AMERICAN UNIVERSITY OF BEIRUT
MAROUN SEMAAN FACULTY OF ENGINEERING & ARCHITECTURE

American University of Beirut
School of Engineering and Architecture
Department of Electrical and Computer Engineering

A full-stack design for the Cybersecurity Website

By
Hashem Khodor
(hmk57@mail.aub.edu)
Dana Kossaybati
(dak39@mail.aub.edu)
Mohammad Khaled Charaf
(mmc51@mail.aub.edu)
Mohammad Ayman Charaf
(mmc50@mail.aub.edu)

A REPORT
submitted to Dr. Ali L Hussein in partial fulfillment of the requirements of the cryptography project for the course EECE455 – Cryptography

November 2023

Table of Contents

Table of Figures

1- Introduction

In the realm of cybersecurity, precision and reliability are paramount. This report provides an in-depth look into the foundation of our encryption project, highlighting the robust algorithms developed. Our platform focuses on providing users with a seamless and efficient encryption experience, eliminating the need for manual customization.

Beginning with an examination of classical ciphers—Hill, Playfair, Monoalphabetic, and Affine—we demonstrate the intuitive nature of our algorithms, designed to operate seamlessly without user intervention. As we traverse into modern encryption techniques, including AES and DES, the report emphasizes the automated sophistication embedded within our cryptographic framework.

The incorporation of polynomial arithmetic within the Galois field GF(2^m) is explored, showcasing the mathematical precision that underpins our encryption process. It's important to note that while the algorithms are mathematically rigorous, the end-user experience is characterized by simplicity and reliability, eliminating the necessity for user-based customization.

This report serves as a testament to the intentional design choices made at our Cybersecurity website, where our focus is on delivering a secure and hassle-free encryption experience for all users. Join us as we unravel the intricacies of our algorithms, demonstrating how our platform seamlessly integrates cutting-edge security with user-friendly functionality.

2-Tools & Languages Used:

Frontend:

HTML

CSS

JavaScript

Backend:

Python

Framework:

Flask

3-Literature Review:

The field of cybersecurity and cryptography has witnessed significant advancements in recent years, driven by the increasing need for secure communication and data protection. The project focuses on the development of a comprehensive website, encompassing a range of cryptographic operations such as AES, DES, classical ciphers, and polynomial arithmetic in Galois fields. This literature review explores the existing knowledge base surrounding these cryptographic techniques, emphasizing their theoretical foundations, practical applications, and the broader context of cybersecurity.

3.1. Advanced Encryption Standard (AES): The Advanced Encryption Standard, established by the National Institute of Standards and Technology (NIST), serves as a cornerstone in modern cryptographic practices[1]. The literature reflects a wealth of research on the design principles, security analyses, and implementation strategies of AES[2]. Notable contributions include the original Rijndael algorithm proposal by Daemen and Rijmen, various optimization techniques for efficient software and hardware implementations[3], and ongoing discussions on potential vulnerabilities and countermeasures.

3.2. Data Encryption Standard (DES): As a precursor to AES, the Data Encryption Standard has been extensively studied in the literature[4]. Early research focused on the development of DES and its subsequent adoption as a federal standard[5]. More recent work explores the vulnerabilities of DES to brute-force attacks and the necessity for stronger encryption algorithms[6]. The evolution of DES into Triple DES (3DES) and its role in legacy systems are also areas of interest in the literature[7].

3.3. Classical Ciphers: Classical ciphers, rooted in historical encryption practices, continue to be relevant in educational and recreational contexts[8]. The literature encompasses studies on the historical significance of classical ciphers, their mathematical foundations, and the educational value of incorporating them into modern cryptographic curricula[9]. The project's inclusion of classical ciphers aligns with this broader interest in preserving and understanding historical cryptographic techniques.

3.4. Polynomial Arithmetic in Galois Fields: The incorporation of polynomial arithmetic in Galois fields is a distinctive feature of the project, aligning with the mathematical underpinnings of modern cryptographic protocols[10]. Literature in this domain spans discussions on finite fields, polynomial arithmetic, and their applications in error-correcting codes and cryptographic algorithms[11]. The project's implementation of polynomial arithmetic in a web-based application contributes to the practical exploration of these theoretical concepts.

In conclusion, the literature surrounding the key components of the project—AES, DES, classical ciphers, and polynomial arithmetic in Galois fields—provides a rich foundation for understanding the theoretical aspects, historical context, and practical considerations associated with these cryptographic techniques. By integrating these elements into a full-stack website, the project not only contributes to the practical implementation of cryptographic algorithms, but also aligns with the broader goals of enhancing user understanding and engagement with cybersecurity principles.

4- References For Literature Review:

1. NIST. (2001). "FIPS PUB 197: Advanced Encryption Standard (AES)." https://csrc.nist.gov/publications/fips/fips197
2. Daemen, J., & Rijmen, V. (2002). "The Design of Rijndael: AES - The Advanced Encryption Standard." Springer.
3. Satoh, A., & Takano, K. (2001). "A Compact Rijndael Hardware Architecture with S-Box Optimization." In International Workshop on Cryptographic Hardware and Embedded Systems (CHES).
4. NIST. (1977). "FIPS PUB 46: Data Encryption Standard (DES)." https://csrc.nist.gov/publications/fips/fips46
5. Diffie, W., & Hellman, M. E. (1977). "Exhaustive Cryptanalysis of the NBS Data Encryption Standard." Computer, 10(6), 74-84.
6. Biham, E., & Shamir, A. (1993). "Differential Cryptanalysis of DES-like Cryptosystems." Journal of Cryptology, 4(1), 3-72.
7. Tuchman, W. (1979). "Hellman Presents No Shortcut Solutions to DES." IEEE Spectrum, 16(7), 40-41.
8. Singh, S. (1999). "The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography." Doubleday.
9. Kahn, D. (1996). "The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet." Scribner.
10. Lidl, R., & Niederreiter, H. (1997). "Introduction to Finite Fields and Their Applications." Cambridge University Press.
11. Blahut, R. E. (2003). "Algebraic Codes for Data Transmission." Cambridge University Press.

5-Challenges

One of the primary challenges faced during the implementation of our project involved the generation of irreducible polynomials in the finite field GF(2^m). This task is critical for the project's functionality, as these polynomials form the basis for polynomial arithmetic in cryptographic operations.

We explored a non-deterministic algorithm for generating irreducible polynomials, leveraging the Berlekamp algorithm for checking irreducibility.

```
GenerateIrreduciblePolynomialsinGF2m(m:int):
    while true:
        P = randomly pick a polynomial in GF(2^m)
        if P is irreducible: //using berlekamp algorithm: O(nlognlogpn)
            return P
```

The proposed algorithm involves the non-trivial task of randomly selecting polynomials and determining their irreducibility using the Berlekamp algorithm. While the algorithmic approach appears theoretically sound, its practical implementation encountered roadblocks due to its inherent complexity.

