A reference implementation of the AES S-Box

Brian Degnan, and Gregory Durgin

***Abstract*—The Advanced Encryption Standard (AES) is a specification for the encryption of electronic data. The substitution-permutation network architecture of AES uses the irreducible polynomial of $x^8 + x^4 + x^3 + x + 1$ that provides a non-linear permutations for the cipher. In this work, we describe the complete circuits and reductions required to implement a non-lookup table based S-Box in hardware for the AES irreducible polynomial in order to create a reference for comparisons. The verification script is written in the BASH shell so that they can be easily modified for integration to different simulation and verification tools.**

***Index Terms*—AES, block cipher, simulation, RFID, IoT**

Motivation

The Advanced Encryption Standard (AES) is a block cipher based on a substitution-permutation network [1]. Although the AES cipher has been approved for over 20 years, we were unable to find a hardware implementation reference that was suitable for comparisons against lightweight ciphers. This document attempts a minimal "nothing up the sleeves" transistor implementation of the non-linear component of AES, the S-Box. This work attempts to be the concisely written culmination of many other AES implementations. There is very little, if any, novel content in this document; however, as we were unable to explicitly find the information required for a research, and this document was created. Our focus of research includes power constrained devices, and the bulk of AES implementations use lookup tables for the Galois Field logic substitutions, or do not clearly explain the circuit implementation [2]–[4]. This work attempts to be a complete description and verification for AES targeting a custom semiconductor implementation that does not use lookup tables. The work is based off of the optimized S-Box by the AES authors [5] combined with an approach alluded to by Satoh et al [6]. This work extends previous work by complete mathematical descriptions, circuit logic implementations, and logic verification scripts.

I. AES

The AES algorithm is a block cipher that uses a 128, 192, or 256 bit keys to operation on a block of 128 bits of data. The shorthand notation for these configurations are AES128/128, AES128/192 and AES128/256 and the algorithm uses n rounds that are 10, 12, and 14 respectively. The AES algorithm is not the focus of this work, but the S-Box that underpins the non-linearity that is used in the cipher, which is the irreducible polynomial of $x^8 + x^4 + x^3 + x + 1$.

A. Review GF(2) Polynomials

As a review of GF(2) polynomials used in this work, GF(2) polynomials represent rings of numbers. As an example, the mapping of coefficients to numbers for $x^2 + x + 1$ is given by the following table:

dec	bin	polynomial
0	000	0
1	001	1
2	010	x
3	011	$x + 1$
4	100	x^2
5	101	$x^2 + 1$
6	110	$x^2 + x$
7	111	$x^2 + x + 1$

AES uses the same irreducible polynomial for every operation, which is $x^8 + x^4 + x^3 + x + 1$. This polynomial results in 9-bits, and is 0x11B in hexadecimal, or 100011011 in binary. You will notice that the 9-bits results in one bit more than you have in an 8-bit byte, and this allows the modulus to be the XOR of 0x1b.

B. Special Properties of GF(2) Polynomials

The logic reductions used in this work are based GF(2) polynomials. The computations on $GF(2^8)$ are done by reducing the field to a lower order composite fields of

$$\begin{aligned} GF(2^2) &\rightarrow GF(2) && : x^2 + x + 1 \\ GF((2^2)^2) &\rightarrow GF(2^2) && : x^2 + x + \varphi \\ GF(((2^2)^2)^2) &\rightarrow GF((2^2)^2) && : x^2 + x + \lambda \end{aligned} \tag{1}$$

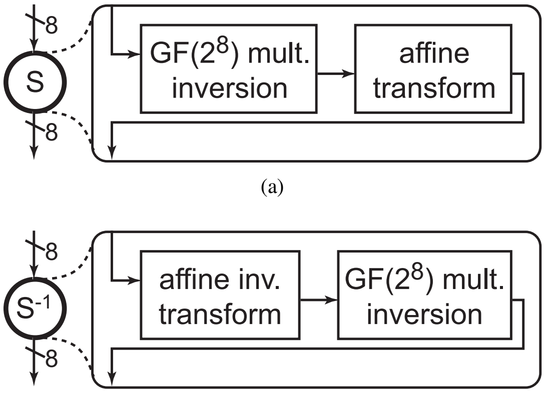
where $\varphi = \{10\}_2$ and $\lambda = \{1100\}_2$. As previously stated, $GF(2^2)$ polynomials can be decomposed to lower-order composite fields where $n_1 x + n_0$. Therefore, and binary number, k, can be split into $k_H x + k_L$. As an example, for $k = \{1001\}_2$, the value k is $k_H x + k_L$, resulting in $\{10\}_2 x + \{01\}_2$, which can be further reduced to $\{1\}_2 x + \{0\}_2$ and $\{0\}_2 x + \{1\}_2$ as high and low terms.

II. AESBASH.SH

In order to facilitate the simulation and representation of hardware that is described in this document, the AES S-Box has been implemented as a virtual logic implementation through the aesbash.sh script [7]. The functions in the script are bitwise, and represent the circuit architectures described in Section III. The script was designed and used under BASH version 3.2.57. The functions are described in Table I and are internally constructed as logic would be in a hardware implementation. For this reason, a multiply of two, 4-bit words is implemented as a single 8-bit input because this

B. Degnan (https://orcid.org/0000-0002-6567-6998) and G. Durgin are with the Department of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, 30332 USA e-mail: (see http://www.propagation.gatech.edu/).

This document was compiled on February 6, 2020.



(a)

(b)

Fig. 1: An illustration of the S-Box architecture is presented. (a) is the Subbyte calculation used in encryption that consists of a multiplicative inversion to an affine transform. (b) is the InvSubbyte calculation used in decryption that consists of an inverse affine transform to the multiplicative inversion module.

format is easily transcribed to SPICE modeling conventions. The archive, [7], also contain test and verification scripts that generate all of the tables in this work.

