Algebraic Cryptanalysis of Advanced Encryption Standard Using Polynomial System Modeling

Stevanus Agustaf Wongso[1,2]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10, Bandung 40132, Indonesia
[1]*13524020@mahasiswa.itb.ac.id*, [2]*stevanus610@gmail.com*

***Abstract*—Advanced Encryption Standard (AES) is highly secure standard for encrypting data that operates on 128 bit blocks. This paper explores algebraic cryptanalysis techniques for AES by modeling the cipher operations as a system of polynomial equations over GF(2). By analyzing how the SubBytes, ShiftRows, MixColumns, and AddRoundKey were operated, we can represented algebraically and demonstrate the theoretical approach to solving the resulting equation system through linearization techniques.**

***Index Terms*—AES, algebraic cryptanalysis, polynomial equations, finite fields, cryptographic analysis**

I. Introduction

Advanced Encryption Standard (AES) is a symmetric encryption algorithm that has been widely used in modern data security. Since 2001, AES has been selected by NIST as a standard for encryption in many cryptographic applications. AES operates on 128-bit blocks and support key sizes of 128, 192, 256 bit. AES has been remain secured against classical attacks such as differential and linear cryptanalysis.

Algebraic cryptanalysis is a cryptographic attack method that represents a cipher as a system of linear equations and attempts to solve them to retrieve the secret key. The core idea is to express each cipher operations (like SubBytes, ShiftRows, MixColumns, and AddRoundKey) as a large system of equation where the key bits are unknown. This approach differs from conventional attacks such as differential and linear cryptanalysis.

In this paper, we demonstrate how each AES operation can be represent as polynomial equation and explore linearization technique for solving this system.

II. AES Structure and Operations

A. AES Overview

AES is a block cipher that encrypts data in fixed size (16 bytes). The algorithm takes 2 inputs, which is the plaintext and also the key. Depending on the key size (128/192/256), AES perform 10, 12 or 14 rounds of transformation. In this paper we will only use several rounds as demonstration. The encryption work by repeatedly transform the plaintext through multiple rounds. Each round applies several operations like SubBytes,ShiftRows, MixColumns, and AddRoundKey. Before the round start, AES will combine the plaintext with the initial encryption Key using XOR operation. This operation is called AddRoundKey. For the remaining round, it will perform the operation we mentioned before. And for the last round, it will skip the MixColumn Operation. For each round, the key will be expanded. It takes the orignal encryption key, then expands it into multiple round key. Each round key is derived from previous key with several transformation including XOR, rotating, substitution, etc, with constant. The strength of AES comes from these complex combination. SubBytes will provide non-linear substitution which make the relation between key and ciphertext more complex. ShiftRows and MixColumns provide Avalanche effect which mean changing one byte of input will change all byte of the result. All of this transformation will make cryptanalysis extremely difficult.

B. Data Representation

In AES, the data is processed in specific structures that are important to understand before exploring the operations.

1) The State: [1] The 128-bit input block is arranged into a 4×4 matrix of bytes called the State. Each byte in the State is denoted as $s[r, c]$ where r is the row index (0-3) and c is the column index (0-3). The State is the primary data structure that is transformed through each round of encryption. The first step of arranging is to copy the input array of bytes $in_0, in_1, ..., in_15$ to the state array s as follows :

$$s_{r,c} = in[r + 4c], \qquad 0 \leq r \leq 3,\ 0 \leq c \leq 3 \tag{1}$$

Then sequence of transformations are applied to the state array. Its final value is copied to the output array of bytes $out_1, out_2, ..., out_15$ as follows :

$$out[r + 4c] = s_{r,c}, \qquad 0 \leq r \leq 3,\ 0 \leq c \leq 3 \tag{2}$$

These steps are illustrated in Fig. 1.

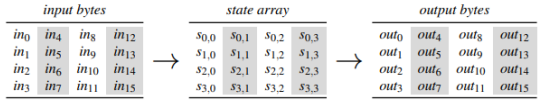

Fig. 1. State matrix represented as array of words. Source: [1]

2) Arrays of Words: [1] A word in AES is a sequence of four bytes. The state matrix can be interpreted as an array of four words, where each column represents one word. Using the notation from Fig. 1, the four columns of the state are denoted as v_0, v_1, v_2, v_3:

$$v_0 = \begin{pmatrix} s_{0,0} \\ s_{1,0} \\ s_{2,0} \\ s_{3,0} \end{pmatrix}, \quad v_1 = \begin{pmatrix} s_{0,1} \\ s_{1,1} \\ s_{2,1} \\ s_{3,1} \end{pmatrix},$$

$$v_2 = \begin{pmatrix} s_{0,2} \\ s_{1,2} \\ s_{2,2} \\ s_{3,2} \end{pmatrix}, \quad v_3 = \begin{pmatrix} s_{0,3} \\ s_{1,3} \\ s_{2,3} \\ s_{3,3} \end{pmatrix}$$

The column index c of the state becomes the word index, and the row index r becomes the byte index within each word. This word-based representation is particularly useful for the MixColumns operation and the key expansion process.

Given a one-dimensional array u of words, $u[i]$ denotes the word indexed by i, and the sequence of four words $u[i], u[i+1], u[i+2], u[i+3]$ is denoted by $u[i..i+3]$.

