Ultra-lightweight 8-bit Multiplicative Inverse Based S-box Using LFSR

Sourav Das

Alcatel-Lucent India Ltd
Email:sourav10101976@gmail.com

Abstract. Most of the lightweight block ciphers are nibble-oriented as the implementation of a 4-bit S-box is much more compact than an 8-bit S-box. This paper proposes a novel implementation of multiplicative inverse for 8-bit S-boxes using LFSR requiring only 138 gate-equivalent. It can be shown that if such S-boxes are adopted for the AES it takes less than 50 gate-equivalent per S-box in parallel implementation. Canright's [1] implementation of the AES S-box is five times more expensive compared to this method for AES-like S-boxes. With this powerful scheme, a lightweight block cipher can be designed using an 8-bit S-box.

Key words: Multiplicative Inverse, AES, LFSR, Lightweight Cryptography

1 Introduction

The goal of lightweight cryptography is to have cryptographic primitives for extremely constrained devices (i.e. with minimal hardware) without sacrificing on the cryptographic strengths. While the AES is suitable for most of the applications, the hardware requirement for the AES is considered to be high for these tiny devices. The AES uses multiplicative inverse for the S-box which is the main contributor of its security. However, the S-box layer in AES is the most hardware resource consuming construct unless the S-boxes are used serially as in [6]. There have been many approaches for reduction of hardware for AES [1], [3], [5], [6], [9], [10], [11]. Among these, the approach of Satoh et al [10] and Canright's approach [1] are the most important ones for reducing the hardware resources of the AES S-box. In Satoh's approach, the subfield arithmetic by breaking the field $GF(2^8)$ to several smaller subfield of $GF(2^4)$ as originally suggested by Rijmen, was further extended using the tower field [8] approach of Paar by breaking up further to $GF(2^2)$. Canright further optimized that approach by considering normal bases in addition to polynomial bases along with optimization in the gates. Till date, this approach provides the tiniest hardware implementation of not only AES S-box but also multiplicative inverse.

This paper takes a completely different approach for implementing the multiplicative inverse and proposes a novel hardware efficient algorithm to find out the multiplicative inverse. It simply uses a maximum length Linear Feedback

Shift Register (LFSR) running both forward and backwards for constant number of times to find out the multiplicative inverse. As LFSR requires a very small hardware, this approach reduces the hardware requirement significantly for the AES-like S-Boxes using multiplicative inverse. Canright's approach requires 85 percent more area compared to this method. One drawback of this method is that full length cycles need to be run for the LFSR. However, as the LFSR transformation has a very minimal critical path, the speed loss is not as high as it seems to be. Hence, the number of cycles is not the right metric to compare the speed. Also, speed is not the main concern in lightweight cryptography. Nevertheless, we also propose a speed improvement for the S-box with an additional hardware. This approach takes 15 cycles where the hardware cost is comparable with Canright's implementation. We provide a better tradeoff using 31 cycles where Canright's approach is 15 percent more expensive. Also, we can easily show that if such S-boxes are adopted for AES (or any other 128/256 bit cipher), it takes around 50 gate-equivalent per S-box in the S-box layer. The number of cycles remains the same across the S-box layer. Hence, in a 256 bit block cipher using these S-boxes, it needs only 1 cycle per bit per round of S-box transformation. As multiplicative inverse has strong cryptographic properties, we believe that these algorithms will help greatly in future design of symmetric key algorithms and hash functions with low hardware count.

This paper is organized as follows. Section 2 provides the schematic of multiplicative inverse implementation using LFSR that can be used for AES. It describes the compact hardware method in Section 2.1 and a method for better speed in Section 2.2. Section 2.3 provides the description of parametrization of the S-box.

2 Multiplicative Inverse Using LFSR

In this section, we present two different hardware implementations for calculating multiplicative inverse. One method of implementation is compact hardware mode where extremely minimal hardware is required. The second method is the speed improvement mode where the speed is improved with additional hardware. Both the methods use maximum length LFSR. See [4] for a detailed description of LFSR. Throughout the rest of the paper referring to LFSR would mean a maximum length LFSR with a given primitive polynomial.

