_LOOP_CNT
        {
           PPK[0] = PPK[0] + S[PPK5 ⊕ Mk16(TK[1],TK[0])]
           PPK[1] = PPK[1] + S[PPK0 ⊕ Mk16(TK[3],TK[2])]
           PPK[2] = PPK[2] + S[PPK1 ⊕ Mk16(TK[5],TK[4])]
           PPK[3] = PPK[3] + S[PPK2 ⊕ Mk16(TK[7],TK[6])]
           PPK[4] = PPK[4] + S[PPK3 ⊕ Mk16(TK[9],TK[8])]
           PPK[5] = PPK[5] + S[PPK4 ⊕ Mk16(TK[11],TK[10])]

           PPK[0] = PPK[0] + RotR1(PPK[5] ⊕ Mk16(TK[13],TK[12]))
           PPK[1] = PPK[1] + RotR1(PPK[0] ⊕ Mk16(TK[15],TK[14]))
           PPK[2] = PPK[2] + RotR1(PPK[1])
           PPK[3] = PPK[3] + RotR1(PPK[2])
           PPK[4] = PPK[4] + RotR1(PPK[3])
           PPK[5] = PPK[5] + RotR1(PPK[4]) + i
           }
```

Figure 4. The Modified Algorithm of Step 2 in Phase 2

More rounds of the step 2 in phase 2 are introduced. PHASE2_LOOP_CNT is the variable that denotes the loop count. How to determine the value of the variable is to be discussed later. The '+i' operation is used here for the same consideration as it is used in the loop of phase 1, that is, to avoid the slide attack [15]. The modified algorithm is immune to the TK recovery attack, since the individual byte of the WEP key is correlated with the every bit of TK and the guess must be made at the all bits of TK instead of certain bytes, which is equivalent to the brutal force attack at TK. We measured the time cost of using different rounds for computing 2^{16} WEP keys on a PC with 1.80 GHz Pentium CPU and 256M memory and listed the results in the table below:

PHASE2_LOOP_CNT	1	2	3	4	5	6	7	8
Average time cost (ms)	46	83	124	160	193	230	266	305

Table 1. The average time cost for computing 2^{16} WEP keys by using different loop count

What interests us is not the specific time cost, but the ratio of the time cost between the modified algorithm and the original one. We denote the time cost of computing 2^{16} WEP keys using only one round in step 2 of phase 2 by t, which is the time cost of the original TKIP key hash function. From table 1 it is known that the time cost is increased by about 0.7 t when every one round computation in step 2 of phase 2 is added. The implementation of TKIP is by means of software update and the computation must be affordable for the computation ability limited hardware. So the optimum loop counts must be determined after an extensive investigation on the early 802.11 market products.

4 Conclusion

WEP has many flaws and fails to achieve its design goal. TKIP provides security enhancements for WEP. In this paper some cryptographic properties of the S-box employed in the TKIP key mixing function are investigated. A countermeasure is presented to defend against the TK recovery attack and its effect on the performance is discussed.

Acknowledgement

Doug Whiting, one of the original authors of TKIP, deserves special thanks for reading the draft and pointing out my misunderstanding about the word 'hash' in the term 'temporal key hash'. He indicated that TKIP was not intended in any sense to be a cryptographic hash function. TKIP is a hash only in the computer science sense, not in the cryptographic sense. While it is certainly interesting to look at TKIP on its own, the research on the combination of 'TKIP+RC4' has more significance.

I would like to thank V.Moen and D.Jonhnston with whom I had email interaction.

Reference

[1] IEEE Standard 802.11i: Medium Access Control (MAC) Security Enhancements. July, 2004. Available from: http://standards.ieee.org/getieee802/download/802.11i-2004.pdf.

[2] J.Daemen and V.Rijmen. The Design of Rijndael: AES - The Advanced Encryption Standard. ISBN 3-540-42580-2. Springer, 2002.

[3] S.Fluhrer, I.Mantin, and A.Shamir. Weaknesses in the Key Scheduling Algorithm of RC4. In SAC’2001. LNCS 2259. Springer-Verlag, pp. 1-24, 2001.

[4] N.Borisov, I.Goldberg, and D.Wagner. Intercepting Mobile Communications: The Insecurity of 802.11. In Proceedings of the Seventh Annual International Conference on Mobile Computing and Networking (MOBICOM 2001). ACM Press, New York. pp. 180-189, 2001. Available from: http://www.isaac.cs.berkeley.edu/isaac/mobicom.pdf.

[5] J.Seberry, X.M.Zhang, and Y.L.Zheng. Relationships Among Nonlinearity Criteria. In EUROCRYPT’94. LNCS 950, Springer-Verlag, pp. 376-388, 1995.

[6] A.F.Webster and S.E.Tavares. On the design of S-boxes. In CRYPTO’85. LNCS 219, Springer-Verlag, pp. 523-534, 1986.

[7] R.Forre. The Strict Avalanche Criterion: Spectral Properties of Boolean Functions and an Extended Definition. In CRYPTO’88. LNCS 403, Springer-Verlag, pp. 450-468, 1990.

[8] E.Biham and A.Shamir. Differential cryptanalysis of DES-like cryptosystems. Journal of Cryptology. Vol.4, No 1, pp. 3-72, 1991.

[9] T.Statoh, T.Iwata, and K.Kurosawa. On Cryptographically Secure Vectorial Boolean Functions. In ASIACRYPT’99. LNCS 1716, Springer-Verlag, pp. 20-28, 1999.

[10] M.Matsui. Linear cryptanalysis method for DES cipher. In EUROCRYPT’93. LNCS 765, Springer-Verlag, pp. 386-397, 1994.

[11] J.Seberry, X.M.Zhang and Y.L.Zheng. Nonlinearly Balanced Boolean Functions and Their Propagation Characteristics. In CRYPTO’93. LNCS 773, Springer-Verlag, pp. 49-60, 1994.

[12] W.Meier, and O.Staffelbach. Nonlinearity criteria for cryptographic functions. In EUROCRYPT’89. LNCS 434, Springer-Verlag, pp. 549-562, 1990.

[13] S.Dubuc. Characterization of Linear Structures. Designs, Codes and Cryptography. Kluwer Academic Publishers, Boston. Vol. 22, pp. 33-45, 2001.

[14] V.Moen, H.Raddum and K.J.Hole. Weakness in the Temporal Key Hash of WPA. ACM SIGMOBILE Computing and Communications Review, Vol.8, Issue 2, ACM Press, pp. 76-83, 2004.

[15] A.Biryukov and D.Wagner. Slide Attacks. In FSE’99. LNCS 1636, Springer-Verlag, pp. 245-259, 1999.

Appendix A

j / i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0.515625	0.531250	0.500000	0.546875	0.515625	0.468750	0.500000	0.468750	0.500000	0.515625	0.515625	0.484375	0.562500	0.453125	0.484375	0.453125
2	0.515625	0.468750	0.531250	0.531250	0.515625	0.531250	0.468750	0.531250	0.531250	0.515625	0.484375	0.562500	0.484375	0.484375	0.453125	0.500000
3	0.500000	0.531250	0.468750	0.546875	0.484375	0.531250	0.531250	0.515625	0.500000	0.500000	0.515625	0.500000	0.500000	0.562500	0.500000	0.531250
4	0.531250	0.531250	0.515625	0.500000	0.515625	0.562500	0.515625	0.500000	0.546875	0.531250	0.531250	0.500000	0.484375	0.500000	0.531250	0.500000
5	0.468750	0.437500	0.531250	0.437500	0.484375	0.453125	0.500000	0.531250	0.531250	0.468750	0.500000	0.468750	0.500000	0.500000	0.500000	0.546875
6	0.515625	0.515625	0.546875	0.515625	0.484375	0.484375	0.531250	0.546875	0.531250	0.515625	0.515625	0.468750	0.546875	0.468750	0.546875	0.531250
7	0.531250	0.515625	0.515625	0.468750	0.546875	0.562500	0.546875	0.453125	0.484375	0.531250	0.531250	0.484375	0.468750	0.468750	0.531250	0.531250
8	0.546875	0.468750	0.546875	0.531250	0.500000	0.500000	0.453125	0.531250	0.515625	0.546875	0.562500	0.453125	0.500000	0.484375	0.531250	0.484375
9	0.500000	0.515625	0.515625	0.484375	0.562500	0.453125	0.484375	0.453125	0.515625	0.531250	0.500000	0.546875	0.515625	0.468750	0.500000	0.468750
