



Universiteit
Leiden
The Netherlands

BSc Computer Science

Representing Block Ciphers
in a Computer Algebra System

Dirk van den Beuken

Supervisors:
Nuša Zidarič & Nele Mentens

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

15/07/2025

Abstract

In this thesis, a package of a Computer Algebra System will be designed used to create symbolic representations of Substitution-Permutation Networks. These representations can be derived using functions of the package. This package will initially include options to retrieve symbolic AES expressions. These expressions can be generated both in finite field and vector form, allowing multiple ways to visualize and manipulate the internal operations of the cipher.

This thesis will be useful for future research, since this package is designed to be extended by other ciphers. This gives the ability to compare ciphers reciprocally. Also, Boolean Functions can be added, allowing for the possibility to analyze cryptographic properties like non-linearity, algebraic degree and algebraic immunity, providing a contribution to the field of cryptography.

Contents

1	Introduction	1
2	Background and Related Work	2
2.1	Related Work and Motivation	2
2.2	Mathematical Background	4
2.2.1	Finite Fields	4
2.2.2	Vector Spaces	5
2.2.3	Multivariate Functions	6
2.2.4	Boolean Functions	7
2.3	Cryptography	7
2.3.1	Substitution-Permutation Network	7
2.3.2	Advanced Encryption Standard	8
2.3.3	Cryptographic Properties	11
2.4	Symbolic Computation and Computer Algebra Systems	12
2.4.1	Groups Algorithms Programming	12
2.4.2	Creating GAP packages	13
3	New GAP package: SYMSPN	14
3.1	Introduction	14
3.2	Designing SYMSPN	14
3.2.1	Structure	14
3.2.2	Record of Symbols	17
3.2.3	Symbolic AES approach	19
3.3	Symbolic AES in a Finite Field	20
3.4	Symbolic AES in a Vector Space	25
3.5	Testing	33
3.6	Profiling and discussion	37
4	Future Work and Conclusion	42
4.1	Symbolic AES as a Boolean Function	42
4.2	GAP functions for analysis	42
4.3	Modelling Similar Ciphers	42
4.4	Conclusion	42
	References	44
	Appendix	45
	A. Creating a GAP package	45
	B. SYMSPN package code	48
	C. Example Usecases of SYMSPN	54

1 Introduction

In cryptography, the focus is on creating cryptographic algorithms that can encrypt plaintext data, and consequently decrypt it. These algorithms are called *ciphers*. Ciphers operate in such a way that data can be encrypted and decrypted efficiently, while ensuring adequate cryptographic strength.

These ciphers are used to provide confidentiality. Confidentiality states that information should only be accessible to authorized parties and remain hidden from unauthorized ones. A cipher accomplishes this by transforming readable data (plaintext) into an unreadable form (ciphertext). In case an outside party intercepts your data, they can only observe the ciphertext, which is not understandable without the secret key.

Cryptography is mainly a mathematical discipline. Finite fields are often used to define the internal operations of ciphers. Finite fields are algebraic structures with a finite number of elements, where addition and multiplication are defined. In binary fields, subtraction corresponds to addition, since adding an element to itself results in zero. Division is effectively implemented as multiplication by the multiplicative inverse of an element. Given the field's closure properties, this makes explicit division operations unnecessary. Elements of finite fields $\mathbb{F}_{2^n}$ can be represented as vectors over $\mathbb{F}_2$ as well, allowing both finite field-based and vector space interpretations.

In this thesis, the following research question is answered:

“How can a computer algebra system package be designed to generate symbolic representations of block ciphers?”

To fulfill this research question, the aim is to create a package in a computer algebra system called GAP (Groups, Algorithms, Programming). This package will symbolically express Substitution-Permutation Networks (SPN). This package should be generalized and scalable so that new ciphers can easily be added to the package. GAP is used as the computational algebra system because it includes its own programming language and it is specifically designed for working with groups, finite fields, and other discrete algebraic structures. Furthermore, GAP is widely used in the scientific domain. Within GAP, my package called SYMSPN (SYMBOLic SPN) will be created, which expresses SPN computations symbolically.

The SYMSPN package will initially be equipped with symbolical expressions of the *Advanced Encryption Standard* (AES) cipher. Both the finite field representations of AES, as well as the vector space representations of AES will be displayed, providing multiple ways to examine AES. The Advanced Encryption Standard is one of the most widely used ciphers in the world. This cipher was standardized by NIST in 2001, and is still the standard today. It was chosen because of its excellent security, high performance and flexibility.

This package will be useful for multiple reasons. Firstly, it allows us to visualize and manipulate SPN (components) symbolically. It is designed to be extended by future projects as well: (i) Boolean representations can be added which gives the opportunity to test cryptographic properties and (ii) other ciphers can also be added so that comparisons between ciphers could take place. This will be useful in the face of *Post Quantum Cryptography* (PQC).

2 Background and Related Work

2.1 Related Work and Motivation

As of today, a lot has been written about mathematical structures that are used in cryptography. Some papers give an introduction about group theory and how it relates to cryptography [Sho22]. These sources are used to familiarize with the basic mathematical knowledge needed to work in the realm of cryptography. *Handbook of Finite Fields* talks at length about groups, finite fields and their implications [MP13]. This book mainly contains definitions about these essential mathematical structures and it provides information about cryptographic properties and Substitution-Permutation Networks as well. Therefore, this book is used as a strong foundation in cryptography today. Because the material in this book is discussed comprehensively, it is valuable as a reference but also for understanding finite field arithmetic and the role it plays in block ciphers.

One very important block cipher that has been created is the *Advanced Encryption Standard* (AES). The book *The Design of Rijndael* talks at length about the AES cipher [DR02]. This includes an overview of how AES has been designed, its behaviour, its functions, but also about the way AES can be expressed. This will be the main source for integrating the AES cipher in the package, since this book focuses on the structure of AES, which will be expressed symbolically. It is especially valuable because it was written by the original designers of AES, providing firsthand insight into their choices and rationale. Another book about AES will also be used [RS05]. It provides an overview of the state of AES research and development four years after its standardization. It also discusses its strengths and weaknesses. The value of this book lies in its comprehensive review of real-world AES performance and its exploration of both structural and algebraic attacks, offering a broader cryptographic context about the cipher.

Other sources will also be used to, among other things, formulate definitions and help in creating cryptographic expressions. One paper [MO24] discusses Boolean functions known as “weightwise affine” and “weightwise quadratic” and investigates their properties. It will mainly be used to extract definitions: Algebraic Normal Form is formally introduced, alongside Boolean functions, both of which are central in describing cryptographic transformations. Understanding these structures is essential for the symbolic representation of SPNs. Another paper [ZMGA24] explores the development of Welch-Gong stream ciphers and delves into expressing cryptographic operations—such as nonlinear transformations—as polynomials over finite fields. This will be especially useful for constructing symbolic expressions in my package. Furthermore, the techniques shown illustrate how to manipulate finite field operations for both theoretical and implementation purposes, which is highly relevant for symbolic computation environments like GAP. Finally, a third paper [Zid20] will be studied to better understand how finite field operations can be symbolically represented and manipulated, since it covers the automated generation of hardware designs for cryptographic algorithms based on finite field arithmetic.

To symbolically express SPNs, a GAP package will be created [Lin07]. This program is great for computational discrete algebra, with a particular emphasis on Computational Group Theory, as documented at www.gap-system.org/. GAP is designed to work well with algebraic structures such as groups, rings, and fields, making it ideal for cryptographic representations. Its language

is optimized for symbolic computation, which allows for the precise handling of algebraic expressions. Users can define custom functions, automate calculations, and explore algebraic properties interactively. GAP also includes an extensive library of algorithms and data structures relevant to finite algebraic systems. This makes it a strong foundation for implementing and analyzing cryptographic schemes. Many GAP packages have already been created, to add additional functionality. These include the FSR package [ZAG19] and the FFCSA package [Zid23]. The FSR (Feedback Shift Register) package contains functions for working with (non-)linear feedback shift registers and filtering functions. It also contains many useful functions like `ReduceMonomialsOverField`, which reduces all exponents of a polynomial such that it fits into a particular field. The FFCSA (Finite Field Constructions, Search and Algorithms) package complements this by focusing on finite field arithmetic, which underpins most non-linear transformations in SPNs. It provides tools for constructing finite fields, generating different bases, and producing matrix and polynomial expressions for arithmetic operations like multiplication. This is vital for SPNs that operate over fields, such as in AES. FFCSA’s ability to output symbolic expressions for field operations makes it a great choice for creating or analyzing SPN layers. FFCSA was designed to enable Design Space Exploration for cryptographic algorithms targeting hardware implementations [ZGA⁺23]. Through its symbolic computation capabilities, FFCSA can generate low-level expressions for submodules such as multipliers, squarers and constant multipliers, which are essential building blocks for SPN layers. Overall, by combining symbolic manipulation, arithmetic algorithm selection mechanisms and support for various field representations, FFCSA extends GAP’s utility from algebraic modeling to practical hardware-oriented representations.

What has not been done before is creating a package that can symbolically represent SPNs, which this thesis is addressing. During this thesis, the above sources will be used to create a new GAP package called **SYMSPN**, which will utilize the FFCSA and FSR packages when constructing symbolic SPN expressions. Initially, the package will be capable of expressing AES in both finite fields and vector spaces. **SYMSPN** is created with scalability in mind, such that it could be extended by adding other symbolic expressions of SPNs. This means that the package is designed including comprehensive error handling, while the initial functions can be utilized for future SPN-implementations as well. This package will primarily visualize the way block ciphers operate, making their properties more comprehensible.

2.2 Mathematical Background

2.2.1 Finite Fields

A finite field is a set with addition and multiplication operations that satisfies field axioms. To understand finite fields technically, groups have to be defined first.

Definition 1.1 [Sho22] A group G is a set of elements with a binary operation $\bullet$ that satisfies the following properties:

1. Closure: If $a, b \in G$, then $a \bullet b \in G$;
 2. Associativity: For every $a, b, c \in G$, we have $(a \bullet b) \bullet c = a \bullet (b \bullet c)$;
 3. Identity: There exists a $i \in G$, such that $i \bullet a = a \bullet i = a$;
 4. Inverse: For every $a \in G$, there exists an inverse element a^{-1} , such that $a^{-1} \bullet a = a \bullet a^{-1} = i$
- If a group also satisfies the commutative property, it is called an *Abelian group*, which states that for all $a, b \in G$, $a \bullet b = b \bullet a$ also holds.

Finite Fields, also called Galois Fields, are fields that have a finite number of elements. These fields are fundamental parts of cryptography, because they enable arithmetic in a well-defined, closed system with a finite number of elements. The order of a finite field $\mathbb{F}$ represents the total number of distinct elements in $\mathbb{F}$. Here, $\mathbb{F}_q$ is the notation used to denote that the finite field contains q elements, where q is a prime value with a positive exponent. By using modulo operations, we can ensure that the field is closed under $\cdot$ and $+$ operations.

Definition 1.2 [Zid20] A (multiplicative) group G is cyclic, if there exists an element $\alpha \in G$ such that for any $a \in G$, there exists an integer i for which $a = \alpha^i$.
The element α is called the generator of the cyclic group and we write $G = \langle \alpha \rangle = \{\alpha^i; i \in \mathbb{Z}\}$.
Given identity element e is a finite group of order n , then $\alpha \in \{e, \alpha, \alpha^2, \dots, \alpha^{n-1}\}$ and $\text{ord}(\alpha) = n$.
That is $\alpha^n = e$.

The multiplicative group is cyclic, as every possible nonzero element within the finite field can be generated using a generator α . Generators don't have to generate all possible elements in a field. For that, primitive generators have to be used:

Definition 1.3 [MP13] An element $\alpha \in \mathbb{F}_q$ which multiplicatively generates the group $\mathbb{F}_q^*$ of all nonzero elements of the field $\mathbb{F}_q$ is a primitive element.

In the case of definition 1.3, an element $\alpha \in \mathbb{F}_q$ is called a primitive element if it can generate every nonzero element $x \in \mathbb{F}_q$ using an integer i such that $\alpha^i = x$.

Definition 1.4 [Zid20] A nonempty set F , together with two binary operations addition $+$ and multiplication $\cdot$ is a field $\mathbb{F} = (F, +, \cdot)$, if:

1. $(F, +)$ is a commutative group with (additive) identity 0
2. $(F \setminus \{0\}, \cdot)$ is a commutative group with (multiplicative) identity 1
3. multiplication is distributive over addition: $\alpha \cdot (\beta + \gamma) = \alpha \cdot \beta + \alpha \cdot \gamma \quad \forall \alpha, \beta, \gamma \in F$

Note that subtraction and division are not separate operations in finite fields. Rather, they are accomplished through addition and multiplication by the respective additive and multiplicative

inverses. Furthermore, for binary fields, subtraction equals addition modulo 2. Thus, addition and multiplication alone are sufficient.

Finite Fields can also be extended. A finite field $\mathbb{F}_2$ can be extended to $\mathbb{F}_{2^4}$ by using an irreducible polynomial of degree 4. For example, $f(x) = x^4 + x + 1$ can be used to perform a finite field extension $\mathbb{F}_2 \xrightarrow{x^4+x+1} \mathbb{F}_{2^4}$. By using finite field extensions, we can build upon smaller finite fields. A finite field which contains two elements - like computers that calculate with 0's and 1's - can thereby be expanded while ensuring that the elements in the finite field can still be represented in a binary notation.

An example of a finite field is $\mathbb{F}_{2^2}$, which can be constructed as $\mathbb{F}_2[x]/(f(x))$ using an irreducible polynomial of degree 2 over $\mathbb{F}_2$, such as $\mathbb{F}_2 \xrightarrow{x^2+x+1} \mathbb{F}_{2^2}$. The elements of $\mathbb{F}_{2^2}$ are $\{\alpha + 1, \alpha, 1, 0\}$, where α is a root of $f(x)$.

When working with extended finite fields, bases are used to represent elements. A polynomial basis is a basis of the form $\{\alpha^{n-1}, \dots, \alpha, 1\}$, where α is a root of an irreducible polynomial of degree n over $\mathbb{F}_q$. Every element in the field extension can be uniquely written as a linear combination of these basis elements along with coefficients.

2.2.2 Vector Spaces

Let $(F, +, \cdot)$ be a field, with unit element 1, and let $(V, \vdash)$ be an Abelian group. Let $\bullet$ be an operation on elements of F and V :

$$\bullet : F \times V \rightarrow V$$

Definition 1.5 [DR02]

The structure $(F, V, \vdash, +, \cdot, \bullet)$ is a vector space over F if the following conditions are satisfied:

1. Distributivity:

$$\forall a \in F, \forall v, w \in V : a \bullet (v \vdash w) = (a \bullet v) \vdash (a \bullet w)$$

$$\forall a, b \in F, \forall v \in V : (a + b) \bullet v = (a \bullet v) \vdash (b \bullet v)$$

2. Associativity: $\forall a, b \in F, \forall v \in V : (a \cdot b) \bullet v = a \bullet (b \bullet v)$

3. Neutral element: $\forall v \in V : 1 \bullet v = v$

The elements of V are called *vectors*, and the elements of F are the *scalars*. The operation $\vdash$ is called the *vector addition*, and $\bullet$ is the *scalar multiplication*.