2.1 Compact Hardware Mode

We begin this section with an introduction of how mathematically multiplicative inverse can be calculated using LFSR. The LFSR transformation can be written as for a single cycle:

$S(t+1) = T \cdot S(t)$

where, T is the LFSR transformation matrix, $S(t)$ is the state of the LFSR at t^{th} time instant or the initial seed and $S(t+1)$ is the state of the LFSR at

$(t+1)^{th}$ time instant i.e. after running one clock cycle. We can generalize the above equation for any number of cycles p as:

$S(t+p) = T^p \cdot S(t)$

It can be noted that for a maximum length LFSR, running $2^n - 1$ cycles gives back the initial state (where n is the length of the LFSR), i.e.:

$S(t+2^n-1) = T^{2^n-1} \cdot S(t) = S(t) \Rightarrow T^{2^n-1} = 1$, which is the identity element.

To calculate the multiplicative inverse of a given input $S(t+p)$, the task is to find out a new state $S(t+\acute{p})$ of the LFSR so that $p + \acute{p} = 2^n - 1$, implying, $S(t+p+\acute{p}) = S(t+2^n-1) = S(t)$ or alternatively, $T^p \cdot T^{\acute{p}} = T^{2^n-1}$. The above implies the following equation for an 8-bit LFSR:

$$S(t+\acute{p}) = S(t+255-p) \tag{1}$$

One way to implement this is to run the LFSR with a particular initial seed till the LFSR state matches the input, then re-initialize the LFSR with the same seed and run it. When the total number of cycles in both the run is 255 (for 8-bit LFSR) the state of the LFSR gives the multiplicative inverse. Comparison of two eight bit variable requires eight XOR gates, eight NOT gates and one eight input NAND gate along with the LFSR circuit. Additionally, eight 2:1 Mux are required for reloading the initial value to the LFSR. However, we find a better optimization as given below.

Note that, the comparison of the LFSR state with the constant initial seed is very easy. It only needs an eight input NAND (or AND) gate along with a few NOT gates. The input to this NAND gate are the LFSR state bits. For the bits that are zero in the initial seed, the corresponding state bits are negated by NOT gates. When the state becomes equal to the constant initial seed, the output of the NAND gate becomes zero.

Then, we use the following algorithm where the comparison is only performed with constant initial seed.

Require: 8-bit LFSR, initial_seed=S(t), S-box_input=S(t+p)

1: Initialize the LFSR with lfsr_state=S-box_input=S(t+p)
2: Run the LFSR in the forward direction till lfsr_state=initial_seed=S(t)
3: Run the LFSR in the reverse direction
4: Stop when total number cycles in both the above steps is 255
5: Output lfsr_state=S(t+$\acute{p}$)

Theorem 1. *The algorithm above outputs the multiplicative inverse of S-box_input, S(t+p).*

Proof. Let, S-box_input correspond to the lfsr_state after running p cycles of LFSR from initial_seed. This is the state of the LFSR at step 1. Then, in step 2, the number of cycles run is, $255-p$, but the LFSR contains the initial_seed=S(t) at this point. The number of cycles run in step 4 is $255-(255-p) = p$. But as mentioned in step 3, the LFSR is running in reverse direction at this point with initial seed as S(t). Hence the state of the LFSR is S(t-p). Since, S(t)=S(t+255),

the state of the LFSR can also be denoted as S(t+255-p). From Equation 1, this state is the multiplicative inverse of S(t+p). □

Hardware Implementation: The practical implementation is as follows.

1. Use eight two-input flip-flops (e.g. scan flip-flops) to store the LFSR state.

2. Arrange one of the inputs of the flip-flops to make the forward LFSR transformation for a given primitive polynomial. Use 2:1 Muxes at the input to load the S-box input on Reset signal. As the combinational logic of the LFSR is applied only on the first flip-flop, the initial loading can also be applied serially where the 2:1 Mux is used only at the first flip-flop. An eight input NAND gate from the existing counter can indicate the completion of eight cycles for the serial loading. The output of this NAND8 gate can go to another 2-input NAND gate to control the Reset signal. The state of the LFSR counter after eight cycles is considered as the initial state for counting the S-box cycles.

3. Arrange the other inputs of the flip-flops to make reverse LFSR transformation for the same primitive polynomial.

4. The output of the LFSR is connected to an 8-input NAND gate (via a few NOT gates) whose output is connected to the Select input of the flip-flops (via a flip-flop so that the output stays there after a match is found). This provides the comparison with the constant seed and the control logic for the LFSR to run in the reverse direction.

