University of Puget Sound
Sound Ideas

Summer Research

Summer 2015

Building an Algebraic Representation of the AES in Sage

Thomas Gagne
tgagne@pugetsound.edu

Follow this and additional works at: http://soundideas.pugetsound.edu/summer_research

 Part of the Algebra Commons, and the Computer Security Commons

Recommended Citation
Gagne, Thomas, "Building an Algebraic Representation of the AES in Sage" (2015). *Summer Research*. Paper 243.
http://soundideas.pugetsound.edu/summer_research/243

This Article is brought to you for free and open access by Sound Ideas. It has been accepted for inclusion in Summer Research by an authorized administrator of Sound Ideas. For more information, please contact soundideas@pugetsound.edu.

Implementing an Algebraic Representation of the AES in Sage

Thomas Gagne
Department of Mathematics and Computer Science
University of Puget Sound

September 23, 2015

1 Introduction

Since its adoption in 2001 by NIST (National Institute of Standards and Technology) as the U.S. national standard for encrypting sensitive data, the Advanced Encryption Standard (also known as AES or Rijndael) cipher has become one of the most popular and widely used encryption ciphers worldwide. While no practically exploitable weaknesses have yet been shown to exist in the AES, the relatively algebraically simple description of the algorithm leads some cryptographers to theorize the existence of an algebraic weakness in the cipher. As a result, experimental algebraic cryptanalysis of the AES has become a topic of great interest in the cryptography community, as the discovery of an algebraic weakness would be devastating to the cipher's security. At the core of this search for potential algebraic weaknesses lies the tool that is an algebraic representation of the AES. An algebraic representation of the cipher is a system of mathematical equations which model the cipher's behavior and therefore can be used by researchers as a tool to study the algebraic properties of the cipher in a more mathematical environment. Despite being one of the primary tools of an algebraic cryptanalyst of the AES though, many algebraic representations are impractical by design. In particular, a fully generalized representation is usually far too cumbersome to work with by hand, while a more simplified representation is easier to work with but incurs a loss of subtle, yet crucial information about the algebraic properties of the AES which would otherwise be provided by a fully generalized representation.

The purpose of this project was to develop a tool to solve this challenge and allow researchers to work with a powerful and descriptive algebraic representation of the AES in a simple environment. This was done by providing a fully generalized algebraic representation of the AES in the mathematical software system Sage. Sage is a computational tool used by mathematicians, researchers, and students, and it provides a powerful environment and collection of tools for working with complex mathematical systems. By implementing a generalized representation in Sage, I aimed to provide this representation in an environment where users can utilize the powerful and useful tools provided by Sage as well as bypass the traditional difficulties of working with a fully generalized representation by allowing Sage to abstract the more challenging aspects of the representation and perform the more difficult work automatically. This allows researchers of the AES to maximize the utility of a generalized representation in a concise and efficient manner, making this project a valuable addition to Sage and to the cryptography community.

2 Description of the AES

For readers unfamiliar with the details of the AES, I provide a brief description of the cipher here which covers the basic details necessary for understanding how we construct a fully generalized algebraic representation later on. For a complete description of the cipher, I recommend Chapter 3 of [1].

The AES (also known as the Rijndael cipher) is a block cipher operating on matrix blocks with 8-bit entries of size $4 \times N_b$, where $4 \leq N_b \leq 8$ is the block length. It is a symmetric key cipher and requires a $4 \times N_k$ block as a key for encryption and decryption, where $4 \leq N_k \leq 8$ is the key length. To encrypt and decrypt data, the AES uses a substitution-permutation network to transform each block by passing each block through the specially designed AES round function a certain number of times. Depending on the block length N_b and the key length N_k, the AES determines the number of rounds, N_r, to apply by using the chart below:

		N_k				
		4	5	6	7	8
N_b	4	10	11	12	13	14
	5	11	11	12	13	14
	6	12	12	12	13	14
	7	13	13	13	13	14
	8	14	14	14	14	14

Each round of the AES consists of the below four steps, except for the final round which omits the `MixColumns` step to make the structure of the encryption algorithm and the decryption algorithm identical:

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

These four steps are known as the round component functions, and all except `AddRoundKey` operate on a single input block. To perform its functionality, `AddRoundKey` additionally requires a $4 \times N_b$ round key block, one of which is generated for each round from the original key via a separate `KeySchedule` function. To maintain consistency in notation though, I will write `AddRoundKey`(A) to denote the function's operation on the arbitrary block A and unless said elsewhere, I will be intentionally ambiguous about which round key is used as it is typically unimportant.

