Galois Field in Cryptography

Christoforus Juan Benvenuto

May 31, 2012

Abstract

This paper introduces the basics of Galois Field as well as its implementation in storing data. This paper shows and helps visualizes that storing data in Galois Fields allows manageable and effective data manipulation, where it focuses mainly on application in computer cryptography. Details on the algorithm for Advanced Encryption Standard (AES), which is an examples of computer cryptography that utilizes Galois Field, will also be included.

Contents

1 Introduction **2**

2 Preliminaries **2**
- 2.1 Galois Field . . . 2
- 2.2 Binary System . . . 3
- 2.3 Bit and Byte . . . 3
- 2.4 ASCII . . . 3
- 2.5 Finite Field Arithmetic . . . 4
  - 2.5.1 Addition and Subtraction . . . 4
  - 2.5.2 Multiplication and Multiplicative Inverse . . . 5

3 Application in Cryptography **7**
- 3.1 Advanced Encryption Standard (AES) . . . 7
  - 3.1.1 State . . . 7
  - 3.1.2 SubBytes . . . 8
  - 3.1.3 ShiftRows . . . 9
  - 3.1.4 MixColumns . . . 9
  - 3.1.5 AddRoundKey . . . 10

1 Introduction

Galois Field, named after Évariste Galois, also known as finite field, refers to a field in which there exists finitely many elements. It is particularly useful in translating computer data as they are represented in binary forms. That is, computer data consist of combination of two numbers, 0 and 1, which are the components in Galois field whose number of elements is two. Representing data as a vector in a Galois Field allows mathematical operations to scramble data easily and effectively.

2 Preliminaries

2.1 Galois Field

The elements of Galois Field $gf(p^n)$ is defined as

$$\begin{aligned} gf(p^n) &= (0, 1, 2, \ldots, p-1) \cup \\ &\quad (p, p+1, p+2, \ldots, p+p-1) \cup \\ &\quad (p^2, p^2+1, p^2+2, \ldots, p^2+p-1) \cup \ldots \cup \\ &\quad (p^{n-1}, p^{n-1}+1, p^{n-1}+2, \ldots, p^{n-1}+p-1) \end{aligned}$$

where $p \in \mathbb{P}$ and $n \in \mathbb{Z}^+$. The order of the field is given by p^n while p is called the characteristic of the field. On the other hand, gf, as one may have guessed it, stands for Galois Field. Also note that the degree of polynomial of each element is at most $n-1$.

Example

$$gf(5) = (0, 1, 2, 3, 4)$$

which consists of 5 elements where each of them is a polynomial of degree 0 (a constant) while

$$\begin{aligned} gf(2^3) &= (0, 1, 2, 2+1, 2^2, 2^2+1, 2^2+2, 2^2+2+1) \\ &= (0, 1, 2, 3, 4, 5, 6, 7) \end{aligned}$$

which consists of $2^3 = 8$ elements where each of them is a polynomial of degree at most 2 evaluated at 2.

2.2 Binary System

In the binary numeral system or base-2 number system, we represents each value with 0 and 1. To convert a decimal numeral system or base-10 number system into binary system, we need to represent a decimal in terms of sums of $a_n 2^n$. That is, if x is the said decimal number then we wish to have

$$x = \sum_{n \in \mathbb{N}} a_n 2^n$$

The coefficients a_n is then written in descending order of n and all leading zeros are then omitted. The final result becomes the binary representation of the decimal x. Ultimately, binary system offers an alternative way of representing the elements of a Galois Field. Both the polynomial and binary representation of an element have their own advantages and disadvantages.