TABLE I: Script functions that emulate hardware blocks

function	args	bit width	description
aes_affine	1	8	calculate affine transform
aes_affineinv	1	8	inverse affine transform
aes_invGF24	1	4	multiplicative inverse of input
aes_isomorphic	1	8	isomorphic transform
aes_isomorphicinv	1	8	inverse isomorphic transform
aes_lambdamultiply	1	4	multiplication by $\lambda = \{1100\}$
aes_multGF2	1	4	mult. of 2, 2-bit numbers
aes_multGF24	1	8	mult. of 2, 4-bit numbers
aes_multiplicativeinversion	1	8	multiplicative inversion
aes_phimultiply	1	2	mult. by constant $\varphi = \{10\}$
aes_squarerGF24	1	4	the square of the input
aes_subbyte	1	8	subbyte calculation
aes_subbyteinv	1	8	inverse subbyte calculation
aesbash_verdep	0	0	check and download deps.
helplatex	0	0	autogenerate this table

III. S-Box construction

The AES S-Box is based on a non-linear boolean function that replaces an element of a finite field with its modular multiplicative inverse. Thus, $x \rightarrow x^{-1}$, where x represents an element of the finite field and x^{-1} denotes its multiplicative inverse in the field. As the S-Box is invertible, there are two modes of operation, Subbyte and InvSubbyte. In the Subbyte case, the data is run through a multiplicative inversion and then an affine transform. To invert the result, the data is run through an affine transform inversion, and then the same multiplicative inversion mathematics as the Subbyte. The software approach to inverse fields is often to use a lookup table, which is not necessarily the best implementation in hardware. An efficient hardware construction for calculating the multiplicative inverse in $GF\left(2^8\right)$ was given by Rijmen in [5], and this work was improved upon by Satoh et al. in [6]. The work described here is a complete implementation of the work in [6] with explanations and verification scripts.

A. SubByte

The SubByte is computed by calculating the Multiplicative Inversion in $GF(2^8)$, described in Section III-E, and then an affine transformation, Af, described in Section III-B, where an 8-bit input becomes an 8-bit output that is then passed to the multiplicative inversion module.

TABLE II: The pre-calculated byte substitution.

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

The pre-calculated values for the SubByte function are reported in Table II, and this output was automatically generated by the script aestest_subbytetable.sh, which iterates through every input. A single input can be evaluated by

```
#!/bin/sh
source aesbash.sh
aes_subbyte "00100000"
```

where "00100000" is the function input, resulting in the output of "10110111".

B. Affine Transformation

The affine transform is defined by:

$$Af(a) = \begin{pmatrix} 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \\ 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \end{pmatrix} \times \begin{pmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} \qquad (2)$$

In equation (2), the input bit a_x is mapped to the affine transform. This results in

$$Af(a) = \begin{pmatrix} af_7 \\ af_6 \\ af_5 \\ af_4 \\ af_3 \\ af_2 \\ af_1 \\ af_0 \end{pmatrix} = \begin{pmatrix} a_7 \oplus a_6 \oplus a_5 \oplus a_4 \oplus a_3 \oplus 0 \\ a_6 \oplus a_5 \oplus a_4 \oplus a_3 \oplus a_2 \oplus 1 \\ a_5 \oplus a_4 \oplus a_3 \oplus a_2 \oplus a_1 \oplus 1 \\ a_4 \oplus a_3 \oplus a_2 \oplus a_1 \oplus a_0 \oplus 0 \\ a_7 \oplus a_3 \oplus a_2 \oplus a_1 \oplus a_0 \oplus 0 \\ a_7 \oplus a_6 \oplus a_2 \oplus a_1 \oplus a_0 \oplus 0 \\ a_7 \oplus a_6 \oplus a_5 \oplus a_1 \oplus a_0 \oplus 1 \\ a_7 \oplus a_6 \oplus a_5 \oplus a_4 \oplus a_0 \oplus 1 \end{pmatrix}, \qquad (3)$$

where each "1" in the transform field corresponds to an XOR, which results in the logic shown in equation (3).

TABLE III: The pre-calculated affine transform logic results.

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	5d	42	1f	00	21	3e	9b	84	a5	ba	e7	f8	d9	c6
10	92	8d	ac	b3	ee	f1	d0	cf	6a	75	54	4b	16	09	28	37
20	80	9f	be	a1	fc	e3	c2	dd	78	67	46	59	04	1b	3a	25
30	71	6e	4f	50	0d	12	33	2c	89	96	b7	a8	f5	ea	cb	d4
40	a4	bb	9a	85	d8	c7	e6	f9	5c	43	62	7d	20	3f	1e	01
50	55	4a	6b	74	29	36	17	08	ad	b2	93	8c	d1	ce	ef	f0
60	47	58	79	66	3b	24	05	1a	bf	a0	81	9e	c3	dc	fd	e2
70	b6	a9	88	97	ca	d5	f4	eb	4e	51	70	6f	32	2d	0c	13
80	ec	f3	d2	cd	90	8f	ae	b1	14	0b	2a	35	68	77	56	49
90	1d	02	23	3c	61	7e	5f	40	e5	fa	db	c4	99	86	a7	b8
a0	0f	10	31	2e	73	6c	4d	52	f7	e8	c9	d6	8b	94	b5	aa
b0	fe	e1	c0	df	82	9d	bc	a3	06	19	38	27	7a	65	44	5b
c0	2b	34	15	0a	57	48	69	76	d3	cc	ed	f2	af	b0	91	8e
d0	da	c5	e4	fb	a6	b9	98	87	22	3d	1c	03	5e	41	60	7f
e0	c8	d7	f6	e9	b4	ab	8a	95	30	2f	0e	11	4c	53	72	6d
f0	39	26	07	18	45	5a	7b	64	c1	de	ff	e0	bd	a2	83	9c

TABLE IV: The pre-calculated byte substitution inversion.

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
10	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
20	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
30	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
40	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
50	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
60	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
70	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
80	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
90	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a0	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b0	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
c0	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d0	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e0	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
f0	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

The pre-calculated values for the affine transform are reported in Table III, and this output was automatically generated by the script aestest_affine.sh, which iterates through every input. A single input can be evaluated by

```
#!/bin/sh
source aesbash.sh
aes_affine "01000010"
```

where “01000010” is the function input, resulting in the output of “10011010”.

C. InvSubByte

The InvSubByte is computed by calculating the inverse affine transformation, Afi, described in Section III-D and then the Multiplicative Inversion in $GF(2^8)$ as described in Section III-E, where an 8-bit input becomes an 8-bit output.