10	0.531250	0.515625	0.484375	0.562500	0.484375	0.484375	0.453125	0.500000	0.515625	0.468750	0.531250	0.531250	0.515625	0.531250	0.468750	0.531250
11	0.500000	0.500000	0.515625	0.500000	0.500000	0.562500	0.500000	0.531250	0.500000	0.531250	0.468750	0.546875	0.484375	0.531250	0.531250	0.515625
12	0.546875	0.531250	0.531250	0.500000	0.484375	0.500000	0.531250	0.500000	0.531250	0.531250	0.515625	0.500000	0.515625	0.562500	0.515625	0.500000
13	0.531250	0.468750	0.500000	0.468750	0.500000	0.500000	0.500000	0.546875	0.468750	0.437500	0.531250	0.437500	0.484375	0.453125	0.500000	0.531250
14	0.531250	0.515625	0.515625	0.468750	0.546875	0.468750	0.546875	0.531250	0.515625	0.515625	0.546875	0.515625	0.484375	0.484375	0.531250	0.546875
15	0.484375	0.531250	0.531250	0.484375	0.468750	0.468750	0.531250	0.531250	0.531250	0.515625	0.515625	0.468750	0.546875	0.562500	0.546875	0.453125
16	0.515625	0.546875	0.562500	0.453125	0.500000	0.484375	0.531250	0.484375	0.546875	0.468750	0.546875	0.531250	0.500000	0.500000	0.453125	0.531250

Table 2. The Avalanche Effect of S-box in TKIP

The element (i, j) in this table represents the probability that the j-th (j=1,2,…,16) output bit will change when the i-th (i=1,2,…,16) input bit is complemented.

Appendix B

Many conclusions in this paper are drawn from the test by running programs. So we include the source codes in the appendix for verification.

Source Code list:

1. The regularity of the S-box test.
2. The avalanche effect of the S-box test.
3. The differ uniformity of the S-box test
4. The linear structure existence of the S-box test.
5. The timing program for different loop counts in step2 of phase 2.

Note:

The differ uniformity test program is time costly. It takes about 50 days to run the program on a PC with 1.80 GHz Pentium CPU and 256M memory.

The timing program is compiled and run in the MS Visual C++ environment for a special timing functions QueryPerformanceCounter() is invoked. This function needs a high-resolution performance counter that must supported by the hardware.When this program is running it is suggested that close other programs to avoid interference.

For the generation of the TKIP S-box we partly refer to David Jonhnston's implementation codes: http://www.deadhat.com/wlancrypto/tkip_key_mixing0.3.c

For the last timing program we partly refer to the implementation codes in IEEE 802.11i: http://standards.ieee.org/getieee802/download/802.11i-2004.pdf

①The regularity of the S-box test:

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

/* The Sbox can be reduced to two 16 bit wide tables, each with 256 entries.*/
/* The second table is the same as the first but with the upper and lower     */
/* bytes swapped. To allow an endian tolerant implementation, the byte        */
/* halves have been expressed independently here.                              */

unsigned int Tkip_Sbox_Lower[256] =
    {
    0xA5,0x84,0x99,0x8D,0x0D,0xBD,0xB1,0x54,
    0x50,0x03,0xA9,0x7D,0x19,0x62,0xE6,0x9A,
    0x45,0x9D,0x40,0x87,0x15,0xEB,0xC9,0x0B,
    0xEC,0x67,0xFD,0xEA,0xBF,0xF7,0x96,0x5B,
    0xC2,0x1C,0xAE,0x6A,0x5A,0x41,0x02,0x4F,
    0x5C,0xF4,0x34,0x08,0x93,0x73,0x53,0x3F,
    0x0C,0x52,0x65,0x5E,0x28,0xA1,0x0F,0xB5,
    0x09,0x36,0x9B,0x3D,0x26,0x69,0xCD,0x9F,
    0x1B,0x9E,0x74,0x2E,0x2D,0xB2,0xEE,0xFB,
    0xF6,0x4D,0x61,0xCE,0x7B,0x3E,0x71,0x97,
    0xF5,0x68,0x00,0x2C,0x60,0x1F,0xC8,0xED,
    0xBE,0x46,0xD9,0x4B,0xDE,0xD4,0xE8,0x4A,
    0x6B,0x2A,0xE5,0x16,0xC5,0xD7,0x55,0x94,
    0xCF,0x10,0x06,0x81,0xF0,0x44,0xBA,0xE3,
    0xF3,0xFE,0xC0,0x8A,0xAD,0xBC,0x48,0x04,
    0xDF,0xC1,0x75,0x63,0x30,0x1A,0x0E,0x6D,
    0x4C,0x14,0x35,0x2F,0xE1,0xA2,0xCC,0x39,
    0x57,0xF2,0x82,0x47,0xAC,0xE7,0x2B,0x95,
    0xA0,0x98,0xD1,0x7F,0x66,0x7E,0xAB,0x83,
    0xCA,0x29,0xD3,0x3C,0x79,0xE2,0x1D,0x76,
    0x3B,0x56,0x4E,0x1E,0xDB,0x0A,0x6C,0xE4,
    0x5D,0x6E,0xEF,0xA6,0xA8,0xA4,0x37,0x8B,
    0x32,0x43,0x59,0xB7,0x8C,0x64,0xD2,0xE0,
    0xB4,0xFA,0x07,0x25,0xAF,0x8E,0xE9,0x18,
    0xD5,0x88,0x6F,0x72,0x24,0xF1,0xC7,0x51,
    0x23,0x7C,0x9C,0x21,0xDD,0xDC,0x86,0x85,
    0x90,0x42,0xC4,0xAA,0xD8,0x05,0x01,0x12,
    0xA3,0x5F,0xF9,0xD0,0x91,0x58,0x27,0xB9,
    0x38,0x13,0xB3,0x33,0xBB,0x70,0x89,0xA7,
    0xB6,0x22,0x92,0x20,0x49,0xFF,0x78,0x7A,
    0x8F,0xF8,0x80,0x17,0xDA,0x31,0xC6,0xB8,
    0xC3,0xB0,0x77,0x11,0xCB,0xFC,0xD6,0x3A
    };

unsigned int Tkip_Sbox_Upper[256] =
    {
    0xC6,0xF8,0xEE,0xF6,0xFF,0xD6,0xDE,0x91,
    0x60,0x02,0xCE,0x56,0xE7,0xB5,0x4D,0xEC,
    0x8F,0x1F,0x89,0xFA,0xEF,0xB2,0x8E,0xFB,
    0x41,0xB3,0x5F,0x45,0x23,0x53,0xE4,0x9B,
    0x75,0xE1,0x3D,0x4C,0x6C,0x7E,0xF5,0x83,
    0x68,0x51,0xD1,0xF9,0xE2,0xAB,0x62,0x2A,
    0x08,0x95,0x46,0x9D,0x30,0x37,0x0A,0x2F,
    0x0E,0x24,0x1B,0xDF,0xCD,0x4E,0x7F,0xEA,
    0x12,0x1D,0x58,0x34,0x36,0xDC,0xB4,0x5B,
    0xA4,0x76,0xB7,0x7D,0x52,0xDD,0x5E,0x13,
    0xA6,0xB9,0x00,0xC1,0x40,0xE3,0x79,0xB6,
    0xD4,0x8D,0x67,0x72,0x94,0x98,0xB0,0x85,
    0xBB,0xC5,0x4F,0xED,0x86,0x9A,0x66,0x11,
    0x8A,0xE9,0x04,0xFE,0xA0,0x78,0x25,0x4B,
    0xA2,0x5D,0x80,0x05,0x3F,0x21,0x70,0xF1,
    0x63,0x77,0xAF,0x42,0x20,0xE5,0xFD,0xBF,
    0x81,0x18,0x26,0xC3,0xBE,0x35,0x88,0x2E,
    0x93,0x55,0xFC,0x7A,0xC8,0xBA,0x32,0xE6,
    0xC0,0x19,0x9E,0xA3,0x44,0x54,0x3B,0x0B,
    0x8C,0xC7,0x6B,0x28,0xA7,0xBC,0x16,0xAD,
    0xDB,0x64,0x74,0x14,0x92,0x0C,0x48,0xB8,
    0x9F,0xBD,0x43,0xC4,0x39,0x31,0xD3,0xF2,
    0xD5,0x8B,0x6E,0xDA,0x01,0xB1,0x9C,0x49,
    0xD8,0xAC,0xF3,0xCF,0xCA,0xF4,0x47,0x10,
    0x6F,0xF0,0x4A,0x5C,0x38,0x57,0x73,0x97,
    0xCB,0xA1,0xE8,0x3E,0x96,0x61,0x0D,0x0F,
    0xE0,0x7C,0x71,0xCC,0x90,0x06,0xF7,0x1C,
    0xC2,0x6A,0xAE,0x69,0x17,0x99,0x3A,0x27,
    0xD9,0xEB,0x2B,0x22,0xD2,0xA9,0x07,0x33,
    0x2D,0x3C,0x15,0xC9,0x87,0xAA,0x50,0xA5,
    0x03,0x59,0x09,0x1A,0x65,0xD7,0x84,0xD0,
    0x82,0x29,0x5A,0x1E,0x7B,0xA8,0x6D,0x2C
    };

unsigned int tkip_sbox(unsigned int index);

/************************************************************/
/* tkip_sbox()                                              */
/* Returns a 16 bit value from a 64K entry table. The Table */
/* is synthesized from two 256 entry byte wide tables.      */
/************************************************************/

unsigned int tkip_sbox(unsigned int index)
{
    unsigned int index_low;
    unsigned int index_high;
    unsigned int left, right;

    index_low = (index % 256);
    index_high = ((index >> 8) % 256);

    left = Tkip_Sbox_Lower[index_low] + (Tkip_Sbox_Upper[index_low] * 256);
    right = Tkip_Sbox_Upper[index_high] + (Tkip_Sbox_Lower[index_high] * 256);

    return (left ^ right);
};

void main()
{
    unsigned int i;
    unsigned int j, k;
    unsigned int sbox[65536];
    unsigned int flag=0;

    for (i=0; i<65536; i++)
      {
        sbox[i]=tkip_sbox(i);
        }

    for (j=0; j<65535; j++)
      {
        for (k=(j+1); k<65536; k++)
          {
            if (sbox[j]==sbox[k])
                {
                  printf("j=%d,k=%d,sbox[%d]=%x\n", j,k,j,sbox[j]);
                  flag=1;
                 }
            }
        }
     if (flag==0)
        printf("no match in sbox is found!\n");
    }
```

