

**American University of Beirut**

**EECE 455 – Cryptography and Network Security**

**Fall 2023-2024**

**Course Project**

*27-November-2023*

***Students: David Abboud - Alexander Menassa***

***Instructor****:* ***Professor Khalil Hariss***

Table of Contents

[Functionality Description 3](#_Toc152063608)

[Theory Summary 5](#_Toc152063609)

[Modular Polynomial Arithmetic 5](#_Toc152063610)

[Representation of a Polynomial 7](#_Toc152063611)

[Addition and Subtraction 8](#_Toc152063612)

[Multiplication 9](#_Toc152063613)

[Modulo Function 11](#_Toc152063614)

[Quotient Function 13](#_Toc152063615)

[Extended Euclidean Algorithm, Inverse, and Division 14](#_Toc152063616)

[Graphical User Interface 16](#_Toc152063617)

[Prospects 18](#_Toc152063618)

# Functionality Description

Through this paper, we will be presenting a project to implement modular polynomial arithmetic in python. Namely, the code will allow a user to perform arithmetic on polynomials in 𝐺𝐹(2^m), for m=2,3,4,5,6,7,8 as well as m=163,233,239,283,409,571.

The code works as follows. First, the user is asked to choose the degree of the polynomial “m”. Upon choosing m, the code will choose the irreducible polynomial m(x) for this m from a predefined list that is written in the backend code.

The user will be asked to enter two polynomials, of degree m-1: Polynomial 1 and Polynomial 2. After clicking on the collect values button, the code saves the input of the user into either Binary, Hexadecimal, Decimal and Polynomial Array format, and converts in the backend each format into an array to make the code more malleable and easier to operate.

Afterwards, the user can select an operation to perform from the following: modulo reduction, finding the inverse, addition, subtraction, multiplication, and division. All the buttons are displayed below the collect values button and can be pressed in any order.

The software displays to the user the result in decimal, hexadecimal, binary, and polynomial forms in a popup window and allows the user to copy and paste any value he wants. The window does not allow any editing so the solutions cannot be altered by the user by mistake.

All of this is implemented in a graphical user interface that we will discuss later in more detail.

# Theory Summary

## Modular Polynomial Arithmetic

Modular Polynomial Arithmetic extends the principles of regular polynomial arithmetic into the realm of modular arithmetic. In modular arithmetic, calculations are performed within a fixed modulus, meaning that the results are considered only within the range of 0 to 1 (modulo 2).

By polynomial, we mean a mathematical expression of the form *anxn + an-1xn-1 + ... + a0*. The highest exponent of *x* is the degree of the polynomial, and *an, an-1, ... , a0* are called coefficients.

We assume the reader is already familiar with usual polynomial arithmetic (same as high school math), however, the user might not be familiar with the modular feature of Galois Fields that restricts the coefficient to certain values in our case zeros and ones.

One important property of modular polynomial arithmetic is its ability to ensure that results remain within a defined range, which can be crucial for computational efficiency and data integrity.

We limit our study to Galois Field 2m which is fundamental in various applications, including error-correcting codes and cryptographic algorithms as seen in the course later.

In GF(2m) , the coefficients of the polynomials are elements of the binary field, meaning they can only take values of 0 or 1. Moreover, arithmetic operations are performed modulo a specific irreducible polynomial of degree m. This irreducible polynomial is chosen carefully to ensure that the resulting field is finite and exhibits specific mathematical properties. There are several complicated algorithms that compute these irreducible polynomials that are outside the scope of our project however we will assume that these polynomials are given. Some examples of these algorithms are: Berlekamp Algorithm, Von

zur Gathen and Kaltofen, etc… These algorithms are quite recent. As we have very little knowledge about them and the math is beyond our capabilities, we decided not to move forward with the implementation of these algorithms. Additionally, these algorithms are not deterministic, in other words, the time complexity is not clearly defined and surely not polynomial. We will assume that they work for small degree polynomials. The citation for this is written below.

Author(s): R. L. Graham, D. E. Knuth, O. Patashnik - Title: Concrete Mathematics: A Foundation for Computer Science -Journal: Mathematics of Computation - Volume: 54 - Issue: 189 - Year: 1990 - Pages: 369-382 - DOI: 10.1090/S0025-5718-1990-0993933-0

## Representation of a Polynomial

A polynomial is represented as a binary vector of degree m-1. Each coefficient is either 0 or 1. For example, a polynomial in GF(2^3) is represented the binary vector [1, 0, 0] in our code, which translates to 1\* x^2 + 0\*x^1 + 0\* x^0. This is the representation used throughout the code. In other words, the code can be analyzed by considering the degree to be in decreasing order following the index of the array:  
for i=0,1,2.. 🡪 Coefficient m-1,m-2,m-3.. onwards.