5. An 8-bit LFSR counter is used. The output of the counter is connected to an 8-input NAND gate (via a few NOT gates) to signal when LFSR state contains the multiplicative inverse of the input. This provides the control logic to indicate the completion of 255 cycles.

6. The circuit diagram is shown in Figure 1.

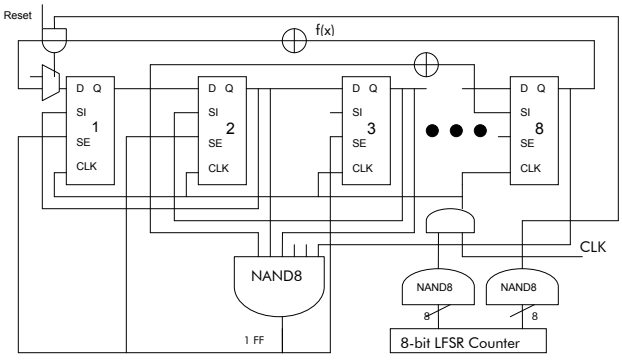

Fig. 1. Circuit Diagram for Compact Hardware Mode

Hardware Cost and Gate Count: We use the primitive polynomial $x^8+x^4+x^3+x^2+1$ which requires three XOR gates for the LFSR feedback function. The total numbers of various gates required to realize the circuit are eight 2-input flip-flops, six 2-input XOR gates, one NAND8 gates and two NOT gates. Using serial loading, the initialization requires 1 mux, 1 NAND8 and 1 NAND gate. The counter requires one NAND8 gate, three XOR gates and eight 1-input flip-flops.

In addition, to avoid the S-box giving zero as output when the input is zero, two NOT gates are added at the input. Using Standard Cells UMCL18G212T3, the total equivalent gate count is $6 \times 16 + 2.67 \times 9 + 4 \times 3 + 0.67 \times 2 + 2.33 \times 1 + 1 \times 1 + 0.67 \times 2 = 138$.

To compare with the existing standards, Canright's implementation takes 253 GE and Satoh's implementation takes 275 GE. Hence Canright's implementation is 85 percent more expensive than this method. The speed is lower in terms of number of cycles. But as stated before, the number of cycles should not be the only metric as the experience shows that the speed reduces drastically when the number of gates is increased due to gate delay. However, as AES uses a polynomial which is irreducible but not primitive, getting the exact AES S-box using this method is not possible. Maximum length LFSRs necessarily need primitive polynomials. Hence, this comparison is with respect to AES-like S-boxes. The AES designers have mentioned that other S-boxes satisfying the same cryptographic properties as with the AES S-box can be used for AES. However, till date that replacement was never attempted as there was no real benefit of doing that. This method generates AES-like S-boxes with same cryptographic properties as in the AES S-box and provides a real benefit of saving the hardware count greatly. By reusing the data state flip-flops for the LFSR and with the common counter for all S-boxes in the S-box layer, the hardware count of AES S-box can be as low as 50 gate equivalent per S-box (see Appendix for a detailed calculation). Hence, we can think about replacing the non-primitive irreducible polynomial of AES with a primitive polynomial that can provide the implementation using LFSR with a great reduction in hardware. Note that, the "super S-box" implementation, Canright's and Satoh's implementations will still be applicable after changing the polynomial. But, since the AES S-box is widely scrutinized and deployed, we leave it at this point for the community to decide on that.

2.2 Speed Improvement Mode

In this mode of implementation, it requires only 15 cycles to calculate the multiplicative inverse. Here, we divide the whole state space of 8-bit LFSR into sixteen zones, each zone is of length 16 states. We denote the zones as $Z_i = (S(t+16 \cdot i), \cdots .S(t+16 \cdot (i+1)-1))$ where $i \in (0, \cdots, 15)$. We generate a mapping of input zone to output zone $Zin_i \rightarrow Zout_j$, where Zin_i and $Zout_j$ denote the input zone and output zone, respectively. For example, if the input is in zone Z_0, then the output zone is Z_{15}; if the input is in zone Z_9, the output zone is Z_6 and so on. In implementation, the zone mapping is simply a mapping between the highest states of the zones i.e. $S(t+16 \cdot (i+1)-1) \rightarrow S(t+16 \cdot ((15-i)+1)-1)$ or, $S(t + 16 \cdot (i + 1) - 1) \rightarrow S(t + 16 \cdot (16 - i) - 1)$.