②The avalanche effect of the S-box test:

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

unsigned int Tkip_Sbox_Lower[256] =
        {
        0xA5,0x84,0x99,0x8D,0x0D,0xBD,0xB1,0x54,
        0x50,0x03,0xA9,0x7D,0x19,0x62,0xE6,0x9A,
        0x45,0x9D,0x40,0x87,0x15,0xEB,0xC9,0x0B,
        0xEC,0x67,0xFD,0xEA,0xBF,0xF7,0x96,0x5B,
        0xC2,0x1C,0xAE,0x6A,0x5A,0x41,0x02,0x4F,
        0x5C,0xF4,0x34,0x08,0x93,0x73,0x53,0x3F,
        0x0C,0x52,0x65,0x5E,0x28,0xA1,0x0F,0xB5,
        0x09,0x36,0x9B,0x3D,0x26,0x69,0xCD,0x9F,
        0x1B,0x9E,0x74,0x2E,0x2D,0xB2,0xEE,0xFB,
        0xF6,0x4D,0x61,0xCE,0x7B,0x3E,0x71,0x97,
        0xF5,0x68,0x00,0x2C,0x60,0x1F,0xC8,0xED,
        0xBE,0x46,0xD9,0x4B,0xDE,0xD4,0xE8,0x4A,
        0x6B,0x2A,0xE5,0x16,0xC5,0xD7,0x55,0x94,
        0xCF,0x10,0x06,0x81,0xF0,0x44,0xBA,0xE3,
        0xF3,0xFE,0xC0,0x8A,0xAD,0xBC,0x48,0x04,
        0xDF,0xC1,0x75,0x63,0x30,0x1A,0x0E,0x6D,
        0x4C,0x14,0x35,0x2F,0xE1,0xA2,0xCC,0x39,
        0x57,0xF2,0x82,0x47,0xAC,0xE7,0x2B,0x95,
        0xA0,0x98,0xD1,0x7F,0x66,0x7E,0xAB,0x83,
        0xCA,0x29,0xD3,0x3C,0x79,0xE2,0x1D,0x76,
        0x3B,0x56,0x4E,0x1E,0xDB,0x0A,0x6C,0xE4,
        0x5D,0x6E,0xEF,0xA6,0xA8,0xA4,0x37,0x8B,
        0x32,0x43,0x59,0xB7,0x8C,0x64,0xD2,0xE0,
        0xB4,0xFA,0x07,0x25,0xAF,0x8E,0xE9,0x18,
        0xD5,0x88,0x6F,0x72,0x24,0xF1,0xC7,0x51,
        0x23,0x7C,0x9C,0x21,0xDD,0xDC,0x86,0x85,
        0x90,0x42,0xC4,0xAA,0xD8,0x05,0x01,0x12,
        0xA3,0x5F,0xF9,0xD0,0x91,0x58,0x27,0xB9,
        0x38,0x13,0xB3,0x33,0xBB,0x70,0x89,0xA7,
        0xB6,0x22,0x92,0x20,0x49,0xFF,0x78,0x7A,
        0x8F,0xF8,0x80,0x17,0xDA,0x31,0xC6,0xB8,
        0xC3,0xB0,0x77,0x11,0xCB,0xFC,0xD6,0x3A
        };

