Efficient Hardware Implementation of ITUbee for Lightweight Application

Juhua Liu[1], Wei Li[1], Guoqiang Bai[2]

[1]*Institute of Microelectronics, Tsinghua University, Beijing, China,*
[2]*Tsinghua National Laboratory for Information Science and Technology, China*

Abstract

Recently, a new lightweight block cryptography algorithm, ITUbee, has been proposed by Ferhat Karakoc in Lightsec 2013. An efficient hardware implementation of ITUbee is presented in this paper. Firstly, we reuse certain module, which takes a big share of hardware resource, to achieve better resource utilization. Secondly, we apply composite field to implement 8-bit S-box instead of the traditional looking up tables (LUTs) to save area requirements. In the end, we conclude that the hardware implementation of ITUbee requires about 6448 GE on 0.18 um technology. The area consumption of ITUbee is roughly 31.2% less than the round-based implementation. And it costs 365.6 GE to implement 8-bit S-box by using composite field, 32.7% less than by using LUTs.

1. Introduction

Nowadays, the increasing applications such like RFID tags and intelligent devices spur us to develop an efficient cryptography algorithm, which meets the security and privacy requirements and can be applied to resource constrained devices at the same time. For that reason, designing lightweight primitives is getting prominent. Block ciphers play an essential role in cryptography applications so that a considerable number of lightweight block ciphers have been proposed. DESXL [1], PRINCE [2], SEA [3] and KATAN [4], for example.

Ferhat Karakoc et al. proposed a new software oriented lightweight block cipher, ITUbee, for resource constrained devices that include a microcontroller and have a limited battery power such as sensor nodes in wireless sensor networks [5]. ITUbee is designed based on a Feistle structure while having no key schedule, which may make ITUbee subjected to related key attacks as observed in GOST cipher [6]. The author came up with a new approach that the round key was injected between two nonlinear operations to mend this weakness.

To evaluate the performance of ITUbee we have implemented the algorithm on hardware and gave the result of Design Compiler. Especially, to reduce the energy consumption of the cipher we applied the S-box based on composite filed to our design. Note that there are two F functions in each round of encryption, which accounts for more than 90% of the total area. Fortunately, this proportion can be reduced dramatically by reusing the F functions, improving its efficiency in terms of energy consumption.

The rest of the paper is organized as follows. In Section 2, we give the compact algorithm description of ITUbee. In Section 3, some details of design rationale of S-Box based on composite filed is shown. We give the hardware architecture of our implementation in Section 4. In Section 5, we give the simulation details and results of implementation. We conclude the paper with Section 6.

2. Description of ITUbee

2.1. Notations

Before we start the descripting, giving the uniform notations throughout this paper makes reading much easier.

||: Concatenation operator.
K_R: The right half of the master key.
K_L: The left half of the master key.
P_R: The right half of the plaintext.
P_L: The left half of the plaintext.
P: 80-bit plaintext.
C_R: The right half of the ciphertext.
C_L: The left half of the ciphertext.
C:80-bitciphertext.
RC_i: The round constant in the i-th round.

2. Algorithm Description

IUTbee algorithm accepts the inputs $P_L, P_R, K_L, K_R, (RC_1, RC_2, ..., RC_{20})$ and outputs the ciphertext C_L, C_R. ITUbee algorithm is designed with a Feistle structure with 80-bit key length and block size, consisting of 20 rounds overall and having key whitening layers at the first and the last round as illustrated in Figure 1. The details of the encryption process are described as below:

ALGORITHM ITUBEE

Input: P_L , P_R , K_L ,K_R ,(RC_1, RC_2, ..., RC_{20})

Output: C_L ,C_R

1 $X_1 \leftarrow P_L \oplus K_L$, $X_0 \leftarrow P_R \oplus K_R$.