C. Round Operations

The general function to execute AES is denoted by $CIPHER()$ and its inverse is denoted by $INVCIPHER()$ As we mention before, the core algorithm for $CIPHER()$ and $INVCIPHER$ is a sequence of fixed transformations of the State. Each rounds of $CIPHER()$ are composed of the following four byte-oriented transformations on the State :

1) SubBytes Transformation: SubBytes transformation is an invertible, non-linear transformation of the state. It uses a substitution table called an S-box which is applied independently to each byte in the state.

Let b denote an input byte to the S-box, and let c denote the constant byte. The output byte b' is constructed by the following two transformations:

1) **Multiplicative Inverse in GF(2^8):**
   Define an intermediate value $\tilde{b}$ as:
$$\tilde{b} = \begin{cases} \{00\} & \text{if } b = \{00\} \\ b^{-1} & \text{if } b \neq \{00\} \end{cases} \tag{3}$$
   where b^{-1} is the multiplicative inverse of b in GF(2^8).
2) **Affine Transformation:**
   Apply the following affine transformation to the bits of $\tilde{b}$ to produce b':
$$\begin{aligned} b'_i = &\tilde{b}_i \oplus \tilde{b}_{(i+4) \bmod 8} \oplus \tilde{b}_{(i+5) \bmod 8} \\ &\oplus \tilde{b}_{(i+6) \bmod 8} \oplus \tilde{b}_{(i+7) \bmod 8} \oplus c_i \end{aligned} \tag{4}$$
   where i represents the bit position (0 to 7), and c_i is the i-th bit of the constant c.

For instance, for $b = \{11010101\}_2$ or in Hex $\{0xD5\}_{16}$ and $c = \{01100011\}_2$, we would find the Multiplication Inverse b^{-1}. To ease this process, we would just look up on $GF(2^8)$ table.

	Y															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	01	8D	F6	CB	52	7B	D1	E8	4F	29	C0	B0	E1	E5	C7
1	74	B4	AA	4B	99	2B	60	5F	58	3F	FD	CC	FF	40	EE	B2
2	3A	6E	5A	F1	55	4D	A8	C9	C1	0A	98	15	30	44	A2	C2
3	2C	45	92	6C	F3	39	66	42	F2	35	20	6F	77	BB	59	19
4	1D	FE	37	67	2D	31	F5	69	A7	64	AB	13	54	25	E9	09
5	ED	5C	05	CA	4C	24	87	BF	18	3E	22	F0	51	EC	61	17
6	16	5E	AF	D3	49	A6	36	43	F4	47	91	DF	33	93	21	3B
7	79	B7	97	85	10	B5	BA	3C	B6	70	D0	06	A1	FA	81	82
X 8	83	7E	7F	80	96	73	BE	56	9B	9E	95	D9	F7	02	B9	A4
9	DE	6A	32	6D	D8	8A	84	72	2A	14	9F	88	F9	DC	89	9A
A	FB	7C	2E	C3	8F	B8	65	48	26	C8	12	4A	CE	E7	D2	62
B	0C	E0	1F	EF	11	75	78	71	A5	8E	76	3D	BD	BC	86	57
C	0B	28	2F	A3	DA	D4	E4	0F	A9	27	53	04	1B	FC	AC	E6
D	7A	07	AE	63	C5	DB	E2	EA	94	8B	C4	D5	9D	F8	90	6B
E	B1	0D	D6	EB	C6	0E	CF	AD	08	4E	D7	E3	5D	50	1E	B3
F	5B	23	38	34	68	46	03	8C	DD	9C	7D	A0	CD	1A	41	1C

Fig. 2. Multiplicative Inverse Table in $GF(2^8)$ used in the AES S-Box
Source: https://i.sstatic.net/Qqo8a.png

From the table, we obtain $b^{-1} = 0xDB$ or in binary $\{11011011\}_2$ and c, and now with affine operation below, we got $b' = \{00000011\}_2$ or in Hex $\{0x03\}_{16}$ The bit representation (LSB-first, bit 0 on the right) is:

$$b'_0 = \tilde{b}_0 \oplus \tilde{b}_4 \oplus \tilde{b}_5 \oplus \tilde{b}_6 \oplus \tilde{b}_7 \oplus c_0 = 1 \oplus 1 \oplus 0 \oplus 1 \oplus 1 \oplus 1 = 1$$
$$b'_1 = \tilde{b}_1 \oplus \tilde{b}_5 \oplus \tilde{b}_6 \oplus \tilde{b}_7 \oplus \tilde{b}_0 \oplus c_1 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 1 \oplus 1 = 1$$
$$b'_2 = \tilde{b}_2 \oplus \tilde{b}_6 \oplus \tilde{b}_7 \oplus \tilde{b}_0 \oplus \tilde{b}_1 \oplus c_2 = 0 \oplus 1 \oplus 1 \oplus 1 \oplus 1 \oplus 0 = 0$$
$$b'_3 = \tilde{b}_3 \oplus \tilde{b}_7 \oplus \tilde{b}_0 \oplus \tilde{b}_1 \oplus \tilde{b}_2 \oplus c_3 = 1 \oplus 1 \oplus 1 \oplus 1 \oplus 0 \oplus 0 = 0$$
$$b'_4 = \tilde{b}_4 \oplus \tilde{b}_0 \oplus \tilde{b}_1 \oplus \tilde{b}_2 \oplus \tilde{b}_3 \oplus c_4 = 1 \oplus 1 \oplus 1 \oplus 0 \oplus 1 \oplus 0 = 0$$
$$b'_5 = \tilde{b}_5 \oplus \tilde{b}_1 \oplus \tilde{b}_2 \oplus \tilde{b}_3 \oplus \tilde{b}_4 \oplus c_5 = 0 \oplus 1 \oplus 0 \oplus 1 \oplus 1 \oplus 1 = 0$$
$$b'_6 = \tilde{b}_6 \oplus \tilde{b}_2 \oplus \tilde{b}_3 \oplus \tilde{b}_4 \oplus \tilde{b}_5 \oplus c_6 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 = 0$$
$$b'_7 = \tilde{b}_7 \oplus \tilde{b}_3 \oplus \tilde{b}_4 \oplus \tilde{b}_5 \oplus \tilde{b}_6 \oplus c_7 = 1 \oplus 1 \oplus 1 \oplus 0 \oplus 1 \oplus 0 = 0$$

To ease the process, we could pre-compute the S-box transformation (multiplicative inverse and also affine transformation) and stored as a 256-byte lookup table, as shown in Fig. 3 :

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

x

Fig. 3. SubBytes Table. Source: [1]

As the result, we obtain the same result from Pre-Compute method, which 0xD5 is transformed into 0x03.

2) ShiftRows Transformation: ShiftRows does cyclic rotation to each row of State Matrix. This operation provides diffusion by ensuring that bytes from different columns are mixed in subsequent operations. The transformation performs a cyclic left rotation on each row:

- Row 0: no shift (unchanged)

- Row 1: left shift by 1 byte position
- Row 2: left shift by 2 byte positions
- Row 3: left shift by 3 byte positions

The shift row operation could be defined as :

$$s'_{r,c} = s_{r,(c+r) \bmod 4} \qquad 0 \leq r, c < 4 \tag{5}$$

where $s_{r,c}$ denotes the byte at row r and column c of the state matrix.

For Example, Consider a state matrix before ShiftRows:

$$\begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix}$$

After ShiftRows:

$$\begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \end{bmatrix}$$

3) MixColumns Transformation: MixColumns is a linear transformation that operates on each column of state matrix. Each column will be multiplied with a fixed polynomial in GF(2) which could create an Avalanche Effect (*a principle where a small change in input could cause the output differs at total*).

The transformation of each column can be represented with a matrix of multiplication in GF(2^8) :

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \qquad 0 \leq c < 4$$

Note: The constant matrix is specified by the AES standard and must not be changed.

The result of each column would be :

$$\begin{aligned} s'_{0,c} &= (\{02\} \cdot s_{0,c}) \oplus (\{03\} \cdot s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\ s'_{1,c} &= s_{0,c} \oplus (\{02\} \cdot s_{1,c}) \oplus (\{03\} \cdot s_{2,c}) \oplus s_{3,c} \\ s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \cdot s_{2,c}) \oplus (\{03\} \cdot s_{3,c}) \\ s'_{3,c} &= (\{03\} \cdot s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \cdot s_{3,c}) \end{aligned}$$

where:

- $\cdot$ denotes multiplication in GF(2^8)
- $\oplus$ denotes addition in GF(2^8), which is the XOR operation
- $\{02\}$ and $\{03\}$ are hexadecimal constants

4) AddRoundKey: AddRoundKey is a column operation where each column is combined with each RoundKey by using XOR operation. RoundKey is a routine that is applied to the key to generate 4*(Nr+1) words. By using KeyExpansion, we could get an unique key for each round. KeyExpansion is the processs of deriving the roundkey from the original encryption key. Operation that is involved with Key Expansion :

1) **RotWord:** A cyclic rotation of the bytes within a word by one position to left.

$$\text{RotWord}([a_0, a_1, a_2, a_3]) = [a_1, a_2, a_3, a_0]$$

2) **SubWord:** Applies the S-box substitution to each byte in the word.

$$\begin{aligned} \text{SubWord}([a_0, a_1, a_2, a_3]) = &[\text{S-box}[a_0], \text{S-box}[a_1], \\ &\text{S-box}[a_2], \text{S-box}[a_3]] \end{aligned}$$

3) **Rcon (Round Constant):** 10 fixed words denoted by rc[j] for $1 \leq i \leq 10$. The value of of rc[j] are given in hexadecimal notation in Fig. 4

j	*Rcon*[*j*]	*j*	*Rcon*[*j*]
1	[01,00,00,00]	6	[20,00,00,00]
2	[02,00,00,00]	7	[40,00,00,00]
3	[04,00,00,00]	8	[80,00,00,00]
4	[08,00,00,00]	9	[1b,00,00,00]
5	[10,00,00,00]	10	[36,00,00,00]

Fig. 4. Round Constant Table. Source: [1]

The Key Expansion algorithm generates a total of $Nb(Nr + 1)$ words, where $Nb = 4$ is the number of columns in the State and Nr is the number of rounds. For AES-128, this produces $4 \times 11 = 44$ words.