The method to calculate the multiplicative inverse is as follows. First we determine the input zone by initializing the LFSR with the input and comparing with the highest state of the sixteen zones. This comparison requires sixteen 8-input NAND gates and an average of 64 NOT gates (four NOT gate each). From the $Zin_i \rightarrow Zout_j$ mapping, we load the highest state in $Zout_j$ in the reverse

LFSR and run till overall 15 cycles complete. At this point the output of the LFSR will contain the multiplicative inverse. This is outlined in the following algorithm.

Require: 8-bit LFSR, the maximum state in zones $S(t+16\cdot i-1)$, zone mapping $S(t+16\cdot(i+1)-1) \rightarrow S(t+16\cdot(16-i)-1)$, S-box_input=S(t+p)

1: Initialize the LFSR with lfsr_state=S-box_input=S(t+p)
2: Run the LFSR in the forward direction till lfsr_state= $S(t+16\cdot(i+1)-1)$ for any $i \in (0\cdots 15)$
3: Load the LFSR with the maximum state of $Zout_i$ i.e. $S(t+16\cdot(16-i)-1)$
4: Run the LFSR in the reverse direction
5: Stop when total number cycles in both the above steps is 15
6: Output lfsr_state=S(t+$\acute{p}$)

Theorem 2. *The algorithm above produces the multiplicative inverse of S-box input, S(t+p)*

Proof. In step2, the state of the LFSR is $S(t+16\cdot(i+1)-1)$ and the number of cycles run is $16\cdot(i+1)-1-p$. In step 3, the state of the LFSR is $S(t+16\cdot(16-i)-1)$. In steps 4 and 5, the number of cycles run is $15-16\cdot(i+1)+1+p$. The state of the LFSR is $S((t+16\cdot(16-i)-1)-(15-16\cdot(i+1)+1+p)) = S(t+256-16\cdot i-1-16+16\cdot i+16-p) = S(t+255-p)$. But, from Equation 1, we have $S(t+255-p) = S(t+\acute{p})$. Hence the algorithm produces the multiplicative inverse of the input. □

Hardware Implementation: The practical implementation is as follows.

1. Keep the LFSR structure same as in Figure 1 except the second input is taken from a mapping module.

2. In this implementation, we need two additional blocks, namely, a comparator module and one mapping module.

3. The comparator module takes the input from the LFSR states and the output is connected to the select input of the flip-flops. Inside the module it contains sixteen 8-input NAND gates in parallel whose inputs are LFSR state bits with some of the bits complemented by NOT gates. The output of the NAND8 gates are connected to a 16-input OR gate whose output is the output of the module.

4. The mapping module is implemented using boolean equations (LUT). Since most of the entries in the mapping table are zeros, the LUT approach does not require large hardware. The input is from LFSR state bits and the output is connected to the second input of the flip-flops. The number of logic gates required will vary and will be dependent on the initial seed ($S(t)$). In our implementation, it required 27 NOT gates, 47 AND gates and 12 OR gates.

5. A 4-bit LFSR counter is used. The output of the counter is connected to a 4-input NAND gate (via a few NOT gates) to signal when LFSR state contains the multiplicative inverse of the input i.e 15 cycles are run.

6. 2:1 Muxes are added in the reverse input as well since a new seed needs to be reloaded when the reverse LFSR starts. The circuit diagram is shown in Figure 2.