Consequently, our team adopted a proactive approach by precomputing a comprehensive set of irreducible polynomials up to m=2000 using sagemath.

Sources for berklemp algorithm:

Divasón, J., Joosten, S.J.C., Thiemann, R. et al. A Verified Implementation of the Berlekamp–Zassenhaus Factorization Algorithm. J Autom Reasoning 64, 699–735 (2020). https://doi.org/10.1007/s10817-019-09526-y

Menezes, A. J., & Blake, I. F. (1993). *Applications of finite fields*. Kluwer Academic Publishers.

6-Code Explanation

Polynomial Arithmetic:

We assessed two different ways of implementing these classes. The first one was purely based on our own code. The second one utilizes the Polynomial class in the library NumPy.

Approach 1: Polynomial Operations Implementation

In Approach 1, we implemented various polynomial operations using custom algorithms. The key components include:

- **Polynomial Multiplication (Two Variations):**
  - **Divide and Conquer Approach:** This method employs a divide-and-conquer strategy for polynomial multiplication, with a worst-case runtime described by [Equation].
  - **Fast Fourier Transform (FFT) Polynomial Multiplication:** Utilizing the FFT technique, this approach achieves a worst-case runtime of O(nlog(n)), where n is the size of the polynomials.

- **Code:**

```python
import numpy as np


def fft_polynomial_multiply(polynomial1, polynomial2):
    # Get the lengths of the input polynomials
    n = len(polynomial1)
    m = len(polynomial2)
    print(n, m)

    # Find the next power of 2 greater than or equal to n+m
    next_pow2 = 1
    while next_pow2 < n + m:
        next_pow2 *= 2

    # Perform FFT on both polynomials
    fft1 = np.fft.fft(polynomial1, next_pow2)
    fft2 = np.fft.fft(polynomial2, next_pow2)

    # Element-wise multiplication in the frequency domain
    result_fft = fft1 * fft2

    # Inverse FFT to get the result in the time domain
    result = np.fft.ifft(result_fft)

    # Round the real part of the result to eliminate small numerical errors
    result = np.round(result.real)

    return list(map(int, result))
```

- **Long Division:**
  - The implementation includes a long division algorithm for polynomials, which is crucial for division and modulus operations.

- **Code:**

```python
def longdivision(first, second):
    # calculates first/second
    xor = lambda x, y: x ^ y
    q = [0 for i in range(len(first) - len(second) + 1)]
    if len(first) == 0 or len(second) == 0:
        return [], []
    first.reverse()
    second.reverse()
    while len(first) >= len(second):
        print(len(first), len(second))
        # print(first, second)
        # print(first, second)

        while len(first) > 0 and first[0] == 0:
            first.pop(0)
        while len(second) > 0 and second[0] == 0:
            second.pop(0)
        if len(first) < len(second):
            break

        index = len(q) - (len(first) - len(second)) - 1
        print(index)
        q[index] = 1
        first = list(
            map(xor, first, second + [0 for i in range(len(first) - len(second))])
        )
        # print(first)

    return q, first
```

- **Extended GCD Algorithm:**
    - The extended greatest common divisor (GCD) algorithm is implemented specifically for polynomials. This algorithm uses a modified version of the extended Euclidean algorithm.

Example Usage:

- The provided script demonstrates the use of these implementations by applying them to two polynomials (**A** and **B**) and computing their extended GCD.

```
def extendedgcdPoly(a, b, MOD):
    import numpy as np
    import copy

    # if b > a:
    # a, b = b, a  # swap them
    r11, r12 = polynomial(0x0), polynomial(0x0)
    r21, r22 = polynomial(0x1), polynomial(0x0)
    r31, r32 = polynomial(0x0), polynomial(0x1)
    Matrix = [["factor", "tnumber", "tcoeff1", "tcoeff2"]]
    # print(f"factor\tnumber\tcoeff1\tcoeff2")
    # print(f"\t{a}\t{r21}\t{r22}")
    # print(f"\t{b}\t{r31}\t{r32}")
    while sum(b.__coeff__) != 0:
        # print(r31,r32)
        print("OKAy")
        print(a, b)
        r11, r12 = r21, r22
        r21, r22 = r31, r32
        # print(a,b)
        factor = a / b
        print("END1")
        r31 = (r11 - factor * r21) % MOD
        print("END2")
        r32 = (r12 - factor * r22) % MOD

        print("END3")
        a, b = b, a % b

        print("END4")
        # print(f"{factor}\t{b}\t{r31}\t{r32}")
        Matrix.append([str(factor), str(b), str(r31), str(r32)])
        print("END5")
```

```python
        import numpy as np
        import pandas

        # print("\n".join(["\t".join([str(cell) for cell in row]) for row in Matrix]))
        # print(pandas.DataFrame(Matrix))
        return a, r21, r22
# A = polynomial([])
# print(A * 23232)
if __name__ == "__main__":
    mod = polynomial([1, 0, 0, 0, 1, 1, 0, 1, 1])
    A = polynomial(0x13A11DBDFD506AAAC623418C4BF534D25B97AE)
    B = polynomial(0x5923442312232444123123124213123124 21)
    GCD, inv1, inv2 = extendedgcdPoly(
        A, B, polynomial(0x13A11DBDFD506AAAC623418C4BF534D25B97A3E)
    )
    print(GCD)
    print(A)
    print((B * inv2) % A)
```

Approach 2: Numpy Polynomial Wrapper Class

In Approach 2, we adopted a different strategy by using the numpy.polynomial library and encapsulating it within a wrapper class. The main features of this approach include:

- **Numpy.Polynomial Wrapper Class:**
  - We encapsulated the functionality of the library numpy.polynomial within a custom wrapper class. This class serves as an interface to perform operations on polynomials.
- **Binary Operation Overloading:**
  - To support operations in the Galois Field GF(2^m), we overloaded the binary operators (**+**, **-**, *****, /, **%**) in the wrapper class. This ensures that polynomial operations adhere to the specific properties of the Galois Field.

