



# Building an Algebraic Representation of AES in Sage

Thomas Gagne

University of Puget Sound Department of Mathematics and Computer Science

Adviser: Professor Rob Beezer



## Introduction

First developed in 2001, the Advanced Encryption Standard (AES) cipher is now one of the most commonly used encryption algorithms worldwide. However, the algebraically simple description of the AES leads some cryptographers to question whether an algebraic weakness in the cipher exists, which would be fatal to the security of the AES. This summer, I studied the algebraic qualities of the AES with the goal of implementing an algebraic representation of the cipher in the open source mathematical software system Sage. Through the steps described below, I wrote a Python class in the Sage source code which embodies the AES' algebraic components and provides tools for studying these components in contexts such as algebraic cryptography and in comparison to other algebraic ciphers. This enables further study of the algebraic properties of the cipher and makes this class a valuable addition to the Sage cryptography library.

## Description of AES

- AES is a block cipher operating on blocks of 128, 196, or 256 bits, arranged in eight bit matrices of dimension  $4 \times 4$ ,  $4 \times 6$ , or  $4 \times 8$  (for simplicity, this poster only considers  $4 \times 4$  matrices).
- Byte entries are arranged and labeled in a  $4 \times 4$  block as such:

|           |           |           |           |
|-----------|-----------|-----------|-----------|
| $A_{0,0}$ | $A_{0,1}$ | $A_{0,2}$ | $A_{0,3}$ |
| $A_{1,0}$ | $A_{1,1}$ | $A_{1,2}$ | $A_{1,3}$ |
| $A_{2,0}$ | $A_{2,1}$ | $A_{2,2}$ | $A_{2,3}$ |
| $A_{3,0}$ | $A_{3,1}$ | $A_{3,2}$ | $A_{3,3}$ |

- To encrypt a block, the cipher repeatedly passes it through a *round function* until the block is sufficiently obfuscated. A single round in AES consists of four *round component functions*:

1. SubBytes
2. ShiftRows
3. MixColumns
4. AddRoundKey

- Encryption additionally requires a *key* to build a unique  $4 \times 4$  block called a *round key* for each round. These are used to encrypt data such that it cannot be decrypted without the given key.

## Building Rijndael-GF

To build an algebraic representation of the AES (known as *Rijndael-GF*), we aim to be able to represent any entry of an encrypted block as a polynomial in terms of the entries of the input block and of the round keys. We construct this functionality in Sage through the below three steps:

1. Consider each entry of the blocks as an element of the finite field:

$$F = GF(2)[x]/(x^8 + x^4 + x^3 + x + 1)$$

Each bit string  $(b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0)$  is then represented as the below element in  $F$ :

$$x^7 \cdot b_7 + x^6 \cdot b_6 + x^5 \cdot b_5 + x^4 \cdot b_4 + x^3 \cdot b_3 + x^2 \cdot b_2 + x \cdot b_1 + x^0 \cdot b_0$$

This gives us a field  $F$  to build polynomials over.

2. For each round component function  $R$ , construct a polynomial  $p$  over  $F$  with entries of  $A$  as indeterminates such that  $p = R(A)_{i,j}$ , where  $A$  is a generic block matrix and  $0 \leq i, j \leq 3$ .
3. Implement this polynomial construction functionality in Sage for each round component function, then compose these functionalities to create polynomials corresponding to a whole encryption.

## Constructing The Polynomials

### 1. SubBytes

SubBytes is a bijective non-linear transformation operating on each individual entry of a block  $A$ . To transform each entry  $A_{i,j}$ , SubBytes first takes that entry's inverse in  $F$ , then passes the result through the below affine transformation over  $GF(2)^8$  to obtain the result  $\text{SubBytes}(A)_{i,j}$ .

$$\text{SubBytes}(A)_{i,j} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \times (A_{i,j})^{-1} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

To construct a polynomial over  $F$  equaling  $\text{SubBytes}(A)_{i,j}$  in terms of the entries of  $A$ , we do:

1. Take  $(A_{i,j})^{-1}$ .
2. Using the method `base_matrix_to_poly()` in Sage, we can build a polynomial which behaves identically to the above affine transformation. This polynomial  $p(x)$  is:

$$p(x) = 05 \cdot x + 09 \cdot x^2 + F9 \cdot x^4 + 25 \cdot x^8 + F4 \cdot x^{16} + 01 \cdot x^{32} + B5 \cdot x^{64} + 8F \cdot x^{128} + 63$$

3. Take  $p((A_{i,j})^{-1})$  to give the result:

$$\text{SubBytes}(A)_{i,j} = 05 \cdot (A_{i,j})^{254} + 09 \cdot (A_{i,j})^{253} + F9 \cdot (A_{i,j})^{251} + 25 \cdot (A_{i,j})^{247} + F4 \cdot (A_{i,j})^{239} + 01 \cdot (A_{i,j})^{223} + B5 \cdot (A_{i,j})^{191} + 8F \cdot (A_{i,j})^{127} + 63$$

### 2. ShiftRows

ShiftRows acts on a block  $A$  by rotating each row of the matrix to the left by as many places as that row's number (starting at 0).