unsigned int Tkip_Sbox_Upper[256] =
        {
        0xC6,0xF8,0xEE,0xF6,0xFF,0xD6,0xDE,0x91,
        0x60,0x02,0xCE,0x56,0xE7,0xB5,0x4D,0xEC,
        0x8F,0x1F,0x89,0xFA,0xEF,0xB2,0x8E,0xFB,
        0x41,0xB3,0x5F,0x45,0x23,0x53,0xE4,0x9B,
        0x75,0xE1,0x3D,0x4C,0x6C,0x7E,0xF5,0x83,
        0x68,0x51,0xD1,0xF9,0xE2,0xAB,0x62,0x2A,
        0x08,0x95,0x46,0x9D,0x30,0x37,0x0A,0x2F,
        0x0E,0x24,0x1B,0xDF,0xCD,0x4E,0x7F,0xEA,
        0x12,0x1D,0x58,0x34,0x36,0xDC,0xB4,0x5B,
        0xA4,0x76,0xB7,0x7D,0x52,0xDD,0x5E,0x13,
        0xA6,0xB9,0x00,0xC1,0x40,0xE3,0x79,0xB6,
        0xD4,0x8D,0x67,0x72,0x94,0x98,0xB0,0x85,
        0xBB,0xC5,0x4F,0xED,0x86,0x9A,0x66,0x11,
        0x8A,0xE9,0x04,0xFE,0xA0,0x78,0x25,0x4B,
        0xA2,0x5D,0x80,0x05,0x3F,0x21,0x70,0xF1,
        0x63,0x77,0xAF,0x42,0x20,0xE5,0xFD,0xBF,
        0x81,0x18,0x26,0xC3,0xBE,0x35,0x88,0x2E,
        0x93,0x55,0xFC,0x7A,0xC8,0xBA,0x32,0xE6,
        0xC0,0x19,0x9E,0xA3,0x44,0x54,0x3B,0x0B,
        0x8C,0xC7,0x6B,0x28,0xA7,0xBC,0x16,0xAD,
        0xDB,0x64,0x74,0x14,0x92,0x0C,0x48,0xB8,
        0x9F,0xBD,0x43,0xC4,0x39,0x31,0xD3,0xF2,
        0xD5,0x8B,0x6E,0xDA,0x01,0xB1,0x9C,0x49,
        0xD8,0xAC,0xF3,0xCF,0xCA,0xF4,0x47,0x10,
        0x6F,0xF0,0x4A,0x5C,0x38,0x57,0x73,0x97,
        0xCB,0xA1,0xE8,0x3E,0x96,0x61,0x0D,0x0F,
        0xE0,0x7C,0x71,0xCC,0x90,0x06,0xF7,0x1C,
        0xC2,0x6A,0xAE,0x69,0x17,0x99,0x3A,0x27,
        0xD9,0xEB,0x2B,0x22,0xD2,0xA9,0x07,0x33,
        0x2D,0x3C,0x15,0xC9,0x87,0xAA,0x50,0xA5,
        0x03,0x59,0x09,0x1A,0x65,0xD7,0x84,0xD0,
        0x82,0x29,0x5A,0x1E,0x7B,0xA8,0x6D,0x2C
        };


unsigned int tkip_sbox(unsigned int index);
unsigned int comp(unsigned int a, unsigned int b, unsigned int k);

unsigned int tkip_sbox(unsigned int index)
{
        unsigned int index_low;
        unsigned int index_high;
        unsigned int left, right;

        index_low = (index % 256);
        index_high = ((index >> 8) % 256);

        left = Tkip_Sbox_Lower[index_low] + (Tkip_Sbox_Upper[index_low] * 256);
        right = Tkip_Sbox_Upper[index_high] + (Tkip_Sbox_Lower[index_high] * 256);

        return (left ^ right);
};

unsigned int comp(unsigned int a, unsigned int b, unsigned int k)
/* compare a and b in the k-th bit location, if equale return 1, if not equal return 0 */
{
        unsigned int temp1, temp2;

        if (k<0 || k>15)    printf("the value of k=%d is overflow!\n", k);
        temp1 = a & (unsigned int)(pow(2,k));
        temp2 = b & (unsigned int)(pow(2,k));
        if ( !(temp1 ^ temp2) )    return 1;
        else    return 0;
```

```
};

void main()
{
    unsigned int i;
    unsigned int j, k;
    unsigned int sbox[65536];
    unsigned int var1,var2;
    unsigned int counter[16][16];
    float prob[16][16];

/* generate all the 65536 values in sbox */
    for (i=0; i<65536; i++)
            sbox[i]=tkip_sbox(i);

/**************************************************/

    for (i=0; i<16; i++)
    {
        for (j=0; j<16; j++)
                counter[i][j]=0;
      }
/**************************************************/

    for (i=0; i<65536; i++)
    {
        var1=sbox[i];
        for (j=0; j<16; j++)
        {
            var2=sbox[(unsigned int)(i^(unsigned int)(pow(2,j)))];
            for (k=0; k<16; k++)
            {
                if ( !comp(var1,var2,k) )    counter[j][k]++;
                }
          }
    }
/**************************************************/
    for (i=0; i<16; i++)
    {
        for (j=0; j<16; j++)
        {
            prob[i][j]=((float)(counter[i][j])) / 65536;
            printf("%-12f", prob[i][j]);
        }
        printf("\n");
    }
/***********************************************/