Vector spaces are fundamental in linear algebra, since they allow us to express finite field elements in binary notation. They consist of sets of elements called vectors, and are expressed as $\mathbb{F}_q^n$, where q denotes the amount of possible elements for every dimension, while n denotes the amount of dimensions. The finite field $\mathbb{F}_{q^n}$ can be viewed as a vector space of dimension n over $\mathbb{F}_q$, and is thereby isomorphic to $\mathbb{F}_q^n$ as a vector space:

$$\mathbb{F}_{q^n} \cong \mathbb{F}_q^n$$

To represent elements of the vector space, a polynomial basis is needed. If we have a vector space $\mathbb{F}_2^8$, for example, we can express the elements in terms of the following basis: $B = \{x^7, x^6, x^5, x^4, x^3, x^2, x, 1\}$

Using this basis, we can express the coordinates from $\mathbb{F}_2 = \{0, 1\}$, with $n = 8$ dimensions. If a hexadecimal value, say $0x63$, is shown, we can express that in vector space $\mathbb{F}_2^8$ as well:

$$\begin{array}{lcl} \text{Polynomial Basis Element:} & x^7 & x^6 & x^5 & x^4 & x^3 & x^2 & x^1 & x^0 & = & B \\ \text{Binary coefficient:} & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & \in & \mathbb{F}_2^8 \end{array}$$

By exercising the dot product we can retrieve the corresponding element in the $\mathbb{F}_{2^8}$ finite field:

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x^7 \\ x^6 \\ x^5 \\ x^4 \\ x^3 \\ x^2 \\ x^1 \\ x^0 \end{bmatrix} = x^6 + x^5 + x + 1 \in \mathbb{F}_{2^8}$$

2.2.3 Multivariate Functions

Multivariate functions are functions that take in multiple variables as input.

Definition 1.6 [Zid20]

Let $\mathbb{F}$ be a finite field with q elements, where q is a prime or a prime power. A multivariate function in t variables $x_0, x_1, \dots, x_{t-1}$ is defined as follows:

$$\begin{aligned} f : \mathbb{F}_q^t &\rightarrow \mathbb{F}_q \\ f(x_0, x_1, \dots, x_{t-1}) &= \sum_{\forall (i_0, i_1, \dots, i_{t-1}) \in \mathbb{Z}_q^t} c_{i_0, i_1, \dots, i_{t-1}} x_0^{i_0} x_1^{i_1} \dots x_{t-1}^{i_{t-1}} \end{aligned}$$

With coefficients $c_{i_0, i_1, \dots, i_{t-1}} \in \mathbb{F}_q$ and where $i_j \in \mathbb{Z}_q$ for $0 \leq j < t$. The sum runs over all possible monomials $x_0^{i_0} x_1^{i_1} \dots x_{t-1}^{i_{t-1}}$, where $\forall x \in \mathbb{F}_q : x^q = x$.

An example of a multivariate function is:

$$f(x, y) = x * \alpha^2 + y^3 * \alpha + 1$$

An important representation of multivariate functions over the finite field $\mathbb{F}_2$ is the Algebraic Normal Form (ANF). In ANF, a function $f : \mathbb{F}_2^t \rightarrow \mathbb{F}_2$ is expressed as a sum of terms, each being a product of distinct variables without exponents greater than one.

Multivariate functions in ANF are central to cryptography due to their clear structure. They allow for easy analysis of cryptographic properties such as algebraic immunity, algebraic degree and non-linearity. Their use can consequently enhance *confusion* and *diffusion*, making them ideal for secure cipher design.

2.2.4 Boolean Functions

A Boolean Function is a function that maps binary inputs to binary outputs: $f : \mathbb{F}_2^n \rightarrow \mathbb{F}_2$

Definition [MP13]

Let n be a positive integer.

- 1) An n -variable Boolean function (or Boolean function in dimension n) is a function defined over the vector space $\mathbb{F}_2^n$ and valued in $\mathbb{F}_2$. *Notation:* $f(x)$, where $x = (x_1, \dots, x_n) \in \mathbb{F}_2^n$.
- 2) The domain $\mathbb{F}_2^n$ can be endowed with the structure of the field $\mathbb{F}_{2^n}$. *Same notation:* $f(x)$, where $x \in \mathbb{F}_{2^n}$.

Boolean functions can be expressed in a variety of ways, including truth tables, algebraic normal form, or as multivariate polynomials over $\mathbb{F}_2$. A simple example of a Boolean function expressed in ANF:

$$f(x_1, x_2, x_3) = x_1x_2 + x_3$$

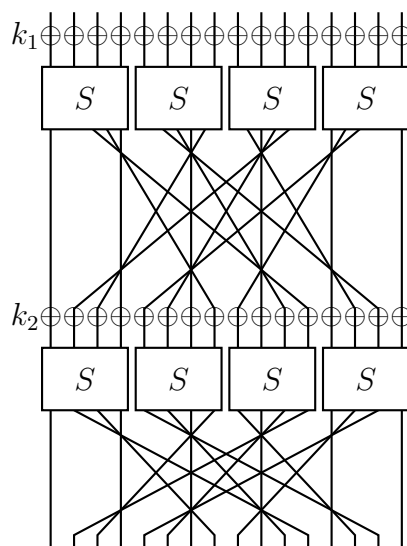
Boolean functions naturally fit into the framework of vector spaces over the field $\mathbb{F}_2$. Since the domain $\mathbb{F}_2^n$ is a n -dimensional vector space of binary vectors, each input to a Boolean function can be viewed as a vector of bits.

Boolean Functions are useful in cryptography, since they enable the construction of non-linear transformations that provide confusion. In block ciphers like AES, Boolean functions are used within substitution boxes, where each output bit can be modeled as a Boolean function of the input bits. By operating over the finite field $\mathbb{F}_2$, Boolean functions can be used to easily construct digital systems that operate on bit-level.

2.3 Cryptography

2.3.1 Substitution-Permutation Network

A Substitution-Permutation Network (SPN) is a cryptographic framework used in block ciphers to transform plaintext into ciphertext through multiple rounds of substitution and permutation operations.



Example of a Substitution-Permutation Network

Definition [MP13]

In an SPN, each round function consists of a few successive layers. The input to a substitution layer is divided into small blocks of bits, say blocks of eight bits each. An S-box is applied to each block. Each S-box is a bijective map, so that the entire substitution layer is also a bijective map. The effect of a substitution layer is local in the sense that an output bit in a particular position depends only on a few of the input bits in its nearby positions. This local effect is compensated by having a permutation layer which permutes its input bits. The round key is usually incorporated at the beginning or at the end of the round function.

Substitution-Permutation Networks are useful because they provide a strong and efficient structure for building secure block ciphers. By utilizing substitution, confusion is accomplished, while diffusion is accomplished using permutations.

2.3.2 Advanced Encryption Standard

The Advanced Encryption Standard (AES) is a symmetric encryption algorithm used to encrypt and decrypt digital data. In 2001, it was adopted by the National Institute of Standards and Technology (NIST) as the new encryption standard, replacing its predecessor: Data Encryption Standard (DES). Due to its relatively short key length, DES had become vulnerable to brute-force attacks [DR02].

The outlining of AES that follows is based on *The Design of Rijndael*[DR02]:

AES is based on the Rijndahl cipher, designed by Joan Daemen and Vincent Rijmen. AES supports key sizes of 128, 192 or 256 bits, unlike the 56-bit key that DES uses. The possibilities for different keys provide flexible levels of security. The block size in AES is fixed at 128 bits, and encryption occurs over multiple rounds that depend on the key size: 10 rounds are used for 128-bit keys, 12 rounds for 192-bit keys and 14 rounds for 256-bit keys.

Each round transforms the initial state through a combination of substitutions, permutations, and arithmetic in a finite field $\mathbb{F}_2 \xrightarrow{f} \mathbb{F}_{2^8}$ with $f(x) = x^8 + x^4 + x^3 + x + 1$ - resulting in an encryption algorithm that incorporates diffusion and confusion.

Before starting the rounds r , the state completes an AddRoundKey step. After that, the steps SubBytes, ShiftRows, MixColumns and AddRoundKey are executed for $N_r - 1$ rounds. The last round consists of SubBytes, ShiftRows and AddRoundKey. The MixColumns step gets dismissed because no extra security improvement is accomplished.

AES processes the plaintext in 128-bit blocks, structured as a 4x4 matrix of bytes, known as a state. The input (plaintext) block is denoted as s , while the generated ciphertext block is denoted as c . Each element of the state s can be denoted as $s_{i,j}$, where $0 \leq i, j \leq 3$.

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$
$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$
$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$

Plaintext represented as block

$c_{0,0}$	$c_{0,1}$	$c_{0,2}$	$c_{0,3}$
$c_{1,0}$	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$
$c_{2,0}$	$c_{2,1}$	$c_{2,2}$	$c_{2,3}$
$c_{3,0}$	$c_{3,1}$	$c_{3,2}$	$c_{3,3}$

Ciphertext represented as block

Before being converted into the ciphertext, the state s has to go through four different steps a number of times during the AES steps. Those steps are outlined in the sections below[DR02].

The SubBytes Step

The SubBytes step is responsible for a nonlinear transformation in the cipher in accordance with a fixed substitution box (S-box). This S-box is derived by taking the multiplicative inverse in the finite field $\mathbb{F}_{2^8}$, followed by an affine transformation. Hence, each input element $a_{i,j}$ will be transformed to an output element $b_{i,j}$, by mapping $a_{i,j} \mapsto b_{i,j}$.

If we take input block a and output block b :

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$\xrightarrow{\text{SubBytes}}$	$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$		$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	$b_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$		$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	$b_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$		$b_{3,0}$	$b_{3,1}$	$b_{3,2}$	$b_{3,3}$

With some element $a_{i,j}$ and constant $c = 0x63$, the affine transformation Aff_8 can be described as follows:

$$Aff_8(a) = L(a) + c$$

The elements are represented in $\mathbb{F}_{2^8}$, modulo the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$.

$$\begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

The ShiftRows Step

During the **ShiftRows** step, the rows will be shifted. By doing this, diffusion is exercised because of interleaving the bytes of different columns. The elements in the block will be unmodified. However, they can be assigned to a different position in the block. For every element $a_{i,j}$ in the block, the element shifts left i indices, cycling back if column of the new position of the element ends up being smaller than 0.

a_0	a_1	a_2	a_3	ShiftRows	a_0	a_1	a_2	a_3
a_4	a_5	a_6	a_7		a_5	a_6	a_7	a_4
a_8	a_9	a_{10}	a_{11}		a_{10}	a_{11}	a_8	a_9
a_{12}	a_{13}	a_{14}	a_{15}		a_{15}	a_{12}	a_{13}	a_{14}

The MixColumns Step

The **MixColumns** step mixes the data within each column of the state matrix by treating each column as a polynomial over $\mathbb{F}_{2^8}$ and multiplying it with a fixed polynomial modulo $x^4 + 1$. This increases diffusion.

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 0x02 & 0x03 & 0x01 & 0x01 \\ 0x01 & 0x02 & 0x03 & 0x01 \\ 0x01 & 0x01 & 0x02 & 0x03 \\ 0x03 & 0x01 & 0x01 & 0x02 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Using this technique, we can transform all elements within a given column at once by changing one column $a_{1,j}, a_{2,j}, a_{3,j}, a_{4,j}$ to a new column $b_{1,j}, b_{2,j}, b_{3,j}, b_{4,j}$.

The AddRoundKey Step

The **AddRoundKey** step is the only operation in AES that directly incorporates the cipher key into the state. It consists of a bitwise XOR between the current state and a round key derived from the main key through the AES key schedule.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$\oplus$	$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$	$=$	$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$		$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$		$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	$b_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$		$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$		$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	$b_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$		$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$		$b_{3,0}$	$b_{3,1}$	$b_{3,2}$	$b_{3,3}$

This step is performed before starting the first round, and is consequently performed as final step during all rounds.

2.3.3 Cryptographic Properties

Boolean and Multivariate functions are integral components in symmetric cryptography, particularly in the context of block ciphers. Their effectiveness is measured by several key cryptographic properties that determine the resistance to various forms of cryptanalysis.

Below, the relevant cryptographic properties are outlined:

Algebraic Degree

The algebraic degree of a multivariate function denotes how complex the function is in terms of how its output depends on the input variables. More specifically, it refers to the largest number of input variables that are multiplied together in any one term of the function. For example, if a function contains a term like $x_1x_3x_7$, then it has degree 3, because three variables are involved.

In cryptography, a higher algebraic degree usually means the function is more difficult to analyze or simplify. This added complexity helps protect against algebraic attacks, where attackers try to solve systems of equations to break the cipher.

Algebraic Immunity

Algebraic immunity measures how resistant a Boolean or multivariate function is to algebraic attacks. These are attacks where an adversary tries to break the cipher by solving equations. Specifically, it tells us how hard it is to find a low-complexity equation that always evaluates to zero when combined with the function. If you have a Boolean function f , then an *annihilator* g is used such that for all $x \in \mathbb{F}_2^n$, either $f(x) \cdot g(x) = 0$ or $(f(x) + 1) \cdot g(x) = 0$.

If such a simple equation exists, the adversary might use it to gain information about the input or key. A function with high algebraic immunity means that no such simple equations exist, which makes it harder for an attacker to exploit the function using algebraic techniques. In essence, the higher algebraic immunity, the more secure the function is against this type of mathematical attack.

Non-linearity

A function has high non-linearity if it behaves very differently from any simple linear function. In cryptography, this is important because linear functions are easier to analyze and predict. The more non-linear a function is, the harder it becomes to approximate or simplify it using linear equations. This makes it more resistant to attacks that try to exploit predictable patterns. Non-linearity, therefore, measures how “far” a function is from being linear, and a high score means the function introduces more complexity and confusion, which is what we aim for in a secure cipher.

When creating a GAP package that symbolically represents ciphers like AES, symbolic cryptographic analysis can be exercised. While analyzing, cryptographic properties can be used as markers for robustness.

2.4 Symbolic Computation and Computer Algebra Systems

2.4.1 Groups Algorithms Programming

Groups, Algorithms, Programming (GAP) is a programming language with libraries used for computational discrete algebra, with a particular emphasis on Computational Group Theory [Lin07]. GAP is particularly useful in the context of cryptographic algorithms when analyzing algebraic structures, such as finite fields and matrix groups.

In GAP, packages can be imported using the commands `ReadPackage("pathtopackage")` and `LoadPackage("packagename")`.

FFCSA

The package *Finite Field Constructions, Search and Algorithms* (FFCSA) is a GAP package designed for implementations of (cryptographic) hardware based on finite field arithmetic [ZGA⁺23]. Using this package, finite fields can be declared, field extensions can be exercised and symbolic representations can be constructed. Furthermore, FFCSA has many functions at its disposal, providing the ability to find the root of a polynomial, generate expressions for finite field multiplication given a particular basis.

Below, a simple GAP program is shown where a finite field is constructed with x as an indeterminate. After that, the finite field can be extended using defining polynomial f .

```
1 gap> SetPackagePath("FSR", "/home/dirk/GAP/FSR");
2 gap> LoadPackage("FSR");
3 gap> SetPackagePath("FFCSA", "/home/dirk/GAP/FFCSA");
4 gap> LoadPackage("FFCSA");
5 gap> # Creating GAP variables (before utilizing FFCSA):
6 gap> K := GF(2);;
7 gap> x := X(K, "x");;
8 gap> f := x^8 + x^4 + x^3 + x + 1;;
9 gap> F := FieldExtension(K, f);;
10 gap> # Use FFCSA to create variables for symbolic computation in F
11 gap> ChooseFieldElms(F);
12
13 variables
14 [ "a_0", "a_1", "a_2", "a_3", "a_4", "a_5", "a_6", "a_7" ]
15 [ "b_0", "b_1", "b_2", "b_3", "b_4", "b_5", "b_6", "b_7" ]
16 [ "d_0", "d_1", "d_2", "d_3", "d_4", "d_5", "d_6", "d_7", "d_8", "d_9",
17   "d_10", "d_11", "d_12",
18   "d_13", "d_14" ]
19 gap> # Use FFCSA to generate the desired polynomial Basis of F
20 gap> BF := GeneratePB(F, RootOfDefiningPolynomial(F));
```

Listing 1: Basic FFCSA example in GAP

2.4.2 Creating GAP packages

GAP packages can be created by anyone. To do that, the files `init.g`, `read.g` and `PackageInfo.g` have to be created first. `init.g` reads the declaration files from the `lib` folder. In these declaration files, the functions of the package are declared. The `read.g` file reads the implementation files from the `lib` folder as well. The actual functions are then loaded in. `PackageInfo.g` contains information about the package, such as the author's name, version number and required installations. Finally, the `tst` folder contains testfiles for testing the functions of the `lib` directory. A more comprehensible explanation of the starting structure of GAP packages can be seen in Appendix A.