2 *for i* =1...20 *do*

2.1 *if* $i \in \{1,3, ... 19\}$

$RK \leftarrow K_R$

2.1 *else*

$RK \leftarrow K_R$

2.3 $X_{i+1} \leftarrow X_{i-1} \oplus F(L(RK \oplus RC_i \oplus F(X_i)))$

3 $C_L \leftarrow X_{20} \oplus K_R$, $C_R \leftarrow X_{21} \oplus K_L$

4 return C_L ,C_R

In each round, there are one *L* function, two *F* functions and XOR operators, the execution order of these operators is shown in figure 1. The definitions of these functions are:

$F(X) = S(L(S(X)))$, $S(a \| b \| c \| d \| e) = s[a] \| s[b] \| s[c] \| s[d] \| s[e]$, $L(a \| b \| c \| d \| e) = (e \oplus a \oplus b) \| (a \oplus b \oplus c) \| (b \oplus c \oplus d) \| (c \oplus d \oplus e) \| (d \oplus e \oplus a)$,where a, b, c, d, e are 8-bit values and s is the S-box used in advanced encryption standard (AES)[5]. The constant RC_i in each round is given in Table1. Note that 16-bit round constant RC_i is XORed with the rightmost 16 bits in each round.

$(K_L \| K_R)$ and $(K_R \| K_L)$ are used as whitening keys at the first and the last round of the encryption algorithm respectively and for even rounds K_L is used while for odd rounds K_R is used. Both of the round keys and whitening keys are derived from the master key directly. The decryption process of ITUbee is the same as the encryption process, while the only difference is that the round keys and constants are used in reversed order [5].

Table 1. Round constants used in ITUbee algorithm

i	RC_i	i	RC_i	i	RC_i	i	RC_i
1	0x1428	6	0x0f23	11	0x0a1e	16	0x0519
2	0x1327	7	0x0e22	12	0x091d	17	0x0418
3	0x1226	8	0x0d21	13	0x081c	18	0x0317
4	0x1125	9	0x0c20	14	0x071b	19	0x0216
5	0x1024	10	0x0b1f	15	0x061a	20	0x0115

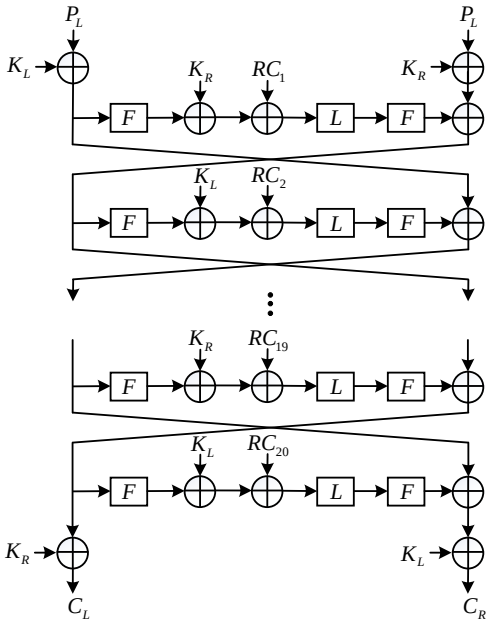


Figure 1. ITUbee encryption algorithm

3. Implementation Of S-Box Using Normal Basis In Composite Filed

To the best of our knowledge, the efficiency of the ITUbee depends on the implementation of S-box involved in F function in each round. The operation of nonlinear multiplication inversion makes S-box the most computational intensive in ITUbee algorithm. There are two mainly approaches in public literature to implement the S-box: using a look up table(LUT) or using a Composite Filed algorithm. Compared with the approach using LUT, the implementation of S-box using composite filed can save area consumption dramatically by performing the 8-bit Galois field inversion of the S-box using subfield of 4 bits and of 2 bits.

Generally, the S-box function with input *a* is defined by two steps: the *multiplicative inverse* in $GF(2^8)$ (see Eq. (1)) and *affine transformation* (see Eq.(2)):

$$c = a^{-1}, \quad (if\ a = 0, then\ c = 0). \qquad (1)$$

$$s = Mc \oplus b, \qquad (2)$$

Where M is a constant bit matrix and b is a byte vector shown below:

$$\begin{pmatrix} s_7 \\ s_6 \\ s_5 \\ s_4 \\ s_3 \\ s_2 \\ s_1 \\ s_0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c_7 \\ c_6 \\ c_5 \\ c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