The pre-calculated values for the InvSubByte function are reported in Table IV, and this output was automatically generated by the script aestest_subbyteinvtable.sh, which iterates through every input. A single input can be evaluated by

```
#!/bin/sh
source aesbash.sh
aes_subbyteinv "00100000"
```

where “00100000” is the function input, resulting in the output of “01010100”.

TABLE V: The pre-calculated inverse affine transform logic results.

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	05	4f	91	db	2c	66	b8	f2	57	1d	c3	89	7e	34	ea	a0
10	a1	eb	35	7f	88	c2	1c	56	f3	b9	67	2d	da	90	4e	04
20	4c	06	d8	92	65	2f	f1	bb	1e	54	8a	c0	37	7d	a3	e9
30	e8	a2	7c	36	c1	8b	55	1f	ba	f0	2e	64	93	d9	07	4d
40	97	dd	03	49	be	f4	2a	60	c5	8f	51	1b	ec	a6	78	32
50	33	79	a7	ed	1a	50	8e	c4	61	2b	f5	bf	48	02	dc	96
60	de	94	4a	00	f7	bd	63	29	8c	c6	18	52	a5	ef	31	7b
70	7a	30	ee	a4	53	19	c7	8d	28	62	bc	f6	01	4b	95	df
80	20	6a	b4	fe	09	43	9d	d7	72	38	e6	ac	5b	11	cf	85
90	84	ce	10	5a	ad	e7	39	73	d6	9c	42	08	ff	b5	6b	21
a0	69	23	fd	b7	40	0a	d4	9e	3b	71	af	e5	12	58	86	cc
b0	cd	87	59	13	e4	ae	70	3a	9f	d5	0b	41	b6	fc	22	68
c0	b2	f8	26	6c	9b	d1	0f	45	e0	aa	74	3e	c9	83	5d	17
d0	16	5c	82	c8	3f	75	ab	e1	44	0e	d0	9a	6d	27	f9	b3
e0	fb	b1	6f	25	d2	98	46	0c	a9	e3	3d	77	80	ca	14	5e
f0	5f	15	cb	81	76	3c	e2	a8	0d	47	99	d3	24	6e	b0	fa

D. Inverse Affine

The inverse affine transform is defined by:

$$Afi(a) = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} \tag{4}$$

In equation (4), the input bit a_x is mapped to the affine transform. This results in

$$Afi(a) = \begin{pmatrix} afi_7 \\ afi_6 \\ afi_5 \\ afi_4 \\ afi_3 \\ afi_2 \\ afi_1 \\ afi_0 \end{pmatrix} = \begin{pmatrix} a_6 \oplus a_4 \oplus a_1 \oplus 0 \\ a_5 \oplus a_3 \oplus a_0 \oplus 0 \\ a_7 \oplus a_4 \oplus a_2 \oplus 0 \\ a_6 \oplus a_3 \oplus a_1 \oplus 0 \\ a_5 \oplus a_2 \oplus a_0 \oplus 0 \\ a_7 \oplus a_4 \oplus a_1 \oplus 1 \\ a_6 \oplus a_3 \oplus a_0 \oplus 0 \\ a_7 \oplus a_5 \oplus a_2 \oplus 1 \end{pmatrix}, \tag{5}$$

where each “1” in the transform field corresponds to an XOR, which results in the logic shown in equation (5).

The pre-calculated values for the inverse affine transform are reported in Table V, and this output was automatically generated by the script aestest_affineinv.sh, which iterates through every input. A single input can be evaluated by

```
#!/bin/sh
source aesbash.sh
aes_affineinv "01000010"
```

where “01000010” is the function input, resulting in the output of “00000011”.

E. Multiplicative Inversion Module

The multiplicative inversion module is implemented as described by Satoh et al. [6], which takes the approach of several 2-degree extensions under bias instead of explicitly applying an 8-degree extension field. In this $GF(2^8)$ field, an element may be represented as $n_1x + n_0$, where n_1 is the

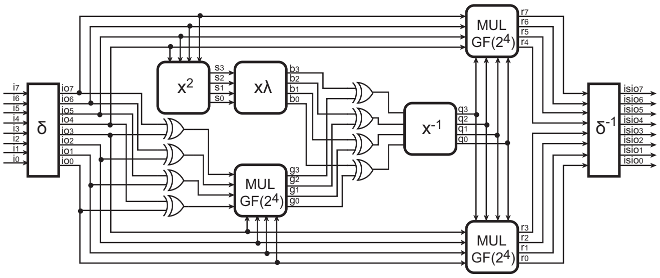


Fig. 2: The illustration depicts the logic that is used to implement the multiplicative inversion in circuits and the aesbash.sh verification code.

most significant nibble and n_0 is the least significant. The multiplicative inverse can be computed using the equation,

$$\begin{aligned}(n_1x+n_0)^{-1} &= n_1\left(n_1^2B+n_1n_0A+n_0^2\right)^{-1}x \\ &\quad+(n_0+n_1A)\left(n_1^2B+n_1n_0A+n_0^2\right)^{-1} \qquad (6)\end{aligned}$$

This equation can then be reduced because any polynomial can be represented as n_1x+n_0 with the irreducible polynomial of x^2+Ax+B. By selecting $A=1$ and $B=\lambda$, the irreducible polynomial becomes $x^2+x+\lambda$, and allows (6) to be reduced to

$$\begin{aligned}(n_1x+n_0)^{-1} &= n_1\left(n_1^2\lambda+n_0\left(n_1+n_0\right)\right)^{-1}x \\ &\quad+(n_0+n_1)\left(n_1^2\lambda+n_0\left(n_1+n_0\right)\right)^{-1}. \qquad (7)\end{aligned}$$

In equation (7), the arithmetic operations are the addition, multiplication, squaring and a multiplicative inversion on a $GF(2^4)$ field. The implementation result is the circuit illustrated in Figure 2; however, the computation of the multiplicative inverse cannot be directly applied to an element based on $GF(2^8)$ without first mapping it to an isomorphic function, δ. The output is then remapped after the multiplicative inversion by the inverse isomorphic function, δ^{-1}. The verification logic for the Multiplicative Inversion Module is reported in Table VI, and this output was automatically generated by the script aestest_multiplicativeinversion.sh, which iterates through every input. A single input can be evaluated by

```
#!/bin/sh
source aesbash.sh
aes_multiplicativeinversion "00000100"
```

where "00000100" in the example above is the binary input representing 0x04.