|           |           |           |           |
|-----------|-----------|-----------|-----------|
| $A_{0,0}$ | $A_{0,1}$ | $A_{0,2}$ | $A_{0,3}$ |
| $A_{1,0}$ | $A_{1,1}$ | $A_{1,2}$ | $A_{1,3}$ |
| $A_{2,0}$ | $A_{2,1}$ | $A_{2,2}$ | $A_{2,3}$ |
| $A_{3,0}$ | $A_{3,1}$ | $A_{3,2}$ | $A_{3,3}$ |

 $\rightarrow$ 

|           |           |           |           |
|-----------|-----------|-----------|-----------|
| $A_{0,0}$ | $A_{0,1}$ | $A_{0,2}$ | $A_{0,3}$ |
| $A_{1,1}$ | $A_{1,2}$ | $A_{1,3}$ | $A_{1,0}$ |
| $A_{2,2}$ | $A_{2,3}$ | $A_{2,0}$ | $A_{2,1}$ |
| $A_{3,3}$ | $A_{3,0}$ | $A_{3,1}$ | $A_{3,2}$ |

To construct a polynomial over  $F$  equaling  $\text{ShiftRows}(A)_{i,j}$ , we simply take:

$$\text{ShiftRows}(A)_{i,j} = A_{i,(j-i) \bmod 4}$$

### 3. MixColumns

MixColumns acts on a block  $A$  by multiplying each column of  $A$  by a matrix over  $F$  in the below transformation:

$$\begin{bmatrix} B_{0,j} \\ B_{1,j} \\ B_{2,j} \\ B_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} A_{0,j} \\ A_{1,j} \\ A_{2,j} \\ A_{3,j} \end{bmatrix}$$

Hence, we construct a polynomial over  $F$  equaling  $\text{MixColumns}(A)_{i,j}$  as:

$$\text{MixColumns}(A)_{i,j} = \begin{cases} A_{0,j} \cdot 02 + A_{1,j} \cdot 03 + A_{2,j} \cdot 01 + A_{3,j} \cdot 01 & \text{if } i = 0 \\ A_{0,j} \cdot 01 + A_{1,j} \cdot 02 + A_{2,j} \cdot 03 + A_{3,j} \cdot 01 & \text{if } i = 1 \\ A_{0,j} \cdot 01 + A_{1,j} \cdot 01 + A_{2,j} \cdot 02 + A_{3,j} \cdot 03 & \text{if } i = 2 \\ A_{0,j} \cdot 03 + A_{1,j} \cdot 01 + A_{2,j} \cdot 01 + A_{3,j} \cdot 02 & \text{if } i = 3 \end{cases}$$

## 4. AddRoundKey

AddRoundKey is simply the entry-wise XORing of a byte matrix  $A$  and a round key byte matrix  $K$ . In Rijndael-GF though, XORing is simply addition in  $F$ , meaning that:

$$\text{AddRoundKey}(A)_{i,j} = A_{i,j} + K_{i,j}$$

## Composing these functionalities

With the polynomial representations of the round component functions fully described, we can now implement them in Sage and compose their functionalities to build a polynomial representation of the entire cipher. I did this by building a `RijndaelGF` class in Python and adding the below qualities to the class:

### 1. RijndaelGF.RoundComponentPolyConstr

- `RoundComponentPolyConstr` is a subclass of `RijndaelGF` which uses the methods described above to build polynomials for particular round component functions. Each created object represents a single round component function  $R$  and has a `__call__(i, j)` method which accepts an index  $i, j$  and returns the polynomial  $p = R(A)_{i,j}$ .

- The `RijndaelGF` class has a `RoundComponentPolyConstr` object for each round component function. These can be accessed with the getter methods:

- `RijndaelGF.sub_bytes_poly_constr()`
- `RijndaelGF.shift_rows_poly_constr()`
- `RijndaelGF.mix_columns_poly_constr()`
- `RijndaelGF.add_round_key_poly_constr()`

### 2. RijndaelGF.compose(f, g)

- `compose(f, g)` is a method which helps build polynomials corresponding to the composition of multiple round component functions.
- Given two `RoundComponentPolyConstr` objects  $f$  and  $g$  which correspond to the functions  $f$  and  $g$  respectively, `compose(f, g)` returns a `RoundComponentPolyConstr` corresponding to  $g \circ f$ .
- Repeated application of `compose` lets us build a `RoundComponentPolyConstr` object which corresponds to the whole cipher rather than a single round component function, which completes our goal of building an algebraic representation of the AES cipher in Sage.

## Code example

```
Sage: rgf = RijndaelGF(4, 4)
Sage: rgf.sub_bytes_poly_constr()(1, 2)
(x^2 + 1)*a12^254 +
(x^3 + 1)*a12^253 +
(x^7 + x^6 + x^5 + x^4 + x^3 + 1)*a12^251 +
(x^5 + x^2 + 1)*a12^247 +
(x^7 + x^6 + x^5 + x^4 + x^2)*a12^239 +
a12^223 +
(x^7 + x^5 + x^4 + x^2 + 1)*a12^191 +
(x^7 + x^3 + x^2 + x + 1)*a12^127 +
(x^6 + x^5 + x + 1)
Sage: rgf.shift_rows_poly_constr()(1, 2)
a13
Sage: rgf.mix_columns_poly_constr()(1, 2)
a02 + (x)*a12 + (x + 1)*a22 + a32
Sage: rgf.add_round_key_poly_constr()(1, 2)
a12 + k012
```