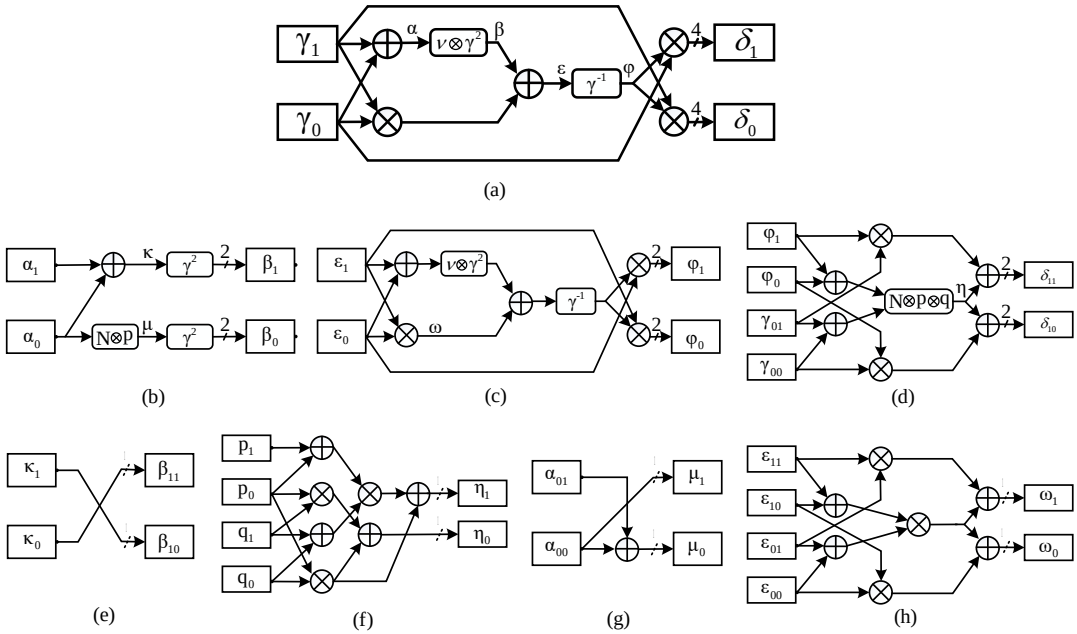


Figure 2. Hierarchical structure

(a) Normal inverter in $GF(2^8)$: $(\gamma_1 Y^{16} + \gamma_0 Y)^{-1} = (\delta_1 Y^{16} + \delta_0 Y)$,the coefficients pair $[\gamma_1, \gamma_0]$ and $[\gamma_1, \gamma_0]$ are of the same bit width, shown at the output of the figure and the same as the below; (b),(c),(d) give the structures of combined operation of squaring then scaling (multiplying) , normal inverter and multiplication in $GF(2^4)$ respectively; (e),(f),(g),(h) then give the structures of squaring (same as inverter),combining the multiplication with scaling by *N*, scaling by *N* and multiplication in $GF(2^2)$ respectively.

Compared with the first substep (multiplicative inverse in *GF* (2^8)), the second substep (affine transformation) is easier to implement on hardware. Therefore, we will focus on the key issue of finding the inverse in *GF* (2^8) . As we know that S-box used in AES algorithm, which is same as the S-box used in ITUbee algorithm, is designed based on the particular Galois field of 8-bit bytes where the bits are coefficients of a polynomial and multiplication is modulo the irreducible polynomial $q(x) = x^8 + x^4 + x^3 + x + 1$, with addition of coefficients modulo 2[7].

Direct calculation of the inverse of a seven-degree polynomial (modulo an eight-degree polynomial using extended Euclidean algorithm) is not quite easy. But calculation of the inversion of a one-degree polynomial, modulo a two-degree polynomial, is pretty easy. Hence, we compact the finding inverse in $GF(2^8)$ into subfield $GF(2^4)$ then into subfield $GF(2^2)$ and finally into subfield $GF(2)$ using a multi-level hierarchical structure, as depicts in Figure 2. Next, we give some details of this hierarchical structure [7].