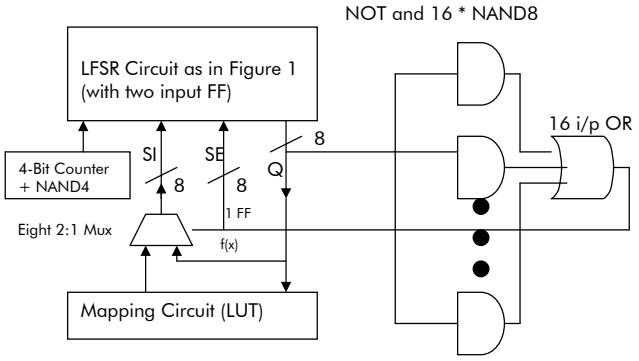

Fig. 2. Circuit Diagram for Speed Improvement Mode

Hardware Cost and Gate Count: We use the primitive polynomial $x^8+x^4+x^3+x^2+1$ which requires three XOR gates for LFSR feedback function. The total numbers of various gates required to realize the circuit are eight 2-input flip-flops, six 2-input XOR gates, for comparator sixteen NAND8 gates along with one sixteen input OR gate, for the mapping module 27 NOT gates, 47 AND gates and 12 OR gates and four 1-input flip-flops and one XOR gate for 4-bit counter. Since, the LFSR is reloaded, eight 2:1 Mux will also be required. The total equivalent gate count is $6\times 8+2.67\times 6+4\times 16+10\times 1+0.67\times 27+1.33\times 47+1.33\times 12+4.67\times 4+2.67\times 1+2.33\times 8=274$ (same as in Satoh's).

In summary, we can achieve the higher speed with this approach. The extra hardware was taken mainly by the comparator and the mapping modules. A better speed vs area trade-off will be achieved with 31 cycles.

Hardware Cost for 31 Cycles: We use the same algorithm above for 31 cycles. In this case, we divide the whole state space into eight zones. Same hardware architecture is used, but in this case the hardware requirements for comparator and mapping modules reduce considerably. For the comparator module, we need eight 8-input NAND gate and one 8-input OR gate. The mapping module in our implementation requires 24 NOT gates, 31 AND gates and 13 OR gates. The counter now requires 5 flip-flops and one XOR gate (with feedback polynomial x^5+x^2+1). The LFSR module requires the same hardware as above i.e. eight 2-input flip-flops, six 2-input XOR gates and eight 2:1 Mux. The total gate equivalent count in this case is, $6\times 8+2.67\times 6+4\times 8+6\times 1+0.67\times 24+1.33\times 31+1.33\times 13+4.67\times 5+2.67\times 1+2.33\times 8=220$.

Hence, we reduced the hardware to a great extent still achieved good speed in terms of the number of cycles. This method is 15 percent better with respect to gate equivalent than Canright's approach.

2.3 Parametrization

The multiplicative inverse S-box using LFSR takes the initial seed, $S(t)$, as a parameter. Selection of a seed doesn't have an impact on the main security properties of the S-box i.e. bias, non-linearity, differential uniformity, SAC etc.

This also provides a great advantage that the intra-S-box linear transformation used in AES is not required for LFSR based implementation of multiplicative inverse. Since the internal linear transformations are different for different seeds, the number of terms in algebraic expression will vary for different seeds. The algebraic degree, however, is always 7, hence this is not a big threat. LFSRs are already known for their excellent statistical properties which are applied to the S-box automatically. The only additional hardware required is for avoiding the zero to zero mapping in the multiplicative inverse. This can be achieved by putting just a couple of NOT gates at the output of forward S-box and in the input of the reverse S-box. However, special care may be needed in selecting the S-box when used in a cipher depending on the linear layer or the structure of the cipher. In order to alleviate any concern with this variable number of terms in algebraic expression, we provide a concrete S-box table and compare the security properties with AES.

2.4 A Concrete S-box

A concrete S-box Table was generated with initial seed as 0x16 and constant value 0x24 that is XORed at the output requiring two NOT gates. These values were used for the specific implementation of *Halka*. The full table is shown in Table 1.