}
```

③The differ uniformity of the S-box test:

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

unsigned int Tkip_Sbox_Lower[256] =
        {
        0xA5,0x84,0x99,0x8D,0x0D,0xBD,0xB1,0x54,
        0x50,0x03,0xA9,0x7D,0x19,0x62,0xE6,0x9A,
        0x45,0x9D,0x40,0x87,0x15,0xEB,0xC9,0x0B,
        0xEC,0x67,0xFD,0xEA,0xBF,0xF7,0x96,0x5B,
        0xC2,0x1C,0xAE,0x6A,0x5A,0x41,0x02,0x4F,
        0x5C,0xF4,0x34,0x08,0x93,0x73,0x53,0x3F,
        0x0C,0x52,0x65,0x5E,0x28,0xA1,0x0F,0xB5,
        0x09,0x36,0x9B,0x3D,0x26,0x69,0xCD,0x9F,
        0x1B,0x9E,0x74,0x2E,0x2D,0xB2,0xEE,0xFB,
        0xF6,0x4D,0x61,0xCE,0x7B,0x3E,0x71,0x97,
        0xF5,0x68,0x00,0x2C,0x60,0x1F,0xC8,0xED,
        0xBE,0x46,0xD9,0x4B,0xDE,0xD4,0xE8,0x4A,
        0x6B,0x2A,0xE5,0x16,0xC5,0xD7,0x55,0x94,
        0xCF,0x10,0x06,0x81,0xF0,0x44,0xBA,0xE3,
        0xF3,0xFE,0xC0,0x8A,0xAD,0xBC,0x48,0x04,
        0xDF,0xC1,0x75,0x63,0x30,0x1A,0x0E,0x6D,
        0x4C,0x14,0x35,0x2F,0xE1,0xA2,0xCC,0x39,
        0x57,0xF2,0x82,0x47,0xAC,0xE7,0x2B,0x95,
        0xA0,0x98,0xD1,0x7F,0x66,0x7E,0xAB,0x83,
        0xCA,0x29,0xD3,0x3C,0x79,0xE2,0x1D,0x76,
        0x3B,0x56,0x4E,0x1E,0xDB,0x0A,0x6C,0xE4,
        0x5D,0x6E,0xEF,0xA6,0xA8,0xA4,0x37,0x8B,
        0x32,0x43,0x59,0xB7,0x8C,0x64,0xD2,0xE0,
        0xB4,0xFA,0x07,0x25,0xAF,0x8E,0xE9,0x18,
        0xD5,0x88,0x6F,0x72,0x24,0xF1,0xC7,0x51,
        0x23,0x7C,0x9C,0x21,0xDD,0xDC,0x86,0x85,
        0x90,0x42,0xC4,0xAA,0xD8,0x05,0x01,0x12,
        0xA3,0x5F,0xF9,0xD0,0x91,0x58,0x27,0xB9,
        0x38,0x13,0xB3,0x33,0xBB,0x70,0x89,0xA7,
        0xB6,0x22,0x92,0x20,0x49,0xFF,0x78,0x7A,
        0x8F,0xF8,0x80,0x17,0xDA,0x31,0xC6,0xB8,
        0xC3,0xB0,0x77,0x11,0xCB,0xFC,0xD6,0x3A
        };

unsigned int Tkip_Sbox_Upper[256] =
        {
        0xC6,0xF8,0xEE,0xF6,0xFF,0xD6,0xDE,0x91,
        0x60,0x02,0xCE,0x56,0xE7,0xB5,0x4D,0xEC,
        0x8F,0x1F,0x89,0xFA,0xEF,0xB2,0x8E,0xFB,
        0x41,0xB3,0x5F,0x45,0x23,0x53,0xE4,0x9B,
        0x75,0xE1,0x3D,0x4C,0x6C,0x7E,0xF5,0x83,
        0x68,0x51,0xD1,0xF9,0xE2,0xAB,0x62,0x2A,
        0x08,0x95,0x46,0x9D,0x30,0x37,0x0A,0x2F,
        0x0E,0x24,0x1B,0xDF,0xCD,0x4E,0x7F,0xEA,
        0x12,0x1D,0x58,0x34,0x36,0xDC,0xB4,0x5B,
        0xA4,0x76,0xB7,0x7D,0x52,0xDD,0x5E,0x13,
        0xA6,0xB9,0x00,0xC1,0x40,0xE3,0x79,0xB6,
        0xD4,0x8D,0x67,0x72,0x94,0x98,0xB0,0x85,
        0xBB,0xC5,0x4F,0xED,0x86,0x9A,0x66,0x11,
        0x8A,0xE9,0x04,0xFE,0xA0,0x78,0x25,0x4B,
        0xA2,0x5D,0x80,0x05,0x3F,0x21,0x70,0xF1,
        0x63,0x77,0xAF,0x42,0x20,0xE5,0xFD,0xBF,
        0x81,0x18,0x26,0xC3,0xBE,0x35,0x88,0x2E,
        0x93,0x55,0xFC,0x7A,0xC8,0xBA,0x32,0xE6,
        0xC0,0x19,0x9E,0xA3,0x44,0x54,0x3B,0x0B,
        0x8C,0xC7,0x6B,0x28,0xA7,0xBC,0x16,0xAD,
        0xDB,0x64,0x74,0x14,0x92,0x0C,0x48,0xB8,
        0x9F,0xBD,0x43,0xC4,0x39,0x31,0xD3,0xF2,
        0xD5,0x8B,0x6E,0xDA,0x01,0xB1,0x9C,0x49,
        0xD8,0xAC,0xF3,0xCF,0xCA,0xF4,0x47,0x10,
        0x6F,0xF0,0x4A,0x5C,0x38,0x57,0x73,0x97,
        0xCB,0xA1,0xE8,0x3E,0x96,0x61,0x0D,0x0F,
        0xE0,0x7C,0x71,0xCC,0x90,0x06,0xF7,0x1C,
        0xC2,0x6A,0xAE,0x69,0x17,0x99,0x3A,0x27,
        0xD9,0xEB,0x2B,0x22,0xD2,0xA9,0x07,0x33,
        0x2D,0x3C,0x15,0xC9,0x87,0xAA,0x50,0xA5,
        0x03,0x59,0x09,0x1A,0x65,0xD7,0x84,0xD0,
        0x82,0x29,0x5A,0x1E,0x7B,0xA8,0x6D,0x2C
        };

unsigned int tkip_sbox(unsigned int index);

unsigned int tkip_sbox(unsigned int index)
{
        unsigned int index_low;
        unsigned int index_high;
        unsigned int left, right;

        index_low = (index % 256);
        index_high = ((index >> 8) % 256);

        left = Tkip_Sbox_Lower[index_low] + (Tkip_Sbox_Upper[index_low] * 256);
        right = Tkip_Sbox_Upper[index_high] + (Tkip_Sbox_Lower[index_high] * 256);

        return (left ^ right);
}

void main()
{
    unsigned int i;
    unsigned int row, col;
    unsigned int sbox[65536];
    unsigned int x,temp,DifferUniform;

    for (i=0; i<65536; i++)
        {
            sbox[i]=tkip_sbox(i);
            }

    for (row=1; row<65536; row++)
        {
            for (col=0; col<65536; col++)
                {
                    temp=0;
                    for (x=0; x<65536; x++)
                          {
                              if ( (sbox[x]^sbox[(x^row)]) == col )
                              temp++;
                              }
                    if (DifferUniform < temp) DifferUniform=temp;
                    }

              /* print the value of differ uniformity every 128 lines of the differ uniform matrix */
              if ( (row % 128) == 0 ) printf("row=%d, DifferUniform=%d\n", row, DifferUniform);
              }

    /* print the final result */
    printf("DifferUniform=%d\n", DifferUniform);
    }
```

④The linear structure existence of the S-box test:

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

unsigned int Tkip_Sbox_Lower[256] =
        {
        0xA5,0x84,0x99,0x8D,0x0D,0xBD,0xB1,0x54,
        0x50,0x03,0xA9,0x7D,0x19,0x62,0xE6,0x9A,
        0x45,0x9D,0x40,0x87,0x15,0xEB,0xC9,0x0B,
        0xEC,0x67,0xFD,0xEA,0xBF,0xF7,0x96,0x5B,
        0xC2,0x1C,0xAE,0x6A,0x5A,0x41,0x02,0x4F,
        0x5C,0xF4,0x34,0x08,0x93,0x73,0x53,0x3F,
        0x0C,0x52,0x65,0x5E,0x28,0xA1,0x0F,0xB5,
        0x09,0x36,0x9B,0x3D,0x26,0x69,0xCD,0x9F,
        0x1B,0x9E,0x74,0x2E,0x2D,0xB2,0xEE,0xFB,
        0xF6,0x4D,0x61,0xCE,0x7B,0x3E,0x71,0x97,
        0xF5,0x68,0x00,0x2C,0x60,0x1F,0xC8,0xED,
        0xBE,0x46,0xD9,0x4B,0xDE,0xD4,0xE8,0x4A,
        0x6B,0x2A,0xE5,0x16,0xC5,0xD7,0x55,0x94,
        0xCF,0x10,0x06,0x81,0xF0,0x44,0xBA,0xE3,
        0xF3,0xFE,0xC0,0x8A,0xAD,0xBC,0x48,0x04,
        0xDF,0xC1,0x75,0x63,0x30,0x1A,0x0E,0x6D,
        0x4C,0x14,0x35,0x2F,0xE1,0xA2,0xCC,0x39,
        0x57,0xF2,0x82,0x47,0xAC,0xE7,0x2B,0x95,
        0xA0,0x98,0xD1,0x7F,0x66,0x7E,0xAB,0x83,
        0xCA,0x29,0xD3,0x3C,0x79,0xE2,0x1D,0x76,
        0x3B,0x56,0x4E,0x1E,0xDB,0x0A,0x6C,0xE4,
        0x5D,0x6E,0xEF,0xA6,0xA8,0xA4,0x37,0x8B,
        0x32,0x43,0x59,0xB7,0x8C,0x64,0xD2,0xE0,
        0xB4,0xFA,0x07,0x25,0xAF,0x8E,0xE9,0x18,
        0xD5,0x88,0x6F,0x72,0x24,0xF1,0xC7,0x51,
        0x23,0x7C,0x9C,0x21,0xDD,0xDC,0x86,0x85,
        0x90,0x42,0xC4,0xAA,0xD8,0x05,0x01,0x12,
        0xA3,0x5F,0xF9,0xD0,0x91,0x58,0x27,0xB9,
        0x38,0x13,0xB3,0x33,0xBB,0x70,0x89,0xA7,
        0xB6,0x22,0x92,0x20,0x49,0xFF,0x78,0x7A,
        0x8F,0xF8,0x80,0x17,0xDA,0x31,0xC6,0xB8,
        0xC3,0xB0,0x77,0x11,0xCB,0xFC,0xD6,0x3A
        };