Now we represent a general element *E* of $GF(2^8)$ as a linear polynomial over $GF(2^4)$, as $g = \lambda_1 y + \lambda_0$, with multiplication modulo an irreducible polynomial $r(y)$ (see Eq.(3)), whose coefficients $[\lambda_1, \lambda_0]$ are in the 4-bit subfield $GF(2^4)$. Although both of normal basis and polynomial basis can decompose the multiplicative inverse in $GF(2^8)$ into its isomorphic subfields, the most compact case uses normal basis for all subfields. Considering this, we choose the normal basis $[Y^{16}, Y]$ to represent the element in $GF(2^8)$ again, $g = \gamma_1 Y^{16} + \gamma_0 Y$,where the $[\gamma_1, \gamma_0] = [g_{7:4}, g_{3:0}]$. Similarly, we get all the irreducible polynomials $s(x)$, $t(w)$ and their normal basis $[X^4, X]$ and $[W^2, W]$ respectively (see Eq.(4) and Eq.(5)). Here we have:

$$r(y) = y^2 + \tau y + \nu = (y + Y^{16})(y + Y) \quad (3)$$

$$s(x) = x^2 + Tx + N = (x + X^4)(x + X) \quad (4)$$

$$t(w) = w^2 + w + 1 = (w + W^{16})(w + W) \quad (5)$$

In Eq. (3) we define the trace $\tau = Y + Y^{16}$ and the norm $\nu = Y \cdot Y^{16}$ (correspondingly $T = X+X^4$, $N = X \cdot X^4$ for Eq.(4) and $W + W^2 = 1$ $W \cdot W^2 = 1$ for Eq.(5). The most efficient choice of trace and norm is to let the trace be unity, here we let $\tau = T = 1$.

In $GF(2^8)$ with a normal basis $[Y^{16}, Y]$, the inverse of $g = (\gamma_1 Y^{16} + \gamma_0 Y)$ modulo $y^2 + \tau y + \nu$ is given by:

$$g^{-1} = (\gamma_1 Y^{16} + \gamma_0 Y)^{-1} = \delta_1 Y^{16} + \delta_0 Y = [\theta^{-1}\gamma_0]Y^{16} + [\theta^{-1}\gamma_1]Y \quad (6)$$

where $\theta = \gamma_1\gamma_0\tau^2 + (\gamma_1^2 + \gamma_0^2)\nu$

then we have:

$$\delta_1 = \theta^{-1}\gamma_0 = [\gamma_1\gamma_0\tau^2 + (\gamma_1^2 + \gamma_0^2)\nu]^{-1}\gamma_0 \quad (7)$$

$$\delta_0 = \theta^{-1}\gamma_1 = [\gamma_1\gamma_0\tau^2 + (\gamma_1^2 + \gamma_0^2)\nu]^{-1}\gamma_1 \quad (8)$$

So finding the inverse of g in GF (28) reduces to an inverse operation and several additions and multiplications in GF (24) , as shown in Figure 2 (a). It easy to handle the addition in GF (24) by bitwise XOR. While for the inverse and multiplication in GF (24) we give their definitions and algorithms using the similar approach above.

In $GF(2^4)$ with a normal basis $[X^4, X]$, the inverse of $\gamma = (\varepsilon_1 X^4 + \varepsilon_0 X)$ modulo $x^2 + Tx + N$ has a same formulation as Eq.(6) except the 2-bit width coefficients.

$$\gamma^{-1} = (\varepsilon_1 X^4 + \varepsilon_0 X)^{-1} = \varphi_1 X^4 + \varphi_0 X = [\Delta^{-1}\varepsilon_0]X^4 + [\Delta^{-1}\varepsilon_1]X \quad (9)$$

where $\Delta = \varepsilon_1\varepsilon_0 T^2 + (\varepsilon_1^2 + \varepsilon_0^2)N$

then we have:

$$\varphi_1 = \Delta^{-1}\varepsilon_0 = [\varepsilon_1\varepsilon_0 T^2 + (\varepsilon_1^2 + \varepsilon_0^2)N]^{-1}\varepsilon_0 \quad (10)$$

$$\varphi_0 = \Delta^{-1}\varepsilon_0 = [\varepsilon_1\varepsilon_0 T^2 + (\varepsilon_1^2 + \varepsilon_0^2)N]^{-1}\varepsilon_1 \quad (11)$$

We can see that finding the inverse of γ means an inverse and several additions and multiplications in *GF* (2^2), shown in Figure 2 (c). As to the multiplication in $GF(2^4)$, $\varphi \cdot \gamma_0$ is defined by (shown Figure 2 (d)):

$$(\varphi_1 X^4 + \varphi_0 X) \cdot (\gamma_{01} X^4 + \gamma_{00} X) = \delta_{11} X^4 + \delta_{10} X \quad (12)$$