Table 1. A Concrete S-box

24	2c	20	dc	26	73	d8	91	25	b7	8f	9c	da	1f	fe	e9
9f	a4	d5	6d	c3	71	32	78	96	db	55	b9	4c	49	6e	42
9a	f9	1d	64	3	5c	a0	0	4a	d7	e3	8e	75	af	b	a
7d	4d	5b	1a	1c	e7	6a	74	10	6	92	29	81	79	17	40
7	7b	69	ca	c8	b8	ef	84	c2	37	3a	98	df	66	12	b6
13	8	5d	fc	47	31	f1	21	8c	14	e1	51	33	19	b3	65
88	4e	90	70	1b	a8	3b	cc	38	15	45	a7	83	39	c	de
a1	3e	c1	b5	eb	7f	ac	a2	1	76	9b	8a	b4	bd	99	16
35	d4	8b	4f	2	54	53	be	52	c7	ea	9	41	c6	f4	b1
58	57	6b	2d	f8	ab	87	7a	f6	59	a3	85	61	3f	9e	ed
63	bf	fd	b2	e8	18	d2	48	7c	95	f	2e	44	ce	5f	a6
f0	8d	3c	f5	46	23	1e	d0	2f	ee	ba	34	6f	5a	4	5e
c5	f2	c4	11	e2	7e	e0	e	dd	bb	9d	62	80	2b	ae	50
aa	97	bc	c9	94	72	e5	d3	77	86	2a	cd	b0	5	d9	d1
e6	e4	a9	ad	d6	56	6c	30	43	ff	89	cb	60	f7	67	cf
a5	36	c0	d	93	fb	82	f3	27	ec	4b	68	22	fa	28	3d

The comparison of security properties of the S-box generated with the above parameters with AES S-box is shown in Table 2. It can be seen that the security properties are essentially same as it is expected. Note that, we have used algebraic normal form to compare the algebraic properties for convenience, unlike the univariate polynomial expression given in original AES specification.

Table 2. Comparison of Security Properties with AES

S-box	Max Alg Term	Min Alg Term	Alg Deg	Diff Uni	Bias	Max SAC	Min SAC	Max NL	Min NL
AES	145	110	7	4	2^{-4}	144	116	114	112
Halka	139	118	7	4	2^{-4}	140	112	114	112

To summarize, we presented a novel method for implementing multiplicative inverse with a great hardware compactness and showed that the method can even be used for 8-bit S-boxes with area better than existing standard. The area requirement in full cycle case is so small that even a lightweight block cipher can be proposed with 8-bit S-boxes.

3 Conclusion

In this paper a novel method of implementing multiplicative inverse using LFSR is proposed. It was shown that if AES used a primitive polynomial, instead of only irreducible polynomial, then it could implement the S-box with less than 50 GE per S-box in AES-256. With this scheme, the security of the lightweight block ciphers can be enhanced greatly.

References

1. D. Canright. A very compact S-box for AES. CHES 2005. LNCS vol. 3659. pp. 441-455. Springer, 2005.
2. C. Carlet. On highly nonlinear S-boxes and their inability to thwart DPA attacks. Indocrypt 2005. LNCS vol. 3797, pp. 49–62, 2005.
3. P. Chodowiec and K. Gaj. Very compact FPGA implementation of the AES algorithm. CHES 2003, LNCS vol. 2279. pp. 319-333. Springer, 2003.
4. R. Lidl and H. Niederreiter. Introduction to Finite Fields and Their Applications. Cambridge. Cambridge Univercity Press. 1994.
5. N. Mentens, L. Batina, B. Preneel and I. Verbauwhede. A systematic evaluation of compact hardware implementations for the Rijndael S-box. CTRSA 2005. LNCS vol. 3376, pp. 323–333. Springer, 2005.
6. A. Moradi, A. Poschmann, S. Ling, C. Paar and H. Wang. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. Eurocrypt 2011. LNCS vol. 6632. pp. 69–88. Springer, 2011.
7. K. Nyberg. Differentially uniform mappings for cryptography. Advances in Cryptology, Eurocrypt'93. LNCS vol. 765. pp. 55–64. Springer-Verlag. 1994.
8. C. Paar. Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields. PhD thesis, Institute for Experimental Mathematics, University of Essen, Germany, 1994.
9. A. Rudra, P. K. Dubey, C. S. Jutla, V. Kumar, J. R. Rao, and P. Rohatgi. Efficient Rijndael encryption implementation with composite field arithmetic. In CHES, 2001, LNCS vol. 2162. pp. 171-184. Springer, 2001.
10. A. Satoh, S. Morioka, K. Takano, and Seiji Munetoh. A compact Rijndael hardware architecture with S-box optimization. In Advances in Cryptology - ASIACRYPT 2001, LNCS vol. 2248, pages 239-254. Springer, 2001.
11. J. Wolkerstorfer, E. Oswald and M. Lamberger. An ASIC implementation of the AES Sboxes. In CT-RSA, LNCS vol. 2271. pp. 67-78. Springer, 2002.