F. Isomorphic Transform

The isomorphic mapping exists to make the mathematics more efficient in the circuit sense. AES involves arithmetic on $GF(2^8)$ elements, and from the hardware perspective, the naive approach is to use lookup tables. The mapping of the operations into a composite field via an isomorphic transform allows for a decreased complexity in circuit logic at the cost of the isomorphic transform circuits.

1) Isomorphic Mapping Module: In Figure 2, the inputs to the isomorphic unit are labeled as i_x, where $_x$ is the bit line number. The outputs of the isomorphic unit are labelled as io.

TABLE VI: The pre-calculated multiplicative inversion module logic results

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	00	01	8d	f6	cb	52	7b	d1	e8	4f	29	c0	b0	e1	e5	c7
10	74	b4	aa	4b	99	2b	60	5f	58	3f	fd	cc	ff	40	ee	b2
20	3a	6e	5a	f1	55	4d	a8	c9	c1	0a	98	15	30	44	a2	c2
30	2c	45	92	6c	f3	39	66	42	f2	35	20	6f	77	bb	59	19
40	1d	fe	37	67	2d	31	f5	69	a7	64	ab	13	54	25	e9	09
50	ed	5c	05	ca	4c	24	87	bf	18	3e	22	f0	51	ec	61	17
60	16	5e	af	d3	49	a6	36	43	f4	47	91	df	33	93	21	3b
70	79	b7	97	85	10	b5	ba	3c	b6	70	d0	06	a1	fa	81	82
80	83	7e	7f	80	96	73	be	56	9b	9e	95	d9	f7	02	b9	a4
90	de	6a	32	6d	d8	8a	84	72	2a	14	9f	88	f9	dc	89	9a
a0	fb	7c	2e	c3	8f	b8	65	48	26	c8	12	4a	ce	e7	d2	62
b0	0c	e0	1f	ef	11	75	78	71	a5	8e	76	3d	bd	bc	86	57
c0	0b	28	2f	a3	da	d4	e4	0f	a9	27	53	04	1b	fc	ac	e6
d0	7a	07	ae	63	c5	db	e2	ea	94	8b	c4	d5	9d	f8	90	6b
e0	b1	0d	d6	eb	c6	0e	cf	ad	08	4e	d7	e3	5d	50	1e	b3
f0	5b	23	38	34	68	46	03	8c	dd	9c	7d	a0	cd	1a	41	1c

TABLE VII: The pre-calculated isomorphic mapping module logic verification

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	00	01	5f	5e	7c	7d	23	22	74	75	2b	2a	08	09	57	56
10	46	47	19	18	3a	3b	65	64	32	33	6d	6c	4e	4f	11	10
20	b0	b1	ef	ee	cc	cd	93	92	c4	c5	9b	9a	b8	b9	e7	e6
30	f6	f7	a9	a8	8a	8b	d5	d4	82	83	dd	dc	fe	ff	a1	a0
40	4b	4a	14	15	37	36	68	69	3f	3e	60	61	43	42	1c	1d
50	0d	0c	52	53	71	70	2e	2f	79	78	26	27	05	04	5a	5b
60	fb	fa	a4	a5	87	86	d8	d9	8f	8e	d0	d1	f3	f2	ac	ad
70	bd	bc	e2	e3	c1	c0	9e	9f	c9	c8	96	97	b5	b4	ea	eb
80	fc	fd	a3	a2	80	81	df	de	88	89	d7	d6	f4	f5	ab	aa
90	ba	bb	e5	e4	c6	c7	99	98	ce	cf	91	90	b2	b3	ed	ec
a0	4c	4d	13	12	30	31	6f	6e	38	39	67	66	44	45	1b	1a
b0	0a	0b	55	54	76	77	29	28	7e	7f	21	20	02	03	5d	5c
c0	b7	b6	e8	e9	cb	ca	94	95	c3	c2	9c	9d	bf	be	e0	e1
d0	f1	f0	ae	af	8d	8c	d2	d3	85	84	da	db	f9	f8	a6	a7
e0	07	06	58	59	7b	7a	24	25	73	72	2c	2d	0f	0e	50	51
f0	41	40	1e	1f	3d	3c	62	63	35	34	6a	6b	49	48	16	17

$$\delta \times i = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} i_7 \\ i_6 \\ i_5 \\ i_4 \\ i_3 \\ i_2 \\ i_1 \\ i_0 \end{pmatrix} \qquad (8)$$

In equation (8), the input bit i_x is mapped to the isomorphic field described by δ, thus, $io=\delta\times i$. This results in

$$\delta \times i = \begin{pmatrix} io_7 \\ io_6 \\ io_5 \\ io_4 \\ io_3 \\ io_2 \\ io_1 \\ io_0 \end{pmatrix} = \begin{pmatrix} i_7 \oplus i_5 \\ i_7 \oplus i_6 \oplus i_4 \oplus i_3 \oplus i_2 \\ i_7 \oplus i_5 \oplus i_3 \oplus i_2 \\ i_7 \oplus i_5 \oplus i_3 \oplus i_2 \oplus i_1 \\ i_7 \oplus i_6 \oplus i_2 \oplus i_1 \\ i_7 \oplus i_4 \oplus i_3 \oplus i_2 \oplus i_1 \\ i_6 \oplus i_4 \oplus i_1 \\ i_6 \oplus i_1 \oplus i_0 \end{pmatrix}, \qquad (9)$$

where each "1" in the isomorphic field δ corresponds to an XOR, which results in the logic shown in equation (9). The verification logic for the Isomorphic Mapping Module is reported in Table VII, and this output was automatically generated by the script aestest_isomorphic.sh, which iterates through every input. A single input can be evaluated by

TABLE VIII: The pre-calculated inverse isomorphic mapping module logic verification