Example

$$19 = \ldots + 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

so the binary representation of 19 is 10011 while the elements of $gf(2^3)$ in binary are

$$gf(2^3) = (001, 010, 011, 100, 101, 110, 111)$$

2.3 Bit and Byte

Each 0 or 1 is called a bit, and since a bit is either 0 or 1, a bit is an element of $gf(2)$. There is also a byte which is equivalent to 8 bits thus is an element of $gf(2^8)$. Since we will be focusing on computer cryptography and as each datum is a series of bytes, we are only interested in Galois Field of order 2 and 2^8 in this paper.
Because computer stores data in bytes, each binary number must be 8 bits long. For number that is less than 8 bits long, leading zeros are added. It follows as well that the biggest number 1 byte can store is $11111111 = 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 255$. Following from the preceding example, 19 is stored as 00010011 in byte.

2.4 ASCII

ASCII stands for American Standard Code for Information Interchange. Since there are exactly 255 characters in ASCII, we can uniquely assign

each character to an element in $gf(2^8)$ or represent it as a byte. The most frequently used ASCII codes and their values are given in the following table

Dec	Chr	Dec	Chr	Dec	Chr	Dec	Chr	Dec	Chr	Dec	Chr
32	Space	48	0	64	@	80	P	96	‘	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	”	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	’	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	∧	110	n	126	∼
47	/	63	?	79	O	95	_	111	o		

where Dec indicates the decimal representation (which can be converted to binary and stored as a byte) and Chr stands for character.

2.5 Finite Field Arithmetic

Unlike working in the Euclidean space, addition (and subtraction) and multiplication in Galois Field requires additional steps.

2.5.1 Addition and Subtraction

An addition in Galois Field is pretty straightforward. Suppose $f(p)$ and $g(p)$ are polynomials in $gf(p^n)$. Let $A = a_{n-1}a_{n-2}\dots a_1a_0$, $B = b_{n-1}b_{n-2}\dots b_1b_0$, and $C = c_{n-1}c_{n-2}\dots c_1c_0$ be the coefficients of $f(p)$, $g(p)$, and $h(p) = f(p) + g(p)$ respectively. If a_k, b_k, and c_k are the coefficients of p^k in $f(p)$, $g(p)$, and $h(p)$ respectively then

$$c_k = a_k + b_k \ (mod\ p)$$

Similarly, if $h(p) = f(p) - g(p)$ then

$$c_k = a_k - b_k \ (mod\ p)$$

where $0 \leq k \leq n-1$. Since computer works in $gf(2^8)$, if a_k and b_k refer to the k^{th} bit in the bytes we wish to add then c_k, the k^{th} bit in the resulting

byte, is given by

$$c_k = a_k + b_k \,(mod\, 2)$$

Since $0 + 1 = 1 + 0 = 1 \,(mod\, 2) = 1$ and $0 + 0 = 0 \,(mod\, 2) = 1 + 1 = 2 \,(mod\, 2) = 0$, we may think of addition as exclusive-or operation which is also known as XOR operation. That is, XOR operation returns 0 if both entries are equal and returns 1 otherwise which also means that subtraction and addition is the same in Galois Field whose characteristic field is 2. Due to the nature of Galois Field, addition and subtraction of two bytes will not go any bigger than $11111111 = 255$, the biggest value one byte can store, and is therefore a safe operation.

Example

Suppose we are working in $gf(2^8)$, then $83 + 249$ is

$$\begin{aligned} 83 + 249 &= (2^6 + 2^4 + 2^1 + 2^0) + (2^7 + 2^6 + 2^5 + 2^4 + 2^3 \\ &= 2^7 + 2 \cdot 2^6 + 2^5 + 2 \cdot 2^4 + 2^3 + 2^1 + 2 \cdot 2^0 \\ &= 2^7 + 2^5 + 2^3 + 2^1 \\ &= 169 \end{aligned}$$

Alternatively, from binary numeral system perspective,

$$\begin{aligned} 83 + 249 &= 01010011 + 11111001 \\ &= 10101010 \\ &= 169 \end{aligned}$$

and the results coincide.