unsigned int Tkip_Sbox_Upper[256] =
        {
```

```
        0xC6,0xF8,0xEE,0xF6,0xFF,0xD6,0xDE,0x91,
        0x60,0x02,0xCE,0x56,0xE7,0xB5,0x4D,0xEC,
        0x8F,0x1F,0x89,0xFA,0xEF,0xB2,0x8E,0xFB,
        0x41,0xB3,0x5F,0x45,0x23,0x53,0xE4,0x9B,
        0x75,0xE1,0x3D,0x4C,0x6C,0x7E,0xF5,0x83,
        0x68,0x51,0xD1,0xF9,0xE2,0xAB,0x62,0x2A,
        0x08,0x95,0x46,0x9D,0x30,0x37,0x0A,0x2F,
        0x0E,0x24,0x1B,0xDF,0xCD,0x4E,0x7F,0xEA,
        0x12,0x1D,0x58,0x34,0x36,0xDC,0xB4,0x5B,
        0xA4,0x76,0xB7,0x7D,0x52,0xDD,0x5E,0x13,
        0xA6,0xB9,0x00,0xC1,0x40,0xE3,0x79,0xB6,
        0xD4,0x8D,0x67,0x72,0x94,0x98,0xB0,0x85,
        0xBB,0xC5,0x4F,0xED,0x86,0x9A,0x66,0x11,
        0x8A,0xE9,0x04,0xFE,0xA0,0x78,0x25,0x4B,
        0xA2,0x5D,0x80,0x05,0x3F,0x21,0x70,0xF1,
        0x63,0x77,0xAF,0x42,0x20,0xE5,0xFD,0xBF,
        0x81,0x18,0x26,0xC3,0xBE,0x35,0x88,0x2E,
        0x93,0x55,0xFC,0x7A,0xC8,0xBA,0x32,0xE6,
        0xC0,0x19,0x9E,0xA3,0x44,0x54,0x3B,0x0B,
        0x8C,0xC7,0x6B,0x28,0xA7,0xBC,0x16,0xAD,
        0xDB,0x64,0x74,0x14,0x92,0x0C,0x48,0xB8,
        0x9F,0xBD,0x43,0xC4,0x39,0x31,0xD3,0xF2,
        0xD5,0x8B,0x6E,0xDA,0x01,0xB1,0x9C,0x49,
        0xD8,0xAC,0xF3,0xCF,0xCA,0xF4,0x47,0x10,
        0x6F,0xF0,0x4A,0x5C,0x38,0x57,0x73,0x97,
        0xCB,0xA1,0xE8,0x3E,0x96,0x61,0x0D,0x0F,
        0xE0,0x7C,0x71,0xCC,0x90,0x06,0xF7,0x1C,
        0xC2,0x6A,0xAE,0x69,0x17,0x99,0x3A,0x27,
        0xD9,0xEB,0x2B,0x22,0xD2,0xA9,0x07,0x33,
        0x2D,0x3C,0x15,0xC9,0x87,0xAA,0x50,0xA5,
        0x03,0x59,0x09,0x1A,0x65,0xD7,0x84,0xD0,
        0x82,0x29,0x5A,0x1E,0x7B,0xA8,0x6D,0x2C
        };

unsigned int tkip_sbox(unsigned int index);

unsigned int tkip_sbox(unsigned int index)
{
        unsigned int index_low;
        unsigned int index_high;
        unsigned int left, right;

        index_low = (index % 256);
        index_high = ((index >> 8) % 256);
        left = Tkip_Sbox_Lower[index_low] + (Tkip_Sbox_Upper[index_low] * 256);
        right = Tkip_Sbox_Upper[index_high] + (Tkip_Sbox_Lower[index_high] * 256);

        return (left ^ right);
};

void main()
{
    unsigned int i;
    unsigned int a, x, temp;
    unsigned int sbox[65536];
    unsigned int IsLinStru, LinStruExist;

    LinStruExist=0;
    for (i=0; i<65536; i++)
        {
            sbox[i]=tkip_sbox(i);
            }

    for (a=1; a<65535; a++)
        {
            x=0;
            IsLinStru=1;
            temp=sbox[x]^sbox[x^a];
            do
            {
                x++;
                if (temp!=(sbox[x]^sbox[x^a]))      IsLinStru=0;
                } while ( (x<65535) && IsLinStru );
            if ( (x==65535) && IsLinStru )
                  {
                     printf("find a linear structure in tkip sbox!\n");
                     printf("F(x)^F(x^a)=constant\n");
                     printf("a=%u, constant=%u\n", a, temp);
                     }
            }