- **Code:**

```python
class polynomial:
    def __init__(self, coeff, mod=2):
        assert type(coeff) in (list, int, numpy.ndarray), "Invalid Type"
        if type(coeff) == int:
            # In this case, the user inputed the coeff in hexadecimal format
            # 0x23542ac -> 00100011....
            __coeff__ = list(map(int, list(bin(coeff))[2:]))
            __coeff__.reverse()
        else:
            __coeff__ = coeff

        self.polynomial = P.Polynomial(P.polytrim(__coeff__))
        self.polynomial.coef %= 2

    def __add__(self, other):
        """ """
        assert type(other) == polynomial, "Invalid Type"
        res = self.polynomial + other.polynomial
        res = P.polytrim(res.coef % 2)
        return polynomial(res, 2)

    def __sub__(self, other):
        """ """
        assert type(other) == polynomial, "Invalid Type"
        return self.__add__(other)

    def __mul__(self, other):
        """"""
        assert type(other) == polynomial, "Invalid Type"
        res = ((self.polynomial * other.polynomial)).coef % 2
        return polynomial(P.polytrim(res), 2)

    def __truediv__(self, other):
        assert type(other) == polynomial, "Invalid Type"
        res = (self.polynomial // other.polynomial).coef % 2
        return polynomial(P.polytrim(res), 2)
        # return super().__truediv__(other)

    def __mod__(self, other):
        assert type(other) == polynomial, "Invalid Type"
        res = (self.polynomial % other.polynomial).coef % 2
        return polynomial(res, 2)

    def __str__(self):
        return self.polynomial.__str__()

    def __eq__(self, other):
        assert type(other) == polynomial, "Invalid Type"
        return self.polynomial.__eq__(other.polynomial)
```

High-Level Overview:

- Approach 2 is characterized by its simplicity and efficiency, as it leverages the optimized polynomial operations provided by the **numpy.polynomial** library. The wrapper class enhances usability and ensures compatibility with operations in GF(2^m).

Conclusion:

In summary, Approach 1, while commendably implementing detailed polynomial operations, faces significant challenges, particularly when confronted with large polynomials where the degree (m) is greater than or equal to 50. The inefficiency of Approach 1 is multifaceted:

- **Divide and Conquer Polynomial Multiplication:**
  - The divide and conquer strategy, while theoretically sound, exhibits practical inefficiencies for larger polynomials. The worst-case runtime equation becomes a limiting factor, resulting in performance degradation as the polynomial degree increases.
  - **Increased Stack Depth:**
    - The divide and conquer approach introduce a disadvantage of increased stack depth. This characteristic can contribute to additional computational overhead, adversely affecting the efficiency and performance of the algorithm, especially for large polynomials.
- **Fast Fourier Transform (FFT) Polynomial Multiplication:**
  - Despite the superior asymptotic complexity of FFT-based multiplication ($O(n \log(n))$), its practical efficiency in Approach 1 may be hindered by non-optimized implementation details and constant factors, leading to suboptimal performance for large polynomials.
- **Long Division and Extended GCD:**
  - The long division and extended GCD algorithms, while essential, may not be fully optimized for scalability when dealing with large polynomials.
- **Performance Bottlenecks:**
  - Custom implementations in Approach 1 introduce potential performance bottlenecks, particularly when compared to more specialized libraries designed for efficient polynomial operations.

Recommendation: Given the challenges observed in Approach 1, especially for large polynomials (m >= 50), and considering the disadvantage of increased stack depth in the divide and conquer strategy, an alternative approach is worth exploring. Approach 2, leveraging the optimized polynomial operations provided by the **numpy.polynomial** library, emerges as a more efficient option for handling large-scale polynomial computations. The inherent optimizations and specialized implementations in libraries like **numpy** make them well-suited for addressing computational challenges associated with higher-degree polynomials. The selection between approaches should be based on the specific requirements and constraints of the application, striking a balance between theoretical soundness and practical efficiency, especially in scenarios involving large polynomial degrees.

7- Testing

1- Home Page Hyperlinks:

Figure 1- Home Page

--> Home: keeps you on the home page
--> Classical: takes you to the classical encryption site
--> DES: takes you to the DES site.
--> Polynomial Arithmetic: takes you to the Polynomial Arithmetic site

2- Classical Encryption Site:

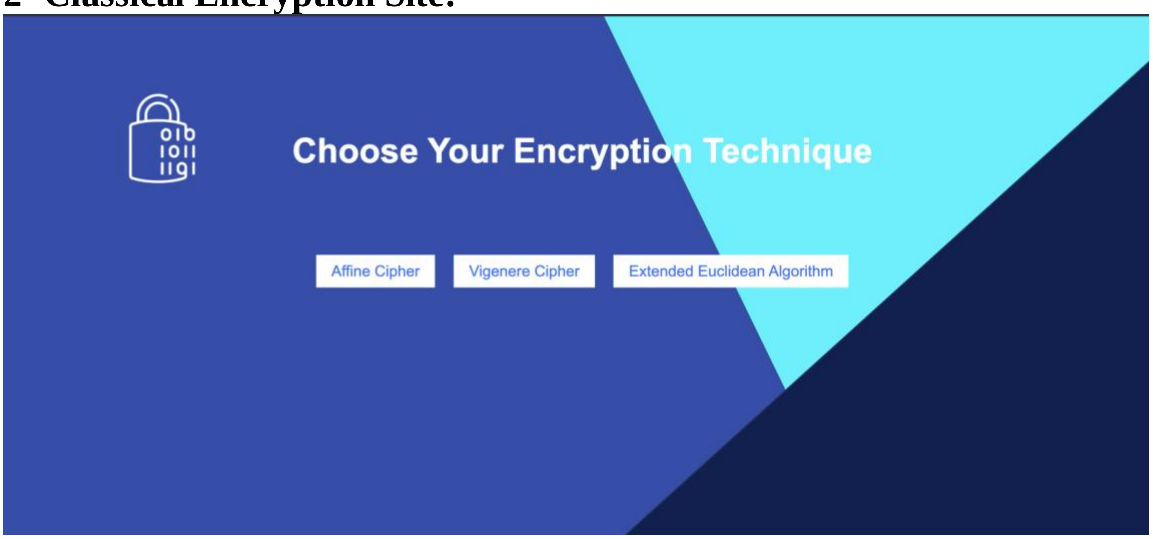


Figure 2 - Classic Encryption Page

--> Affine Cipher: Allows use to insert input for Affine Cipher encryption: <-

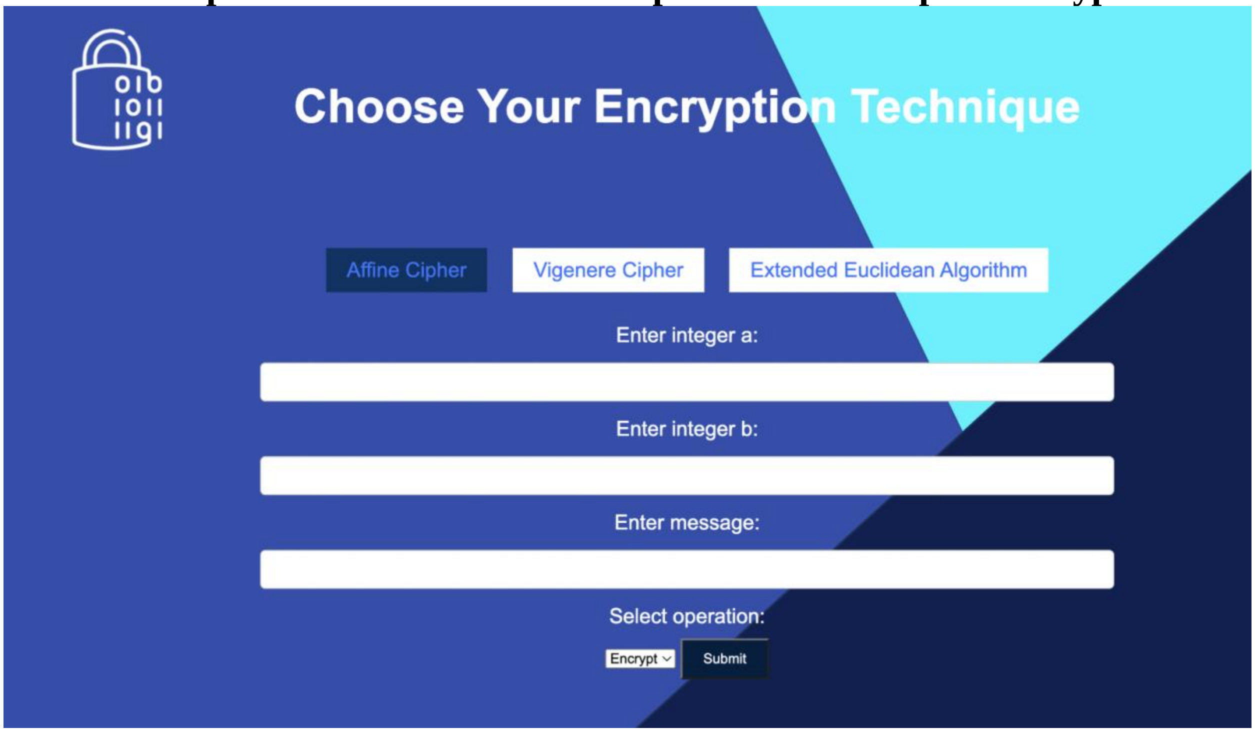


Figure 3 - Affine Cipher

--> Test Case: a = 11, b = 17, Message = Hello, Operation = Encrypt

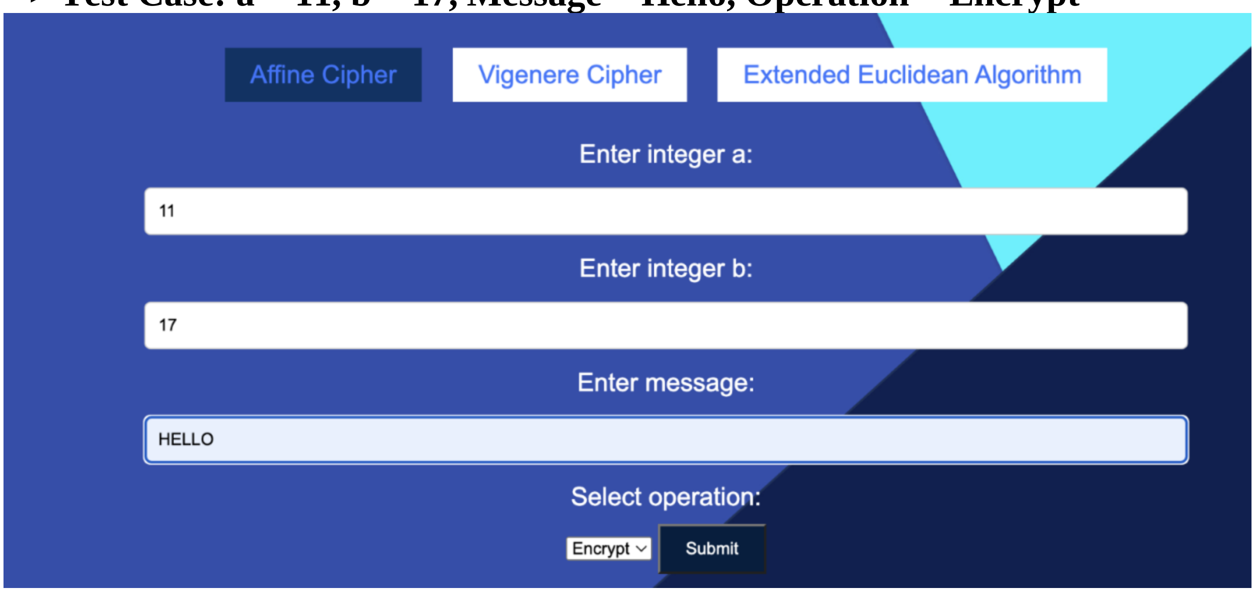


Figure 4 - Test case 1

Output:

Figure 5 - Output of Test case 1

--> Test Case: a = 11, b = 17, Message = Hello, Operation = Decrypt

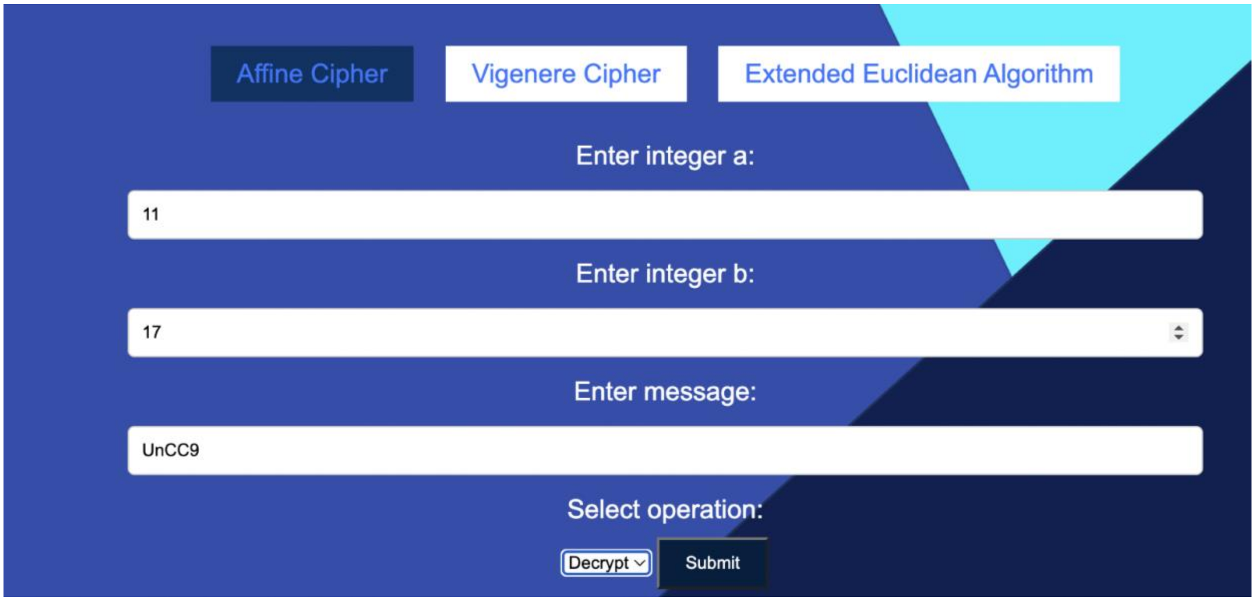


Figure 6- Test Case 2

Output:

Figure 7 - Output of Test Case 2

--> Vigenere Cipher: Allows use to insert input for Vigenere Cipher encryption: <--

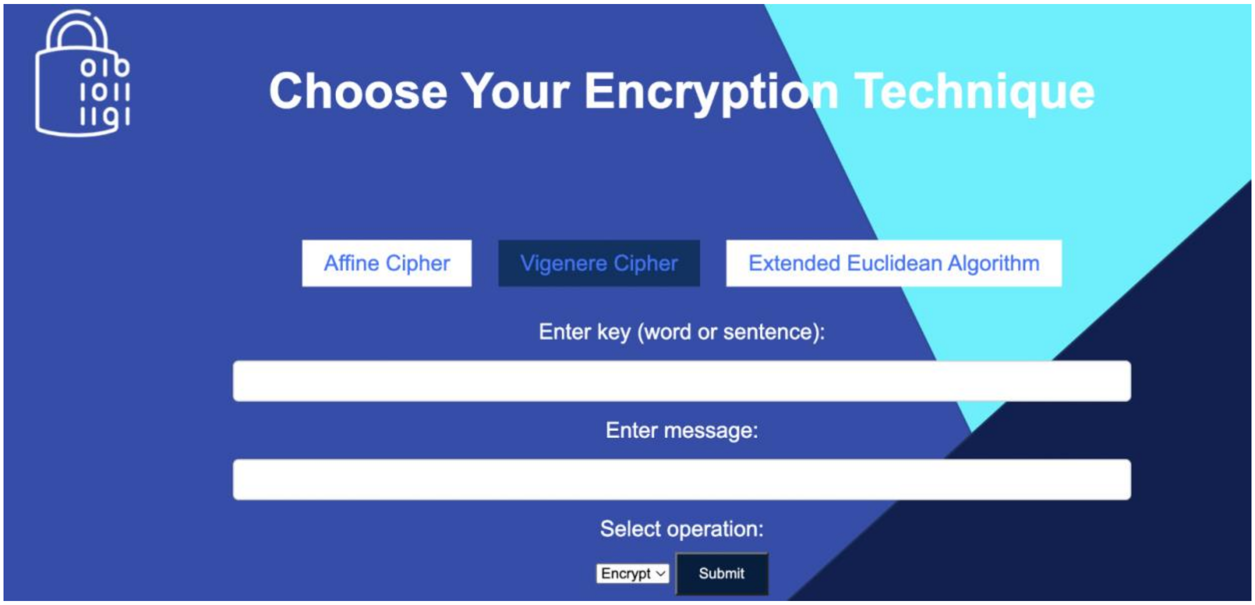


Figure 8 - Vigenere Cipher

--> Test Case: key = deceptive, message = wearediscoveredsaveyourself, Operation = Encrypt <---

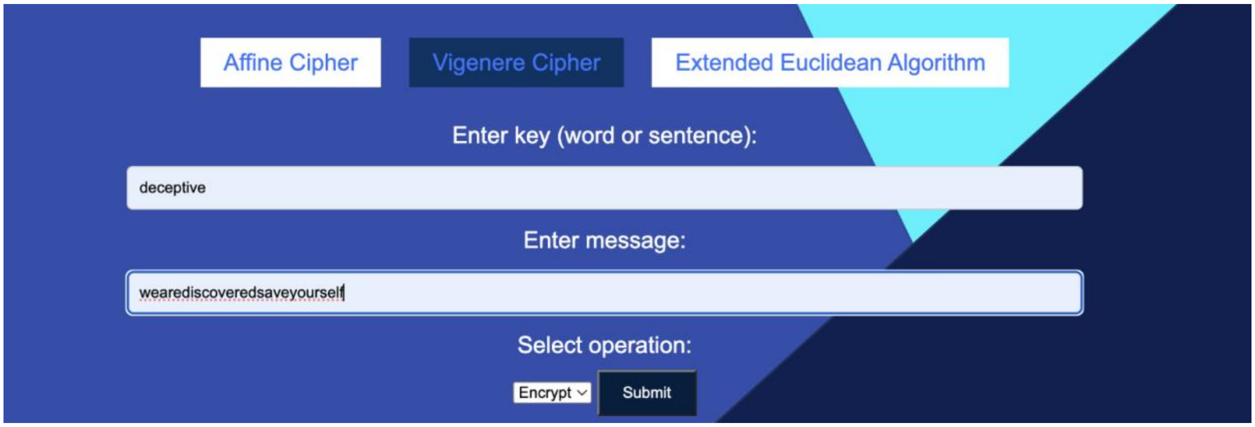


Figure 9 - Test Case 3

Output:

Figure 10 - Output of Test Case 3

--> Test Case: key = deceptive, message
=JsmFDGAXqBJqFDGKFJrMAIQVwQt
, Operation = Decrypt

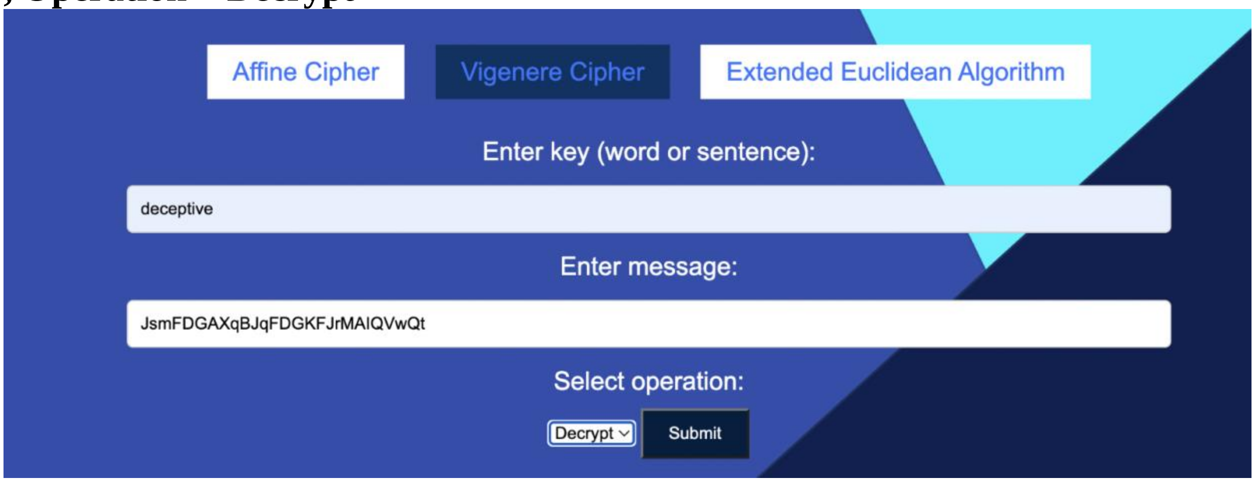


Figure 11 - Test Case 4

Output:

Figure 12 - Output of Test Case 4

--> Extended Euclidean Algorithm: Allows use to insert input for Extended Euclidean Algorithm: <--

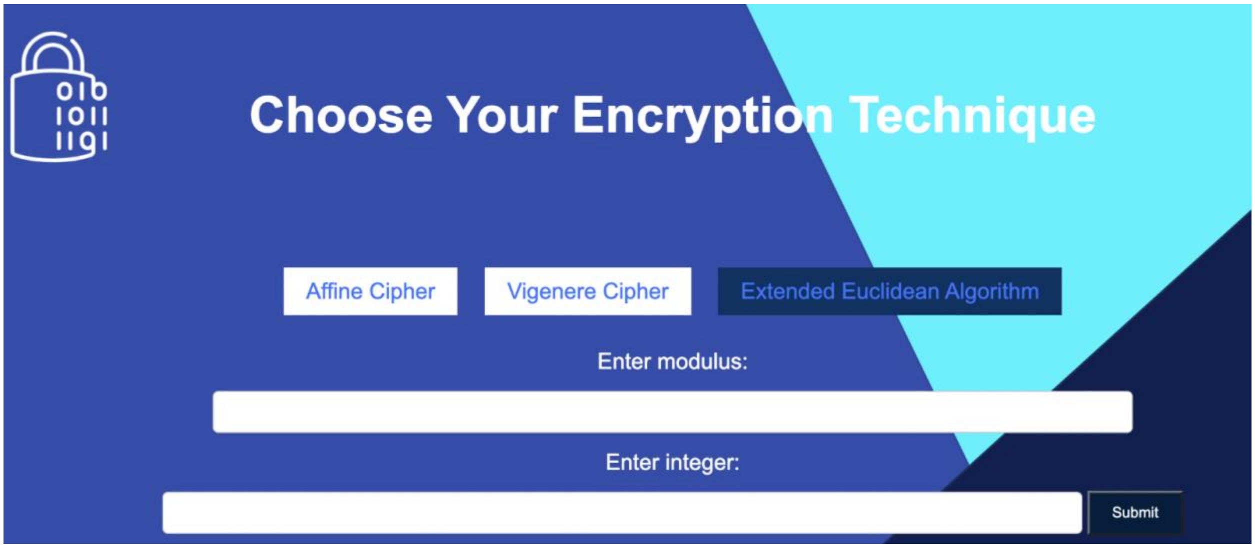


Figure 13 - Extended Euclidean Algorithm

--> Test Case: modulus = 5, integer = 2

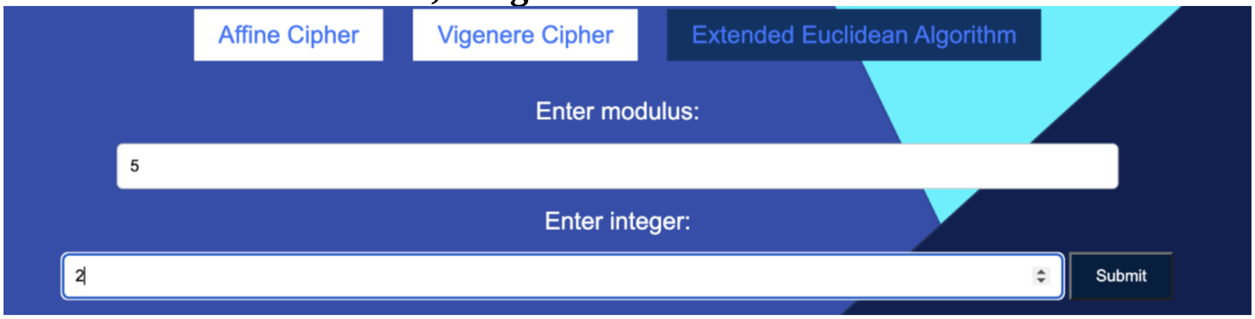


Figure 14 - Test Case 5

Output:

Figure 15 - Output of Test Case 5

3- DES Site:

Figure 16 - DES Page

Note: User can enter number or key in hexadecimal or binary by prefix his input either with 0x or 0b.
--> Test Case: Number = 0x02468aceeca86420, Key = 0x0f1571c947d9e859

Figure 17 - Test Case 6

Output: Answer with table showing rounds of DES

Round	Ki	Li	Ri
IP		0x5a005a00	0x3cf03c0f
1	0x7833c320da70	0x3cf03c0f	0xbad22845
2	0x2b1a74ca48d8	0xbad22845	0x99e9b723
3	0x8c78d881d31d	0x99e9b723	0xbae3b9e
4	0x1667789316a0	0xbae3b9e	0x42415649
5	0xce5d01d80b25	0x42415649	0x18b3fa41
6	0x4bab4d126a9c	0x18b3fa41	0x9616fe23
7	0x9f48b713191	0x9616fe23	0x67117cf2
8	0x710deaa3202b	0x67117cf2	0xc11bfc09
9	0x129ab83347c3	0xc11bfc09	0x887fbc6c
10	0x9c38661e8103	0x887fbc6c	0x600f7e8b
11	0xa26e4cc66544	0x600f7e8b	0xf596506e
12	0x48772468a3c8	0xf596506e	0x738538b8
13	0xc09d79f0d40b	0x738538b8	0xc6a62c4e
14	0xc5e2634e162a	0xc6a62c4e	0x56b0bd75
15	0xa3df829c7968	0x56b0bd75	0x75e8fd8f
16	0xa6120b4d4c25	0x75e8fd8f	0x25896490
IP-1		0xda02ce3a	0x89ecac3b

Figure 18 - Output of Test Case 6

4- AES Site:

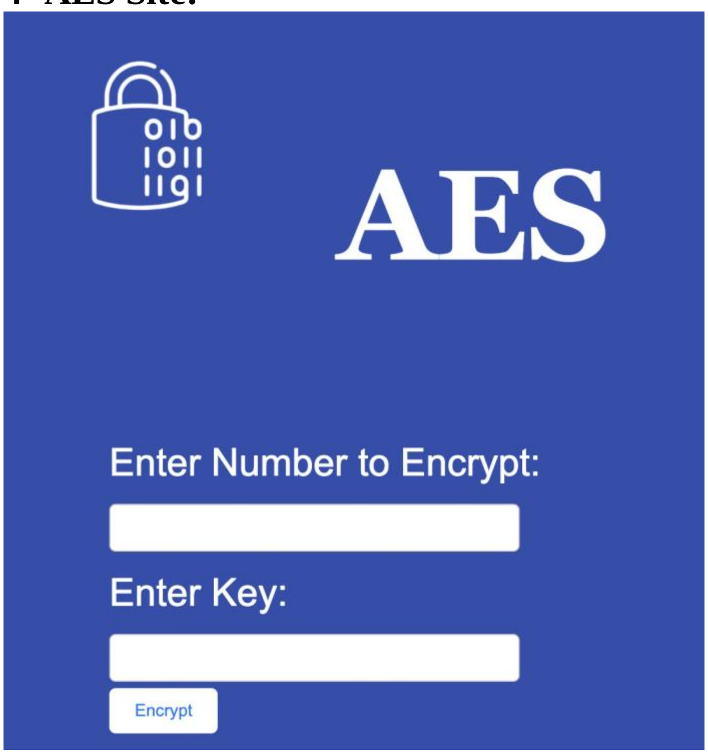


Figure 19 - AES Page

Note: User can enter number or key in hexadecimal or binary by prefix his input either with 0x or 0b.

--> Test Case: Number = 0x0123456789abcdeffedcba9876543210, Key = 0x0f1571c947d9e8590cb7add6af7f6798

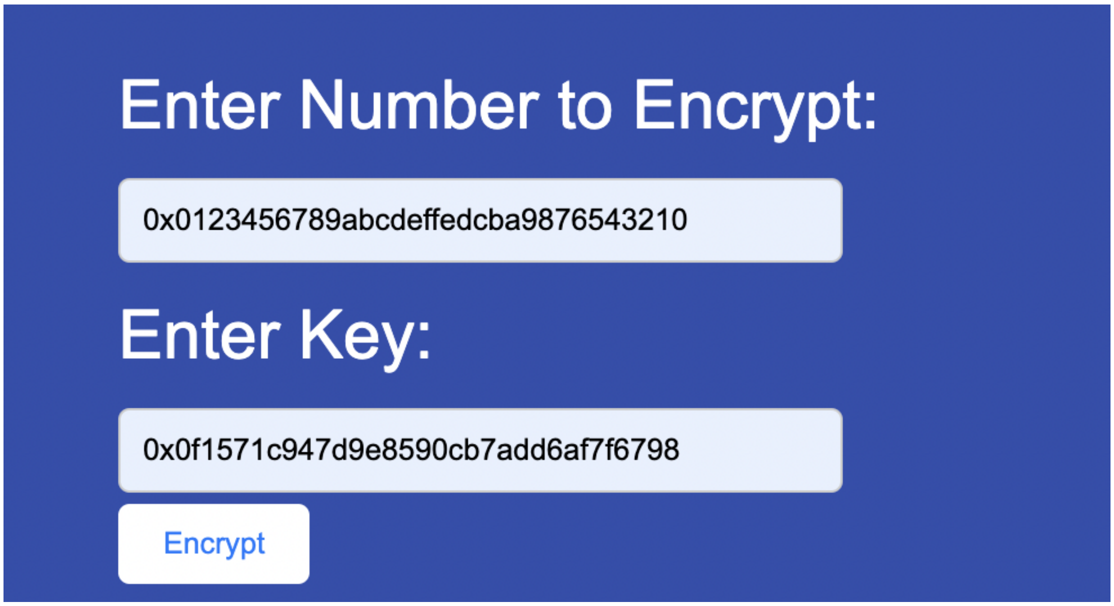


Figure 20 - Test Case 7

Output: Answer with table showing rounds of AES

Key Words	Auxillary Function
w0 = 0f1571c9 w1 = 47d9e859 w2 = 0cb7add6 w3 = af7f6798	RotWord(w3) = 7f6798af = x1 SubWord(1) = d2854679 = y1 RCON(1) = 01000000 y1 xor Rcon(1) = d3854679 = z1
w4 = w0 xor z1 = 0xdc9037b0 w5 = w4 xor w1 = 0xaf7f6798 w6 = w5 xor w2 = 0xcb7add6 w7 = w6 xor w3 = 0x47d9e859	RotWord(w7) = 8115a738 = x2 SubWord(2) = 0c595c07 = y2 RCON(2) = 02000000 y2 xor Rcon(2) = 0e595c07 = z2
w8 = w4 xor z2 = 0xd2c96bb7 w9 = w8 xor w5 = 0x388115a7 w10 = w9 xor w6 = 0x97fe723f w11 = w10 xor w7 = 0x9b49dfe9	RotWord(w11) = ffd3c6e6 = x3 SubWord(3) = 1666b48e = y3 RCON(3) = 04000000 y3 xor Rcon(3) = 1266b48e = z3
w12 = w8 xor z3 = 0xc0afdf39 w13 = w12 xor w9 = 0xe6ffd3c6 w14 = w13 xor w10 = 0xde7ec661 w15 = w14 xor w11 = 0x4980b45e	RotWord(w15) = ae7ec0b1 = x4 SubWord(4) = e4f3bac8 = y4 RCON(4) = 08000000 y4 xor Rcon(4) = ecf3bac8 = z4
w16 = w12 xor z4 = 0x2c5c65f1 w17 = w16 xor w13 = 0xb1ae7ec0 w18 = w17 xor w14 = 0x5751ad06 w19 = w18 xor w15 = 0x892f6b67	RotWord(w19) = 8cdd5043 = x5 SubWord(5) = 64c1531a = y5 RCON(5) = 10000000 y5 xor Rcon(5) = 74c1531a = z5
w20 = w16 xor z5 = 0x589d36eb w21 = w20 xor w17 = 0x438cdd50 w22 = w21 xor w18 = 0xf222a390 w23 = w22 xor w19 = 0xa5730e96	RotWord(w23) = 4046bd4c = x6 SubWord(6) = 095a7a29 = y6 RCON(6) = 20000000 y6 xor Rcon(6) = 295a7a29 = z6
w24 = w20 xor z6 = 0x71c74cc2 w25 = w24 xor w21 = 0x4c4046bd w26 = w25 xor w22 = 0xfcc9bed w27 = w26 xor w23 = 0xfdee387d	RotWord(w27) = a5a9efcf = x7 SubWord(7) = 06d3df8a = y7 RCON(7) = 40000000 y7 xor Rcon(7) = 46d3df8a = z7
w28 = w24 xor z7 = 0x37149348 w29 = w28 xor w25 = 0xcfa5a9ef w30 = w29 xor w26 = 0x83e5ef52 w31 = w30 xor w27 = 0x8c2974bf	RotWord(w31) = 7da14af7 = x8 SubWord(8) = ff32d668 = y8 RCON(8) = 80000000 y8 xor Rcon(8) = 7f32d668 = z8
w32 = w28 xor z8 = 0x48264520 w33 = w32 xor w29 = 0xf77da14a w34 = w33 xor w30 = 0x38d808a5 w35 = w34 xor w31 = 0xbb3de7f7	RotWord(w35) = be0b383c = x9 SubWord(9) = ae2b07eb = y9 RCON(9) = 1b000000 y9 xor Rcon(9) = b52b07eb = z9
w36 = w32 xor z9 = 0xfd0d42cb w37 = w36 xor w33 = 0x3cbe0b38 w38 = w37 xor w34 = 0xcbc3aa72 w39 = w38 xor w35 = 0xf31ba2d7	RotWord(w39) = 6b4156f9 = x10 SubWord(10) = 7f83b199 = y10 RCON(10) = 36000000 y10 xor Rcon(10) = 4983b199 = z10
w40 = w36 xor z10 = 0xb48ef352 w41 = w40 xor w37 = 0xf96b4156 w42 = w41 xor w38 = 0xc5d54a6e w43 = w42 xor w39 = 0xe16e01c	