2.5.2 Multiplication and Multiplicative Inverse

Multiplication in Galois Field, however, requires more tedious work. Suppose $f(p)$ and $g(p)$ are polynomials in $gf(p^n)$ and let $m(p)$ be an irreducible polynomial (or a polynomial that cannot be factored) of degree at least n in $gf(p^n)$. We want $m(p)$ to be a polynomial of degree at least n so that the product of two $f(p)$ and $g(p)$ does not exceed $11111111 = 255$ as the product needs to be stored as a byte. If $h(p)$ denotes the resulting product then

$$h(p) = (f(p) \cdot g(p)) \,(mod\, m(p))$$

On the other hand, the multiplicative inverse of $f(p)$ is given by $a(p)$ such that

$$(f(p) \cdot a(p)) \,(mod\, m(p)) = 1$$

Note that figuring out the product of two polynomials and the multiplicative inverse of a polynomial requires both reducing coefficients modulo p and reducing polynomials modulo $m(p)$. The reduced polynomial can be calculated easily with long division while the best way to compute the multiplicative inverse is by using Extended Euclidean Algorithm. The details on the calculations in $gf(2^8)$ is best explained in the following example.

Example

Suppose we are working in $gf(2^8)$ and we take the irreducible polynomial modulo $m(p)$ to be $p^8 + p^6 + p^5 + p^1 + p^0$. To calculate $84 \cdot 13$, we need to go through several steps. First, we compute the product of the polynomial and reduce the coefficients modulo 2.

$$\begin{aligned} 84 \cdot 13 &= ((2^6 + 2^4 + 2^2) \cdot (2^3 + 2^2 + 2^0)) \ (mod\, m(p)) \\ &= (2^9 + 2^8 + 2^7 + 2 \cdot 2^6 + 2^5 + 2 \cdot 2^4 + 2^2) \ (mod\, m(p)) \\ &= (2^9 + 2^8 + 2^7 + 2^5 + 2^2) \ (mod\, m(p)) \end{aligned}$$

Then we use long division to compute the reduced polynomial as follows

Remainder	Quotient
$2^9 + 2^8 + 2^7 + 2^5 + 2^2$	
$2^8 + 2^6 + 2^5 + 2^1 + 2^0$	
2^0	$2^1 + 2^0$

Where the last entry in the first column is the product we seek for. Since the product is 1, it follows that 84 and 13 are multiplicative inverse pairs.

Example

Now pretend that we do not know the multiplicative inverse of 84. Then to calculate the multiplicative inverse we will use Extended Euclidean Algorithm. Unlike long division, we need to keep track of the auxiliary when we work with Extended Euclidean Algorithm as follows

Remainder	Quotient	Auxiliary
$2^8 + 2^6 + 2^5 + 2^1 + 2^0$		0
$2^6 + 2^4 + 2^2$		1
$2^5 + 2^4 + 2^1 + 2^0$	2^2	2^2
2^0	$2^1 + 2^0$	$2^3 + 2^2 + 1$

The first two rows in the Remainder column are always the modulo polynomial followed by the polynomial we wish to invert. The first two rows in

the Auxiliary are always 0 and 2^0. The remainder and the quotient in row n is then calculated from the division of the remainders in row $n-1$ and $n-2$ while the auxiliary in row n is given by the sum of the auxiliary in row $n-2$ and the product of the quotient and the auxiliary in row $n-1$ until the last remainder equals to 2^0. The final entry in Auxiliary happens to be the multiplicative inverse of the product, thus the multiplicative inverse of 84 is $2^3 + 2^2 + 2^0 = 13$ which agrees with the preceding example.

3 Application in Cryptography

The most popular and widely used application of Galois Field is in Cryptography. Since each byte of data are represented as a vector in a finite field, encryption and decryption using mathematical arithmetic is very straightforward and is easily manipulable.
In the 1970's, IBM developed Data Encryption Standard (DES). However, given that DES uses humble 56-bit key and technology advances rapidly, a supercomputer was able to break the key in less than 24 hours thus a more sophisticated algorithm was necessary. In 2001, Vincent Rijmen and John Daemon came up with a more complicated algorithm called Rijndael and it has been the Advanced Encryption Standard (AES) ever since.