During encryption, the byte entries of a block are sometimes interpreted as belonging to the finite field:

$$F = \frac{GF(2)[x]}{x^8 + x^4 + x^3 + x + 1}$$

Each arbitrary byte $(b_7b_6b_5b_4b_3b_2b_1b_0)$ is described as corresponding to the below element in F:

$$x^7 \cdot b_7 + x^6 \cdot b_6 + x^5 \cdot b_5 + x^4 \cdot b_4 + x^3 \cdot b_3 + x^2 \cdot b_2 + x^1 \cdot b_1 + x^0 \cdot b_0$$

Because byte strings and the elements of F are easily interchangeable, I will typically represent the elements of F by their corresponding hex string representations. For example, the byte string `6F` corresponds to the element $x^6 + x^5 + x^3 + x^2 + x^1 + x^0$ of F and vice versa.

Below I describe the specific details of each of the four round component functions and their inverses:

2.1 SubBytes

SubBytes is a non-linear substitution transformation operating on each entry of a block individually. To transform each entry of a block, SubBytes first takes that element's inverse in F (mapping the element 0 to itself), then transforms the result by an affine transformation over $GF(2)^8$. To be exact:

$$\texttt{SubBytes}(A)_{i,j} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \times (A_{i,j})^{254} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

The inverse transformation $\texttt{SubBytes}^{-1}$ is represented by the below transformation:

$$\texttt{SubBytes}^{-1}(A)_{i,j} = \left(\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \times A_{i,j} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \right)^{254}$$

2.2 ShiftRows

ShiftRows operates on a block by cyclically shifting each row of the block a certain number of columns to the left. Given a block length N_b, the number of columns each row R_i is shifted by is found below:

N_b	R_0	R_1	R_2	R_3
4	0	1	2	3
5	0	1	2	3
6	0	1	2	3
7	0	1	2	4
8	0	1	3	4

The inverse transformation simply shifts the rows to the right the correct number of columns.

2.3 MixColumns

MixColumns operates on a block by interpreting each column $[A_{0,i}, A_{1,i}, A_{2,i}, A_{3,i}]$ of the block as the polynomial $y^3 \cdot A_{0,i} + y^2 \cdot A_{1,i} + y^1 \cdot A_{2,i} + y^0 \cdot A_{3,i}$ from the polynomial ring $F[y]$, then multiplying the resulting polynomial by the fixed polynomial $c(y) = \texttt{03}{\cdot}y^3{+}\texttt{01}{\cdot}y^2{+}\texttt{01}{\cdot}y^1{+}\texttt{02}$ modulo $(y^4 + 1)$.

The inverse transformation $\texttt{MixColumns}^{-1}$ acts similarly to the normal transformation except that the transformation instead multiplies polynomials by the fixed polynomial $d(y) = \texttt{0B}{\cdot}y^3{+}\texttt{OD}{\cdot}y^2{+}\texttt{09}{\cdot}y^1{+}\texttt{0E}$ modulo $(y^4 + 1)$.

2.4 `AddRoundKey`

`AddRoundKey` is the simple entry-wise XORing of the current block matrix with the round key generated for the current round. Because the XOR operation is its own inverse, `AddRoundKey` is also its own inverse.

3 Constructing the Algebraic Representation

When I set out to construct an algebraic representation of the AES, I aimed to be able to build a system of algebraic equations which behaved analogously to the whole cipher. The system of equations I constructed to build this algebraic representation follows the below form:

If C is a function operating on blocks which corresponds to the whole cipher function or just a component of it, then each entry of the output matrix $C(A)_{i,j}$ can be represented as a polynomial over F with variables being the entries of the generic input block A. Some polynomials of this form will additionally include variables which are round key entries. Because the `KeySchedule` generates the round keys from the entries of the original key block K, it is possible to represent each round key entry as a polynomial over F with variables being the entries of K. Hence, one possible representation of the whole cipher constructs polynomials equaling $C(A)_{i,j}$ with variables being the entries of the generic block A and the key block K. However, many forms of cryptanalysis are based on the assumption that round keys are independent and have no relation to the original key. To prevent excluding this generalized representation from being used in those forms of cryptanalysis, I decided to make the key entries of the polynomial $C(A)_{i,j}$ come from the entries of N_r independent round keys, which are denoted as $K^{(1)}, K^{(2)}, ..., K^{(N_r)}$.