In the `lib` folder, declaration files and implementation files are usually present. This keeps the code organized and well-structured, which makes extending the package easier. Also, future users of the package can easily see what functions are available and with which parameters they can be called. Using `InstallGlobalFunction`, function can be created. Via `InstallMethod`, multiple functions under the same name can be created. This is useful in case different representations need different parameters.

While designing packages, it is important to implement error handling properly. Parameters have to be checked and user input should conform with the capabilities of the package. This minimizes the presence of bugs. To check if the package works properly, test files can be written in the `tst` folder. These files utilize the functions in the `lib` folder and throw errors in case a bug is present, or if the code doesn't function as intended.

For more information about GAP and the creation of packages, the official website can be accessed: www.gap-system.org/

3 New GAP package: SYMSPN

3.1 Introduction

To express Substitution-Permutation Networks in a symbolical way, a new GAP package is created as the centerpiece of this project. The package is called **SYMSPN**, which stands for *Symbolic Substitution-Permutation Networks*. This package is meant to express symbolic SPN transformations in Algebraic Normal Form. This is useful because it can be used to derive cryptographic properties like confusion and diffusion. Additionally, **SYMSPN** provides a structured way to symbolically represent the internal components of SPNs, such as S-box layers and permutation layers, which allows for systematic analysis and transformation of cryptographic primitives. By working symbolically, it enables automated reasoning about properties such as nonlinearity and algebraic degree.

This package will be equipped with functions for symbolic AES permutations. AES is chosen because it is a well-known cipher, and is widely used as a standard for encryption. AES expressions will be created both in finite field representation $\mathbb{F}_{2^8}$ and vector representation $\mathbb{F}_2^8$. Expressing AES in both finite fields and vector spaces is useful because they capture different aspects of the algorithm. The finite field view is essential for modeling the nonlinear algebraic operations, like S-boxes and inverses, while the vector space view over $\mathbb{F}_2$ helps model AES as a series of linear transformations, which is useful as a prelude for expressing AES in a Boolean Function.

One of the main goals of this package, is that this package can be extended by future work: other substitution-permutation networks can be added to create a wide array of symbolic SPN expressions. In the following section, the making of this package will be discussed: The structure will be outlined and the choices regarding the package-design will be explained.

3.2 Designing SYMSPN

The SYMSPN package should be able to construct symbolic expressions of substitution permutation networks. In order to do that, the package has to be designed with scalability and robustness in mind. This means that the functions should be flexible, such that they can work with different state sizes, expressions and matrices. The package should be equipped with good error handling as well to prevent a breakdown. Finally, the package should have a clear structure that can be modified easily. This makes expanding the package in the future more accessible.

The design of the package is discussed in section 3.2, but also in section 3.3 and 3.4. Section 3.2 discusses the general aspects of the package design. In section 3.3 and 3.4, the functions are listed and described. There, more specific design choices are explained concerning those specific functions.

3.2.1 Structure

To create the SYMSPN package, the `lib`, `doc` and `tst` folder are created first, containing the functions, documentation and test files respectively. By separating these aspects of the package, we obtain a clear structure throughout the package. The `read.g` and `init.g` are also provided. These files read the implementation files and declaration files respectively.

```

1 Print("SYMSPN init.g: Reading declaration files...\n");
2 ReadPackage("SYMSPN", "lib/general.gd");
3 ReadPackage("SYMSPN", "lib/ffSPN.gd");
4 ReadPackage("SYMSPN", "lib/vecSPN.gd");
5 Print("SYMSPN init.g: Done reading declaration files!\n");

```

Listing 2: init.g file in SYMSPN

```

1 Print("SYMSPN read.g: Reading implementation files...\n");
2 ReadPackage("SYMSPN", "lib/globals.gi");
3 ReadPackage("SYMSPN", "lib/general.gi");
4 ReadPackage("SYMSPN", "lib/ffSPN.gi");
5 ReadPackage("SYMSPN", "lib/vecSPN.gi");
6 Print("SYMSPN read.g: Done reading implementation files!\n");

```

Listing 3: read.g file in SYMSPN

By splitting the lib files up into declaration- and implementation files, the package stays structured while also being more easy to comprehend. It can also reduce compilation time and helps with documentation and testing.

New users can acquire knowledge about the functions that are included along with their parameters by looking at the declaration files, providing an easy way to comprehend the functions of the package. Meanwhile, the implementation files stay clean by only containing the actual functions. There are a total of four implementation files and three declaration files.

Declaration files:

- general.gd
- ffSPN.gd
- vecSPN.gd

Implementation files:

- globals.gi
- general.gi
- ffSPN.gi
- vecSPN.gi

The file `globals.gi` contains global variables. These are polynomials, finite fields, generators and more. These variables can be passed on as parameters when utilizing functions. This helps the user to formulate the function calls more quickly, speeding up the utilization of the functions by the user. For example, polynomial $f_1 = x^8 + x^4 + x^3 + x^2 + 1$ is already present in `globals.gi`, meaning that the user can immediately pass on `f1` as a parameter. Since this file isolates the variables,

new variables can be added easily. This file is loaded in first before the other implementation and declaration files so that all functions in the package can utilize the variables if needed. The use of these global variables is optional: both the global variables from `globals.gi` as well as local variables can be used throughout all functions. So the user is also capable of creating new variables at any time and utilize them.

The `general.gd` and `general.gi` files contain functions that are applicable to different representations of SPNs. For example, both elements in finite fields and elements in vector spaces can utilize these functions since the parameters are the same and the type of operations are the same. Right now, the `general` files contain the functions `shiftRows` and `mixColumns`.

However, in some cases the different representations require different functions. That is where the `ffSPN` and `vecSPN` files come into play. First, these files contain the functions `initSymbolsFFE` and `initSymbolsVEC` to retrieve records with symbols and variables for finite field and vector space representations. More about these two functions is discussed in section 3.2.2. The other functions within the `ffSPN` and `vecSPN` files are `subBytes`, `roundAES` and `roundAESElm` for implementations: whichever operation is chosen, they are called under the same name. Using `InstallMethod`, multiple functions can be created with the same name while GAP method selection mechanisms figure out which function to chose based on the parameters that are given. By designing the package this way, the user experience is more pleasant, since there are not ambiguous function names for different representations.

Apart from the `lib` folder, the `tst/` folder contains three files. These files are meant to showcase that AES can be expressed using this package.

- `testffAES.g`
- `testvecAES.g`
- `compareAES.g`

Two files, `testffAES.g` and `testvecAES.g`, are meant for testing AES in finite fields and vector spaces respectively. It checks if all elements of the new state are transformed in the same manner, thereby checking the consistency of transformation across the state. `compareAES.g` is created to test if those two representations correspond with each other, to proof their correctness. A further explanation of the test files can be found at section 3.5.

In the `doc` folder, a file called `symspn.txt` is present. This concise file describes the package `SYMSPN` and explains what it is for. Also, it lists all functions that are present and briefly describes them. Finally, example commands are presented to utilize the package and to execute the test files.

Finally, `PackageInfo.g` is created, which stores general information about the package, including the package name, version number, and dependencies. While designing this package, FSR version 1.2.2 and FFCSA version 0.9.4 were used. This will therefore be the required packages to use `SYMSPN`. The full file `PackageInfo.g` can be seen in Appendix B.

The whole package structure is as follows:

```
.
├── doc
│   └── symspn.txt
├── init.g
├── read.g
├── lib
│   ├── ffSPN.gd
│   ├── ffSPN.gi
│   ├── vecSPN.gd
│   ├── vecSPN.gi
│   ├── general.gd
│   ├── general.gi
│   └── globals.gi
├── PackageInfo.g
├── tst
│   ├── compareAES.g
│   ├── testffaes.g
│   └── testvecAES.g
```

Throughout all codes in the package, comments are provided. These comments concisely explain the code, making it more comprehensible for new users to understand and extend the package. Every function is equipped with a comment to indicate what the function does: making it very easy to grasp what the function is designed for. Within the functions, comments are also provided to, among other things, indicate error handling and what operations are being executed.

Error handling is also built in. The functions expect a certain type of parameter, like matrices or polynomials. So before using the parameters, the type of the parameters will be checked in the function. Also, the variables itself have to be appropriate. For example, when creating a state, the parameters `rows` and `cols` that are used to construct the state have to be a positive number, otherwise a state could not exist. Lastly, functions like `roundAES` and `roundAESElm` call multiple functions consequently. The error handling of all those functions will be done beforehand. That way, an atomic operation is ensured. If not all error handling is being done beforehand, the state could be transformed partially. By utilizing this manner of error handling, an operation is either performed or it is not.

3.2.2 Record of Symbols

The files in the `lib` folder contain various functions to perform arithmetic operations, like substituting bytes, shifting rows and mixing columns. These functions require many variables and constants in order to operate. For example, the functions initially use a finite field K , containing two elements. This finite field can be extended using a polynomial, resulting in an extended finite field. From there, a generator can be created along with its corresponding basis. Finally, the operations are

performed on a state with an arbitrary number of rows and columns, containing symbolic elements. To pass all of these functions along as parameters every time a function is called, will be very tedious. Passing on many parameters for every function deteriorates the user experience. Furthermore, `InstallMethod` has a hardcoded maximum of six parameters, limiting the capability to pass on many parameters. Because of these two reasons, the functions `initSymbolsFFe` and `initSymbolsVEC` are created. These functions take in the number of rows, the number of columns and a polynomial as parameters. The finite field extension will be done using the polynomial parameter. Consequently, the adequate state with its symbolic elements will be created based on the rows and columns parameters.

The functions consequently return a record containing all necessary symbols that can be utilized by other functions. This includes the parameters that were given, the basis and generator that have been created, but also the state itself. The full record can be viewed at Listing 4, line 27-36.

Now, these records will be passed on as one parameter to other functions. This gives other functions the ability to access the general symbols (including the state) at any time, while also drastically limiting the amount of parameters that are passed to the functions.

The full `initSymbolsFFe` function of finite field elements can be seen below. The vector space representation is very similar, and can also be seen at section 3.4.

```

1 InstallGlobalFunction("initSymbolsFFe", function(rows, cols, f)
2   ### Generate default symbolic values for finite fields
3   local K, F, gen, BF, row, col, stateRow, state, elm;
4   # Check parameters
5   # Check parameters rows, cols and f
6   if not IsInt(rows) and not IsInt(cols) then
7     Error("Rows and Columns should be integers");
8   fi;
9   if rows < 1 or cols < 1 then
10    Error("Number of rows and columns should be at least one");
11  fi;
12  if not IsPolynomial(f) then
13    Error("Polynomial should be given");
14  fi;
15  K := GF(2);
16  F := FieldExtension(K, f);
17  gen := RootOfDefiningPolynomial(F);
18  BF := GeneratePBdownto(F, gen);
19  # Generating State with its elements
20  state := [];
21  for row in [1..rows] do
22    stateRow := [];
23    for col in [1..cols] do
24      elm := Concatenation("s", String(row), String(col));
25      Add(stateRow, X(F, elm));
26    od;
27    Add(state, stateRow);
28  od;

```

```

29     return rec(
30         K := K,
31         F := F,
32         f := f,
33         BF := BF,
34         gen := gen,
35         rows := rows,
36         cols := cols,
37         state := state
38     );
39 end);

```

Listing 4: initSymbolsFFe function

3.2.3 Symbolic AES approach

From this point on, AES in symbolic finite field expressions can be created and AES in symbolic vector expressions can be created.

The package should be able to execute the steps `SubBytes`, `ShiftRows` and `MixColumns` during every round. These three steps will all be executed by a dedicated function. The `AddRoundKey` step will be avoided. That is because this step doesn't provide direct symbolic insight. In symbolic AES modelling, `AddRoundKey` adds arbitrary key variables via XOR, which obscures the algebraic structure. Since the key values are secret and thus not universal, symbolic key variables would only expand the expression.

Hence, the steps that will be executed are `SubBytes`, `ShiftRows` and `MixColumns`.

$$(\text{MixColumns} \circ \text{ShiftRows} \circ \text{SubBytes})(s)$$

Where s is the state that will be transformed by these steps.

For both the finite field representation and vector representation, functions will be created to retrieve every possible element of the new state (transformed state) after AES round(s). There will also be a function that retrieves the whole new state after AES round(s). The amount of rounds can be determined by a parameter. Whether these functions transform the state, depends on the parameter `transformState`. Within functions, that variable will be used to determine if the state itself should be transformed, or whether just a copy should be transformed. The transformed state will be returned.

```

1  if not transformState then
2      # Transform copied state
3      symbolscopy := ShallowCopy(symbols);
4      symbolscopy.state := List(symbols.state, row -> ShallowCopy(row));
5      state := symbolscopy.state;
6  else
7      # Transform state
8      state := symbols.state;

```

Listing 5: Checking if state should be transformed

In conclusion, dedicated functions will be created for the steps `SubBytes`, `ShiftRows` and `MixColumns`. By designing the package this way, the package's structure is more clear and it allows for unit tests to be written. Moreover, modifications in future work can be done more easily. In section 3.3 and 3.4, the functions itself will be shown. There, design choices are described in more detail.

3.3 Symbolic AES in a Finite Field

To create symbolic AES expressions in a finite field, the function `initSymbolsFFE` is called first. A finite field $\mathbb{F}_2$, named K , is created within this function. From there, a finite field extension can be created using parameter polynomial $f_1 = x^8 + x^4 + x^3 + x^2 + 1$, with intermediate x . This finite field extension allows symbols $\in \mathbb{F}_{2^8}$. For symbolic finite field computations, we use f_1 instead of $f_2 = x^8 + x^4 + x^3 + x + 1$, because we can then express `SubBytes` as a linearized polynomial [DR02]. From here, generator α can be created, along with a polynomial basis BF .

Along with the above variables, `state` will be created again. The size of `state` is based on the parameters `rows=4` and `cols=4`. This is the block that AES operates on. This block will be transformed in the AES-steps. This block consists of four rows, each consisting of four columns, creating the space for 16 symbolic elements (variables) in total. These variables are `s11` up till `s44`, where the first number indicates the row number, and the second number indicates the column number of the block state. These elements can be created as `s11 := X(F, "s11");`. This allows us to display the AES elements symbolically, while also giving the opportunity to evaluate the symbolic statements during the test phase.

All variables made during this function will be returned in a dictionary (record) format, as discussed before. This way, multiple variables can be obtained by just one function call.

SubBytes

The subbytes step operates on the individual elements of the state, and uses the polynomial $x^8 + x^4 + x^3 + x^2 + 1$ while doing so [DR02].

The subbytes step can then be performed in the following formula, which will be passed as a parameter:

$$L(x) = \alpha^2 x + \alpha^{199} x^2 + \alpha^{99} x^4 + \alpha^{185} x^8 + \alpha^{197} x^{16} + x^{32} + \alpha^{96} x^{64} + \alpha^{232} x^{128} + \alpha^{195}$$

Where x is the inverse of the state element. This variable can be derived by Fermat's Little Theorem, which is also passed on as a parameter:

$$\text{Inv}(x) = x^{254}$$

To exercise this transformation on the whole state, two for loops are created: (i) the outer loop to iterate through the rows, and (ii) the inner loop to iterate through the columns, mapping the elements one by one. To make sure the elements stay within bounds of the finite field, the function

ReduceMonomialsOverField is used.