3.1 Advanced Encryption Standard (AES)

Before data scrambling and encryption can begin, the data must first be arranged in a state or a matrix of bytes. The algorithm for Advanced Encryption Standard (AES) consists of smaller, sub-algorithms namely SubBytes, ShiftRows, MixColumns, and AddRoundKey, where each method will be explained in details below. Note that the following explanation only applies to AES with 128-bit key. The algorithm for other variations such as AES with 192-bit and 256-bit keys slightly differs.

3.1.1 State

AES breaks data into states or matrix of bytes of predetermined size and encrypt every state independently of each other. Putting together bytes into a state has no cryptographic significance, yet it is an important process without whom other operations cannot be conducted. In Rijndael with a 128-bit key, an array or a matrix with 4 rows and 4 columns is formed where each entry is a byte or 8 bits so that there are essentially 16 bytes in total. Additionally, we may also think of a 128-bit state space as a vector in $gf(2^8)^{16}$

where each component of the vector is a byte. If A denotes the said matrix and $a_{(i,j)}$ for $0 \leq i, j \leq 3$ refers to each element in the matrix then a state is defined to be

$$A = \begin{bmatrix} a_{(0,0)} & a_{(0,1)} & a_{(0,2)} & a_{(0,3)} \\ a_{(1,0)} & a_{(1,1)} & a_{(1,2)} & a_{(1,3)} \\ a_{(2,0)} & a_{(2,1)} & a_{(2,2)} & a_{(2,3)} \\ a_{(3,0)} & a_{(3,1)} & a_{(3,2)} & a_{(3,3)} \end{bmatrix}$$

3.1.2 SubBytes

In this method each byte, each element in the matrix is replaced using the Rijndael's S-Box. This method is broken down into two stages. In the first stage each byte is replaced with its multiplicative inverse. In case of a byte whose value is 0 which does not have a multiplicative inverse, the byte remains 0. The second stage involves performing invertible affine transformation on each byte. In this case, if $x = x_0x_1x_2 \ldots x_7x_8$ is a vector whose elements are the binary representation of the byte in ascending order of power, then the output would be $A \cdot x + b$ where A is an 8×8 matrix and b is a vector which represents the coefficients (again, in ascending order) of some constant. Rijndael specifies the modulo polynomial $m(p)$ for finding the multiplicative inverse to be $m(p) = p^8 + p^4 + p^3 + p^1 + p^0$ while the affine transformation is defined to be

$$A \cdot x + b = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

where $b = 2^6 + 2^5 + 2^1 + 2^0 = 99$. Note that since multiplicative inverse comes in pairs and the affine transformation is invertible, it is possible to revert scrambled data back to its original state so that decryption can be performed to retrieve the data. If $y = y_0y_1y_2 \ldots y_7y_8$ is the input byte, the

inverse affine transformation is as follow

$$C \cdot y + d = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

3.1.3 ShiftRows

ShiftRows is perhaps the simplest operation to scramble data. Like the name suggest, ShiftRows shifts row n to the left by $n-1$ unit, so that the first row remains unchanged while the second row is shifted to the left by 1, third row by 2, and fourth row by 3. That is, if we let A and A' to denote the state before and after performing ShiftRows then if A is given by

$$A = \begin{bmatrix} a_{(0,0)} & a_{(0,1)} & a_{(0,2)} & a_{(0,3)} \\ a_{(1,0)} & a_{(1,1)} & a_{(1,2)} & a_{(1,3)} \\ a_{(2,0)} & a_{(2,1)} & a_{(2,2)} & a_{(2,3)} \\ a_{(3,0)} & a_{(3,1)} & a_{(3,2)} & a_{(3,3)} \end{bmatrix}$$

then A' is given by

$$A' = \begin{bmatrix} a_{(0,0)} & a_{(0,1)} & a_{(0,2)} & a_{(0,3)} \\ a_{(1,1)} & a_{(1,2)} & a_{(1,3)} & a_{(1,0)} \\ a_{(2,2)} & a_{(2,3)} & a_{(2,0)} & a_{(2,1)} \\ a_{(3,3)} & a_{(3,0)} & a_{(3,1)} & a_{(3,2)} \end{bmatrix}$$

Where we can deduce by observation that this operation is easily invertible.