Using this representation, the whole cipher becomes represented in the form of $4 \cdot N_b$ polynomials over F. In particular, these polynomials belong to the polynomial ring:

$$F[A_{0,0}, A_{0,1}, ..., A_{0,N_b}, A_{1,0}, ..., A_{3,N_b}, K^{(0)}_{0,0}, K^{(0)}_{0,1}, ..., K^{(0)}_{0,N_k}, K^{(0)}_{1,0}, K^{(r)}_{3,3}]$$

where A is a generic input block and $K^{(r)}_{i,j}$ is the i,jth entry of the rth generic round key.

To be able to construct these $4 \cdot N_b$ polynomials corresponding to arbitrary components of the cipher we must first be able to construct them for each of the four round component functions which make up a single round. Once we can construct these, we can link these polynomials together in order to build polynomials corresponding to more complex aspects of the cipher. Below I describe how the algebraic representation for each round component function was derived:

4 The Algebraic Representations of Round Component Functions

4.1 `SubBytes`

Let a be the function over F which corresponds to the inversion step of `SubBytes` and let b be the function over $GF(2)^8$ which corresponds to the affine transformation step of `SubBytes`. Building a polynomial representation of a is trivial, simply being $a(A)_{i,j} = (A_{i,j})^{254}$, since this maps each element to its inverse and maps 0 to 0. Building a polynomial representation of b is more difficult however, as we must construct a polynomial over F which behaves identically under evaluation to an affine transformation over $GF(2)^8$. A straightforward solution to this is to use Lagrangian interpolation to build this polynomial. Following this method, if we let $F = \{x_0, x_1, ..., x_{255}\}$ be an indexing of the elements of F and let $V : F \to GF(2)^8$ be a field

isomorphism from F to $GF(2)^8$, then a polynomial p over F which under evaluation behaves identically to b can be found as such:

$$p(\alpha) = \sum_{i=0}^{255} b(V(x_i)) \cdot l_i(x_i), \forall \alpha \in F$$

$$l_i(\alpha) = \prod_{\substack{j=0 \\ j \neq i}}^{255} \frac{\alpha - x_j}{x_i - x_j} = \frac{\alpha - x_0}{x_i - x_0} ... \frac{\alpha - x_{i-1}}{x_i - x_{j-1}} \cdot \frac{\alpha - x_{i+1}}{x_i - x_{j+1}} ... \frac{\alpha - x_{255}}{x_i - x_{255}}, \forall \alpha \in F$$

This method works because each function l_i has the behavior that:

$$l_i(x_k) = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{if } i \neq k \end{cases}$$

meaning that $p(x_k) = b(V(x_0)) \cdot 0 + ... + b(V(x_{k-1})) \cdot 0 + b(V(x_k)) \cdot 1 + b(V(x_{k+1})) \cdot 0 + ... + 0$, making p the polynomial we are interested in.

However, while researching how to calculate this polynomial I encountered a much more elegant and efficient method to calculate p, which is important if we ever wish to expand the generalization of this tool in the future to other ciphers or other forms of the AES. As some preliminaries, this method requires an understanding of the field trace function and of dual bases of finite fields, which I informally describe here. For readers interested in a more in-depth discussion of this subject, I recommend Appendix A of [1].

The trace function is defined as $Tr : GF(p^n) \to GF(p), Tr(x) = \sum_{i=0}^{n-1} x^{p^i}$. It is not difficult to prove that the trace function is linear over $GF(p)$, that is:

$$Tr(x + y) = Tr(x) + Tr(y), \forall x, y \in GF(p^n)$$

$$Tr(ax) = aTr(x), \forall a \in GF(p), \forall x \in GF(p^n)$$

For the second definition, we remind the reader that since $GF(p^n) \cong GF(p)^n$, we can represent the elements of $GF(p^n)$ as n-dimensional vectors over $GF(p)$ and that we can pick a basis of this vector space to be $\boldsymbol{e} = [e_0, e_1, ..., e_{n-1}]$ where $e_i \in GF(p^n), 0 \leq i \leq n-1$. Using this basis, we can represent each element a in $GF(p^n)$ as $a = \sum_{i=0}^{n-1} a_i e_i$ where $a_i \in GF(p), 0 \leq i \leq n-1$ and $[a_0, a_1, ..., a_{n-1}]$ are the coordinates of a as a vector. Given the basis $\boldsymbol{e}$, the dual basis of $\boldsymbol{e}$ is defined to be the basis $\boldsymbol{d} = [d_0, d_1, ..., d_{n-1}]$ such that for all $0 \leq i, j \leq n-1$, $Tr(d_i e_j) = \delta(i, j)$, where δ is the Kronecker delta function. The dual basis' primary utility comes from the below expression, which is derived easily since the trace function is linear over $GF(p)$:

$$Tr(d_j a) = Tr\left(d_j \sum_{i=0}^{n-1} a_i e_i\right) = \sum_{i=0}^{n-1} a_i Tr(d_j e_i) = a_j$$

Using the dual basis and the trace function we can easily calculate a polynomial p over F which behaves identically to the affine transformation b. To do this, let M be the 8×8 matrix used in the affine transformation b. Then, letting $b' = M \cdot \alpha$ where $\alpha \in GF(2)^8$:

$$b' = \sum_{i=0}^{7} b'_i e_i \tag{1}$$

and

$$\begin{aligned} b'_i &= \sum_{j=0}^{7} M_{i,j}\alpha_j \\ &= \left(\sum_{j=0}^{7} M_{i,j} Tr(\alpha d_j)\right) + c \\ &= \sum_{j=0}^{7} M_{i,j} \sum_{t=0}^{7} \alpha^{p^t} d_j^{p^t} \end{aligned} \tag{2}$$

Substituting (2) into (1), we arrive at:

$$\begin{aligned} b' &= \sum_{i=0}^{7}\sum_{j=0}^{7} M_{i,j} \sum_{t=0}^{7} \alpha^{p^t} d_j^{p^t} e_i \\ &= \sum_{t=0}^{7} \left(\sum_{i=0}^{7}\sum_{j=0}^{7} M_{i,j} d_j^{p^t} e_i\right) \alpha^{p^t} \end{aligned}$$

which gives the final result:

$$b = \sum_{t=0}^{7} \left(\sum_{i=0}^{7}\sum_{j=0}^{7} M_{i,j} d_j^{p^t} e_i\right) \alpha^{p^t} + c$$

where c is the added constant of the affine transformation. This method of calculating a polynomial corresponding to b is not only in some aspects simpler than the method involving Lagrangian interpolation, but it is also much more efficient, especially when using large fields. Because I hope this project can grow to involve other ciphers and other forms of AES which may involve much larger fields in the future, it is important that I provide this efficient method of creating these polynomials in my implementation. For this reason, I opted to use this algorithm to calculate the polynomial corresponding to the affine transformation step of `SubBytes`.

Using this algorithm, I found the polynomial corresponding to the affine transformation step of `SubBytes` to be:

$$\begin{aligned} p(\alpha) = {} & \texttt{05} \cdot \alpha + \texttt{09} \cdot \alpha^2 + \texttt{F9} \cdot \alpha^4 + \texttt{25} \cdot \alpha^8 + \\ & \texttt{F4} \cdot \alpha^{16} + \texttt{01} \cdot \alpha^{32} + \texttt{B5} \cdot \alpha^{64}\ \texttt{8F} \cdot \alpha^{128} + \texttt{63} \end{aligned}$$

Combining this with the inversion step of `SubBytes` gives:

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

I then used the algorithm once more to calculate the polynomial corresponding to `SubBytes`$^{-1}$ to be:

$$\begin{aligned}\text{SubBytes}^{-1}(A)_{i,j} = (05 \cdot (A_{i,j}) + FE \cdot (A_{i,j})^2 + 7F \cdot (A_{i,j})^4 + \\ 5A \cdot (A_{i,j})^8 + 78 \cdot (A_{i,j})^{16} + 59 \cdot (A_{i,j})^{32} + \\ DB \cdot (A_{i,j})^{64} + 8F \cdot (A_{i,j})^{128} + 05)^{255}\end{aligned}$$

4.2 `ShiftRows`

Compared to `SubBytes`, `ShiftRows` has a far simpler algebraic representation. Recall that the offset each row is shifted by is determined by the below chart:

N_b	R_0	R_1	R_2	R_3
4	0	1	2	3
5	0	1	2	3
6	0	1	2	3
7	0	1	2	4
8	0	1	3	4

Using this chart, it is extraordinarily simple to derive the algebraic representation of `ShiftRows` to be:

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

$$\text{ShiftRows}^{-1}(A)_{i,j} = A_{i,(j+R_i) \bmod N_b}$$

4.3 `MixColumns`

While the description of `MixColumns` might already seem to be fairly algebraic in nature, its description does not properly fit into the form of algebraic representation I decided upon for this implementation. Hence, to build an algebraic representation of `MixColumns` I show here how to represent it as a matrix transformation and then use this matrix to construct the equations in the algebraic representation. Recall that `MixColumns` transforms each column $[A_{i,0}, A_{i,1}, A_{i,2}, A_{i,3}]$ of a block into the column $[B_{i,0}, B_{i,1}, B_{i,2}, B_{i,3}]$ with the following transformation over the polynomial ring $F[\alpha]$:

$$B_{i,0} \cdot \alpha^0 + B_{i,1} \cdot \alpha^1 + B_{i,2} \cdot \alpha^2 + B_{i,3} \cdot \alpha^3 =$$

$$(A_{i,0} \cdot \alpha^0 + A_{i,1} \cdot \alpha^1 + A_{i,2} \cdot \alpha^2 + A_{i,3} \cdot \alpha^3) \cdot (02 \cdot \alpha^0 + 01 \cdot \alpha^1 + 01 \cdot \alpha^2 + 03 \cdot \alpha^3) (\bmod \alpha^4 + 1)$$

Multiplying this product out and sorting according to the resulting powers of α, this is equivalent to:

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

Using this matrix, I constructed the algebraic representation of `Mixcolumns` to be:

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

4.4 `AddRoundKey`

Of all the round component functions, `AddRoundKey` has the simplest algebraic representation. `AddRoundKey` is simply the entry-wise XORing of the generic block matrix A and the arbitrary generic round key $K^{(r)}$ and because XORing is simply addition in F, therefore:

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

Additionally, F has characteristic 2, making `AddRoundKey` its own inverse transformation. Hence, the algebraic representation of $\texttt{AddRoundKey}^{-1}$ is the same as above.

5 Composing these Algebraic Representations

With the algebraic representations of each of the round component functions constructed, all that remains is to construct a means of composing these representations. In particular, for any two functions f and g for which we have algebraic representations for, we must be able to construct an algebraic representation for $g \circ f$. The algorithm I used to perform this follows as such:

1. Suppose f and g are functions with known algebraic representations and that we wish to find a polynomial equaling $g(f(A))_{i,j}$ for an arbitrary input block A, where $0 \leq i \leq N_b$ and $0 \leq j \leq 4$.

2. Calculate

$$g(A)_{i,j} = A_{0,0} \cdot a_{0,0} + ... + A_{0,N_b} \cdot a_{0,N_b} + A_{1,0} \cdot a_{1,0} + ... + A_{3,N_b} \cdot a_{3,N_b} + K^{(0)}_{0,0} \cdot k^{(0)}_{0,0} + ... + K^{(N_r)}_{3,N_b} \cdot k^{(N_r)}_{3,N_b}$$

   where $a_{i,j}, k^{(r)}_{i,j} \in F$ for all $1 \leq i \leq 4$, $1 \leq j \leq N_b$, and $1 \leq r \leq N_r$.

3. Replace each $A_{i,j}$ in the above equation with $f(A)_{i,j}$, giving the result:

$$g(f(A))_{i,j} = f(A)_{0,0} \cdot a_{0,0} + ... + f(A)_{0,N_b} \cdot a_{0,N_b} + f(A)_{1,0} \cdot a_{1,0} + ... + f(A)_{3,N_b} \cdot a_{3,N_b} + K^{(r)}_{0,0} \cdot k^{(0)}_{0,0} + ... + K^{(N_r)}_{3,N_b} \cdot k^{(N_r)}_{3,N_b}$$

Note that because the round key entries are constant in AES, we do not replace the $K_{i,j}^{(r)}$ values in the expression with $f(K^{(r)})_{i,j}$.

Using this algorithm, we can easily compose the algebraic representations of each of the round component functions in any order to create an algebraic representation for the whole cipher or for just a component of it.