Note: In this paper, we focus only on the encryption process (CIPHER) and its algebraic representation for cryptanalysis purposes. The inverse operation for decryption are not discussed, as they are not relevant to the algebraic attack methodology. We also only use one round as demonstration to ease the process.

III. Algebraic Representation

Algebraic Cryptanalysis is an attack method that models cipher as a system of polynomial equations. Solving this equation by treating ciphertext and plaintext as known value and the key bit as unknown, we could retrieve the secret key.

A. Basic Concept

Consider a cipher that transforms plaintext P into ciphertext C using key K:

$$C = E(P, K) \tag{6}$$

In algebraic cryptanalysis, we express each bit of the encryption process as polynomial equations. If we denote plaintext bits as p_i, ciphertext bits as c_i, and key bits as k_i, the cipher operations become a system of equations:

$$f_j(p_0, p_1, ..., k_0, k_1, ...) = c_j \qquad \text{for each output bit } j$$

The attack succeeds when we can solve this system to find the key variables k_i.

B. Demonstration with Simple Linear Cipher

To illustrate this concept, we will use a simple cipher. Consider a 4-bit cipher with 4-bit key and the encryption only involve XOR Operation between key and plaintext.

$$c_i = p_i \oplus k_i \qquad \text{for } i = 0, 1, 2, 3$$

Example: Let plaintext $P = 1101$ and ciphertext = $C = 0111$. The system of equations becomes :

$$\begin{aligned} p_0 \oplus k_0 = c_0 \quad &\Rightarrow \quad 1 \oplus k_0 = 0 \quad \Rightarrow \quad k_0 = 1 \\ p_1 \oplus k_1 = c_1 \quad &\Rightarrow \quad 1 \oplus k_1 = 1 \quad \Rightarrow \quad k_1 = 0 \\ p_2 \oplus k_2 = c_2 \quad &\Rightarrow \quad 0 \oplus k_2 = 1 \quad \Rightarrow \quad k_2 = 1 \\ p_3 \oplus k_3 = c_3 \quad &\Rightarrow \quad 1 \oplus k_3 = 1 \quad \Rightarrow \quad k_3 = 0 \end{aligned}$$

Now, we could retrieve the key bits : $K = 1010$. This cipher is easily broken because:

a) The equations are linear which mean each equation contains only one degree terms.
b) Each key has exactly 1 equation, no mixing between variables.

What about if we adjust to a more complex cipher? Let's try with the Affine Cipher.
Assume plaintext $P =$ "ALGEO" and $C =$ "OLKUW". The Affine Cipher uses two key components a and b with encryption equation:

$$c_i = (a \cdot p_i + b) \mod 26 \tag{7}$$

We convert the plaintext and ciphertext into integer mapping (A=0, B=1, ..., Z=25):

$$\begin{aligned} P &= [0, 11, 6, 4, 14] \\ C &= [14, 11, 10, 20, 22] \end{aligned}$$

From the first character, we get:

$$\begin{aligned} c_0 &= (a \cdot p_0 + b) \mod 26 \\ 14 &= a \cdot 0 + b \\ b &= 14 \end{aligned}$$

Substituting $b = 14$ into the second equation:

$$\begin{aligned} c_1 &= (a \cdot p_1 + b) \mod 26 \\ 11 &= 11a + 14 \pmod{26} \\ 11a &= -3 \pmod{26} \\ 11a &= 23 \pmod{26} \end{aligned}$$

Since $GCD(11, 26) = 1$, this mean 11 has multiplicative inverse modulo 26 which is 19 because $11 \cdot 19 = 209 = 8 \cdot 26 + 1 \equiv 1 \pmod{26}$.

$$a = 19 \cdot 23 \mod 26 = 437 \mod 26 = 21$$

Thus, the key is retrieved: $a = 21$, $b = 14$.
We can verify with the remaining characters:

$$\begin{aligned} c_2 &= (21 \cdot 6 + 14) \mod 26 = 140 \mod 26 = 10 \\ c_3 &= (21 \cdot 4 + 14) \mod 26 = 98 \mod 26 = 20 \\ c_4 &= (21 \cdot 14 + 14) \mod 26 = 308 \mod 26 = 22 \end{aligned}$$

By this demonstration, we could see that even a more complex cipher can still be easily broken with algebraic analysis. This happens because all of these ciphers have **linear equations**.
Question: What happens when we apply this method to a non-linear cipher?
In AES, the SubBytes operation applies multiplicative inversion in $GF(2^8)$, which produces high-degree polynomial equations. We will discuss how this non-linearity affects algebraic cryptanalysis in the following section.