Where

$$\delta_{11} = \eta \oplus (\varphi_1 \otimes \gamma_{01}) = (N \otimes (\varphi_1 \oplus \varphi_0)(\gamma_{01} \oplus \gamma_{00})) \oplus (\varphi_1 \otimes \gamma_{01}) \quad (13)$$

$$\delta_{10} = \eta \oplus (\varphi_0 \otimes \gamma_{00}) = (N \otimes (\varphi_1 \oplus \varphi_0)(\gamma_{01} \oplus \gamma_{00})) \oplus (\varphi_0 \otimes \gamma_{00}) \quad (14)$$

To compact the logic and simplify the circuit, we combine the operations of squaring and scaling by the norm ν ($\nu = N^2 X$)(shown in Figure 2 (a)):

$$\nu \otimes (\alpha_1 X^4 + \alpha_0 X)^2 = \beta_1 X^4 + \beta_0 X = [(\alpha_1 + \alpha_0)^2]X^4 + (\alpha_0 \otimes N)^2 X \quad (15)$$

Obviously, we can write the coefficients:

$$\beta_1 = (\alpha_1 + \alpha_0)^2, \beta_0 = (\alpha_0 \otimes N)^2 \quad (16)$$

Note we continue decomposing the operation in $GF(2^4)$ into $GF(2^2)$,note that into $GF(2^2)$ the inverse is the same as squaring, which is free with a normal basis (shown in figure 2 (e)):

$$(k_1 W^2 \oplus k_0 W)^{-1} = (k_1 W^2 \oplus k_0 W)^2 = k_0 W^2 \oplus k_1 W \quad (17)$$

The multiplication in $GF(2^2)$ has a same structure as in $GF(2^4)$ except the scaling norm is 1 (show in Figure 2(h)). Up to now the remaining operation need in subfield $GF(2^2)$ is scaling by $N = W^2$ and combined operation of multiplication with scaling by *N* (shown in Figure 2 (g),(f)).

$$N \otimes (\alpha_{01} W^2 \oplus \alpha_{00} W) = [\alpha_{00}]W^2 + [\alpha_{00} + \alpha_{01}]W \quad (18)$$

$$N \otimes (p_1 W^2 + p_0 W) \otimes (q_1 W^2 + q_0 W) = \eta_1 W^2 + \eta_0 W = [(p_0 \otimes q_0) \oplus ((p_1 \oplus p_0) \otimes (q_1 \oplus q_0))]W^2 + [(p_0 \otimes q_0) \oplus (p_1 \otimes q_1)]W \quad (19)$$

Where

$$\eta_1 = (p_0 \otimes q_0) \oplus ((p_1 \oplus p_0) \otimes (q_1 \oplus q_0)) \quad (20)$$

$$\eta_0 = (p_0 \otimes q_0) \oplus (p_1 \otimes q_1) \quad (21)$$

In $GF(2)$, $\otimes$ means AND and $\oplus$ means XOR bitwise [7].

4. Hardware Implementation

In order to reduce area consumption, we proposed an efficient hardware implementation by reusing certain module and applying composite field to implement 8-bit S-box. As depicted in Figure 1, the F module consists of two 8-bit S-boxes and is reused two times. 8-bit S-box, as non-linear layer, costs a large proportion of area resource. Therefore, we divide one round into three cycles, so that the F module can be reused.

Three cycles are used to implement one round. Firstly, we use two 80-bit registers to store the state and key respectively. Additionally, one 40-bit register is used to store initial value for later swapping in the last cycle. And one 16-bit register is used to store round constant. The output at the end of each cycle will be stored in the registers and reused as the input data at the beginning of the next cycle. Consequently, in the

first cycle, one F module is used. In the second cycle, one 40-bit XOR, one 16-bit XOR and one L module are used. In the third cycle, F module can be used again. Therefore, F module is used two times in one round.