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	00	01	bc	bd	5d	5c	e1	e0	0c	0d	b0	b1	51	50	ed	ec
10	1f	1e	a3	a2	42	43	fe	ff	13	12	af	ae	4e	4f	f2	f3
20	bb	ba	07	06	e6	e7	5a	5b	b7	b6	0b	0a	ea	eb	56	57
30	a4	a5	18	19	f9	f8	45	44	a8	a9	14	15	f5	f4	49	48
40	f1	f0	4d	4c	ac	ad	10	11	fd	fc	41	40	a0	a1	1c	1d
50	ee	ef	52	53	b3	b2	0f	0e	e2	e3	5e	5f	bf	be	03	02
60	4a	4b	f6	f7	17	16	ab	aa	46	47	fa	fb	1b	1a	a7	a6
70	55	54	e9	e8	08	09	b4	b5	59	58	e5	e4	04	05	b8	b9
80	84	85	38	39	d9	d8	65	64	88	89	34	35	d5	d4	69	68
90	9b	9a	27	26	c6	c7	7a	7b	97	96	2b	2a	ca	cb	76	77
a0	3f	3e	83	82	62	63	de	df	33	32	8f	8e	6e	6f	d2	d3
b0	20	21	9c	9d	7d	7c	c1	c0	2c	2d	90	91	71	70	cd	cc
c0	75	74	c9	c8	28	29	94	95	79	78	c5	c4	24	25	98	99
d0	6a	6b	d6	d7	37	36	8b	8a	66	67	da	db	3b	3a	87	86
e0	ce	cf	72	73	93	92	2f	2e	c2	c3	7e	7f	9f	9e	23	22
f0	d1	d0	6d	6c	8c	8d	30	31	dd	dc	61	60	80	81	3c	3d

```
#!/bin/sh
source aesbash.sh
aes_isomorphic "00000100"
```

where “00000100” in the example above is the binary input representing 0x04.

2) Inverse Isomorphic Module: In Figure 2, the inputs to the isomorphic inversion unit are labeled as r_x, where $_x$ is the bit line number. The outputs of the isomorphic unit are labelled as $isio$.

$$\delta^{-1} \times r = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} r_7 \\ r_6 \\ r_5 \\ r_4 \\ r_3 \\ r_2 \\ r_1 \\ r_0 \end{pmatrix} \qquad (10)$$

In equation (10), the input bit r_x is mapped to the isomorphic inversion field described by δ^{-1}, thus, $io = \delta^{-1} \times i$. This results in

$$\delta^{-1} \times r = \begin{pmatrix} isio_7 \\ isio_6 \\ isio_5 \\ isio_4 \\ isio_3 \\ isio_2 \\ isio_1 \\ isio_0 \end{pmatrix} = \begin{pmatrix} r_7 \oplus r_6 \oplus r_5 \oplus r_1 \\ r_6 \oplus r_2 \\ r_6 \oplus r_5 \oplus r_1 \\ r_6 \oplus r_5 \oplus r_4 \oplus r_2 \oplus r_1 \\ r_5 \oplus r_4 \oplus r_3 \oplus r_2 \oplus r_1 \\ r_7 \oplus r_4 \oplus r_3 \oplus r_2 \oplus r_1 \\ r_5 \oplus r_4 \\ r_6 \oplus r_5 \oplus r_4 \oplus r_2 \oplus r_0 \end{pmatrix} \qquad (11)$$

where each “1” in the isomorphic field δ^{-1} corresponds to an XOR, which results in the logic shown in equation (11).

The verification logic for the Inverse Isomorphic Mapping Module is reported in Table VIII, and this output was automatically generated by the script aestest_invisomorphic.sh, which iterates through every input. A single input can be evaluated by

```
#!/bin/sh
source aesbash.sh
aes_isomorphicinv "00000100"
```

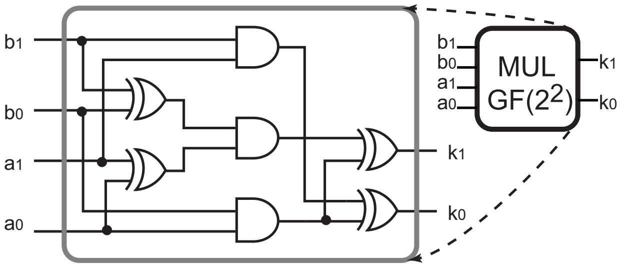


Fig. 3: The illustration reports the implementation logic structure of the $GF(2^2)$ multiplication module.

TABLE IX: The pre-calculated $GF(2^2)$ Multiplication Module logic verification

	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

where “00000100” in the example above is the binary input representing 0x04.

G. $GF(2^4)$ Addition

Addition in a Galois Field, $GF(2)$, is simply the XOR between elements.

H. $GF(2^2)$ Multiplication

The $GF(2^2)$ Multiplication Module operates on two, 2-bit inputs and results in a 2-bit output. For the elements in $GF(2^2)$, let the output, k, be the product of inputs a and b, so that $k = ab$, where $k = \{k_1, k_0\}$, $a = \{a_1, a_0\}$ and $b = \{b_1, b_0\}$. Mapping the inputs into a $GF(2)$ polynomial results in

$$\begin{aligned} k &= k_1x + k_0 = (a_1a_0)(b_1b_0) \\ k &= k_1x + k_0 = (a_1x + a_0)(b_1x + b_0) \qquad (12) \\ k &= k_1x + k_0 = \left(a_1b_1x^2 + a_1b_0x + a_0b_1x + a_0b_0\right) \end{aligned}$$

which can be further reduced because $x^2 = x+1$. The resulting equation is

$$\begin{aligned} k = k_1x + k_0 &= a_1b_1(x+1) + a_1b_0x + a_0b_1x + a_0b_0 \\ &= (a_1b_1 + a_1b_0 + a_0b_1)x + (a_1b_1 + a_0b_0), \end{aligned} \qquad (13)$$

which can be implemented as AND and XOR logic. The logical representation of (13) is

$$k = \begin{cases} k_1 = a_1b_1 \oplus a_1b_0 \oplus a_0b_1 \\ k_0 = a_1b_1 \oplus a_0b_0 \end{cases} \qquad (14)$$

and this logic is implemented in Figure 3.

The verification of the logic for the $GF(2^2)$ Multiplication Module is reported in Table IX, and this output was automatically generated by the script aestest_multGF22.sh, which iterates through every input. The function expects as single 4-bit input which represents the two, 2-bit values being multiplied, as $a_1a_0b_1b_0$. This is due to a limitation in the bashbignumbers.sh dependency which has nibble boundaries. A single input can be evaluated by

```
#!/bin/sh
source aesbash.sh
aes_multGF2 "0111"
```

where “0111” in the example above is the binary input of $a = \{0, 1\}$ and $b = \{1, 1\}$ resulting in an output of $k = \{1, 1\}$.