IV. Algebraic Modeling of AES

After understanding how linear ciphers can be easily broken by Algebraic Attack, now we apply the same method to AES. The key challenge here is to handle the non-linear equations (SubBytes operation).
Suppose we have plaintext "ALJABAR GEOMETRI", converted to hex:

$$\begin{aligned} P = &\texttt{41 4C 4A 41 42 41 52 20} \\ &\texttt{47 45 4F 4D 45 54 52 49} \end{aligned}$$

And we get ciphertext:

$$\begin{aligned} C = &\texttt{1D 5A 5C 31 52 8C CE 8E} \\ &\texttt{70 C5 ED 23 48 0C 9E D3} \end{aligned}$$

Arranging into state matrix:

$$P = \begin{bmatrix} \texttt{41} & \texttt{42} & \texttt{47} & \texttt{45} \\ \texttt{4C} & \texttt{41} & \texttt{45} & \texttt{54} \\ \texttt{4A} & \texttt{52} & \texttt{4F} & \texttt{52} \\ \texttt{41} & \texttt{20} & \texttt{4D} & \texttt{49} \end{bmatrix}, \quad C = \begin{bmatrix} \texttt{1D} & \texttt{52} & \texttt{70} & \texttt{48} \\ \texttt{5A} & \texttt{8C} & \texttt{C5} & \texttt{0C} \\ \texttt{5C} & \texttt{CE} & \texttt{ED} & \texttt{9E} \\ \texttt{31} & \texttt{8E} & \texttt{23} & \texttt{D3} \end{bmatrix}$$

Our goal is to find the key matrix K:

$$K = \begin{bmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{bmatrix}$$

Now, we need to analyze each operation on AES. In this paper KeyExpansion is not included.

A. AddRoundKey : Initial Operation on AES

We know that the initial step on AES is to do XOR operation on the plaintext with initial Key.
Let S be the state after AddRoundKey operation:

$$S = \begin{bmatrix} P_{0,0} \oplus k_{0,0} & P_{0,1} \oplus k_{0,1} & P_{0,2} \oplus k_{0,2} & P_{0,3} \oplus k_{0,3} \\ P_{1,0} \oplus k_{1,0} & P_{1,1} \oplus k_{1,1} & P_{1,2} \oplus k_{1,2} & P_{1,3} \oplus k_{1,3} \\ P_{2,0} \oplus k_{2,0} & P_{2,1} \oplus k_{2,1} & P_{2,2} \oplus k_{2,2} & P_{2,3} \oplus k_{2,3} \\ P_{3,0} \oplus k_{3,0} & P_{3,1} \oplus k_{3,1} & P_{3,2} \oplus k_{3,2} & P_{3,3} \oplus k_{3,3} \end{bmatrix}$$

This gives us 16 equations where each $S_{r,c} = P_{r,c} \oplus k_{r,c}$.

B. SubBytes Operation

After AddRoundKey, each byte of state S passes through the S-box. SubBytes is the only non-linear operation in AES. Understanding why it is non-linear is crucial for algebraic cryptanalysis.
A function f is **linear** if it satisfies the following property:

$$f(a \oplus b) = f(a) \oplus f(b)$$

This means the function can be distributed over XOR operation. If a function is linear, we can separate and solve each variable independently.
A function is **non-linear** if:

$$f(a \oplus b) \neq f(a) \oplus f(b)$$

Non-linear functions mix the inputs in a way that cannot be separated. Let us test the function using S-box :

$$\begin{aligned} a &= \texttt{0x10} \\ b &= \texttt{0x20} \\ a \oplus b &= \texttt{0x30} \end{aligned}$$

Looking up the S-box table (Fig. 3):

$$\begin{aligned} \text{Sbox}[\texttt{10}] &= \texttt{CA} \\ \text{Sbox}[\texttt{20}] &= \texttt{B7} \\ \text{Sbox}[\texttt{30}] &= \texttt{04} \end{aligned}$$

If S-box were linear, then $\text{Sbox}[a \oplus b]$ should equal $\text{Sbox}[a] \oplus \text{Sbox}[b]$:

$$\begin{aligned} \text{Sbox}[a] \oplus \text{Sbox}[b] &= \texttt{CA} \oplus \texttt{B7} = \texttt{7D} \\ \text{Sbox}[a \oplus b] &= \text{Sbox}[\texttt{30}] = \texttt{04} \end{aligned}$$

Since $\texttt{7D} \neq \texttt{04}$, the S-box is **non-linear**. This mean :

$$\text{Sbox}[P_{i,j} \oplus K_{i,j}] \neq \text{Sbox}[P_{i,j}] \oplus \text{Sbox}[K_{i,j}]$$

After proofing that S-box operation is not linear, then come up a question "How can we solve this Algebraicly?".
Well, we actually could see that:

$$\begin{aligned} x_{r,c} &= P_{r,c} \oplus K_{r,c} \quad \text{(SubBytes input)} \\ t_{r,c} &= \text{Sbox}[x_{r,c}] \quad \text{(SubBytes output)} \end{aligned}$$

Now, we have the S-box output matrix:

$$T = \begin{bmatrix} t_{0,0} & t_{0,1} & t_{0,2} & t_{0,3} \\ t_{1,0} & t_{1,1} & t_{1,2} & t_{1,3} \\ t_{2,0} & t_{2,1} & t_{2,2} & t_{2,3} \\ t_{3,0} & t_{3,1} & t_{3,2} & t_{3,3} \end{bmatrix}$$