      if (LinStruExist==0)
            printf("no linear structure in sbox is found!\n");
      }
```

⑤The timing program for different loop counts in step2 of phase 2:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <Windows.h>


typedef unsigned char byte;    /* 8-bit byte (octet) */
typedef unsigned short u16b; /* 16-bit unsigned word */
typedef unsigned long u32b;    /* 32-bit unsigned word */

/* macros for extraction/creation of byte/u16b values */
#define RotR1(v16) ((((v16) >> 1) & 0x7FFF) ^ (((v16) & 1) << 15))
#define Lo8(v16) ((byte)( (v16) & 0x00FF))
#define Hi8(v16) ((byte)(((v16) >> 8) & 0x00FF))
#define Lo16(v32) ((u16b)( (v32) & 0xFFFF))
#define Hi16(v32) ((u16b)(((v32) >>16) & 0xFFFF))
#define Mk16(hi,lo) ((lo) ^ (((u16b)(hi)) << 8))

/* select the Nth 16-bit word of the Temporal Key byte array TK[] */
#define TK16(N) Mk16(TK[2*(N)+1],TK[2*(N)])

/* S-box lookup: 16 bits --> 16 bits */
#define _S_(v16) (Sbox[0][Lo8(v16)] ^ Sbox[1][Hi8(v16)])

/* fixed algorithm "parameters" */
#define PHASE1_LOOP_CNT 8 /* this needs to be "big enough" */
#define TA_SIZE 6 /* 48-bit transmitter address */
#define TK_SIZE 16 /* 128-bit Temporal Key */
#define P1K_SIZE 5 /* 80-bit Phase1 key */ /* Note here P1K_SIZE=5 not 10 ! */
#define RC4_KEY_SIZE 16 /* 128-bit RC4KEY (104 bits unknown) */

#define PHASE2_LOOP_CNT 1 /* test the different time cost by changing this value ! */

/* 2-byte by 2-byte subset of the full AES S-box table */
const u16b Sbox[2][256]= /* Sbox for hash (can be in ROM) */
{ {
0xC6A5,0xF884,0xEE99,0xF68D,0xFF0D,0xD6BD,0xDEB1,0x9154,
0x6050,0x0203,0xCEA9,0x567D,0xE719,0xB562,0x4DE6,0xEC9A,
0x8F45,0x1F9D,0x8940,0xFA87,0xEF15,0xB2EB,0x8EC9,0xFB0B,
0x41EC,0xB367,0x5FFD,0x45EA,0x23BF,0x53F7,0xE496,0x9B5B,
0x75C2,0xE11C,0x3DAE,0x4C6A,0x6C5A,0x7E41,0xF502,0x834F,
0x685C,0x51F4,0xD134,0xF908,0xE293,0xAB73,0x6253,0x2A3F,
0x080C,0x9552,0x4665,0x9D5E,0x3028,0x37A1,0x0A0F,0x2FB5,
0x0E09,0x2436,0x1B9B,0xDF3D,0xCD26,0x4E69,0x7FCD,0xEA9F,
0x121B,0x1D9E,0x5874,0x342E,0x362D,0xDCB2,0xB4EE,0x5BFB,
0xA4F6,0x764D,0xB761,0x7DCE,0x527B,0xDD3E,0x5E71,0x1397,
0xA6F5,0xB968,0x0000,0xC12C,0x4060,0xE31F,0x79C8,0xB6ED,
0xD4BE,0x8D46,0x67D9,0x724B,0x94DE,0x98D4,0xB0E8,0x854A,
0xBB6B,0xC52A,0x4FE5,0xED16,0x86C5,0x9AD7,0x6655,0x1194,
0x8ACF,0xE910,0x0406,0xFE81,0xA0F0,0x7844,0x25BA,0x4BE3,
0xA2F3,0x5DFE,0x80C0,0x058A,0x3FAD,0x21BC,0x7048,0xF104,
0x63DF,0x77C1,0xAF75,0x4263,0x2030,0xE51A,0xFD0E,0xBF6D,
0x814C,0x1814,0x2635,0xC32F,0xBEE1,0x35A2,0x88CC,0x2E39,
0x9357,0x55F2,0xFC82,0x7A47,0xC8AC,0xBAE7,0x322B,0xE695,
0xC0A0,0x1998,0x9ED1,0xA37F,0x4466,0x547E,0x3BAB,0x0B83,
0x8CCA,0xC729,0x6BD3,0x283C,0xA779,0xBCE2,0x161D,0xAD76,
0xDB3B,0x6456,0x744E,0x141E,0x92DB,0x0C0A,0x486C,0xB8E4,
0x9F5D,0xBD6E,0x43EF,0xC4A6,0x39A8,0x31A4,0xD337,0xF28B,
0xD532,0x8B43,0x6E59,0xDAB7,0x018C,0xB164,0x9CD2,0x49E0,
0xD8B4,0xACFA,0xF307,0xCF25,0xCAAF,0xF48E,0x47E9,0x1018,
0x6FD5,0xF088,0x4A6F,0x5C72,0x3824,0x57F1,0x73C7,0x9751,
0xCB23,0xA17C,0xE89C,0x3E21,0x96DD,0x61DC,0x0D86,0x0F85,
0xE090,0x7C42,0x71C4,0xCCAA,0x90D8,0x0605,0xF701,0x1C12,
0xC2A3,0x6A5F,0xAEF9,0x69D0,0x1791,0x9958,0x3A27,0x27B9,
0xD938,0xEB13,0x2BB3,0x2233,0xD2BB,0xA970,0x0789,0x33A7,
0x2DB6,0x3C22,0x1592,0xC920,0x8749,0xAAFF,0x5078,0xA57A,
0x038F,0x59F8,0x0980,0x1A17,0x65DA,0xD731,0x84C6,0xD0B8,
0x82C3,0x29B0,0x5A77,0x1E11,0x7BCB,0xA8FC,0x6DD6,0x2C3A,
},

{ /* second half of table is byte-reversed version of first! */
0xA5C6,0x84F8,0x99EE,0x8DF6,0x0DFF,0xBDD6,0xB1DE,0x5491,
0x5060,0x0302,0xA9CE,0x7D56,0x19E7,0x62B5,0xE64D,0x9AEC,
0x458F,0x9D1F,0x4089,0x87FA,0x15EF,0xEBB2,0xC98E,0x0BFB,
0xEC41,0x67B3,0xFD5F,0xEA45,0xBF23,0xF753,0x96E4,0x5B9B,
0xC275,0x1CE1,0xAE3D,0x6A4C,0x5A6C,0x417E,0x02F5,0x4F83,
0x5C68,0xF451,0x34D1,0x08F9,0x93E2,0x73AB,0x5362,0x3F2A,
0x0C08,0x5295,0x6546,0x5E9D,0x2830,0xA137,0x0F0A,0xB52F,
0x090E,0x3624,0x9B1B,0x3DDF,0x26CD,0x694E,0xCD7F,0x9FEA,
0x1B12,0x9E1D,0x7458,0x2E34,0x2D36,0xB2DC,0xEEB4,0xFB5B,
0xF6A4,0x4D76,0x61B7,0xCE7D,0x7B52,0x3EDD,0x715E,0x9713,
0xF5A6,0x68B9,0x0000,0x2CC1,0x6040,0x1FE3,0xC879,0xEDB6,
0xBED4,0x468D,0xD967,0x4B72,0xDE94,0xD498,0xE8B0,0x4A85,
0x6BBB,0x2AC5,0xE54F,0x16ED,0xC586,0xD79A,0x5566,0x9411,
0xCF8A,0x10E9,0x0604,0x81FE,0xF0A0,0x4478,0xBA25,0xE34B,
0xF3A2,0xFE5D,0xC080,0x8A05,0xAD3F,0xBC21,0x4870,0x04F1,
0xDF63,0xC177,0x75AF,0x6342,0x3020,0x1AE5,0x0EFD,0x6DBF,
0x4C81,0x1418,0x3526,0x2FC3,0xE1BE,0xA235,0xCC88,0x392E,
0x5793,0xF255,0x82FC,0x477A,0xACC8,0xE7BA,0x2B32,0x95E6,
0xA0C0,0x9819,0xD19E,0x7FA3,0x6644,0x7E54,0xAB3B,0x830B,
0xCA8C,0x29C7,0xD36B,0x3C28,0x79A7,0xE2BC,0x1D16,0x76AD,
0x3BDB,0x5664,0x4E74,0x1E14,0xDB92,0x0A0C,0x6C48,0xE4B8,
0x5D9F,0x6EBD,0xEF43,0xA6C4,0xA839,0xA431,0x37D3,0x8BF2,
0x32D5,0x438B,0x596E,0xB7DA,0x8C01,0x64B1,0xD29C,0xE049,
0xB4D8,0xFAAC,0x07F3,0x25CF,0xAFCA,0x8EF4,0xE947,0x1810,
0xD56F,0x88F0,0x6F4A,0x725C,0x2438,0xF157,0xC773,0x5197,
0x23CB,0x7CA1,0x9CE8,0x213E,0xDD96,0xDC61,0x860D,0x850F,
0x90E0,0x427C,0xC471,0xAACC,0xD890,0x0506,0x01F7,0x121C,
0xA3C2,0x5F6A,0xF9AE,0xD069,0x9117,0x5899,0x273A,0xB927,
0x38D9,0x13EB,0xB32B,0x3322,0xBBD2,0x70A9,0x8907,0xA733,
0xB62D,0x223C,0x9215,0x20C9,0x4987,0xFFAA,0x7850,0x7AA5,
0x8F03,0xF859,0x8009,0x171A,0xDA65,0x31D7,0xC684,0xB8D0,
0xC382,0xB029,0x775A,0x111E,0xCB7B,0xFCA8,0xD66D,0x3A2C,
}
};

void Phase1(u16b *P1K,const byte *TK,const byte *TA,u32b IV32);
void Phase2(byte *RC4KEY,const byte *TK,const u16b *P1K,u16b IV16);

/*
*********************************************************************
* Routine: Phase 1 -- generate P1K, given TA, TK, IV32
*
* Inputs:
* TK[] = Temporal Key [128 bits]
* TA[] = transmitter's MAC address [ 48 bits]
* IV32 = upper 32 bits of IV [ 32 bits]
* Output:
* P1K[] = Phase 1 key [ 80 bits]
*
* Note:
* This function only needs to be called every 2**16 frames,
* although in theory it could be called every frame.
*