I. $GF(2^2)$ Multiplication by constant

The multiplication by a constant, φ, is a special case of the $GF(2^2)$ Multiplication Module described in Section III-H. The constant $\varphi = \{1, 0\}$ is a fixed value on the “b” input to $GF(2^2)$ multiplication. Starting from equation (13), we can reduce the logic to

$$k = (a_1 + a_0)\,x + a_1, \tag{15}$$

which reduces to logic based on an XOR as

$$k = \begin{cases} k_1 = a_1 \oplus a_0 \\ k_0 = a_1 \end{cases}, \tag{16}$$

and test vector was not created for this module as it can be done by inspection.

J. $GF(2^4)$ Multiplication

The $GF(2^4)$ Multiplication Module operates on two, 4-bit inputs and results in a 4-bit output. For the elements in $GF(2^4)$, let the output, k, be the product of inputs a and b, so that $k = ab$, where $k = \{k_3, k_2, k_1, k_0\}$, $a = \{a_3, a_2, a_1, a_0\}$ and $b = \{b_3, b_2, b_1, b_0\}$. Mapping the inputs into a $GF(2)$ polynomial results in

$$\begin{aligned} k &= \underbrace{k_3k_2}_{k_H}\underbrace{k_1k_0}_{k_L} = \Big(\underbrace{a_3a_2}_{a_H}\underbrace{a_1a_0}_{a_L}\Big)\Big(\underbrace{b_3b_2}_{b_H}\underbrace{b_1b_0}_{b_L}\Big) \\ &= k_Hx + k_L = (a_Hx + a_L)\,(b_Hx + b_L) \\ &= (a_Hb_H)\,x^2 + (a_Hb_L + a_Lb_H)\,x + a_Lb_L \end{aligned} \tag{17}$$

to which $x^2 = x + \varphi$ can be substituted. The φ constant is defined in Section III-I. The resulting equation is

$$\begin{aligned} k &= (a_Hb_H)\,(x + \varphi) + (a_Hb_L + a_Lb_H)\,x + a_Lb_L \\ &= (a_Hb_H + a_Hb_L + a_Lb_H)\,x + a_Hb_H\varphi + a_Lb_L \end{aligned} \tag{18}$$

and this logic is implemented in the circuit illustrated in Figure 4. The verification logic for the $GF(2^4)$ Multiplication Module is reported in Table X, and this output was automatically generated by the script aestest_multGF24.sh, which iterates through every input. A single input can be evaluated by

```
#!/bin/sh
source aesbash.sh
aes_multGF24 "00100100"
```

where “00100100” in the example above is the binary input of $a = \{0, 0, 1, 0\}$ and $b = \{0, 1, 0, 0\}$ resulting in an output of $k = \{1, 0, 0, 0\}$.

TABLE X: The pre-calculated $GF(2^4)$ Multiplication Module logic verification

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
2	0	2	3	1	8	a	b	9	c	e	f	d	4	6	7	5
3	0	3	1	2	c	f	d	e	4	7	5	6	8	b	9	a
4	0	4	8	c	6	2	e	a	b	f	3	7	d	9	5	1
5	0	5	a	f	2	7	8	d	3	6	9	c	1	4	b	e
6	0	6	b	d	e	8	5	3	7	1	c	a	9	f	2	4
7	0	7	9	e	a	d	3	4	f	8	6	1	5	2	c	b
8	0	8	c	4	b	3	7	f	d	5	1	9	6	e	a	2
9	0	9	e	7	f	6	1	8	5	c	b	2	a	3	4	d
a	0	a	f	5	3	9	c	6	1	b	e	4	2	8	d	7
b	0	b	d	6	7	c	a	1	9	2	4	f	e	5	3	8
c	0	c	4	8	d	1	9	5	6	a	2	e	b	7	f	3
d	0	d	6	b	9	4	f	2	e	3	8	5	7	a	1	c
e	0	e	7	9	5	b	2	c	a	4	d	3	f	1	8	6
f	0	f	5	a	1	e	4	b	2	d	7	8	3	c	6	9

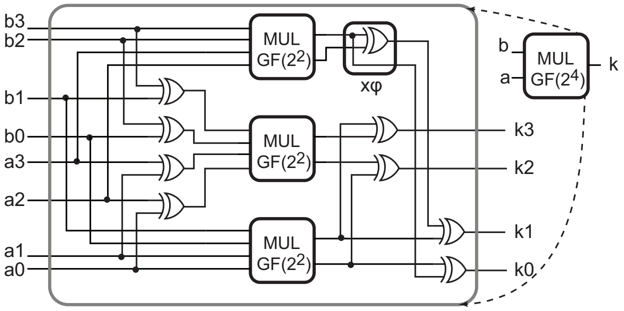


Fig. 4: The illustration reports the implementation logic structure of the $GF(2^4)$ multiplication module. Note the additional XOR on the most significant multiplier that creates the constant multiplication, $x\varphi$, in $GF(2^2)$ that is described in Section III-I.

K. $GF(2^4)$ Squaring Module

The $GF(2^4)$ Squaring Module operates one 4-bit input and results in a 4-bit output. For the elements in $GF(2^4)$, let the output, k, be the product of input a^2, so that $k = aa$, where $k = \{k_3, k_2, k_1, k_0\}$, and $a = \{a_3, a_2, a_1, a_0\}$. Mapping the inputs into a $GF(2)$ polynomial results in

$$\begin{aligned} k &= \underbrace{k_3k_2}_{k_H}\underbrace{k_1k_0}_{k_L} = \Big(\underbrace{a_3a_2}_{a_H}\underbrace{a_1a_0}_{a_L}\Big)^2 \\ &= k_Hx + k_L = (a_Hx + a_L)^2 \\ &= (a_Hb_H)\,x^2 + (a_Hb_L + a_Lb_H)\,x + a_Lb_L \\ &= {a_H}^2x^2 + {a_L}^2. \end{aligned} \tag{19}$$