It is also conventional to represent such a polynomial as the binary number 100, thus can be seen as 4 in hexadecimal or 4 in decimal. In this example 100 refers to x^2+x^1+1, the same logic applies to all binary numbers. The hexadecimal and decimal representation are conversions from binary to hex or dec.

## Addition and Subtraction

First, addition of such numbers is equivalent to subtraction, both are a XOR. this is due to simple observation and can be easily proved by considering all possible cases for two bits x1 and x2, and seeing that x1-x2= x1+ x2 = x1 XOR x2 in this field. If the polynomials are of different length, we XOR The extra bits of the longest with zero. This is called calculation considerations since it simplifies the calculation logic from applying modulo to just XORing.

Thus, Addition of polynomials in GF(2) is performed by XORing corresponding bits. This is because XOR is equivalent to addition in GF(2). For example:

[1,0,1,1]⊕[1,1,0,1]=[0,1,1,0]

In GF(2m), subtraction is the same as addition, because we have 3 cases, either the result of subtraction is ai – b­i. If ai – b­i = 0, then ai + b­­*­*i = 2%2 = 0, if ai – b­i = -1%2 = 1, then ai + b­i = 1%2 = 1 and if ai – b­i = 1%2=1, then ai + b­i = 1%2, which leads us to say that in GF(2m), addition and subtraction are equivalent.

## Multiplication

Multiplication in GF(2^m) is performed modulo an irreducible polynomial of degree m. 𝑚(𝑥) has no divisors other than itself & 1 say it is irreducible (or prime) polynomial. This polynomial is chosen such that it cannot be factored into lower-degree polynomials. The choice of this irreducible polynomial is crucial in defining the field. We choose these polynomials based on a research paper.

This result is obtained by multiplying each term of the first polynomial by each term of the second polynomial and then combining like terms using XOR. During the multiplication process, each term in the product is computed modulo 2, and then the result is reduced modulo the chosen irreducible polynomial. This reduction is essential to keep the degree of the result less than m.

If we implement a standard polynomial multiplication function, the time complexity of this function is O(n^2), where n is the degree of the polynomials being multiplied. Surely there are more efficient algorithms for polynomial multiplication using divide and conquer and fast Fourier transforms, but these are beyond the scope of this project and only offer substantial benefit for polynomials of extremely large degree.

So, we will multiply using the regular way, with a slight modification: we will do modulo 2 (actually, we do this by doing multiplication directly on bitwise operations, namely, Multiplication of polynomials in GF(2) is achieved through shift and XOR operations) at every step to directly incorporate modular reduction during the polynomial multiplication loop. This will save memory and potentially improve performance. At the end, we will do modulo the irreducible polynomial, which brings the polynomial back withing the allowed degree, or into the GF we are working in. This is necessary since higher degrees are not in the field.

## Modulo Function

To do Modulo the irreducible polynomial, we will actually build a more generalized function: the function Mod that takes as input two polynomials X and Y and returns Z Where Z= X mod Y.

We immediately see that if Y = M where M is the irreducible polynomial, we can use the above function directly to complete our polynomial multiplication.

Mathematically, if *L* is the polynomial being reduced and *M* is the irreducible polynomial, the reduction can be expressed as:

*L*(*x*)=*Q*(*x*)⋅*M*(*x*)+*R*(*x*)

where *Q*(*x*) is the quotient, *M*(*x*) is the irreducible polynomial, and)*R*(*x*) is the remainder.

In the context of polynomial arithmetic, modulo reduction involves subtracting a multiple of an irreducible polynomial from the polynomial being reduced. The irreducible polynomial is chosen such that its degree is higher than the degrees of the polynomials involved.

The algorithm for modulo reduction is akin to the manual long division process. You align the highest degree terms of both polynomials, subtract, and then bring down the next term until the degree of the polynomial being reduced is less than the degree of the irreducible polynomial. The Mod function repeatedly performs this process until the degree of the polynomial being reduced is less than the degree of the irreducible polynomial.

We make this function serve another purpose also, if we give it as input a polynomial of degree already less than that of M, then it appends zeros to the left so that the length of the array becomes M-1.

Modulo ensure that the desired max degree of the polynomial is preserved and that we are still in the allowed GF. This function is mostly used in multiplication since the degree can exceed the maximum degree allowed, thus modulo function brings it back to the allowed max degree of the resulting polynomial.

## Quotient Function

Inspired by this function, we develop a ‘quot’ function which returns the Q(x) quotient polynomial. Note that this function performs X//Y, where neither need be an irreducible polynomial.

The way quot works is elegant. We start with temp as a slice of the dividend, of equal length as divisor.