4 Appendix

4.1 Notes on Gate Equivalent

We have not done the actual implementation on ASIC as we don't have those tools. Instead, we have used Xilinx 7i FPGA to check the hardware. Gate equivalents for various hardware primitives that are used in this paper for estimation are given in the following Table. These figures are mainly taken from the thesis of Poschman (http://eprint.iacr.org/2009/516.pdf) to have a fare comparison with PRESENT. Note that, NAND8 does not exist in those libraries. But ATL 60 and ATLS60 series datasheet shows that a NAND8 gate needs 3.5 times the site count of a NAND2 gate. The datasheet can be found in the following link: http://www.datasheetcatalog.org/datasheet/atmel/DOC0388.PDF. So, I think we can safely assume that if the support of NAND8 gate is provided in the library used in PRESENT, the gate equivalent count will be 4. If the reader is not convinced with that assumption, the error margin is really less. We have only three NAND8 gates in the compact circuit. In the worst case, the NAND8 gate will require 7 GE by combining 2-input NAND gates. In that case, the gate equivalent count of compact hardware mode will be 138+3(7-4)=147.

Table 3. Gate Vs Gate Equivalent Count

Gate	GE	Gate	GE	Gate	GE	Gate	GE
NOT	0.67	NAND, NOR	1	2:1 MUX	2.33	NAND8	4
XOR	2.67	AND, OR	1.33	2-input FF	6		

4.2 A Note on Prior Art on LFSR Based Multiplicative Inverse

The possibility of generation of multiplicative inverse using LFSR is existing ever since the LFSR was invented. The novelty of this work is the hardware efficient usage of it which is simple but unpublished using excellent engineering techniques. Even though this work was done independently and we searched exhaustively later, there could still be a possibility that LFSR has been used in some fashion inside some other literature. One such work was found in: http://www.kemt.fei.tuke.sk/personal/drutarovsky/publications/fpl2009.pdf.

The above paper uses two LFSRs and runs them in both forward and reverse directions, performs a matrix multiplication after the LFSR transformations and then caches both the LFSRs outputs using the LUTs to get the AES S-box. While this can be a good strategy for FPGA, it is very clear that such an approach would require much bigger ASIC gate equivalent count (two different LFSRs flip-flops and XORs, one 8-bit counter requiring one more LFSR flip-flops and XORs, LUT where boolean equations would take a large hardware, XORs for two matrix multiplications, the control logic and Muxes) than both Satoh's and Canright's approach. Clearly, the approach taken in *Halka* is much more optimized with much more efficient algorithm for implementation. It uses a single LFSR flip-flops and a small control logic; no matrix multiplications or LUTs. Using a different initial seed, this also gets a free linear transformation for *Halka* S-box. The algorithm is also different here as it cleverly loads the S-box

input as the initial seed and runs it till the LFSR state becomes the initial seed and then running it on reverse direction till the number of cycles is 255. The contribution of this paper over the paper mentioned above should now be quite obvious for the LFSR based algorithms.

4.3 Per S-box Gate Equivalent Count for AES or in a Larger Cipher

Here we show the per S-box Gate Equivalent Count if this S-box were adopted for AES when the S-boxes are implemented in parallel. This is applicable for any other new cipher that plans to use 8-bit S-boxes. It was shown in Section 5 that for each S-box the hardware requirement sans the counter is 76.67 GE. The counter requires a hardware of 61.33 GE that can be reused across the S-boxes. Now, the storage of the each data state requires a two-input flip-flop with gate equivalent count 6. However, the S-box flip-flops can be reused to keep the data state. In other words, the data state flip-flops can be reused for the S-box. The initialization of the data state flip-flops (i.e. mux/second input of the 2-input flip-flops) can also be common with the S-box. So we can subtract the flip-flop hardware requirement of the S-box. However, we need to add a 2:1 mux for loading the data state in various rounds. This makes the hardware requirement of the S-box sans the counter as $76.67 - 8 \times 6 + 8 \times 2.33 = 47.31$. To this, we need to add the share of each S-box for the counter. For AES-128, there are 16 S-boxes (excluding the key-scheduling part). Hence, the share of the counter per S-box is 61.33/16=3.83. Thus, the hardware requirement per S-box is 47.31+3.83=51.14 GE.

For AES-256, the hardware requirement per S-box is 47.31+61.33/32=49.22.