The reuse of F module saves a significant amount of area in hardware implementation. The L module can be implemented with simple XORs and bit operations. Figure 3 shows the datapath of ITUbee, which performs one round in 3 clocks and needs 61 clocks in total to implement 20 rounds. For comparison, we also implement a round-based architecture, which performs one round in one clock. In this way, one 40-bit register for storing initial value is saved, but the F module is used two in one single cycle, which increase the area extremely.

The 8-bit S-Box in ITUbee is the S-box used in AES. Except for using LUTs, which is simple 256 case statements in hardware, we can apply the mathematical formula to compute S-box. As we give in section 3, we use composite field to solve the complex inverse calculation, which turns out to be more efficient in hardware,

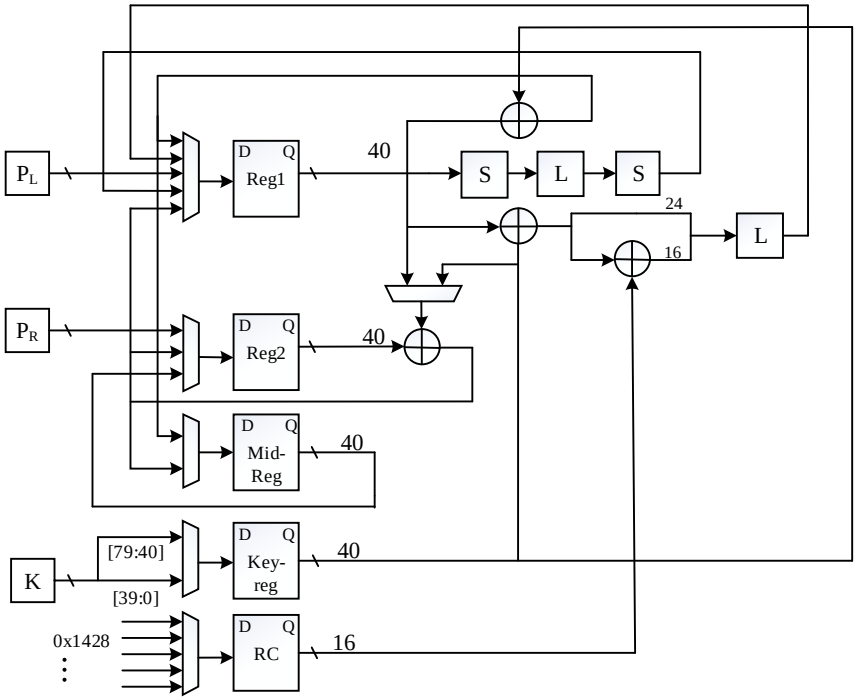


Figure 3. datapath of the hardware implementation of ITUbee

The reuse of F module saves a significant amount of area in hardware implementation. The L module can be implemented with simple XORs and bit operations. Figure 3 shows the datapath of ITUbee, which performs one round in 3 clocks and needs 61 clocks in total to implement 20 rounds. For comparison, we also implement a round-based architecture, which performs one round in one clock. In this way, one 40-bit register for storing initial value is saved, but the F module is used two in one single cycle, which increase the area extremely.

The 8-bit S-Box in ITUbee is the S-box used in AES. Except for using LUTs, which is simple 256 case statements in hardware, we can apply the mathematical formula to compute S-box. As we give in section 3, we use composite field to solve the complex inverse calculation, which turns out to be more efficient in hardware.

5. Results

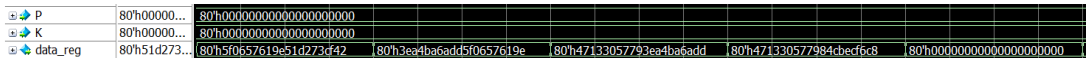
We implemented the proposed design in verilog DHL and synthesized it on 0.18 um CMOS technology to check its hardware complexity. In this area-optimized implementation, a 40-bit width datapath was used. In order to compare the area requirements independently it is common to state the area as gate equivalents (GE). One GE is equivalent to the area which is required by the two-input NAND gate with the lowest driving strength of the corresponding technology. The area in GE is

derived by dividing the area in um^2 by the area of a two-input NAND gate. Encrypting 80-bit plaintext with an 80-bit key occupies about 6748 GE and requires 61 clock cycles.

An example of ITUbee as test vector is illustrated in Table 2. The simulation results are shown in Figure 4.