At each iteration of the while loop, we pop zeros at the beginning of the slice. If length of slice after that is equal to length of divisor, then we append 1 to the quotient, and we perform Divisor XOR temp , we store that in temp, extending it with 1 new bit from dividend. If the length of slice is less than that of divisor, then we append 0 to quotient, and we “bring down” a digit from dividend.

The subtraction is a XOR remember? We are lucky to be in GF(2), we are only working in binary numbers and subtraction is the same as addition in our case.

## Extended Euclidean Algorithm, Inverse, and Division

The above functions so far are efficient as they operate directly with binary numbers, exploiting the fact that we are in GF(2). Had we not been, we would need to do more complicated operation, like finding the inverse of the coefficients of M in the field, so we can divide the leading coefficients. But here, its easier since the inverse of 1 is 1, and we design the function in such a way so it never would encounter a case where it needs to invert 0, as 0 has no inverse.

We just discovered now why its important for p=2 to be prime in general, so we can find the multiplicative inverse.

In addition, the polynomial M should be prime/ irreducible so we can find an inverse of any polynomial X mod M. note that we can easily find the additive inverse of X in GF(2^m) as it will also be X. we are thus interested with the multiplicative inverse.

Here, if we use the Euclidean algorithm , it would tell us whether gcd(X,M) and thus if it is 1, then X and M are relatively prime. Thus, we can find X-1 mod (M). but as we choose M to be prime, we don’t need to check for this, instead we must find a and b such that aX+ bY=1.

But, since we can , we also write the code for the Euclidean algorithm.

Now, we create a code for the EUA. We mimic the traditional Extended Euclidean Algorithm, making use of all our already created functions. We use it to build the inverse function.

We are asked to also provide a function to perform division in this field, so we understand a/b as a\* b inverse . Thus, we now use the multiplication and inverse functions to build a division function.

Lastly, we make use of simple auxiliary functions to allow us to freely go back and forth between our polynomial vector representation, a printed polynomial representation, binary strings, hexadecimal strings, and decimal numbers.

# Graphical User Interface

The entire backend code serves as the backbone of the GUI. The GUI was developed using Tkinter library in python. This library allows the building of windows, and is based on a grid system. The grid is a table with columns and rows. We used this grid system to allow the user to first select the degree of the GF(2m). The selection is saved as it serves to the implementation of all backend functions. After selecting ‘m’, the user has the option to chose his preferred format of input:  
Binary, Hexadecimal, Decimal and Array Polynomial format. Binary, Hexadecimal and Decimal allow the user to input a string of length m. The polynomial array option is a more user friendly way to input the coefficients of the polynomial as it displays m boxes where each box is a coefficient for an term x^i. The code allow for the user to input 2 polynomials as most operations work with two polynomials such as addition, subtraction, multiplication and division. While inverse works on only one polynomial.   
After inputting the polynomials and picking which operation to perform, the code executes the functions calling the backend code functions. In order for that to work, we imported the file into the GUI file which allows the use of the functions in the backend code.

Additionally, the operations are displayed in a popup window that uses all four types of formats. The print and inverse buttons will display both polynomial respectively, and the other operations will display the result in all four formats as well.

There advantage of this type of GUI is that it can be developed into becoming an application since it is code based and not website based. It becomes a tool on our computers. The Code can also be adapted to work for as many degrees as we want. The display is visually pleasing and is easy to follow. The advantage in this GUI is that the operations are easy to test and to make sure of the results since all buttons display almost everything.

# Prospects

As a future project, we can implement our code on hardware, for example on assembly. There are many advantages in doing so. Assembly is hardware specific, and allows the use of XOR, AND, OR etc… operations very efficiently since it communicates directly with the hardware via an assembler. This project will be quite challenging since each machine has its own hardware thus each machine has its own optimized way to perform these operations. This is great when we are talking about huge degrees, since Euclidean algorithm can be lengthy and if performed with solid and optimized hardware can become very efficient is done via binary operations. The same can be done with addition, subtraction, multiplication etc… since its all about XORs and shifts.

In addition, we can improve our GUI by allowing the session to be reused. Our current version does not allow several sessions at once, it has a bug which does not take in the values and saves them when changing from BIN to Array or vice versa. Additionally, we can make it web based, or application based as well as we can improve the visuals with better visibility in colors or font.

Also, we can make the code work for more values of m, and more than one irreducible polynomial for the same m. This is quite simple since the core of the code exists all we have to do is extend the arrays and add irreducible polynomials to the pregiven set of polynomial that are embedded in our code.

We can also add a function to calculate them but is challenging in our current level. This would require us to fully understand the studies done on these algorithms and study how they work. These algorithms work on specific hardware and require specific programming languages to code which we lack the knowledge and skill to do in our current state.