3.1.4 MixColumns

The next method messes with columns instead of rows. MixColumns takes each column in the state and perform linear transformation on it. Since each column has 4 rows, the matrix transformation has to be 4×4 and is defined as follow

$$M = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

And since linear transformation has an inverse, this operation is invertible. In fact, the matrix used to revert the state back to its original standing is given by

$$M^{-1} = \begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{bmatrix}$$

3.1.5 AddRoundKey

AddRoundKey is the most important stage as it is the stage that provides uniqueness to the encryption thus is inevitably a complex operation by itself. The output of AddRoundKey fully depends on the key or the password specified by the user. In this stage a subkey, which is the same size as the state, is computed from the main key using Rijndael's Key Schedule. Once a subkey is derived, the sum of the subkey and the state is calculated. This process consists of three parts: Rotate, Rcon, and SubByes. The first part is to rotate or to shift the bytes that form the keyword 8 bits to the left, which is similar to what happens to the second row in ShiftRows. The second part is to apply sub-operation called Rcon; And the third part is to perform Rijndael's S-Box whose details have been dissected in SubBytes. Another step of this operation is to expand the main key until we have enough subkeys. However this paper will not tackle this process further due to its complexity.

Example

Suppose we wish to encrypt a sentence or a string which we may think of as an array of ASCII characters. For simplicity sake, let us try to encrypt "Fun Cryptography" which consists of exactly 16 characters with AES. Converting each character to its respective ASCII code and assigning all bytes into a state yields

$$\begin{bmatrix} 70 & 117 & 110 & 32 \\ 67 & 114 & 121 & 112 \\ 116 & 111 & 103 & 114 \\ 97 & 112 & 104 & 121 \end{bmatrix} = \begin{bmatrix} 01000110 & 01110101 & 01101110 & 00010000 \\ 01000011 & 01110010 & 01111001 & 01110000 \\ 01110100 & 01101111 & 01100111 & 01110010 \\ 01100001 & 01110000 & 01101000 & 01111001 \end{bmatrix}$$

Again to keep the example simple, we will perform each sub-algorithm once. SubBytes then scramble the above state to

$$\begin{bmatrix} 01011010 & 10011101 & 10011111 & 10110111 \\ 00011010 & 01000000 & 10110110 & 01010001 \\ 10010010 & 10101000 & 10000101 & 01000000 \\ 11101111 & 01010001 & 01000101 & 10110110 \end{bmatrix}$$

followed by ShiftRows

$$\begin{bmatrix} 01011010 & 10011101 & 10011111 & 10110111 \\ 01000000 & 10110110 & 01010001 & 00011010 \\ 10000101 & 01000000 & 10010010 & 10101000 \\ 10110110 & 11101111 & 01010001 & 01000101 \end{bmatrix}$$

and lastly MixColumns, omitting AddRoundKey

$$\begin{bmatrix} 11100111 & 00011000 & 00100100 & 01110000 \\ 00101010 & 10101011 & 00111001 & 01100011 \\ 00010101 & 01100101 & 11110111 & 10100111 \\ 10101011 & 11110110 & 00000011 & 10100100 \end{bmatrix} = \begin{bmatrix} 231 & 24 & 36 & 112 \\ 42 & 171 & 57 & 99 \\ 21 & 101 & 247 & 167 \\ 171 & 246 & 3 & 164 \end{bmatrix}$$

which does not quite translate to a human readable sentence. The decryption process is as simple as reading the above steps backwards.

References

[1] Harris Nover, Algebraic Cryptanalysis of AES, 1-6.

[2] National Institute of Standards and Technology, Advanced Encryption Standard, FIPS 197 (2011).