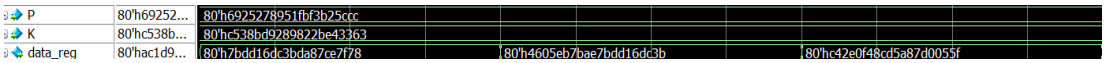
Table 2. A sample test of encryption of ITUbee

Plaintext	Key	Ciphertext
00000000000000000000	00000000000000000000	471330577984cbecf6c8
00000000000000000000	00000000000000000080	761b8299b3f6a99f0838
6925278951fbf3b25ccc	c538bd9289822be43363	c42e0f48cd5a87d0055f



(a)

(b)



(c)

Figure 4. simulation results of ITUbee

Specifically, in the above implementation, the area requirement is mostly occupied by S-boxes registers. 80-bit state register costs 612GE. 40-bit key register costs 306 GE. 40-bit middle register costs 306 GE. 16-bit round constant register costs 122.4 GE. one 8-bit S-box requires 356.6 GE. L module consumes 170.3 GE. 40-bit XOR costs 106.4 GE.

Table 3 gives the comparison between LUTs implementation and composite field implementation of 8-bit S-box, and table 4 gives the comparisons between proposed implementation and round-based implementation of ITUbee.

Table 3. Area comparison between LUTs and composite field

Composite-field(GE)	LUTs(GE)
543.8	356.6

Table 4. Area comparison between proposed and round-based implementation

	Combinational area(GE)	Noncombinational(GE)
Proposed (6747.9)	5497.7	1350
round-based (9812.8)	87757	10371

Conclusively, the proposed hardware implementation of ITUbee requires about 6448 GE on 0.18 um technology. The area consumption of ITUbee is roughly 31.2% less than the round-based implementation. And it costs 365.6 GE to implement 8-bit S-box by using composite field, 32.7% less than by using LUTs.

6. Conclusion

There is a great improvement in terms of area consumption by using the composite field to implement the S-box compared with the approach using LUTs. In our hardware architecture, we reuse the S-box in F function, which consists of ten 8-bit S-boxes and area occupancy proportion is more than 90%, by dividing each round into 3 clock cycles to reduce the area consumption further. We implemented the proposed design in and synthesized it on 0.18um CMOS technology, the results show that the area is saved by 32.7% by using composite field S-box compared with using LUTs, and 31.2% by reusing S-box compares with not reusing.

7. Acknowledgments

This work is supported by the National Natural Science Foundation of China (Grants 61472208) and National Key Basic Research Program of China (Grant 2013CB338004).

8. References

[1] Leader, G., Paar, C., Poschmann, A., Schramm, K., (2007): New Lightweight DES Variants. In: Biryukov, A. (ed) FSE LNCS, Vol. 4593, pp196-210. Springer, Heidelberg (2007).

[2] Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E.B., Knezevic, M., Knudsen, L.R., Leander, G., Nikov, V., Paar, C., Rechberger, C., Rombouts, P., Thomsen, S.S., Yalçın, T. (2012): PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 208–225. Springer, Heidelberg.

[3] Standaert, F.-X., Piret, G., Gershenfeld, N., Quisquater, J.-J. (2006): SEA: A Scalable Encryption Algorithm for Small Embedded Applications. In: Domingo-Ferrer, J., Posegga, J., Schreckling, D. (eds.) CARDIS 2006. LNCS, vol. 3928, pp. 222–236. Springer, Heidelberg.

[4] De Cannière, C., Dunkelman, O., Knežević, M. (2009) : KATAN and KTANTAN — A Family of Small and Efficient Hardware-Oriented Block Ciphers. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 272–288. Springer, Heidelberg.

[5] Ferhat Karakoc, Hüseyin Demirci. & A. Emre Harmanci, (2013): ITUbee: A Software Oriented Lightweight Block Cipher, Lightsec, LNCS 8162, pp16-27.

[6] Zabotin, I.A., Glazkov, G.P., Isaeva, V.B. (1989): Cryptographic Protection for Information Processing Systems. Cryptographic Transformation Algorithm. Government Standard of the USSR, GOST 28147-89.

[7] D.Canright, , (2005) ,A very compact S-box for AES, Springer.