The naive approach would be to simply put the same values into the a and b inputs of the $GF(2^4)$ Multiplication Module described in Section III-J; however, the circuit complexity can be decreased using the reductions described in equation (1). The x^2 term can be reduced by setting $x^2 = x + \varphi$, resulting in

$$\begin{aligned} k &= {a_H}^2x^2 + {a_L}^2 \\ &= {a_H}^2\,(x + \varphi) + {a_L}^2 \\ &= \underbrace{{a_H}^2x}_{k_H} + \underbrace{{a_H}^2\varphi + {a_L}^2}_{k_L} \end{aligned} \tag{20}$$

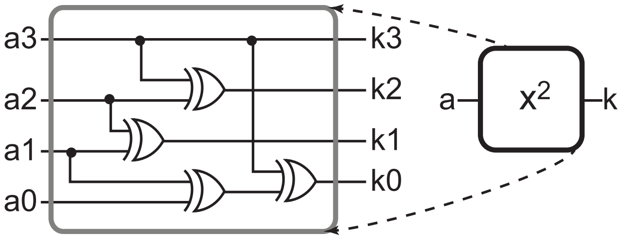


Fig. 5: The illustration reports the implementation logic structure of the $GF(2^4)$ squaring module.

TABLE XI: The pre-calculated $GF(2^4)$ Squarer logic verification

input	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
output	0	1	3	2	6	7	5	4	d	c	e	f	b	a	8	9

which is now in terms of $GF(2^2)$. Starting from the high term, k_H, and using the substitutions in (1) we can show that

$$\begin{aligned}
k_H &= a_H^2 = (a_3a_2)^2 \\
k_H &= a_H^2 = (a_3x + a_2)^2 \\
k_H &= a_3^2x^2 + a_3a_2x + a_3a_2x + a_2^2 = a_3x^2 + a_2 \qquad (21)\\
k_H &= a_3(x+1) + a_2 \\
k_H &= k_3x + k_2 = \underbrace{a_3x}_{k_3} + \underbrace{a_3 + a_2}_{k_2}.
\end{aligned}$$

Continuing the decomposition with the low term, k_L, where $\varphi = \{10\}$, results in

$$\begin{aligned}
k_L &= a_H^2\varphi + a_L^2 = (a_3a_2)^2\{10\} + (a_1a_0)^2 \\
k_L &= (a_3x + a_2)^2\left(\{1\}x + \{0\}\right) + (a_1x + a_0)^2 \\
k_L &= \left(a_3^2x^2 + a_3a_2x + a_2a_3x + a_2^2\right)x \\
&\quad + \left(a_1^2x^2 + a_1a_0x + a_0a_1x + a_0^2\right) \qquad (22)\\
k_L &= a_3x^3 + a_2x + a_1x^2 + a_0 \\
k_L &= a_3(1) + a_2x + a_1(x+1) + a_0 \\
k_L &= k_1x + k_0 = (a_2 + a_1)x + (a_3 + a_1 + a_0),
\end{aligned}$$

where the simplification is from $x^3 = x^2 + x = (x+1) + x = 1$. Combining (21) and (22) results in the final logic equation of

$$k = \begin{cases} k_3 = a_3 \\ k_2 = a_3 \oplus a_2 \\ k_1 = a_2 \oplus a_1 \\ k_0 = a_3 \oplus a_1 \oplus a_0 \end{cases} \qquad (23)$$

which can be implemented as 4 XOR gates. The logic is implemented in the circuit illustrated in Figure 5. The verification logic for the $GF(2^4)$ Squaring Module is reported in Table XI, and this output was automatically generated by the script aestest_sqrGF24.sh, which iterates through every input. A single input can be evaluated by

```
#!/bin/sh
source aesbash.sh
aes_squarerGF24 "0100"
```

where "0100" in the example above is the binary input of $a = \{0,1,0,0\}$ resulting in an output of $k = \{0,1,1,0\}$.

L. $GF(2^4)$ *Multiplication by constant*

The $GF(2^4)$ Multiplication by Constant Module operates one 4-bit input, a 4-bit constant and results in a 4-bit output. For the elements in $GF(2^4)$, let the output, k, be the product of input a^2, so that $k = a\lambda$, where $k = \{k_3, k_2, k_1, k_0\}$, $a = \{a_3, a_2, a_1, a_0\}$ and $\lambda = \{1100\}$. Mapping the inputs into a $GF(2)$ polynomial results in

$$\begin{aligned}
k &= \underbrace{k_3k_2}_{k_H}\underbrace{k_1k_0}_{k_L} = k_Hx + k_L \\
&= k_Hx + k_L = \big(\underbrace{a_3a_2}_{a_H}\underbrace{a_1a_0}_{a_L}\big)\big\{\underbrace{11}_{\lambda_H}\underbrace{00}_{\lambda_L}\big\} \\
&= (a_Hx + a_L)(\lambda_Hx + \lambda_L) \\
&= a_H\lambda_Hx^2 + a_L\lambda_Hx, \qquad (24)
\end{aligned}$$

where $\lambda_L = \{00\}$, which cancels out many of the terms. As with the squaring circuit, the naive approach would be to simply put the same values into the a and b inputs of the $GF(2^4)$ Multiplication Module described in Section III-J; however, the circuit complexity can be decreased using the reductions described in equation (1). The substitution of $x^2 = x + \varphi$ reduces equation (24) to

$$\begin{aligned}
k &= a_H\lambda_H(x + \varphi) + a_L\lambda_Hx \\
k &= \big(\underbrace{a_H\lambda_H + a_L\lambda_H}_{k_H}\big)x + \big(\underbrace{a_H\lambda_H\varphi}_{k_L}\big), \qquad (25)
\end{aligned}$$

where k_H and k_L can be further reduced. Starting with the k_H term in 25, the reduction follow as

$$\begin{aligned}
k_H =& k_3x + k_2 \\
=& a_H\lambda_H + a_L\lambda_H \\
=& (a_3a_2)\{11\} + (a_1a_0)\{11\} \\
=& (a_3x + a_2)(x+1) + (a_1x + a_0)(x+1) \\
=& (a_3)x^2 + (a_3 + a_2)x + a_2 \\
& + a_1x^2 + (a_1 + a_0)x + a_0 : x^2 = x + 1 \qquad (26)\\
=& a_3(x+1) + (a_3 + a_2)x + a_2 \\
& + a_1(x+1) + (a_1 + a_0)x + a_0 \\
=& (a_3 + a_3 + a_2 + a_1 + a_1 + a_0)x \\
& + (a_3 + a_2 + a_1 + a_0) \\
=& k_3x + k_2 = (a_2 + a_0)x + (a_3 + a_2 + a_1 + a_0).
\end{aligned}$$