*********************************************************************
*/
void Phase1(u16b *P1K,const byte *TK,const byte *TA,u32b IV32)
{
int i;

/* Initialize the 80 bits of P1K[] from IV32 and TA[0..5] */
```

```
P1K[0] = Lo16(IV32);
P1K[1] = Hi16(IV32);
P1K[2] = Mk16(TA[1],TA[0]); /* use TA[] as little-endian */
P1K[3] = Mk16(TA[3],TA[2]);
P1K[4] = Mk16(TA[5],TA[4]);

/* Now compute an unbalanced Feistel cipher with 80-bit block */
/* size on the 80-bit block P1K[], using the 128-bit key TK[] */
for (i=0; i < PHASE1_LOOP_CNT ;i++)
{ /* Each add operation here is mod 2**16 */
P1K[0] += _S_(P1K[4] ^ TK16((i&1)+0));
P1K[1] += _S_(P1K[0] ^ TK16((i&1)+2));
P1K[2] += _S_(P1K[1] ^ TK16((i&1)+4));
P1K[3] += _S_(P1K[2] ^ TK16((i&1)+6));
P1K[4] += _S_(P1K[3] ^ TK16((i&1)+0));
P1K[4] += i; /* avoid "slide attacks" */
}
}
/*
*********************************************************************
* Routine: Phase 2 -- generate RC4KEY, given TK, P1K, IV16
*
* Inputs:
* TK[] = Temporal Key [128 bits]
* P1K[] = Phase 1 output key [ 80 bits]
* IV16 = low 16 bits of IV counter [ 16 bits]
* Output:
* RC4KEY[] = the key used to encrypt the frame [128 bits]
*
* Note:
* The value {TA,IV32,IV16} for Phase1/Phase2 must be unique
* across all frames using the same key TK value. Then, for a
* given value of TK[], this TKIP48 construction guarantees that
* the final RC4KEY value is unique across all frames.
*
* Suggested implementation optimization: if PPK[] is "overlaid"
* appropriately on RC4KEY[], there is no need for the final
* for loop below that copies the PPK[] result into RC4KEY[].
*
*********************************************************************
*/
void Phase2(byte *RC4KEY,const byte *TK,const u16b *P1K,u16b IV16)
{
int i, j;
```

```
u16b PPK[6]; /* temporary key for mixing */

/* all adds in the PPK[] equations below are mod 2**16 */
for (i=0;i<5;i++) PPK[i]=P1K[i]; /* first, copy P1K to PPK */
PPK[5] = P1K[4] + IV16; /* next, add in IV16 */

for (j=0; j < PHASE2_LOOP_CNT; j++)
{
/* Bijective non-linear mixing of the 96 bits of PPK[0..5] */
PPK[0] += _S_(PPK[5] ^ TK16(0)); /* Mix key in each "round" */
PPK[1] += _S_(PPK[0] ^ TK16(1));
PPK[2] += _S_(PPK[1] ^ TK16(2));
PPK[3] += _S_(PPK[2] ^ TK16(3));
PPK[4] += _S_(PPK[3] ^ TK16(4));
PPK[5] += _S_(PPK[4] ^ TK16(5)); /* Total # S-box lookups == 6 */

/* Final sweep: bijective, linear. Rotates kill LSB correlations */
PPK[0] += RotR1(PPK[5] ^ TK16(6));
PPK[1] += RotR1(PPK[0] ^ TK16(7)); /* Use all of TK[] in Phase2 */
PPK[2] += RotR1(PPK[1]);
PPK[3] += RotR1(PPK[2]);
PPK[4] += RotR1(PPK[3]);
PPK[5] += RotR1(PPK[4]);
PPK[5] += j;
}
/* At this point, for a given key TK[0..15], the 96-bit output */
/* value PPK[0..5] is guaranteed to be unique, as a function */
/* of the 96-bit "input" value {TA,IV32,IV16}. That is, P1K */
/* is now a keyed permutation of {TA,IV32,IV16}. */

/* Set RC4KEY[0..3], which includes cleartext portion of RC4 key */
RC4KEY[0] = Hi8(IV16); /* RC4KEY[0..2] is the WEP IV */
RC4KEY[1] =(Hi8(IV16) | 0x20) & 0x7F; /* Help avoid FMS weak keys */
RC4KEY[2] = Lo8(IV16);
RC4KEY[3] = Lo8((PPK[5] ^ TK16(0)) >> 1);

/* Copy 96 bits of PPK[0..5] to RC4KEY[4..15] (little-endian) */
for (i=0;i<6;i++)
   {
      RC4KEY[4+2*i] = Lo8(PPK[i]);
      RC4KEY[5+2*i] = Hi8(PPK[i]);
      }

}
```

```
/*
**********************************************************************
* Main
**********************************************************************
*/
void main()
{
unsigned int j;
byte TA[TA_SIZE]={0x10,0x22,0x33,0x44,0x55,0x66};
byte
TK[TK_SIZE]={0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,0x0B,0X0C,0x0D,
0x0E,0x0F};
byte RC4KEY[RC4_KEY_SIZE];
u16b IV16;
u32b IV32;
u16b P1K[P1K_SIZE];

double Phase1Millisecond, Phase2Millisecond;
double Phase1Second, Phase2Second;


LARGE_INTEGER Frequency, *lpFrequency;
LARGE_INTEGER Phase1Start,Phase2Start,Phase2End;
LARGE_INTEGER *lpPhase1Start, *lpPhase2Start, *lpPhase2End;

lpPhase1Start=&Phase1Start;
lpPhase2Start=&Phase2Start;
lpPhase2End=&Phase2End;

lpFrequency=&Frequency;
if (QueryPerformanceFrequency(lpFrequency))
   {
   printf("the installed hardware support a high-resolution performance counter!\n");
   printf("The LowPart of Frequency is: %d\n", lpFrequency->LowPart);
   printf("The HighPart of Frequency is %d\n", lpFrequency->HighPart);
   }
else
   {
   printf("the installed hardware does not support a high-resolution performance counter!\n");
   exit(1);
   }
srand((unsigned)time(NULL));
IV32=rand() + (rand() << 16);

TA[0] = TA[0] & 0xFC;      /* Clear I/G and U/L bits in OUI */

if   (!QueryPerformanceCounter(lpPhase1Start))
       {
          printf("QueryPerformanceCounter in Phase 1 start failed!\n");
          exit(1);
          }

Phase1(P1K,TK,TA,IV32);

if   (!QueryPerformanceCounter(lpPhase2Start))
       {
          printf("QueryPerformanceCounter in Phase 2 start failed!\n");
          exit(1);
          }

IV16=0;
for   (j=0;j<65536;j++)
       {
          Phase2(RC4KEY,TK,P1K,IV16);
          IV16++;
          }

if   (!QueryPerformanceCounter(lpPhase2End))
       {
          printf("QueryPerformanceCounter in Phase 2 end failed!\n");
          exit(1);
          }

Phase1Millisecond=(float)(1000)   *   (lpPhase2Start->LowPart   -   lpPhase1Start->LowPart   +
4294967296*(lpPhase2Start->HighPart - lpPhase1Start->HighPart) ) / lpFrequency->LowPart;
Phase1Second=Phase1Millisecond / 1000;
Phase2Millisecond=(float)(1000)   *   (lpPhase2End->LowPart   -   lpPhase2Start->LowPart   +
4294967296*(lpPhase2Start->HighPart - lpPhase1Start->HighPart)) / lpFrequency->LowPart;
Phase2Second=Phase2Millisecond / 1000;

printf("In Phase1, %f millisecond elapsed.\n", Phase1Millisecond);
printf("In Phase1, %f second elapsed.\n", Phase1Second);
printf("In Phase2, %f millisecond elapsed.\n", Phase2Millisecond);
printf("In Phase2, %f second elapsed.\n", Phase2Second);
}
```