C. ShiftRows Operation

After SubBytes, the state matrix T undergoes ShiftRows transformation. This operation performs cyclic left shifts on each row:

$$u_{r,c} = t_{r,(c+r) \bmod 4} \quad \text{for } 0 \leq r, c \leq 3$$

Applying to our S-box output matrix T:

$$T = \begin{bmatrix} t_{0,0} & t_{0,1} & t_{0,2} & t_{0,3} \\ t_{1,0} & t_{1,1} & t_{1,2} & t_{1,3} \\ t_{2,0} & t_{2,1} & t_{2,2} & t_{2,3} \\ t_{3,0} & t_{3,1} & t_{3,2} & t_{3,3} \end{bmatrix} \rightarrow \begin{bmatrix} t_{0,0} & t_{0,1} & t_{0,2} & t_{0,3} \\ t_{1,1} & t_{1,2} & t_{1,3} & t_{1,0} \\ t_{2,2} & t_{2,3} & t_{2,0} & t_{2,1} \\ t_{3,3} & t_{3,0} & t_{3,1} & t_{3,2} \end{bmatrix}$$

D. MixColumns Operation

After ShiftRows, each column of state T is transformed by MixColumns. This operation multiplies each column by a fixed matrix in $GF(2^8)$:

$$\begin{bmatrix} v_{0,c} \\ v_{1,c} \\ v_{2,c} \\ v_{3,c} \end{bmatrix} = \begin{bmatrix} \texttt{02} & \texttt{03} & \texttt{01} & \texttt{01} \\ \texttt{01} & \texttt{02} & \texttt{03} & \texttt{01} \\ \texttt{01} & \texttt{01} & \texttt{02} & \texttt{03} \\ \texttt{03} & \texttt{01} & \texttt{01} & \texttt{02} \end{bmatrix} \begin{bmatrix} T_{0,c} \\ T_{1,c} \\ T_{2,c} \\ T_{3,c} \end{bmatrix} \quad \text{for } c = 0, 1, 2, 3$$

Expanding for each output byte:

$$\begin{aligned} v_{0,c} &= (\texttt{02} \cdot T_{0,c}) \oplus (\texttt{03} \cdot T_{1,c}) \oplus T_{2,c} \oplus T_{3,c} \\ v_{1,c} &= T_{0,c} \oplus (\texttt{02} \cdot T_{1,c}) \oplus (\texttt{03} \cdot T_{2,c}) \oplus T_{3,c} \\ v_{2,c} &= T_{0,c} \oplus T_{1,c} \oplus (\texttt{02} \cdot T_{2,c}) \oplus (\texttt{03} \cdot T_{3,c}) \\ v_{3,c} &= (\texttt{03} \cdot T_{0,c}) \oplus T_{1,c} \oplus T_{2,c} \oplus (\texttt{02} \cdot T_{3,c}) \end{aligned}$$

Substituting ShiftRows output into MixColumns, for column 0:

$$\begin{aligned} c_{0,0} &= (\texttt{02} \cdot t_{0,0}) \oplus (\texttt{03} \cdot t_{1,1}) \oplus t_{2,2} \oplus t_{3,3} \\ c_{1,0} &= t_{0,0} \oplus (\texttt{02} \cdot t_{1,1}) \oplus (\texttt{03} \cdot t_{2,2}) \oplus t_{3,3} \\ c_{2,0} &= t_{0,0} \oplus t_{1,1} \oplus (\texttt{02} \cdot t_{2,2}) \oplus (\texttt{03} \cdot t_{3,3}) \\ c_{3,0} &= (\texttt{03} \cdot t_{0,0}) \oplus t_{1,1} \oplus t_{2,2} \oplus (\texttt{02} \cdot t_{3,3}) \end{aligned}$$

Since $t_{r,c} = \text{Sbox}[P_{r,c} \oplus K_{r,c}]$, the complete global equation for $c_{0,0}$ is:

$$\begin{aligned} c_{0,0} = &(\texttt{02} \cdot \text{Sbox}[P_{0,0} \oplus K_{0,0}]) \\ &\oplus (\texttt{03} \cdot \text{Sbox}[P_{1,1} \oplus K_{1,1}]) \\ &\oplus \text{Sbox}[P_{2,2} \oplus K_{2,2}] \\ &\oplus \text{Sbox}[P_{3,3} \oplus K_{3,3}] \end{aligned} \tag{8}$$

Here we get total 16 global equations. Now, we need to look into the bit level.

V. Examine each Bytes into Polynomial Equations

For byte $K_{0,0}$, we define bit-level variables:

$$\begin{aligned} P_{0,0} &= (p_7, p_6, p_5, p_4, p_3, p_2, p_1, p_0) \\ &= \texttt{0x41} = (0, 1, 0, 0, 0, 0, 0, 1) \\ K_{0,0} &= (k_7, k_6, k_5, k_4, k_3, k_2, k_1, k_0) \quad \text{(Key)} \end{aligned}$$

A. AddRoundKey Equations

Let $x = P_{0,0} \oplus K_{0,0}$ be the S-box input. At bit level:

$$\begin{aligned}
x_0 &= p_0 \oplus k_0 = 1 \oplus k_0 \\
x_1 &= p_1 \oplus k_1 = 0 \oplus k_1 = k_1 \\
x_2 &= p_2 \oplus k_2 = 0 \oplus k_2 = k_2 \\
x_3 &= p_3 \oplus k_3 = 0 \oplus k_3 = k_3 \\
x_4 &= p_4 \oplus k_4 = 0 \oplus k_4 = k_4 \\
x_5 &= p_5 \oplus k_5 = 0 \oplus k_5 = k_5 \\
x_6 &= p_6 \oplus k_6 = 1 \oplus k_6 \\
x_7 &= p_7 \oplus k_7 = 0 \oplus k_7 = k_7
\end{aligned}$$

From this operation we get about 8 linear equations.