Design Choices: Because both expression $L(x)$ and $\text{Inv}(x)$ are given as polynomials, other polynomials could also be given. This makes future implementations of new SPNs applicable. For example, if a different algorithm is designed that uses the `subBytes` step with x^3 instead of the inverse, it can still pass this new expression on as a parameter. The same principle holds for the linearized polynomial. The `ReduceMonomialsOverField` function is also used in accordance with the finite field that is given via `symbols` to make sure the elements stay within the adequate finite field.

```

1 InstallMethod(subBytes, "subBytes with polynomials",
2 [IsRecord, IsPolynomial, IsPolynomial],
3 function(symbols, expr1, expr2)
4     ### subBytes step using two expressions
5     local symbolscopy, F, state, row, col, rows, cols, expoVar;
6     F := symbols.F;
7     rows := symbols.rows;
8     cols := symbols.cols;
9     ### Check transformState: Omitted for brevity
10    for row in [1..rows] do
11        for col in [1..cols] do
12            expoVar := ReduceMonomialsOverField(F, Value(expr1, state[
13                row][col]));
14            state[row][col] := ReduceMonomialsOverField(F, Value(expr2,
15                expoVar));
16        od;
17    od;
18    return state;
19 end);

```

Listing 6: subBytes function using polynomial parameters

ShiftRows

The `ShiftRows` step simply shifts rows 2, 3 and 4. Row 2 will be shifted one to the left, row 3 will be shifted two to the left, and row 4 will be shifted 3 to the left. The elements are just transpositioned, while not modifying the values itself. For implementation in GAP, the FFCSA function `ROL()` can be used to rotate left. A code line like `state[2] := ROL(state[2], 1);` can be used to shift the second row one to the left. This type of operation is also exercised on row two and three respectively using a list. This list states how much each row should shift. In the case of AES, the list `[0,1,2,3]` is passed on. Error handling is also present to ensure that every row has a shift-value.

Design Choices: By rotating the state one by one using a for loop instead of hardcoding, the `shiftRows` function can operate on any state length.

```

1 InstallGlobalFunction("shiftRows", function(symbols, shiftArr,
2     transformState)
3     ### ShiftRows step, where shiftArr denoted the shifting per row
4     local symbolscopy, state, row;

```

```

4      # Check parameters shiftArr and transformState
5      if not IsList(shiftArr) then
6          Error("shiftArr should be a given");
7      fi;
8      if not IsBool(transformState) then
9          Error("transformState should be a Boolean");
10     fi;
11     ### Check transformState: Omitted for brevity
12     # Check if there is a shift value for every row
13     if Length(shiftArr) <> Length(state) then
14         Error("Length shiftArr is not equal to amount of state rows\n");
15     fi;
16     # Shifting the rows
17     for row in [1..Length(shiftArr)] do
18         state[row] := ROL(state[row], shiftArr[row]);
19     od;
20
21     return state;
22 end);

```

Listing 7: shiftRows function

MixColumns

The `MixColumns` step operates independently on four-element columns and mixes them linearly by multiplication with the following matrix[DR02]:

$$\begin{bmatrix} \alpha^{25} & \alpha & 1 & 1 \\ 1 & \alpha^{25} & \alpha & 1 \\ 1 & 1 & \alpha^{25} & \alpha \\ \alpha & 1 & 1 & \alpha^{25} \end{bmatrix}$$

In GAP, this can be implemented by first declaring a matrix called `MixM1`, with generator α being integrated. This matrix will be passed on to the function. The size of the matrix is checked, to ensure that matrix multiplication is possible given the state size. After that, the columns are filtered out of the input block `state`. Consequently, matrix `MixM1` can be multiplied by those columns. We can store the results directly in `state`.

Design Choices: Before applying the matrix multiplication, error handling is executed. The number of columns of the matrix has to be equal to the amount of rows of the state. Otherwise, matrix multiplication is not possible. After that, the columns are filtered out one by one. By doing that, the matrix parameter `mat` can be multiplied by those columns. Then the resulting columns can be loaded into the state.

```

1 InstallMethod(mixColumns, "mixColumns step in finite field",
2 [IsRecord, IsMatrix, IsBool],
3 function(symbols, mat, transformState)
4     ### MixColumns step, with matrix mat

```

```

5      local symbolscopy, state, row, col, rows, cols, newCol, newCols,
        stateCol;
6      rows := symbols.rows;
7      cols := symbols.cols;
8      ### Check transformState: Omitted for brevity
9      # Check if matrix multiplication can take place
10     if Length(state) <> Length(mat[1]) then
11         Error("Length Mixcolumns Matrix is different from length of
            state");
12     fi;
13     # Extract Mixcolumns matrix's columns and calculate new columns
14     newCols := [];
15     for col in [1..cols] do
16         # For every row of the state, take a certain column index
17         stateCol := List(state, row -> row[col]);
18         newCol := mat * stateCol;
19         Add(newCols, newCol);
20     od;
21     # Update state
22     for row in [1..rows] do
23         for col in [1..cols] do
24             state[row][col] := newCols[col][row];
25         od;
26     od;
27     return state;
28 end);

```

Listing 8: MixColumns function

roundAES

The function **roundAES** will be used to execute all previous steps consequently, and return the state after AES round(s). To do that, it needs all parameters that are used in the functions discussed previously: the symbols record, the two polynomials for **subBytes** and **params** to use during the **shiftRows** and **mixColumns** steps. Flag parameters are also provided to (i) specify the amount of rounds that need to be run and (ii) whether the state itself should be transformed. The transformed state will be returned. Finally, there is also the same function but with default flags. Those flags are set such that one AES round is executed without transforming the state itself.

Design Choices: Because the **InstallMethod** can only handle up to six parameters, some variables are grouped in lists, instead of solely individual parameters. In this case, **params** contains the parameters for all steps after **subBytes**. The **flags** parameter list contains the amount of rounds and the **transformState** variable. The expressions for the **subBytes** step are not given as a list, since this is the main point of difference with the vector space representation: the vector space representation uses a function, matrix and vector. This means that we can rely on GAP's method selection mechanisms to choose between the two functions (which have the same function name), instead of relying on cumbersome techniques to separate the two functions when making a function call. For more information about the vector space implementation, see section 3.4.

The error handling steps of the functions `shiftRows` and `mixColumns` are already performed in this function. By doing this, an error can be thrown before transforming the state (partially). This ensures an atomic transformation. After that, the `subBytes`, `shiftRows` and `mixColumns` steps can be walked through in sequence by calling the corresponding functions.

A for loop is used for executing a certain number of rounds instead of recursively calling the function. The first reason that, now, error handling does not have to be repeated constantly, making it more efficient. Furthermore, the for loop makes sure that the `transformState` variable is not too cumbersome: if you would execute two rounds, but don't want to transform the actual state, then you have to pass on the current copied state instead of simply `symbols` (which contains the initial state), making the function overly complicated. So using a for loop is way more efficient and less arduous.

```

1 InstallMethod(roundAES, "roundAES in finite field with default flags",
2 [IsRecord, IsPolynomial, IsPolynomial, IsList],
3 function(symbols, expr1, expr2, params)
4     return roundAES(symbols, expr1, expr2, params, [1, false]);
5 end);
6
7 InstallMethod(roundAES, "roundAES in finite field",
8 [IsRecord, IsPolynomial, IsPolynomial, IsList, IsList],
9 function(symbols, expr1, expr2, params, flags)
10     ### Execute AES round(s) in finite fields
11     ### Return final state
12     local symbolscopy, state, shiftArr, mat, round, rounds,
13         transformState;
14     ### Error Handling: Omitted for brevity
15     ### Check transformState: Omitted for brevity
16     ### Error Handling: Omitted for brevity
17     for round in [1..rounds] do
18         subBytes(symbolscopy, expr1, expr2, true);
19         shiftRows(symbolscopy, shiftArr, true);
20         mixColumns(symbolscopy, mat, true);
21     od;
22     return state;
23 end);

```

Listing 9: roundAES function

roundAESElm

This function also executes AES round(s) as well, but returns only one element. It takes in the same parameters as the `roundAES` function, and even uses that function to transform the state. From that transformed state, only one element will be returned. The parameters `elmRow` and `elmCol` are used to specify which element should be returned. An error is thrown if the row or column number is not within bounds of the state. It is also possible to call this function without flags: default flags will be provided then.

Design Choices: This function utilizes the `roundAES` function. Because of that, error handling and calling functions can be kept to a minimum within `roundAESElm`. Hereby, redundant code is avoided. The parameters and their order is the same as in `roundAES`, except for the added `pos`, which contains the coordinates (position) of the state element that needs to be returned.

```

1 InstallMethod(roundAESElm,
2 "Obtain AES element in finite field after rounds with default flags",
3 [IsRecord, IsPolynomial, IsPolynomial, IsList, IsList],
4 function(symbols, expr1, expr2, params, pos)
5     return roundAESElm(symbols, expr1, expr2, params, pos, [1, false]);
6 end);
7
8 InstallMethod(roundAESElm, "Obtain AES element in finite field after
9     rounds",
10 [IsRecord, IsPolynomial, IsPolynomial, IsList, IsList, IsList],
11 function(symbols, expr1, expr2, params, pos, flags)
12     ### Execute AES round(s) in finite fields
13     #### Return one element of final state
14     local state, elmRow, elmCol;
15     # Check if pos[] is valid
16     if Length(pos) <> 2 then
17         Error("Row and Column are required");
18     fi;
19     elmRow := pos[1];
20     elmCol := pos[2];
21     if not IsInt(elmRow) and not IsInt(elmCol) then
22         Error("Row and Column should be integers");
23     fi;
24     if elmRow < 1 or elmRow > symbols.rows then
25         Error("element row is outside of state");
26     fi;
27     if elmCol < 1 or elmCol > symbols.cols then
28         Error("element column is outside of state");
29     fi;
30     # Call roundAES and consequently return one state element
31     state := roundAES(symbols, expr1, expr2, params, flags);
32     return state[elmRow][elmCol];
33 end);

```

Listing 10: `roundAESElm` function

3.4 Symbolic AES in a Vector Space

To create a symbolic AES expressions in vector space, variables have to be created first, just like the finite field representation. In this case, the function `initSymbolsVEC` will be used. We begin with a finite field $\mathbb{F}_2$, named K . Every element is a vector, and the vector consists of eight binary values $\mathbb{F}_2$. From there, a finite field extension can be created. For the symbolic AES in vector representation, we use the standard AES polynomial $f_2(x) = x^8 + x^4 + x^3 + x + 1$ with intermediate x . From here,

a root β and a corresponding polynomial basis BG can be created. This allows us to express the symbols in $\mathbb{F}_2$ and use basis BG to represent all possible $\mathbb{F}_2^8$ elements.

The symbolic elements will be different from the finite field implementation. Instead of creating elements of the AES block directly, every element is a vector (element $\in \mathbb{F}_2^8$), consisting of 8 elements $\in \mathbb{F}_2$ for every vector. These vectors make up the AES-block **state**.

The function `initSymbolsVEC` will almost be the same as the `initSymbolsFFe` function, apart from creating the state with its symbolic elements.

Design Choices: The only real difference within the function is the state that is being created. In this case, vectors, consisting of components (entries), have to be created. These entries are made in top down order.

```

1 # Generating State with its elements
2 state := [];
3 for row in [1..rows] do
4     stateRow := [];
5     for col in [1..cols] do
6         elm := [];
7         for component in [1..8] do
8             # Adding vector components top down
9             comp := Concatenation("s", String(row), String(col), "_",
10                String(8-component));
11             Add(elm, X(K, comp));;
12         od;
13         Add(stateRow, elm);;
14     od;
15 Add(state, stateRow);
16 od;

```

Listing 11: Creating state in `initSymbolsVEC`

SubBytes

This **SubBytes** step uses a matrix, L and the corresponding constant vector c , during the affine transformation as described in section 2.3.2.

$$Aff_8(a) = L(a) + c$$

To transform all elements of the state, an outer loop is created to iterate over the rows and an inner loop is created to iterate over the columns. By doing that, each state element can be transformed one by one. The transformation is done by utilizing the FFCSA function `FFA_inv_matrixU()` to compute the inverse of the vector element. This function computes the inverse in matrix form by calling `FFA_exp_matrixU` with exponent $2^{\deg(f)} - 2$, which corresponds to the multiplicative inverse in $\mathbb{F}_{2^n}$. `FFA_exp_matrixU` computes repeated squaring of a vector under a matrix representation, by expressing the exponent into binary values. Finally, the inverse will be multiplied by matrix L

to consequently add `c_vector` to the result. These three variables are all given as parameters to the function.

Design Choices: Whereas the finite field version of `subBytes` uses expressions, the vector space implementation uses a function, matrix and list as parameters. This way the transformation is, just like in the finite field representation, up to the user. This creates consistency between the two functions, while also allowing for a wide variety of possibilities to transform the state. Moreover, because this `subBytes` variant uses a function, matrix and vector as parameters instead of two polynomials, the GAP selection mechanisms can discriminate between the two functions.

```

1 InstallMethod(subBytes, "subBytes with function and matrix",
2 [IsRecord, IsFunction, IsMatrix, IsList, IsBool],
3 function(symbols, func, mat, vec, transformState)
4     ### subBytes step using a function, a matrix and a vector constant
5     local BF, symbolscopy, state, c_vector, expoVar, row, col, rows,
6         cols, cell;
7     BF := symbols.BF;
8     rows := symbols.rows;
9     cols := symbols.cols;
10    ### Check transformState: Omitted for brevity
11    # Check if matrix multiplication is possible
12    if Length(mat) <> Length(vec) then
13        Error("Length of matrix and length of vector not the same");
14    fi;
15    # Execute subBytes step
16    for row in [1..rows] do
17        for col in [1..cols] do
18            expoVar := func(BF, state[row][col]);
19            state[row][col] := MultMatrixEvec(mat, expoVar) + vec;
20        od;
21    od;
22    return state;
23 end);

```

Listing 12: SubBytes for Vector implementation

ShiftRows

The **ShiftRows** step can be executed in the same manner as the finite field implementation: Shifting rows 2, 3 and 4 using `ROL()`. Row 2 will be shifted one to the left, row 3 will be shifted two to the left, and row 4 will be shifted 3 to the left. A line like `state[2] := ROL(state[2], 1);` can be used to shift the second row one to the left. This type of operation is also exercised on row two and three respectively. For the vector space representation of AES, the exact same function is used as the finite field representation.

MixColumns

Since the **MixColumns** step uses polynomial $f_2 = x^8 + x^4 + x^3 + x + 1$ instead of $f_1 = x^8 + x^4 + x^3 + x^2 + 1$, a different matrix is required. Since **MixM1** uses α^{25} and α , the corresponding elements for f_2 have to be derived. This can be done using GAP:

```

1 gap> BF1; # Basis for finite field extended by f1
2 Basis( GF(2^8), [ Z(2^8)^7, Z(2^8)^6, Z(2^8)^5, Z(2^8)^4, Z(2^8)^3, Z
   (2^8)^2, Z(2^8), Z(2)^0 ] )
3 gap> BF2; # Basis for finite field extended by f2
4 Basis( GF(2^8), [ Z(2^8)^175, Z(2^8)^150, Z(2^8)^125, Z(2^8)^100, Z(2^8)
   ^75, Z(2^8)^50, Z(2^8)^25, Z(2)^0 ] )
5 gap> alpha; # Generator for finite field extended by f1
6 Z(2^8)
7 gap> beta; # Generator for finite field extended by f2
8 Z(2^8)^25
9 gap> alpha^25 = beta;
10 true
11 gap> alpha = beta + One(F2);
12 true

```

Listing 13: Corresponding elements of MixColumns matrices

Hence, the corresponding matrix:

$$\begin{bmatrix} \beta & 1+\beta & 1 & 1 \\ 1 & \beta & 1+\beta & 1 \\ 1 & 1 & \beta & 1+\beta \\ 1+\beta & 1 & 1 & \beta \end{bmatrix}$$

The **MixColumns** step will multiply this matrix with the state columns. However, using this matrix in the vector space implementation is not possible since multiplying with β would result outside $\mathbb{F}_2$. Instead of multiplying the state elements with $\mathbb{F}_{2^8}$ elements like in the finite field implementation, this implementation will multiply the state elements with matrices. This means that **MixM2** will consist of matrices. **MixM2** is given below, alongside the matrices it consists of.

$$M_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad M_2 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad M_3 = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$MixM2 = \begin{bmatrix} M_2 & M_3 & M_1 & M_1 \\ M_1 & M_2 & M_3 & M_1 \\ M_1 & M_1 & M_2 & M_3 \\ M_3 & M_1 & M_1 & M_2 \end{bmatrix}$$

Where all 0's and 1's are elements of $\mathbb{F}_2$.