6 Implementation Details

The second step of this project involves implementing the algorithms which calculate these algebraic representations into Sage, as well as implementing the algorithm for composing representations and other algorithms for polynomial evaluation. Sage is built on top of the programming language Python and so to do this I made a Python class known as `RijndaelGF`, since my representation is closely related to the more generic version of AES known as Rijndael-GF. This class embodies all the tools and algorithms necessary for working with algebraic representations of AES by being constructible as a `RijndaelGF` object which then provides various methods for accessing these tools. Additionally, I added the functionality of being able to perform full encryptions and decryptions with a `RijndaelGF` object as a demonstration of the implementation's correctness.

6.1 `RijndaelGF.Round_Component_Poly_Constr`

Since for a function f, the algebraic representation of f simply means that we can create a polynomial equaling $f(A)_{i,j}$ in terms of the entries of A, it makes some sense to implement this functionality as a method of `RijndaelGF`. However, there are two issues caused by this. First, this would either force users to construct their own functions for building algebraic representations corresponding to composed functions, or the implementation would have to automatically build these functions. The first choice is problematic in that we are forcing the user to perform an action much better suited for Sage to automatically perform, while the second is problematic because such a created function reveals no information about its intended purpose and can be vague and difficult to use. Additionally, we should be able to work with the algebraic representation of a particular function in the same manner as with the algebraic representations of other functions. This is impossible to enforce if we implement this functionality as a method, whereas if we construct an object which embodies an algebraic representation we can enforce this much more easily.

For these reasons, I chose to embody the algebraic representations of functions with my own class called `Round_Component_Poly_Constr`. Objects of this class can be used to construct polynomials from the algebraic representations of a function by invoking the `__call__()` method. To be exact, if `f` is the `Round_Component_Poly_Constr` object corresponding to the function f, then we can calculate the polynomial $f(A)_{i,j}$ by simply calling `f(i,j)`. To provide the basics for working with representations of components of the cipher, I added a `Round_Component_Poly_Constr` object to the `RijndaelGF` class for each round component function. Additionally, this class is not restricted to just the round component functions, despite the name. As is described in the section below, one can use the `compose(f,g)` method in order to create new `Round_Component_Poly_Constr` objects corresponding to the composition of multiple functions. Alternatively, users can create their own method of the form `f_poly_constr(i,j)` which implements the algebraic representation for some other function f and returns $f(A)_{i,j}$ and pass this method to the class constructor as `Round_Component_Poly_Constr(f_poly_constr)` in order to create a `Round_Component_Poly_Constr` corresponding to the function f.

6.2 RijndaelGF.compose(f,g)

In order to build Round_Component_Poly_Constr objects corresponding to the composition of multiple functions, we require a compose(f,g) method. For reasons which will soon become obvious, I decided to implement the compose(f,g) method to have two distinct functionalities. First, if both f and g are Round_Component_Poly_Constr objects corresponding to the functions f and g respectively, then compose(f,g) will return a new Round_Component_Poly_Constr object corresponding to the function $g \circ f$. Second, if the argument f is a Round_Component_Poly_Constr object corresponding to the function f and the argument g is the polynomial $g(A)_{i,j}$, then compose(f,g) returns the polynomial $g(f(A))_{i,j}$. The reason for these two functionalities is that the first functionality is used for creation of new Round_Component_Poly_Constr objects while the second functionality is what actually creates the polynomials for the objects created by the first functionality. It is because these two functionalities are so tightly linked that I decided to have compose perform both these functionalities rather than creating separate methods.

My implementation of compose(f,g) is described in the below pseudocode. Note that the first case is an implementation of the algorithm described in Section 5.

```
def compose(f, g):
        if is_polynomial(g):
                f_vals = [f(i,j) for i in range(4) for j in range(Nb)]
                return g.evaluate_variables(f_vals)
        else:
                gof = lambda i, j: return compose(f, g(i,j))
                return new Round_Component_Poly_Constr(gof)
```

A particularly useful feature of this implementation of compose is that the actual calculation of polynomials is delayed when creating new Round_Component_Poly_Constr objects. This makes the computational cost of creating a Round_Component_Poly_Constr object corresponding to many composed functions trivial, as it delays polynomial calculations until the user explicitly requests it.

6.3 Additional Details of the Implementation

Using the provided Round_Component_Poly_Constr objects and the compose method of the RijndaelGF class, it is now possible to use this implementation to create and interact with a fully generalized algebraic representation of the entire cipher as well as representations of smaller components of the cipher. While my primary goal for this project was accomplished at this point, I recognized that the RijndaelGF class required various other components to be implemented in order to make the constructed algebraic representations easier to work with and more powerful. I describe some of the more important components here.