B. SubBytes Equations

The S-box consists of two operations:

a) Multiplicative inverse: $y = x^{-1}$ in $GF(2^8)$
b) Affine transformation: $t = A \cdot y \oplus \texttt{0x63}$

To express x^{-1} algebraically, we use the definition:

$$x \cdot y = 1 \quad \text{in } GF(2^8)$$

Since $1 = \texttt{0x01} = 00000001_2$, the product $x \cdot y$ must satisfy:

- Bit 0 of result = 1
- Bits 1-7 of result = 0

Expanding the multiplication at bit level over $GF(2)$ with irreducible polynomial, each bit of the result becomes a polynomial equation. This gives us **8 quadratic equations**:
Bit 0:

$$\begin{aligned}
1 = {} & x_0y_0 \oplus x_1y_7 \oplus x_2y_6 \oplus x_3y_5 \oplus x_4y_4 \oplus x_5y_3 \\
& \oplus x_6y_2 \oplus x_7y_1 \oplus x_3y_7 \oplus x_4y_6 \oplus x_5y_5 \oplus x_6y_4 \\
& \oplus x_7y_3 \oplus x_4y_7 \oplus x_5y_6 \oplus x_6y_5 \oplus x_7y_4 \oplus x_6y_7 \\
& \oplus x_7y_6
\end{aligned}$$

Bit 1-7:

$$\begin{aligned}
0 = {} & x_0y_1 \oplus x_1y_0 \oplus x_2y_7 \oplus x_3y_6 \oplus x_4y_5 \oplus x_5y_4 \\
& \oplus x_6y_3 \oplus x_7y_2 \oplus x_4y_7 \oplus x_5y_6 \oplus x_6y_5 \\
& \oplus x_7y_4 \oplus x_5y_7 \oplus x_6y_6 \oplus x_7y_5 \oplus x_7y_7
\end{aligned}$$

Each equation contains terms of the form x_iy_j, which are **degree-2 (quadratic)** polynomials. This nonlinearity is what makes algebraic cryptanalysis of AES computationally difficult.

C. Affine Transformation Equations

After obtaining $y = x^{-1}$, the affine transformation computes the S-box output:

$$t = A \cdot y \oplus c \tag{9}$$

where A is an 8×8 binary matrix and $c = \texttt{0x63} = 01100011_2$.

Expanding at bit level, we get **8 linear equations**:

$$\begin{aligned}
t_0 &= y_0 \oplus y_4 \oplus y_5 \oplus y_6 \oplus y_7 \oplus 1 \\
t_1 &= y_0 \oplus y_1 \oplus y_5 \oplus y_6 \oplus y_7 \oplus 1 \\
t_2 &= y_0 \oplus y_1 \oplus y_2 \oplus y_6 \oplus y_7 \\
t_3 &= y_0 \oplus y_1 \oplus y_2 \oplus y_3 \oplus y_7 \\
t_4 &= y_0 \oplus y_1 \oplus y_2 \oplus y_3 \oplus y_4 \\
t_5 &= y_1 \oplus y_2 \oplus y_3 \oplus y_4 \oplus y_5 \oplus 1 \\
t_6 &= y_2 \oplus y_3 \oplus y_4 \oplus y_5 \oplus y_6 \oplus 1 \\
t_7 &= y_3 \oplus y_4 \oplus y_5 \oplus y_6 \oplus y_7
\end{aligned}$$

These equations link the inverse bits y_i to the S-box output bits t_i.

D. Combining ShiftRows and MixColumns

After SubBytes operations, the state passes through ShiftRows and MixColumns before becoming the ciphertext.
To recover the key, we work backwards from the ciphertext:

a) Apply InvMixColumns to ciphertext C
b) Apply InvShiftRows
c) This gives us the S-box outputs $t_{r,c}$ (known values)

Once $t_{r,c}$ is known, we substitute into the affine equations to constrain y_i, then use the inverse constraint $x \cdot y = 1$ and AddRoundKey equations to solve for key bits k_i.

E. Summary for a single byte

From the full set of equations derived from AddRoundKey, SubBytes, and the affine transformation, our objective is to determine which variables remain unknown after systematic substitution.
We begin by observing that the plaintext bits p_i are known, and the S-box output bits t_i can be recovered from the ciphertext by applying $\mathrm{InvMixColumns}$ followed by $\mathrm{InvShiftRows}$. Consequently, both p_i and t_i are treated as constants(known).
From the AddRoundKey operation, the S-box input bits are expressed as

$$x_i = p_i \oplus k_i,$$

which allows us to eliminate all variables x_i by rewriting them as linear expressions in the key bits k_i.
Next, consider the affine transformation in the SubBytes operation:

$$t = A \cdot y \oplus c.$$

Since t and c are known and A is invertible, the inverse bits y_i can be uniquely determined as

$$y = A^{-1}(t \oplus c).$$

Thus, all variables y_i are also eliminated and become known constants.