MixM2 allows for the possibility of linear transformations M_1 , M_2 and M_3 while staying within $\mathbb{F}_2$. M_1 is an identity matrix, and will therefore keep the vector unchanged when multiplied. M_2 corresponds with multiplication by β . It accomplishes this because the diagonal of 1's is moved one index to the right. Multiplying this matrix with a vector is the same as shifting all entries of a vector one to the left. Only the first entry of the vector is shifted outside the scope of the vector. Since the polynomial $f_2 = x^8 + x^4 + x^3 + x + 1$ is used, $x^8 = x^4 + x^3 + x + 1$ can be inferred. Hence, the first entry of the vector can be added to index 0, 1, 3 and 4. Vector element `state[1][1]` would look like the following when multiplied with M_2 :

Before multiplication: $[s_{11.7}, s_{11.6}, s_{11.5}, s_{11.4}, s_{11.3}, s_{11.2}, s_{11.1}, s_{11.0}]$

After multiplication: $[s_{11.6}, s_{11.5}, s_{11.4}, s_{11.3} + s_{11.7}, s_{11.2} + s_{11.7}, s_{11.1}, s_{11.0} + s_{11.7}, s_{11.7}]$

Matrix M_3 corresponds with multiplication by $\beta + 1$. Multiplying by M_3 is the same as multiplying by M_2 and adding multiplication by M_1 . Hence, $M_3 = M_2 \oplus M_1$. Vector element `state[1][1]` when multiplied with M_3 :

Before multiplication: $[s_{11.7}, s_{11.6}, s_{11.5}, s_{11.4}, s_{11.3}, s_{11.2}, s_{11.1}, s_{11.0}]$

After multiplication: $[s_{11.7} + s_{11.6}, s_{11.6} + s_{11.5}, s_{11.5} + s_{11.4}, s_{11.4} + s_{11.7} + s_{11.3}, s_{11.3} + s_{11.7} + s_{11.2}, s_{11.2} + s_{11.1}, s_{11.1} + s_{11.7} + s_{11.0}, s_{11.0} + s_{11.7}]$

A verification that β indeed corresponds with M_2 and $\beta + 1$ corresponds with M_3 is given below.

```

1 gap> gen; # Primitive root
2 Z(2^8)
3 gap> # beta should correspond with M2
4 gap> (M2 * IntFFExt(BF2, gen)) * BF2 = beta * gen;
5 true
6 gap> # beta + 1 should correspond with M3
7 gap> (M3 * IntFFExt(BF2, gen)) * BF2 = (beta + One(F2)) * gen;
8 true
9 gap>

```

Listing 14: Verification that beta and beta+1 correspond with M_2 and M_3

Design Choices: In GAP, matrices are lists of lists. When there is just one list consisting of lists, it is recognized as a matrix, as is the case with **MixM1**. In case there is a matrix consisting of matrices, it will be too complex for GAP's type-checking mechanisms to recognize it as a matrix. Consequently, GAP's type-checking mechanisms will recognize it as a list. This means that GAP method selection mechanisms can differentiate between the two functions, since **MixM1** will be seen as a matrix and **MixM2** will be seen as a list. The parameter will still be called `mat` to make clear that **MixM2** will be treated as a matrix within the function, even if it is not recognized as such by GAP's type-checking mechanisms.

Because **MixM2** is seen as a list, direct multiplication with columns during the `mixColumns` step is not possible. Therefore, multiplication is done element by element, as showcased in Listing 15.

```

1 InstallMethod(mixColumns, "MixColumns step within vector space",
2 [IsRecord, IsList, IsBool],
3 function(symbols, mat, transformState)
4     ### mixColumns step with a list consisting of matrices
5     local rows, cols, symbolscopy, state, stateRow, row, col, stateCol,
6         copy, i;
7     rows := symbols.rows;
8     cols := symbols.cols;
9     ### Check transformState: Omitted for brevity
10    # Check if matrix multiplication is possible
11    if Length(state) <> Length(mat[1]) then
12        Error("Length Mixcolumns Matrix is different from length of
13            state");
14    fi;
15    # Check if mat[] consists of matrices
16    for row in [1..Length(mat[1])] do
17        for col in [1..Length(mat)] do
18            if not IsMatrix(mat[row][col]) then
19                Error("Matrix should consist of matrices");
20            fi;
21        od;
22    od;
23    # Execute mixColumns step by multiplying matrix by vector elements
24    copy := List(state, row -> List(row, cell -> ShallowCopy(cell)));
25    for col in [1..cols] do
26        stateCol := List(copy, stateRow -> stateRow[col]);
27        for row in [1..Length(stateCol)] do
28            state[row][col] := state[row][col] * Zero(K);
29            # Multiply all row elements of state column
30            # with all column elements of matrix row
31            for i in [1..Length(stateCol)] do
32                state[row][col] := state[row][col] + mat[row][i] *
33                    stateCol[i];
34            od;
35        od;
36    od;
37    return state;
38 end);

```

Listing 15: MixColumns step for Vector implementation

roundAES

The function `roundAES` in the vector representation works in the same manner as in the finite field representation. After obtaining the symbols, the functions `subBytes`, `shiftRows` and `mixColumns` will be executed consequently. However, there are two differences: (i) the parameters that are passed on have to be different, since `subBytes` uses a function, matrix and vector as parameters and (ii) the error handling to ensure an atomic operation is different because the parameters are different. A version of `roundAES` with default flags is also given.

Design Choices: Just like with the finite field representation, some parameters have to be given in lists to comply with the six parameters limit. To stay consistent, there is a `params` list for the `shiftRows` and `mixColumns` steps, and there is a `flags` list for the amount of rounds and `transformState`. The only difference in parameter data types is for `subBytes`. In this case, `func` will be used to denote the function that is being passed on. Because a function parameter differs from a polynomial parameter (as seen in the finite field version), GAP's method selection mechanisms can be utilized. The other `subBytes` parameters are given as a list.

```
1 InstallMethod(roundAES, "roundAES in vector space with default flags",
2 [IsRecord, IsFunction, IsList, IsList],
3 function(symbols, func, params1, params2)
4     return roundAES(symbols, func, params1, params2, [1, false]);
5 end);
6
7 InstallMethod(roundAES, "roundAES in vector space",
8 [IsRecord, IsFunction, IsList, IsList, IsList],
9 function(symbols, func, params1, params2, flags)
10     ### Execute AES round(s) in vector space
11     ### Return final state
12     local symbolscopy, state, mat1, vec, shiftArr, mat2, row, col, round
13         , rounds, transformState;
14     ### Error Handling: Omitted for brevity
15     ### Check transformState: Omitted for brevity
16     ### Error Handling: Omitted for brevity
17     # Execute AES steps in sequence
18     for round in [1..rounds] do
19         subBytes(symbolscopy, func, mat1, vec, true);
20         shiftRows(symbolscopy, shiftArr, true);
21         mixColumns(symbolscopy, mat2, true);
22     od;
23     return state;
24 end);
```

Listing 16: roundAES function

roundAESElm

The function `roundAESElm` performs the same operations as in the finite field representation: It utilizes the `roundAES` function and returns just one element of the state. The difference lies in the parameters that are being used, which need to comply with the vector space variant of `roundAES`.

Design Choices: To stay consistent with the `roundAES` function, the same parameter types are given in the same order. The only difference is the parameters for the coordinates of the state element that has to be returned: that is given as a list parameter `pos`.

```
1 InstallMethod(roundAESElm,
2 "Obtain AES element in vector space after rounds with default flags",
3 [IsRecord, IsFunction, IsList, IsList, IsList],
4 function(symbols, func, params1, params2, pos)
5     return roundAESElm(symbols, func, params1, params2, pos, [1,false]);
6 end);
7
8 InstallMethod(roundAESElm, "Obtain round AES in vector space",
9 [IsRecord, IsFunction, IsList, IsList, IsList, IsList],
10 function(symbols, func, params1, params2, pos, flags)
11     ### Execute AES round(s) in vector space
12     ### Return one element of final state
13     local state, elmRow, elmCol;
14     # Check if pos[] is valid
15     if Length(pos) <> 2 then
16         Error("Row and Column are required");
17     fi;
18     elmRow := pos[1]; elmCol := pos[2];
19     if not IsInt(elmRow) and not IsInt(elmCol) then
20         Error("Row and Column should be integers");
21     fi;
22     if elmRow < 1 or elmRow > symbols.rows then
23         Error("element row is outside of state");
24     fi;
25     if elmCol < 1 or elmCol > symbols.cols then
26         Error("element column is outside of state");
27     fi;
28     # Call roundAES and consequently return one state element
29     state := roundAES(symbols, func, params1, params2, flags);
30     return state[elmRow][elmCol];
31 end);
```

Listing 17: roundAESElm function

3.5 Testing

Now that all necessary functions are written for both the finite field implementation and vector implementation, test cases can be written. There are three test files: `testffAES.g`, `testvecAES.g` and `compareAES.g`.

The expressions can be tested using the `Value()` function. This function can assign concrete values to symbolic elements, resulting in a concrete result. The line thereafter indicates what the resulting output should be.

`testffAES.g`

After creating the GAP code for symbolic finite field expressions, a test case can be written to check if all elements of the state are the same after one round of AES. The file `testffAES.g` is used for this purpose.

All elements are transformed with the same formula during the `subBytes` step. The elements are not modified in the `shiftRows` step, and the `mixColumns` step uses the same row-elements in a different order. Hence, if all symbolic elements get assigned the same value, the result should be the same for all elements: Whichever element we take, the resulting output is the same.

To test this, the symbols are retrieved first using the `initSymbolsFFe` function. After that, all state elements are tested with the `Value()` function. For both options, exclusively Ones (`One(F)`) and Zeros (`Zero(F)`) are tested. Since both these values are elements of $\mathbb{F}_2$, we can also plug these into the vector representation later.

Given that (i) 1^{254} is equal to 1, (ii) $(\beta + 1) + \beta + 1 + 1$ can be reduced to 1 and (iii) the `shiftRows` step doesn't transform the state, only the affine transformation step will determine the output that needs to be checked. The affine transformation can in this case be described as:

$$\alpha^{232} \cdot x^{128} + \alpha^{96} \cdot x^{64} + x^{32} + \alpha^{197} \cdot x^{16} + \alpha^{185} \cdot x^8 + \alpha^{99} \cdot x^4 + \alpha^{199} \cdot x^2 + \alpha^2 \cdot x + \alpha^{195}$$

Where the universal variable $x = 1$ can be used since all values are 1. This results in $Z(2^8)^{243}$, as also shown in the listing below:

```
1 gap> affExpr;  
2 Z(2^8)^232*x^128+Z(2^8)^96*x^64+x^32+Z(2^8)^197*x^16+Z(2^8)^185*x^8+Z  
   (2^8)^99*x^4+Z(2^8)^199*x^2+Z(2^8)^2*x+Z(2^8)^195  
3 gap> Value(affExpr, [x], [One(symbols.F)]);  
4 Z(2^8)^243
```

Listing 18: Result in case variable x is equal to 1

In the case of the Zeros: only α^{195} remains after the affine transformation. Hence, $Z(2^8)^{195}$ should be the result, where $Z(2^8)$ is denoted as the generator.

```
1 symbolsFFe := initSymbolsFFe(4, 4, f1);  
2 K := symbolsFFe.K;  
3 F := symbolsFFe.F;  
4 BF := symbolsFFe.BF;  
5 alpha := symbolsFFe.gen;
```

```

6 # Finite Field Elements before AES round
7 stateFFe := symbolsFFe.state;
8 # Finite Field Elements after AES round
9 newStateFFe := roundAES(symbolsFFe, invExpr, affExpr, [[0,1,2,3], MixM1
  ], [1, false]);
10 # Obtain initial state elements
11 rows := symbolsFFe.rows;
12 cols := symbolsFFe.cols;
13 stateSymbolsFFe := [];
14 for row in [1..rows] do
15   for col in [1..cols] do
16     Add(stateSymbolsFFe, stateFFe[row][col]);
17   od;
18 od;
19 Ones := [];
20 Zeros := [];
21 for i in stateSymbolsFFe do Add(Ones, One(F)); od;
22 for i in stateSymbolsFFe do Add(Zeros, Zero(F)); od;
23 # Test out elements, given all values are one
24 expected := Z(2^8)^243;
25 for row in [1..rows] do
26   for col in [1..cols] do
27     val := Value(newStateFFe[row][col], stateSymbolsFFe, Ones);
28     if val <> expected then
29       Error("Test failed at position [", row, "][", col, "]");
30     fi;
31   od;
32 od;
33 # Test out elements, given all values are zero
34 expected := Z(2^8)^195;
35 for row in [1..rows] do
36   for col in [1..cols] do
37     val := Value(newStateFFe[row][col], stateSymbolsFFe, Zeros);
38     if val <> expected then
39       Error("Test failed at position [", row, "][", col, "]");
40     fi;
41   od;
42 od;

```

Listing 19: testfile for finite field implementation

testvecAES.g

This test file does the same as `testvecAES.g`, but then for the vector expressions: It checks if all elements give the same value if exclusively Ones and Zeros are given to the `Value` function. In GAP, `One` and `Zero` return the corresponding multiplicative identity and the corresponding additive identity for the algebraic structure in question.

The symbolic variables for vector representations are obtained using `initSymbolsVEC`. Then, arrays

for the Ones and Zeros are created, along with the array of symbols. The $\text{One}(F)$, in $\mathbb{F}_{2^8}$, that was used in the finite field implementation corresponds with $[\text{Zero}(K), \text{Zero}(K), \text{Zero}(K), \text{Zero}(K), \text{Zero}(K), \text{Zero}(K), \text{Zero}(K), \text{One}(K)]$ in $\mathbb{F}_2^8$, that will be used in the vector space implementation. Likewise, $\text{Zero}(F)$ in $\mathbb{F}_{2^8}$ corresponds with $[\text{Zero}(K), \text{Zero}(K), \text{Zero}(K), \text{Zero}(K), \text{Zero}(K), \text{Zero}(K), \text{Zero}(K), \text{Zero}(K)]$ in $\mathbb{F}_2^8$.

The `Value` function will thereafter test the value of every element. By multiplying the vector by a basis (dot product), one element in $\mathbb{F}_{2^8}$ can be obtained, which makes the result more convenient for checking. If we would plug in the Ones, and thereafter multiply the results by the basis, we would receive $Z(2^8)^{243}$ as the result.