Currently, while a user is able to construct algebraic representations corresponding to any imaginable component of the cipher, a created Round_Component_Poly_Constr object cannot be used outside a purely theoretical application. In order to allow users to use these objects in a practical manner, I added the method RijndaelGF.apply_poly(state, poly_constr). This method accepts two arguments: a block matrix called state, and a Round_Component_Poly_Constr object called poly_constr, and this method returns a new block matrix where each i,jth entry of the matrix equals the polynomial poly_constr(i,j) evaluated by setting its variables equal to the entries of state. In short, if poly_constr corresponds to the function

f, then `apply_poly(state, poly_constr)` returns f(`state`). This allows users to construct algebraic representations corresponding to certain functions, then be able to pass block matrices through these functions without ever having to explicitly define the function beyond its algebraic representation. In addition, the argument `state` can be a generic state matrix, meaning that `poly_constr(state)` returns a matrix of polynomials which describes the entire algebraic representation. This gives us a succinct way to generate a complete description of an algebraic representation of a function. Finally, this method makes it easy to add the functionality of the round component functions to the `RijndaelGF` class, as is shown in below example:

```
def sub_bytes(self, state):
        return self.apply_poly(state, self.sub_bytes_poly_constr())
```

The second addition I added to this implementation deals with key variables. Previously, I pointed out that because the round key entries are generated from the original key, it is therefore possible to express all round key entries as polynomials with variables being the entries of the original key. As many forms of cryptanalysis assume round key entries to be independent however, I decided to have this implementation use independent round key entries by default. In order to preserve the functionality of representing round key entries in terms of the original key though, I added an `expand_key_poly(row, col, round)` method to the `RijndaelGF` class to do this. When called, `expand_key_poly(row, col, round)` returns a polynomial composed of entries of the original key which equals the `row`th, `col`th entry of the `round`th round key. Although I have not described the details of how the `KeySchedule` generates round keys in this paper, this method simply backtracks through the `KeySchedule` algorithm to arrive at the answer.

The final important addition I added to this implementation includes various methods which allow the user to perform full encryptions and decryptions. This functionality is built through repeated application of the `apply_poly` method with the built-in `Round_Component_Poly_Constr` objects for each of the round component functions. There are two benefits to implementing this functionality: first, we can input the official AES testing vectors and prove the correctness of this implementation. Second, since `apply_poly` accepts generic block matrices as input and since the method `encrypt` is built upon this method, we can simply call `encrypt(generic_block, generic_key)` to get a block matrix whose every entry is a polynomial representing the application of the entire cipher. While this is likely too computationally intensive to ever be practically useful, it acts as a practical example of how the structure of this implementation allows us to perform these powerful operations by reusing other aspects of the implementation.

7 Conclusion

By adding this implementation to Sage, I have successfully completed my goal of providing users the ability to generate a fully generalized algebraic representation of the whole AES cipher and its subcomponents in a simple and powerful mathematical computing environment. Additionally, I have allowed room for this project to grow to include representations of other variants of the AES by making the representation as generalized as possible as well as allowing this representation's internal behavior to be modified to fit these other representations. Furthermore, I have structured this implementation such that it is not only a useful tool in itself but that it also establishes a framework which can be used as a general model for implementing the algebraic representations of other ciphers and algorithms in Sage. For these reasons, I believe the result of this project is both a powerful and useful tool for algebraic cryptanalysts and researchers of the AES as well as a valuable contribution to Sage and its cryptographic libraries.

References

[1] Joan Daemen and Vincent Rijmen, *The Design of Rijndael*. Springer-Verlag, 2002. ISBN: 3 540 42580 2

[2] Federal Information Processing Standards Publication 197, *Announcing the ADVANCED ENCRYPTION STANDARD (AES)*. United States National Institute of Standards and Technology (NIST), 2001.

[3] Niels Ferguson, Richard Schroeppel, and Doug Whiting, *A simple algebraic representation of Rijndael*. LNCS 2259. Springer Verlag, 2001.

[4] S. Murphy and M.J.B. Robshaw, *Essential Algebraic Structure Within the AES*. LNCS 2442. ISBN: 3 540 45708 9