After these substitutions, the only remaining unknowns are the key bits k_i.
We now substitute both $x_i = p_i \oplus k_i$ and the known values of y_i into the inverse constraint

$$x \cdot y = 1 \quad \text{in } GF(2^8).$$

Each resulting equation is of the form

$$\sum_j (p_i \oplus k_i)\, y_j = c,$$

where y_j and p_i are constants. As a result, all previously quadratic terms reduce to linear expressions in the key bits.
Therefore, for a single byte, the system reduces to eight linear equations in eight unknown key bits. If these equations are linearly independent, the entire byte of the round key can be uniquely recovered.
Extending this reasoning to the full AES state, the sixteen S-boxes yield a total of 128 linear equations in 128 key bit variables. Under ideal conditions and ignoring the key schedule, this implies that the full round key is theoretically recoverable from the ciphertext and plaintext pair.

F. Summary of Complete System

We now extend the previous analysis from a single S-box to the full AES encryption.
For the complete AES state, there are sixteen bytes processed by the SubBytes operation in each round. For each byte, the plaintext bits are known and the corresponding S-box output bits can be recovered from the ciphertext by applying the $InvMixColumns$ and $InvShiftRows$. Hence, for every S-box, both the input plaintext bits and the output bits of the affine transformation are treated as known constants.
As in the single-byte case, the AddRoundKey operation allows the elimination of the S-box input variables:

$$x_{r,c,i} = p_{r,c,i} \oplus k_{r,c,i},$$

where (r, c) denotes the byte position in the state and i the bit index. Thus, all internal variables $x_{r,c,i}$ are rewritten as linear expressions in the round key bits.
Similarly, the affine transformation equations

$$t_{r,c} = A \cdot y_{r,c} \oplus c$$

can be inverted since A is nonsingular and $t_{r,c}$ is known. This uniquely determines each inverse variable $y_{r,c}$, eliminating all $y_{r,c,i}$ from the system.
After these substitutions, the only remaining unknowns in the entire system are the round key bits $k_{r,c,i}$.
Substituting the expressions for $x_{r,c,i}$ and the known constants $y_{r,c,i}$ into the multiplicative inverse constraints

$$x_{r,c} \cdot y_{r,c} = 1 \quad \text{in } GF(2^8),$$

all quadratic terms collapse into linear expressions. Consequently, each S-box contributes eight linear equations in eight key bit variables.
For one round of AES, the sixteen S-boxes together yield a system of 128 linear equations in 128 round key bit variables. If the resulting system has full rank, the entire round key is uniquely determined.
However, this result applies only to an isolated round key. In the full AES cipher, round keys are not independent but are linked by the key schedule. When multiple rounds are modeled simultaneously, equations from different rounds introduce dependencies among key bits, while additional auxiliary variables reappear due to repeated SubBytes operations. This coupling often causes rank deficiency in the global system, leading to multiple algebraic solutions and preventing direct recovery of the master key.
Therefore, although the algebraic system for a single round can theoretically determine all round key bits after elimination, extending the approach to the full AES cipher does not result in a practical key recovery attack.

ACKNOWLEDGMENT

Above all, the author expresses gratitude to God Almighty for the strength, health, and perseverance throughout the completion of this work. Sincere thanks are also given to the author's family for their continuous support, encouragement, and patience. Lastly, appreciation is extended to friends and peers who provided discussions, motivation, and moral support during the learning and writing process.
The author would also like to express sincere gratitude to Prof. Dr. Rinaldi Munir, S.T., M.T., lecturer at Institut Teknologi Bandung, for his guidance and lectures in the *Algebra and Geometry* course during the third semester.
The author also acknowledges the contribution of artificial intelligence tools, which were utilized as a mentoring aid throughout the writing process. These tools assisted in proofreading, improving clarity of explanations, refining mathematical narration, and verifying the logical flow of algebraic modeling and implementation. The final analysis, interpretations, and conclusions remain the sole responsibility of the author.

REFERENCES

[1] National Institute of Standards and Technology (2001) Advanced Encryption Standard (AES). (Department of Commerce, Washington, D.C.), Federal Information Processing Standards Publication (FIPS) NIST FIPS 197-upd1, updated May 9, 2023. https://doi.org/10.6028/NIST.FIPS.197-upd1
[2] S. Gueron, "Intel Advanced Encryption Standard (AES) New Instructions Set," Intel Corporation, White Paper 323641-001, 2010. [Online]. Available: https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf
[3] N. Courtois and J. Pieprzyk, "Cryptanalysis of Block Ciphers with Overdefined Systems of Equations," in *Advances in Cryptology – ASIACRYPT 2002*, Lecture Notes in Computer Science, vol. 2501, Springer, Berlin, Heidelberg, 2002, pp. 267–287.

[4] J. Daemen and V. Rijmen, *The Design of Rijndael: AES – The Advanced Encryption Standard*, Springer-Verlag, Berlin, Heidelberg, 2002.

PERSONAL STATEMENT

I hereby declare that the paper I have written is my own work, not an adaptation or translation of someone else's work, and does not contain any form of plagiarism.

Bandung, December 24, 2025

Stevanus Agustaf Wongso
13524020