When plugging in the Zeros, only constant $c = Z(2^8)^{195}$ should be returned.

```

1  ### Shortened for brevity
2  for i in [1..Length(stateSymbolsVEC)] do
3      if i mod 8 = 0 then
4          Add(Ones, One(K));
5      else
6          Add(Ones, Zero(K));
7      fi;
8  od;
9  for i in stateSymbolsVEC do Add(Zeros, Zero(K)); od;
10 # Test out elements, given all values are one
11 expected := Z(2^8)^243;
12 for row in [1..rows] do
13     for col in [1..cols] do
14         val := Value(newStateVEC[row][col] * BG, stateSymbolsVEC, Ones);
15         if val <> expected then
16             Error("Test failed at position [" , row, "][" , col, "]");
17         fi;
18     od;
19 od;
20 # Test out elements, given all values are zero
21 expected := Z(2^8)^195;
22 for row in [1..rows] do
23     for col in [1..cols] do
24         val := Value(newStateVEC[row][col] * BG, stateSymbolsVEC, Zeros)
25         ;
26         if val <> expected then
27             Error("Test failed at position [" , row, "][" , col, "]");
28         fi;
29     od;
30 od;

```

Listing 20: testfile for Vector implementation

compareAES.g

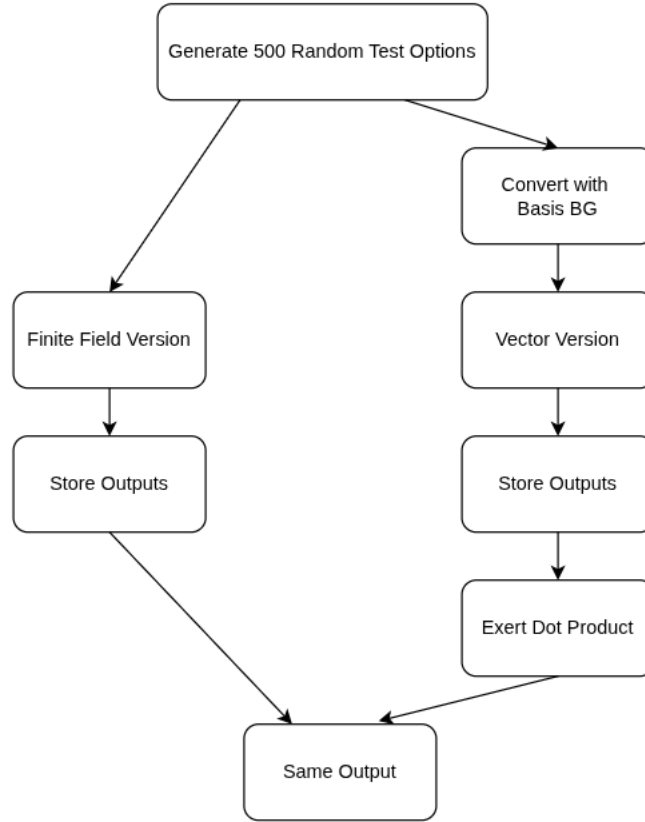
The finite field implementation of one round AES executes the steps in a different way than the vector implementation of one round AES. However, they should both provide the same output if the same input values are given. To test this, the testfile `compareAES.g` is created.

This file starts with the same variables as seen in `testffaes.g` and `testvecAES.g`. After that, a random value α^i , with some integer i , will be chosen to use as input for the finite field implementation using array `constsFF`. That same value will be assigned to `constsVEC`, but then in the right format for the vectors. Then, a random element of the state will be chosen. The finite field element and vector element can conclusively be derived using `Value()`. The result of the finite field representation and the result of the vector representation should be the same, otherwise an error is given. This process is repeated 500 times to test out a variety of different value options and state element options. The test program should not throw an error, thereby indicating the correctness of generating both finite field and vector representations.

```
1 ### Same FF variables and vector variables: shortened for brevity
2
3 # Testing out if resultVEC is the same as resultFFe (500x)
4 for i in [1..500] do
5     # Create Constants (constsFFe = constsVEC)
6     constsFFe := [];
7     for elmNr in [1..16] do Add(constsFFe, alpha^Random([0..254])); od;
8     constsVEC := [];
9     for elmFFe in constsFFe do for elmVEC in IntFFExt(BG, elmFFe) do Add(
        constsVEC, elmVEC*One(K)); od; od;
10
11     # Determine Row and Column (constsFFe = constsVEC)
12     elmRow := Random([1..4]);
13     elmCol := Random([1..4]);
14     elementFFe := newStateFFe[elmRow][elmCol];
15     resultFFe := Value(elementFFe, stateSymbolsFFe, constsFFe);
16     elementVEC := newStateVEC[elmRow][elmCol] * BG;
17     resultVEC := Value(elementVEC, stateSymbolsVEC, constsVEC);
18     if resultFFe <> resultVEC then
19         Error("Mismatch detected.");
20     fi;
21 od;
```

Listing 21: testfile for comparing Finite Field and Vector implementation

When running `compare.g`, the output of the finite field implementation is the same as the output of the vector representation multiplied with the basis (dot product), because $\mathbb{F}_{2^8} \iff \mathbb{F}_2^8$.



This is exactly the goal of the two implementations: They come up with different expressions, but they converge on the same output when the same constants are given. If that would not be the case, they would be different ciphers.

3.6 Profiling and discussion

Now that **SYMSPN** has been created, profiling can be done. Profiling is useful because it generated performance statistics and helps to identify bottlenecks in the package. Since the finite field representations and vector space representations are computed in a different way, both variants need to be modeled.

While working on this section, the Razer Blade 15 Base Model (Late 2020) has been used. The following specifications apply:

- CPU: 10th Gen Intel Core i7-10750H processor 6 Cores / 12 Threads, 2.6 GHz / 5.0 GHz (Base / Max Turbo), 64KB L1 Cache per Core, 256KB L2 Cache per Core, 12 MB Shared L3 Cache
- Memory: 16GB dual-channel DDR4-2933 MHz (8GB x 2)

When executing one round of AES, the following execution times were derived, which were obtained using the **Runtime** function.

Step	Finite Field	Vector Space
SubBytes	1008	36052
ShiftRows	0	0
MixColumns	2	19

Table 1: Time (ms) for executing every step

To understand the differences in execution time between the steps and between the implementations, profiling methods will be used.

Profiling Methods

GAP has profiling functionalities built in. With these profiling functions, you can observe how much each function has been called, and the time it took to complete those functions. The function `ProfileGlobalFunctions()` profiles all global functions to track how often and how long each is called during program execution. `ProfileOperationsAndMethods()` also enables profiling, but counts the operations and method calls. It helps to identify which operations are most frequently used. The profile statistics can be viewed using `DisplayProfile()`. It will print out every function along with the number of callings. The amount of milliseconds that the function was in process (both including and excluding child processes) is also shown. With `ClearProfile()`, the profiling can start again. This is useful if you want to profile functions one by one in an isolated manner.

With the following GAP code, the functions of SYMSPN have been profiled:

```

1 gap> SetPackagePath("SYMSPN", "/home/dirk/GAP/SYMSPN");
2 gap> LoadPackage("SYMSPN");
3 gap> ProfileGlobalFunctions(true);
4 gap> ProfileOperationsAndMethods(true);
5
6 gap> ### FINITE FIELD
7 gap> symbols := initSymbolsFFe(4, 4, f1);;
8 gap> DisplayProfile();
9 gap> ClearProfile();
10 gap> subBytes(symbols, invExpr, affExpr, true);;
11 gap> DisplayProfile();
12 gap> ClearProfile();
13 gap> shiftRows(symbols, [0,1,2,3], true);;
14 gap> DisplayProfile();
15 gap> ClearProfile();
16 gap> mixColumns(symbols, MixM1, true);;
17 gap> DisplayProfile();
18 gap> ClearProfile();
19
20 gap> ### VECTOR SPACE
21 gap> symbols := initSymbolsVEC(4, 4, f2);;
22 gap> DisplayProfile();
23 gap> ClearProfile();
24 gap> subBytes(symbols, FFA_inv_matrixU, L, c_vector, true);;

```

```

25 gap> DisplayProfile();
26 gap> ClearProfile();
27 gap> shiftRows(symbols, [0,1,2,3], true);;
28 gap> DisplayProfile();
29 gap> ClearProfile();
30 gap> mixColumns(symbols, MixM2, true);;
31 gap> DisplayProfile();
32 gap> ClearProfile();
33
34 gap> ProfileGlobalFunctions(false);
35 gap> ProfileOperationsAndMethods(false);

```

Listing 22: Profiling code for SYMSPN functions

The total results of the functions are as follows:

Function (Finite Field)	Self/ms	Stor/kb
initSymbolsFFe	48	10592
subBytes	4518	52901
shiftRows	1	0
mixColumns	6	179
Function (Vector Space)	Self/ms	Stor/kb
initSymbolsVEC	41	189
subBytes	107883	7140120
shiftRows	0	0
mixColumns	67	6149

Table 2: GAP profiling results for finite field and vector-based functions

self/ms measures the amount of time a function spent executing its own code, excluding the time spent in any functions it called. Note that it takes longer this time because of the profiling methods. **stor/kb** shows the amount of memory allocated directly by that function during its execution.

Interpreting the results

What is clear when looking at Table 1, is that the `subBytes` function takes the longest time to execute. This is absolutely the case for the vector representation, where it took more than 36 seconds to complete this step while the finite field representation took about one second. The `SubBytes` step can be calculated easily via one formula in the finite field representation $\mathbb{F}_{2^8}$, while that same step appears much more cumbersome in the vector space version $\mathbb{F}_2^8$. Especially calculating the inverse is way easier in finite field representation, since Fermat's Little Theorem can be used: taking $s \mapsto s^{254}$ gives the inverse. On the other hand, the vector representation cannot use an exponentiation like that, which makes computing the inverse more computationally exhaustive. This is the main difference in terms of execution time between the two implementations.

We can also see in Table 2 that the `subBytes` step uses the most memory: approximately 53 megabytes in the finite field implementation, and 7,1 gigabytes in the vector space implementation. The required amount of memory should be kept in mind if multiple rounds of AES will be executed:

The memory capacity should be big enough to fulfill those rounds.

When looking at the vector space implementation in Table 2 specifically, we can see that 107588 milliseconds out of the 107883 milliseconds can be attributed to the function `FFA_inv_matrixU`, which was given as a parameter to the `subBytes` function. `FFA_inv_matrixU` takes a lot of time primarily because it relies on `FFA_exp_matrixU`, which computes repeated multiplications of symbolic vectors. When investigating further, we find that this function spends the bulk of the execution time on `ReduceMonomialsOverField`, with 103433 milliseconds, to ensure that the computations stay within the vector space $\mathbb{F}_2^8$.

In case a more efficient algorithm is designed to compute the symbolic inverse of a vector element, the performance of one round AES could be improved drastically.

Multiple Rounds

So far, only one round of AES has been discussed. That is because executing multiple rounds of AES takes a very long time and requires a much more powerful computer. On the laptop that has been used for profiling, finishing multiple rounds of AES is not feasible given the time constraints. This is the case for both the finite field variant and the vector variant.

To showcase that much more time is needed to compute more than one round, an experiment is set up. One thing to note is that this experiment is for profiling purposes only. This experiment is designed to illustrate that executing multiple rounds is very computationally exhaustive, and hence is not meant to be used for non-testing purposes.

The experiment is as follows:

For both the finite field and vector representation, one round of AES is executed. Then, all state elements are set to their initial state, except for the first state element (`state[1][1]`). This way, we can isolate just one element for transformation. The whole state element for finite field representation is as follows:

$$\begin{aligned} &\alpha^{27} * s11^{254} + \alpha^3 * s22^{254} + \alpha^2 * s33^{254} + \alpha^2 * s44^{254} + \alpha^{224} * s11^{253} + \alpha^{200} * s22^{253} + \alpha^{199} * s33^{253} + \\ &\alpha^{199} * s44^{253} + \alpha^{124} * s11^{251} + \alpha^{100} * s22^{251} + \alpha^{99} * s33^{251} + \alpha^{99} * s44^{251} + \alpha^{210} * s11^{247} + \alpha^{186} * s22^{247} + \\ &\alpha^{185} * s33^{247} + \alpha^{185} * s44^{247} + \alpha^{222} * s11^{239} + \alpha^{198} * s22^{239} + \alpha^{197} * s33^{239} + \alpha^{197} * s44^{239} + \alpha^{25} * \\ &s11^{223} + \alpha * s22^{223} + s33^{223} + s44^{223} + \alpha^{121} * s11^{191} + \alpha^{97} * s22^{191} + \alpha^{96} * s33^{191} + \alpha^{96} * s44^{191} + \alpha^2 * \\ &s11^{127} + \alpha^{233} * s22^{127} + \alpha^{232} * s33^{127} + \alpha^{232} * s44^{127} + \alpha^{195} \end{aligned}$$

Transforming the first element is also too computationally exhaustive. Therefore, only a small part of the expression will be used. From there, we build up the length of that expression part. So we start off with just the first term, then the second term is added, then the third term, etc. This process repeats until more than one hour of execution time is needed. This process is also repeated for the vector implementation. There, the terms of the first entry of the first state will be build up, similar to the finite field representation. The first component of the vector representation:

$$\begin{aligned} &s11_7 * s11_6 * s11_5 * s11_4 * s11_3 * s11_1 * s11_0 + s11_7 * s11_6 * s11_5 * s11_4 * s11_2 * s11_1 * \\ &s11_0 + s22_7 * s22_6 * s22_5 * s22_4 * s22_3 * s22_2 * s22_0 + s22_7 * s22_6 * s22_5 * s22_4 * s22_2 * \\ &s22_1 * s22_0 + s33_7 * s33_6 * s33_5 * s33_4 * s33_3 * s33_2 * s33_0 + s33_7 * s33_6 * s33_5 * s33_4 * \\ &s33_3 * s33_1 * s33_0 + s44_7 * s44_6 * s44_5 * s44_4 * s44_3 * s44_2 * s44_0 + s44_7 * \dots \end{aligned}$$

The results are as follows:

Finite Field expression	time (ms)
$\alpha^{27} * s11^{254}$	686
$\alpha^{27} * s11^{254} + \alpha^3 * s22^{254}$	58562
$\alpha^{27} * s11^{254} + \alpha^3 * s22^{254} + \alpha^2 * s33^{254}$	> 3600000
Vector Space expression	time (ms)
$s11_7 * s11_6 * s11_5 * s11_4 * s11_3 * s11_1 * s11_0$	29212
$s11_7 * s11_6 * s11_5 * s11_4 * s11_3 * s11_1 * s11_0 + s11_6 * \dots$ (5 terms)	38810
$s11_7 * s11_6 * s11_5 * s11_4 * s11_3 * s11_1 * s11_0 + s11_6 * \dots$ (10 terms)	56327
$s11_7 * s11_6 * s11_5 * s11_4 * s11_3 * s11_1 * s11_0 + s11_6 * \dots$ (20 terms)	128605
$s11_7 * s11_6 * s11_5 * s11_4 * s11_3 * s11_1 * s11_0 + s11_6 * \dots$ (40 terms)	2908454

Table 3: GAP profiling results for finite field and vector-based functions

As can be seen in Table 3, the time to compute a second AES round will grow exponentially. Just a few terms of one state element quickly causes more than one hour of execution time. Therefore, computing multiple rounds of AES symbolically is practically infeasible.

However, there are potential ways to reduce the execution time. Right now, GAP only uses one CPU core at a time, as can be seen using the Unix command “top”.

Using multiple cores in parallel can help speed up symbolic SPN cipher computation by distributing independent or repeated tasks across multiple CPU cores. Ciphers that have multiple rounds with independent or structurally similar computations (like substitution and permutation layers) can be executed in parallel. Even within a round, applying substitutions to different symbolic elements can be parallelized since the elements are independent of each other during that step.

GAP has a limited number of options when it comes to parallel computing, but *HPC-GAP* is one of them [BHJ⁺16]. HPC-GAP is a modified version of the standard GAP interpreter that includes built-in support for multi threading and parallel computing.

Apart from using parallelism to reduce execution time, a lookup table might also be used. Using a lookup table for symbolic SPNs can significantly reduce computation time by avoiding repeated symbolic computations. Instead of recomputing the symbolic output during the subBytes, shiftRows or mixColumns steps, precomputed symbolic expressions can be stored and reused. This reduces the execution time of symbolic operations into a fast table lookup. It also allows symbolic simplification to be done once during preprocessing, rather than during every execution.