The k_L term can be decomposed in a similar manner with (26) as

$$\begin{aligned}
k_L =& k_1x + k_0 \\
=& a_H\lambda_H\varphi \\
=& (a_3a_2)\{11\}\{10\} \\
=& (a_3x + a_2)(x+1)(x) \\
=& a_3x^3 + a_3x^2 + a_2x^2 + a_2x : x^2 = x + 1, x^3 = 1 \\
=& a_3(1) + a_3(x+1) + a_2(x+1) + a_2x \qquad (27)\\
=& (a_3 + a_2 + a_2)x + (a_3 + a_3 + a_2) \\
=& k_1x + k_0 = (a_3)x + (a_2).
\end{aligned}$$

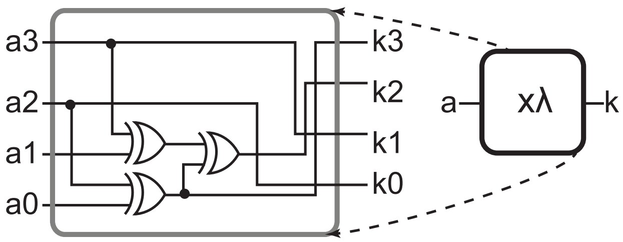


Fig. 6: The illustration reports the implementation logic structure of the $GF(2^4)$ constant multiplication module.

TABLE XII: The pre-calculated constant λ multiplicative logic verification

input	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
output	0	c	4	8	d	1	9	5	6	a	2	e	b	7	f	3

The equations (26) and (27) can then be used to create the logic description of

$$k = \begin{cases} k_3 = a_3 \oplus a_2 \\ k_2 = a_3 \oplus a_2 \oplus a_1 \oplus a_0 \\ k_1 = a_3 \\ k_0 = a_2 \end{cases} \tag{28}$$

which can be implemented as 3 XOR gates because $a_2 \oplus a_0$ occurs twice. The logic is implemented in the circuit illustrated in Figure 6. The verification logic for the $GF(2^4)$ Constant Multiplication Module is reported in Table XII, and this output was automatically generated by the script aestest_lambdaGF24.sh, which iterates through every input. A single input can be evaluated by

```
#!/bin/sh
source aesbash.sh
aes_lambdamultiply "0100"
```

where "0100" in the example above is the binary input of $a = \{0, 1, 0, 0\}$ resulting in an output of $k = \{1, 1, 0, 1\}$.

M. $GF(2^4)$ Inversion

The $GF(2^4)$ Inversion Module calculates the inversion of the input a, where $k = a^{-1}$. The calculated results in reported in Table XIII, and the logic was created by minimizing the Karnaugh Map resulting in

$$k = \begin{cases} k_3 = & a_3 \oplus a_3a_2a_1 \oplus a_3a_0 \oplus a_2 \\ k_2 = & a_3a_2a_1 \oplus a_3a_2a_0 \oplus a_3a_0 \oplus a_2 \\ & \oplus a_2a_1 \\ k_1 = & a_3 \oplus a_3a_2a_1 \oplus a_3a_1a_0 \oplus a_2 \\ & \oplus a_2a_2 \oplus a_1 \\ k_0 = & a_3a_2a_1 \oplus a_3a_2a_0 \oplus a_3a_1 \\ & a_3a_1a_0 \oplus a_3a_0 \oplus a_2 \oplus a_2a_1 \\ & a_2a_1a_0 \oplus a_1 \oplus a_0 \end{cases} \tag{29}$$

which represents the circuit logic in lieu of a lookup table. The verification logic for the $GF(2^4)$ Inversion Module is reported in Table XIII, and this output was automatically generated by the script aestest_invGF24.sh, which iterates through every input. A single input can be evaluated by

TABLE XIII: The pre-calculated $GF(2^4)$ inversion logic verification

a	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
k^{-1}	0	1	3	2	f	c	9	b	a	6	8	7	5	e	d	4

```
#!/bin/sh
source aesbash.sh
aes_invGF24 "0100"
```

where "0100" in the example above is the binary input of $a = \{0, 1, 0, 0\}$ resulting in an output of $k = \{1, 1, 1, 1\}$.

IV. Conclusion

The focus of this work was a reference S-Box for AES with a focus on the circuit implementation for the multiplicative inversion module. The complete mathematical justification has been presented, and is complimented a BASH script for logic verification that can be used for comparisons against outputs using SPICE.

Acknowledgment

This research was supported in part by an appointment to the Intelligence Community Postdoctoral Research Fellowship Program at the Georgia Institute of Technology, administered by Oak Ridge Institute for Science and Education through an interagency agreement between the U.S. Department of Energy and the Office of the Director of National Intelligence.

Brian Degnan would also like thank Ella Rose for her assistance in all things cryptography, Professor Gregory Durgin for his continued support, and Suigin Maeda for general thoughts in the field of mathematics.

References

[1] NIST, "197, Advanced Encryption Standard (AES), National Institute of Standards and Technology, US Department of Commerce, November 2001," *http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf*, 2001.

[2] 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 *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2001, pp. 171–184.

[3] Z. Wang, X. Zhang, S. Wang, Z. Hao, and Z. Zheng, "Application of the composite field in the design of an improved aes s-box based on inversion," in *The Third International Conference on Communications, Computation, Networks and Technologies*, 2014, pp. 23–29.

[4] S. Gueron and S. Mathew, "Hardware implementation of aes using area-optimal polynomials for composite-field representation gf (2^ 4)^ 2 of gf (2^ 8)," in *Computer Arithmetic (ARITH), 2016 IEEE 23nd Symposium on*. IEEE, 2016, pp. 112–117.

[5] V. Rijmen, "Efficient Implementation of the Rijndael S-box," *Katholieke Universiteit Leuven, Dept. ESAT. Belgium*, 2000.

[6] A. Satoh, S. Morioka, K. Takano, and S. Munetoh, "A compact rijndael hardware architecture with s-box optimization," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2001, pp. 239–254.

[7] B. Degnan, "aesbash.sh," *https://github.com/bpdegnan/aes/tree/master/aes-sbox/emulate*, 01 Nov 2017.