In appendix C, examples codes are provided to showcase that multiple rounds do work in case less complex expressions and functions are used. These codes are meant to show that the SYMSPN package is capable of executing multiple rounds without problems.

4 Future Work and Conclusion

The SYMSPN package is designed in such a way that it can be expanded. This opens up for utilization of future projects. A Boolean function can be created, which paves the way for a cryptographic analyzer. This package can also be expanded to include other ciphers, so that comparisons can take place.

4.1 Symbolic AES as a Boolean Function

After symbolically expressing AES in finite fields and vector spaces, a Boolean function can also be added. A Boolean function is a function that operates solely on binary inputs - values that are either 0 or 1. We can express these boolean functions in Algebraic Normal Form over field $\mathbb{F}_2$, where elements can be added (XOR) and multiplied (AND).

4.2 GAP functions for analysis

Using Boolean expressions shows the behaviour of SPNs at the level of individual bits. This low-level perspective is particularly valuable for analyzing cryptographic strength at the bit-level. This means that a cryptographic analyzer can be created. This analyzer, as its name implies, can analyze the cryptographic properties like non-linearity and algebraic immunity. This is useful for checking the cryptographic robustness and building efficient hardware versions - tasks not feasible with the current finite field/vector representations.

Cryptographic properties can be analyzed by creating new functions. Example functions would be `compute_algebraic_immunity(expr)`, `get_algebraic_degree(expr)` and `measure_nonlinearity(expr)`.

4.3 Modelling Similar Ciphers

The SYMSPN package can be expanded to encompass other ciphers. Block ciphers like *Lumora* have been developed for *Post Quantum Cryptography* (PQC) [SG25]. Lumora has a structure similar to AES, with `SubCell`, `ShiftRows` and `MixColumn` operations. Because of their similar structure, it would be very interesting to compare the symbolical representations of AES and Lumora: The similarities and differences can be compared, the complexity of both algorithms can be inferred, and cryptographic properties at a symbolic level can be assessed. This way, we can conclude what the desirable properties are for a given cipher.

4.4 Conclusion

To conclude, in this thesis a new GAP package has been created called SYMSPN. This package is designed to encompass symbolic expressions of substitution-permutation networks in a Computer Algebra System. Right now, the SYMSPN package contains the block cipher AES. Symbolic AES expressions can be created in both finite fields and vector spaces. Both in finite field and vector form, the expressions of all elements can be retrieved using the functions provided in the package.

This package is useful because it allows visualization and manipulations of SPNs symbolically. Since concrete values can be assigned to the symbolic elements, the cryptographic strength of a cipher can be derived. This package has been designed such that it be expanded by future implementations of other ciphers. Hence, the package is created with generalization and scalability in mind. By extending this package, new features can be added. For instance, a Boolean Function can be added with an analyzer of cryptographic properties. Other substitution-permutation networks can also be added, like the Lumora cipher. By integrating multiple ciphers within the package, ciphers can be compared and robustness can be measured reciprocally.

References

- [BHJ⁺16] Reimer Behrends, Kevin Hammond, Vladimir Janjic, Alexander Konovalov, Steve Linton, Hans-Wolfgang Loidl, Patrick Maier, and Phil Trinder. Hpc-gap: engineering a 21st-century high-performance computer algebra system. *Concurrency and Computation: Practice and Experience*, 28(13):3606–3636, 2016.
- [DR02] Joan Daemen and Vincent Rijmen. *The design of Rijndael*, volume 2. Springer, 2002.
- [Lin07] Steve Linton. Gap: groups, algorithms, programming. *ACM Communications in Computer Algebra*, 41(3):108–109, 2007.
- [MO24] Pierrick Méaux and Yassine Ozaim. On the cryptographic properties of weightwise affine and weightwise quadratic functions. *Discrete Applied Mathematics*, 355:13–29, 2024.
- [MP13] Gary L Mullen and Daniel Panario. *Handbook of finite fields*, volume 17. CRC press Boca Raton, 2013.
- [RS05] Hans Dobbertin Vincent Rijmen and Aleksandra Sowa. Encryption standard–aes. 2005.
- [SG25] Susanta Samanta and Guang Gong. Lumora: A family of permutation based wide-block ciphers for pqc zksnark applications. 2025.
- [Sho22] Dawson M Shores. Cryptography through the lens of group theory. 2022.
- [ZAG19] Nusa Zidaric, Mark Aagaard, and Guang Gong. Feedback shift register package. 2019.
- [ZGA⁺23] Nuša Zidarič, Guang Gong, Mark Aagaard, Aleksandar Jurišić, and Olexandr Konovalov. Ffcsa-finite field constructions, search, and algorithms. *ACM Communications in Computer Algebra*, 57(2):57–64, 2023.
- [Zid20] Nusa Zidaric. Automated design space exploration and datapath synthesis for finite field arithmetic with applications to lightweight cryptography. 2020.
- [Zid23] Nusa Zidaric. Finite field constructions, search and algorithms. 2023.
- [ZMGA24] N Zidarič, Kalikinkar Mandal, Guang Gong, and M Aagaard. The welch-gong stream cipher-evolutionary path. *Cryptography and Communications*, 16(1):129–165, 2024.

Appendix

A. Creating a GAP package

To create a package in GAP, a folder has to be created first, say `mypack/`. Within that folder, three new folders can be created: `lib/`, `doc/` and `tst/`.

- `lib/`: Contains the actual library code
- `doc/`: Contains documents about the library
- `tst/`: Contains test files to test if the library files function properly

In addition to those three folders, the files `init.g`, `read.g` and `PackageInfo.g` are also present.

`init.g`

The `init.g` file reads the declaration files that are present in the `lib/` folder. Reading in the package can be done with `ReadPackage()`.

```
1 ReadPackage("mypack", "lib/mypack.gd");
2 Print("mypack declaration files loaded.\n");
```

Listing 23: `init.g` file

`read.g`

The `read.g` file reads the implementation files that are present in the `lib/` folder. `ReadPackage()` is also used for this `read.g` file.

```
1 ReadPackage("mypack", "lib/mypack.gi");
2 Print("mypack implementation files loaded.\n");
```

Listing 24: `read.g` file

`PackageInfo.g`

The `PackageInfo.g` file, as its name implies, contains a lot of information about the package itself. This includes the version number, the date and the coder, among other things. More essential information is also present, like `GAPVersion`, which states what minimum GAP version is required to be able to use this package. Conclusively, the test file is also denoted.

```
1 SetPackageInfo( rec(
2   PackageName := "mypack",
3   Version := "1.0",
4   Date := "2025-04-16",
5   Persons := [
6     rec(
7       LastName := "van den Beuken",
8       FirstNames := "Dirk",
```

```

9      IsAuthor := true,
10     IsMaintainer := true,
11     Email := "dirk.vandenbeuken@gmail.com"
12   )
13 ],
14 TestFile := "tst/test.tst",
15 ) );

```

Listing 25: PackageInfo.g file

lib folder

The lib folder is the most important part of the package, since it includes the actual code the package is designed for.

This code is broken up into two type of files: (i) the declaration files, denoted with the `.gd` extension, and (ii) the implementation files, denoted with the `.gi` extension. The declaration files serve as header files. They usually contain a `DeclareGlobalFunction` folder, which refers to functions of the implementation files.

```

1 DeclareGlobalFunction("HelloWorld");
2 DeclareGlobalFunction("ArrayPlusOne");

```

Listing 26: lib/mypack.gd file

The implementation files, consequently, contain the functions that the declaration files invoke. This can be done using the `InstallGlobalFunction`.

```

1 InstallGlobalFunction("HelloWorld", function()
2   Print("Hello World\n");
3 end);
4
5 InstallGlobalFunction("ArrayPlusOne", function(array)
6   local newArray, number;
7   newArray := [];
8   for number in array do
9     Add(newArray, number + 1);
10  od;
11  return newArray;
12 end);

```

Listing 27: lib/mypack.gi file

doc folder

The doc folder contains the documentation files of the package. These documents provide explanations of the package's functions, usage examples, theoretical background, and developer notes. GAP uses the GAPDoc system for documentation, which involves writing documentation in XML format and compiling it into readable formats like PDF and HTML.

tst folder

The tst folder contains the test files, to make sure the code of the lib files function properly. In order to test, a function can be called. In the line below, you can note the return values you should receive. If you execute the test script, it should confirm the correct return value.

```
1 gap> ArrayPlusOne([1,2,3,4,5]);  
2 [ 2, 3, 4, 5, 6 ]
```

Listing 28: tst/test.tst file

B. SYMSPN package code

```
1 Print("SYMSPN init.g: Reading declaration files...\n");
2 ReadPackage("SYMSPN", "lib/general.gd");
3 ReadPackage("SYMSPN", "lib/ffSPN.gd");
4 ReadPackage("SYMSPN", "lib/vecSPN.gd");
5 Print("SYMSPN init.g: Done reading declaration files!\n");
```

Listing 29: init.g file

```
1 Print("SYMSPN read.g: Reading implementation files...\n");
2 ReadPackage("SYMSPN", "lib/globals.gi");
3 ReadPackage("SYMSPN", "lib/general.gi");
4 ReadPackage("SYMSPN", "lib/ffSPN.gi");
5 ReadPackage("SYMSPN", "lib/vecSPN.gi");
6 Print("SYMSPN read.g: Done reading implementation files!\n");
```

Listing 30: read.g file

```
1 K := GF(2);
2 x := X(K, "x");
3 f1 := x^8+x^4+x^3+x^2+1;
4 F1 := FieldExtension(K, f1);
5 alpha := RootOfDefiningPolynomial(F1);
6 BF1 := GeneratePBdownto(F1, alpha);
7 invExpr := x^254;
8 affExpr := alpha^2 * x + alpha^199 * x^2 + alpha^99 * x^4 + alpha^185 *
    x^8 + alpha^197 * x^16 + x^32 + alpha^96 * x^64 + alpha^232 * x^128 +
    alpha^195;
9 MixM1 := [
10 [alpha^25, alpha, One(K), One(K)],
11 [One(K), alpha^25, alpha, One(K)],
12 [One(K), One(K), alpha^25, alpha],
13 [alpha, One(K), One(K), alpha^25]
14 ];;
15
16
17 f2 := x^8+x^4+x^3+x+1;
18 F2 := FieldExtension(K, f2);
19 beta := RootOfDefiningPolynomial(F2);
20 BF2 := GeneratePBdownto(F2, beta);
21 L:= [
22 [One(K), One(K), One(K), One(K), One(K), Zero(K), Zero(K), Zero(K)],
23 [Zero(K), One(K), One(K), One(K), One(K), One(K), Zero(K), Zero(K)],
24 [Zero(K), Zero(K), One(K), One(K), One(K), One(K), One(K), Zero(K)],
25 [Zero(K), Zero(K), Zero(K), One(K), One(K), One(K), One(K), One(K)],
26 [One(K), Zero(K), Zero(K), Zero(K), One(K), One(K), One(K), One(K)],
27 [One(K), One(K), Zero(K), Zero(K), Zero(K), One(K), One(K), One(K)],
28 [One(K), One(K), One(K), Zero(K), Zero(K), Zero(K), One(K), One(K)],
```

```

29 [One(K), One(K), One(K), One(K), Zero(K), Zero(K), Zero(K), One(K)]
30 ];
31 c_vector := [0, 1, 1, 0, 0, 0, 1, 1]*One(K);
32 M1 := [
33 [One(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K)],
34 [Zero(K), One(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K)],
35 [Zero(K), Zero(K), One(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K)],
36 [Zero(K), Zero(K), Zero(K), One(K), Zero(K), Zero(K), Zero(K), Zero(K)],
37 [Zero(K), Zero(K), Zero(K), Zero(K), One(K), Zero(K), Zero(K), Zero(K)],
38 [Zero(K), Zero(K), Zero(K), Zero(K), Zero(K), One(K), Zero(K), Zero(K)],
39 [Zero(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K), One(K), Zero(K)],
40 [Zero(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K), One(K)]
41 ];
42 M2 := [
43 [Zero(K), One(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K)],
44 [Zero(K), Zero(K), One(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K)],
45 [Zero(K), Zero(K), Zero(K), One(K), Zero(K), Zero(K), Zero(K), Zero(K)],
46 [One(K), Zero(K), Zero(K), Zero(K), One(K), Zero(K), Zero(K), Zero(K)],
47 [One(K), Zero(K), Zero(K), Zero(K), Zero(K), One(K), Zero(K), Zero(K)],
48 [Zero(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K), One(K), Zero(K)],
49 [One(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K), One(K)],
50 [One(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K)]
51 ];
52 M3 := [
53 [One(K), One(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K)],
54 [Zero(K), One(K), One(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K)],
55 [Zero(K), Zero(K), One(K), One(K), Zero(K), Zero(K), Zero(K), Zero(K)],
56 [One(K), Zero(K), Zero(K), One(K), One(K), Zero(K), Zero(K), Zero(K)],
57 [One(K), Zero(K), Zero(K), Zero(K), One(K), One(K), Zero(K), Zero(K)],
58 [Zero(K), Zero(K), Zero(K), Zero(K), Zero(K), One(K), One(K), Zero(K)],
59 [One(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K), One(K), One(K)],
60 [One(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K), Zero(K), One(K)]
61 ];
62 MixM2 := [
63 [M2, M3, M1, M1],
64 [M1, M2, M3, M1],
65 [M1, M1, M2, M3],
66 [M3, M1, M1, M2]
67 ];

```

Listing 31: globals.gi

```

1 SetPackageInfo( rec(
2   PackageName := "SYMSPN",
3   Subtitle := "Generating Symbolic Expressions of Substitution-
4     Permutation Networks",
5   Version := "1.0",
6   Date := "2025-07-15",

```

```

6  ArchiveFormats := "tar.gz",
7  AbstractHTML := "This package provides functions to symbolically
    express SPNs",
8  Persons := [
9      rec(
10         LastName := "van den Beuken",
11         FirstNames := "Dirk",
12         IsAuthor := true,
13         IsMaintainer := true,
14         Email := "dirk.vandenbeuken@gmail.com"
15     )
16 ],
17 License := "GPL-2.0-or-later",
18 GAPVersion := "4.8",
19 Autoload := true,
20 Dependencies := rec(
21     NeededOtherPackages := [
22         [ "FSR", "1.2.2" ],
23         [ "FFCSA", "0.9.4" ]
24     ]
25 ),
26 AvailabilityTest := ReturnTrue,
27 TestFiles := ["tst/compareAES.g", "tst/testffaES.g", "tst/testvecAES
    .g"]
28 ) );

```

Listing 32: PackageInfo.g file

```

1  InstallMethod(roundAES, "roundAES in finite field",
2  [IsRecord, IsPolynomial, IsPolynomial, IsList, IsList],
3  function(symbols, expr1, expr2, params, flags)
4      ### Execute AES round(s) in finite fields
5      ### Return final state
6      local symbolscopy, state, shiftArr, mat, round, rounds,
          transformState;
7      # Check flags[] is valid
8      if Length(flags) <> 2 then
9          Error("Two flags are required: rounds and transformState");
10     fi;
11     rounds := flags[1];
12     transformState := flags[2];
13     if not IsBool(transformState) then
14         Error("transformState should be a Boolean");
15     fi;
16     if not IsInt(rounds) and rounds < 1 then
17         Error("Number of rounds has to be a positive number");
18     fi;
19     # Check if params[] is valid

```

```

20  if Length(params) <> 2 then
21      Error("2 parameters in list expected");
22  fi;
23  shiftArr := params[1];
24  mat := params[2];
25  if not IsList(shiftArr) then
26      Error("shiftArr must be a list");
27  fi;
28  if not IsMatrix(mat) then
29      Error("mat must be a matrix");
30  fi;
31  if not transformState then
32      # Transform copied state
33      symbolscopy := ShallowCopy(symbols);
34      symbolscopy.state := List(symbols.state, row -> ShallowCopy(row)
35      );
36      state := symbolscopy.state;
37  else
38      # Transform state
39      symbolscopy := symbols;
40      state := symbols.state;
41  fi;
42  # Error handling of shiftrows and mixcolumns already
43  # to ensure atomic operation
44  if Length(shiftArr) <> Length(state) then
45      Error("Length shiftArr is not equal to amount of state rows");
46  fi;
47  if Length(state) <> Length(mat[1]) then
48      Error("Length Mixcolumns Matrix is different from length of
49      state");
50  fi;
51  for round in [1..rounds] do
52      subBytes(symbolscopy, expr1, expr2, true);
53      shiftRows(symbolscopy, shiftArr, true);
54      mixColumns(symbolscopy, mat, true);
55  od;
56  return state;
57 end);

```

Listing 33: roundAES function for FF

```

1  InstallMethod(roundAES, "roundAES in vector space",
2  [IsRecord, IsFunction, IsList, IsList, IsList],
3  function(symbols, func, params1, params2, flags)
4      ### Execute AES round(s) in vector space
5      ### Return final state
6      local symbolscopy, state, mat1, vec, shiftArr, mat2, row, col, round
          , rounds, transformState;

```

```

7   # Check if flags[] is valid
8   if Length(flags) <> 2 then
9       Error("Two flags are required: rounds and transformState");
10  fi;
11  rounds := flags[1];
12  transformState := flags[2];
13  if not IsBool(transformState) then
14      Error("transformState should be a Boolean");
15  fi;
16  if not IsInt(rounds) and rounds < 1 then
17      Error("Number of rounds has to be a positive number");
18  fi;
19  # Check if params1[] and params2[] are valid
20  if Length(params1) <> 2 then
21      Error("2 items in params1[] expected");
22  fi;
23  if Length(params2) <> 2 then
24      Error("2 items in params2[] expected");
25  fi;
26  mat1 := params1[1];
27  vec := params1[2];
28  shiftArr := params2[1];
29  mat2 := params2[2];
30  if not IsMatrix(mat1) then
31      Error("For params1[1]: matrix must be given");
32  fi;
33  if not IsList(vec) then
34      Error("For params1[2]: list must be given");
35  fi;
36  if not IsList(shiftArr) then
37      Error("For params2[1]: list must be given");
38  fi;
39  if not IsList(mat2) then
40      Error("For params2[2]: matrix must be given");
41  fi;
42  for row in [1..Length(mat2[1])] do
43      for col in [1..Length(mat2)] do
44          if not IsMatrix(mat2[row][col]) then
45              Error("For params2[2]: matrix should consist of matrices
46                  ");
47          fi;
48      od;
49  od;
50  if not transformState then
51      # Transform copied state
52      symbolscopy := ShallowCopy(symbols);
53      symbolscopy.state := List(symbols.state, row -> List(row, cell
54          -> ShallowCopy(cell)));

```

```

53     state := symbolscopy.state;
54 else
55     # Transform state
56     symbolscopy := symbols;
57     state := symbols.state;
58 fi;
59 # Error handling of subbytes, shiftrows and mixcolumns already
60 # to ensure atomic operation
61 if Length(mat1) <> Length(vec) then
62     Error("Length of matrix and length of vector not the same");
63 fi;
64 if Length(shiftArr) <> Length(state) then
65     Error("Length shiftArr is not equal to amount of state rows");
66 fi;
67 if Length(state) <> Length(mat2[1]) then
68     Error("Length Mixcolumns Matrix is different from length of
        state");
69 fi;
70 # Execute AES steps in sequence
71 for round in [1..rounds] do
72     subBytes(symbolscopy, func, mat1, vec, true);
73     shiftRows(symbolscopy, shiftArr, true);
74     mixColumns(symbolscopy, mat2, true);
75 od;
76 return state;
77 end);

```

Listing 34: roundAES function for VEC

C. Example Usecases of SYMSPN

```

1 gap> # One round in seperate steps
2 gap> symbols := initSymbolsFFe(4, 4, f1);
3 rec( BF := Basis( GF(2^8), [ Z(2^8)^7, Z(2^8)^6, Z(2^8)^5, Z(2^8)^4, Z
   (2^8)^3, Z(2^8)^2, Z(2^8), Z(2)^0 ] ), F := GF(2^8), K := GF(2), cols
   := 4, f := x^8+x^4+x^3+x^2+Z(2)^0, gen := Z(2^8), rows := 4,
4   state := [ [ s11, s12, s13, s14 ], [ s21, s22, s23, s24 ], [ s31, s32,
   s33, s34 ], [ s41, s42, s43, s44 ] ] )
5 gap> subBytes(symbols, invExpr, affExpr, true);;
6 gap> shiftRows(symbols, [0,1,2,3], true);;
7 gap> mixColumns(symbols, MixM1, true);;
8 gap> symbols.state;
9 [ [ Z(2^8)^27*s11^254+Z(2^8)^3*s22^254+Z(2^8)^2*s33^254+Z(2^8)^2*s44
   ^254+Z(2^8)^224*s11^253+Z(2^8)^200*s22^253+Z(2^8)^199*s33^253+Z(2^8)
   ^199*s44^253+Z(2^8)^124*s11^251+Z(2^8)^100*s22^251+Z(2^8)^99*s33^251+
10  ### Shortened for brevity
11 gap>
12 gap> # One round in one go
13 gap> symbols := initSymbolsFFe(4, 4, f1);;
14 gap> roundAES(symbols, invExpr, affExpr, [[0,1,2,3],MixM1], [1,true]);;
15 gap> symbols.state;
16 [ [ Z(2^8)^27*s11^254+Z(2^8)^3*s22^254+Z(2^8)^2*s33^254+Z(2^8)^2*s44
   ^254+Z(2^8)^224*s11^253+Z(2^8)^200*s22^253+Z(2^8)^199*s33^253+Z(2^8)
   ^199*s44^253+Z(2^8)^124*s11^251+Z(2^8)^100*s22^251+Z(2^8)^99*s33^251+
17  ### Shortened for brevity
18 gap>
19 gap> # One element (state[1][1]) after one round
20 gap> symbols := initSymbolsFFe(4, 4, f1);;
21 gap> roundAESElm(symbols, invExpr, affExpr, [[0,1,2,3], MixM1], [1, 1],
   [1, true]);;
22 Z(2^8)^27*s11^254+Z(2^8)^3*s22^254+Z(2^8)^2*s33^254+Z(2^8)^2*s44^254+Z
   (2^8)^224*s11^253+Z(2^8)^200*s22^253+Z(2^8)^199*s33^253+Z(2^8)^199*
   s44^253+Z(2^8)^124*s11^251+Z(2^8)^100*s22^251+Z(2^8)^99*s33^251+Z
   (2^8)^99*s44^251+Z(2^8)^210*s11^247+Z(2^8)^186*s22^247+Z(2^8)^185*s33
   ^247+Z(2^8)^185*s44^247+Z(2^8)^222*s11^239+Z(2^8)^198*s22^239+Z(2^8)
   ^197*s33^239+Z(2^8)^197*s44^239+Z(2^8)^25*s11^223+Z(2^8)*s22^223+s33
   ^223+s44^223+Z(2^8)^121*s11^191+Z(2^8)^97*s22^191+Z(2^8)^96*s33^191+Z
   (2^8)^96*s44^191+Z(2^8)^2*s11^127+Z(2^8)^233*s22^127+Z(2^8)^232*s33
   ^127+Z(2^8)^232*s44^127+Z(2^8)^195

```

Listing 35: One round of AES in finite fields

```

1 gap> # One round in seperate steps
2 gap> symbols := initSymbolsVEC(4, 4, f2);
3 rec( BF := Basis( GF(2^8), [ Z(2^8)^175, Z(2^8)^150, Z(2^8)^125, Z(2^8)
   ^100, Z(2^8)^75, Z(2^8)^50, Z(2^8)^25, Z(2)^0 ] ), F := GF(2^8), K :=
   GF(2), cols := 4, f := x^8+x^4+x^3+x+Z(2)^0, gen := Z(2^8)^25,

```

```

4  rows := 4, state := [ [ [ s11_7, s11_6, s11_5, s11_4, s11_3, s11_2,
    s11_1, s11_0 ], [ s12_7, s12_6, s12_5, s12_4, s12_3, s12_2, s12_1,
    s12_0 ], [ s13_7, s13_6, s13_5, s13_4, s13_3, s13_2, s13_1, s13_0
    ],
5      [ s14_7, s14_6, s14_5, s14_4, s14_3, s14_2, s14_1, s14_0 ] ],
    [ [ s21_7, s21_6, s21_5, s21_4, s21_3, s21_2, s21_1, s21_0
    ], [ s22_7, s22_6, s22_5, s22_4, s22_3, s22_2, s22_1, s22_0
    ] ],
6      [ s23_7, s23_6, s23_5, s23_4, s23_3, s23_2, s23_1, s23_0 ], [
    s24_7, s24_6, s24_5, s24_4, s24_3, s24_2, s24_1, s24_0 ] ],
7      [ [ s31_7, s31_6, s31_5, s31_4, s31_3, s31_2, s31_1, s31_0 ], [
    s32_7, s32_6, s32_5, s32_4, s32_3, s32_2, s32_1, s32_0 ], [
    s33_7, s33_6, s33_5, s33_4, s33_3, s33_2, s33_1, s33_0 ],
8      [ s34_7, s34_6, s34_5, s34_4, s34_3, s34_2, s34_1, s34_0 ] ],
    [ [ s41_7, s41_6, s41_5, s41_4, s41_3, s41_2, s41_1, s41_0
    ], [ s42_7, s42_6, s42_5, s42_4, s42_3, s42_2, s42_1, s42_0
    ] ],
9      [ s43_7, s43_6, s43_5, s43_4, s43_3, s43_2, s43_1, s43_0 ], [
    s44_7, s44_6, s44_5, s44_4, s44_3, s44_2, s44_1, s44_0 ] ]
    ] )
10 gap> subBytes(symbols, FFA_inv_matrixU, L, c_vector, true);;
11 gap> shiftRows(symbols, [0,1,2,3], true);;
12 gap> mixColumns(symbols, MixM2, true);;
13 gap> symbols.state;
14 [ [ [ s11_7*s11_6*s11_5*s11_4*s11_3*s11_1*s11_0+s11_7*s11_6*s11_5*s11_4*
    s11_2*s11_1*s11_0+s22_7*s22_6*s22_5*s22_4*s22_3*s22_2*s22_0+s22_7*
    s22_6*s22_5*s22_4*s22_2*s22_1*s22_0+s33_7*s33_6*s33_5*s33_4*s33_3*
15 ### Shortened for brevity
16 gap>
17 gap> # One round in one go
18 gap> symbols := initSymbolsVEC(4, 4, f2);;
19 gap> roundAES(symbols, FFA_inv_matrixU, [L, c_vector], [[0,1,2,3], MixM2
    ], [1, true]);;
20 gap> symbols.state;
21 [ [ [ s11_7*s11_6*s11_5*s11_4*s11_3*s11_1*s11_0+s11_7*s11_6*s11_5*s11_4*
    s11_2*s11_1*s11_0+s22_7*s22_6*s22_5*s22_4*s22_3*s22_2*s22_0+s22_7*
    s22_6*s22_5*s22_4*s22_2*s22_1*s22_0+s33_7*s33_6*s33_5*s33_4*s33_3*
22 ### Shortened for brevity
23 gap>
24 gap> # One element (state[1][1]) after one round
25 gap> symbols := initSymbolsVEC(4, 4, f2);;
26 gap> roundAESElm(symbols, FFA_inv_matrixU, [L, c_vector], [[0,1,2,3],
    MixM2], [1, 1], [1, true]);;
27 [ s11_7*s11_6*s11_5*s11_4*s11_3*s11_1*s11_0+s11_7*s11_6*s11_5*s11_4*
    s11_2*s11_1*s11_0+s22_7*s22_6*s22_5*s22_4*s22_3*s22_2*s22_0+s22_7*
    s22_6*s22_5*s22_4*s22_2*s22_1*s22_0+s33_7*s33_6*s33_5*s33_4*s33_3*
28 ### Shortened for brevity

```

Listing 36: One round of AES in vector space

```

1 gap> symbols := initSymbolsFFe(4, 4, f1);;
2 gap> # Using dummy exponent variable
3 gap> expr1 := x^3;
4 x^3
5 gap> # Using dummy transformation polynomial
6 gap> expr2 := alpha^2 * x + alpha^199 * x^2 + alpha^99 * x^4;
7 Z(2^8)^99*x^4+Z(2^8)^199*x^2+Z(2^8)^2*x
8 gap> roundAES(symbols, expr1, expr2, [[0,1,2,3], MixM1], [2, true]);;
9 gap> symbols.state;
10 [ Z(2^8)^82*s11^144+Z(2^8)^241*s11^96*s22^48+Z(2^8)^237*s11^96*s33^48+Z
    (2^8)^237*s11^96*s44^48+Z(2^8)^145*s11^48*s22^96+Z(2^8)^137*s11^48*
    s33^96+Z(2^8)^137*s11^48*s44^96+Z(2^8)^13*s12^144+Z(2^8)^113*s12^96*
    s23^48+Z(2^4)*s12^96*s34^48+Z(2^8)^13*s12^96*s41^48+Z(2^8)^213*s12
    ^48*s23^96+Z(2^8)^21*s12^48*s34^96+Z(2^8)^13*s12^48*s41^96+Z(2^8)^12*
    s13^144+Z(2^8)^12*s13^96*s24^48+Z(2^8)^112*s13^96*s31^48+Z(2^8)^16*
11 ### Shortened for brevity

```

Listing 37: Multiple rounds in finite fields

```

1 gap> symbols := initSymbolsVEC(4, 4, f2);;
2 gap> # Using dummy function to showcase multiple rounds
3 gap> func1 := function(BF, s)
4 >         return BF * One(K);
5 > end;
6 function( BF, s ) ... end
7 gap> roundAES(symbols, func1, [L, c_vector], [[0,1,2,3], MixM2], [2,
    true]);;
8 gap> symbols.state;
9 [ [ [ Z(2^8)^13, Z(2^8)^115, Z(2^8)^142, Z(2^8)^193, Z(2^8)^232, Z(2^8)
    ^252, Z(2^8)^146, Z(2^8)^103 ], [ Z(2^8)^13, Z(2^8)^115, Z(2^8)^142,
    Z(2^8)^193, Z(2^8)^232, Z(2^8)^252, Z(2^8)^146, Z(2^8)^103 ],
10     [ Z(2^8)^13, Z(2^8)^115, Z(2^8)^142, Z(2^8)^193, Z(2^8)^232, Z
        (2^8)^252, Z(2^8)^146, Z(2^8)^103 ], [ Z(2^8)^13, Z(2^8)^115, Z
        (2^8)^142, Z(2^8)^193, Z(2^8)^232, Z(2^8)^252, Z(2^8)^146, Z
        (2^8)^103 ] ],
11 [ [ Z(2^8)^13, Z(2^8)^115, Z(2^8)^142, Z(2^8)^193, Z(2^8)^232, Z(2^8)
    ^252, Z(2^8)^146, Z(2^8)^103 ], [ Z(2^8)^13, Z(2^8)^115, Z(2^8)
    ^142, Z(2^8)^193, Z(2^8)^232, Z(2^8)^252, Z(2^8)^146, Z(2^8)^103 ],
12 ### Shortened for brevity

```

Listing 38: Multiple rounds in vector space

```

1 gap> SetPackagePath("SYMSPN", "/path/to/SYMSPN");
2 gap> LoadPackage("SYMSPN");
3 gap> ReadPackage("SYMSPN", "tst/testffAES.g");

```

```
4 true
5 gap> ReadPackage("SYMSPN", "tst/testvecAES.g");
6 true
7 gap> ReadPackage("SYMSPN", "tst/compareAES.g");
8 true
```

Listing 39: Executing test files