Start of Round	After SubBytes	After Shift Rows	After Mix Columns	Round Key
01 89 fe 76 23 ab dc 54 45 cd ba 32 67 ef 98 10				0f 47 0c af 15 d9 b7 7f 71 e8 ad 67 c9 59 d6 98
0e ce f2 d9 36 72 6b 2b 34 25 17 55 ae b6 4e 88	ab 8b 89 35 05 40 7f f1 18 3f f0 fc e4 4e 2f c4	ab 8b 89 35 40 7f f1 05 f0 fc 18 3f c4 e4 4e 2f	b9 94 57 75 e4 8e 16 51 47 20 9a 3f c5 d6 f5 3b	dc 9b 97 38 90 49 fe 81 37 df 72 15 b0 e9 3f a7
65 0f c0 4d 74 c7 e8 d0 70 ff e8 2a 75 3f ca 9c	4d 76 ba e3 92 c6 9b 70 51 16 9b e5 9d 75 74 de	4d 76 ba e3 c6 9b 70 92 9b e5 51 16 de 9d 75 74	8e 22 db 12 b2 f2 dc 92 df 80 f7 c1 2d c5 1e 52	d2 49 de e6 c9 80 7e ff 6b b4 c6 d3 b7 5e 61 c6
5c 6b 05 f4 7b 72 a2 6d b4 34 31 12 9a 9b 7f 94	4a 7f 6b bf 21 40 3a 3c 8d 18 c7 c9 b8 14 d2 22	4a 7f 6b bf 40 3a 3c 21 c7 c9 8d 18 22 b8 14 d2	b1 c1 0b cc ba f3 8b 07 f9 1f 6a c3 1d 19 24 5c	c0 89 57 b1 af 2f 51 ae df 6b ad 7e 39 67 06 c0
71 48 5c 7d 15 dc da a9 26 74 c7 bd 24 7e 22 9c	a3 52 4a ff 59 86 57 d3 f7 92 c6 7a 36 f3 93 de	a3 52 4a ff 86 57 d3 59 c6 7a f7 92 de 36 f3 93	d4 11 fe 0f 3b 44 06 73 cb ab 62 37 19 b7 07 ec	2c a5 f2 43 5c 73 22 8c 65 0e a3 dd f1 96 90 50
f8 b4 0c 4c 67 37 24 ff ae a5 c1 ea e8 21 97 bc	41 8d fe 29 85 9a 36 16 e4 06 78 87 9b fd 88 65	41 8d fe 29 9a 36 16 85 78 87 e4 06 65 9b fd 88	2a 47 c4 48 83 e8 18 ba 84 18 27 23 eb 10 0a f3	58 fd 0f 4c 9d ee cc 40 36 38 9b 46 eb 7d ed bd
72 ba cb 04 1e 06 d4 fa b2 20 bc 65 00 6d e7 4e	40 f4 1f f2 72 6f 48 2d 37 b7 65 4d 63 3c 94 2f	40 f4 1f f2 6f 48 2d 72 65 4d 37 b7 2f 63 3c 94	7b 05 42 4a 1e d0 20 40 94 83 18 52 94 c4 43 fb	71 8c 83 cf c7 29 e5 a5 4c 74 ef a9 c2 bf 52 ef
0a 89 c1 85 d9 f9 c5 e5 d8 f7 f7 fb 56 7b 11 14	67 a7 78 97 35 99 a6 d9 61 68 68 0f b1 21 82 fa	67 a7 78 97 99 a6 d9 35 68 0f 61 68 fa b1 21 82	ec 1a c0 80 0c 50 53 c7 3b d7 00 ef b7 22 72 e0	37 bb 38 f7 14 3d d8 7d 93 e7 08 a1 48 f7 a5 4a
db a1 f8 77 18 6d 8b ba a8 30 08 4e ff d5 d7 aa	b9 32 41 f5 ad 3c 3d f4 c2 04 30 2f 16 03 0e ac	b9 32 41 f5 3c 3d f4 ad 30 2f c2 04 ac 16 03 0e	b1 1a 44 17 3d 2f ec b6 0a 6b 2f 42 9f 68 f3 b1	48 f3 cb 3c 26 1b c3 be 45 a2 aa 0b 20 d7 72 38
f9 e9 8f 2b 1b 34 2f 08 4f c9 85 49 bf bf 81 89	99 1e 73 f1 af 18 15 30 84 dd 97 3b 08 08 0c a7	99 1e 73 f1 18 15 30 af 97 3b 84 dd a7 08 08 0c	31 30 3a c2 ac 71 8c c4 46 65 48 eb 6a 1c 31 62	fd 0e c5 f9 0d 16 d5 6b 42 e0 4a 41 cb 1c 6e 56
cc 3e ff 3b a1 67 59 af 04 85 02 aa a1 00 5f 34	4b b2 16 e2 32 85 cb 79 f2 97 77 ac 32 63 cf 18	4b b2 16 e2 85 cb 79 32 77 ac f2 97 18 32 63 cf	4b b2 16 e2 85 cb 79 32 77 ac f2 97 18 32 63 cf	b4 ba 7f 86 8e 98 4d 26 f3 13 59 18 52 4e 20 76
ff 08 69 64 0b 53 34 14 84 bf ab 8f 4a 7c 43 b9				

Figure 21- Output of Test Case 7

5- Polynomial Arithmetic Site:

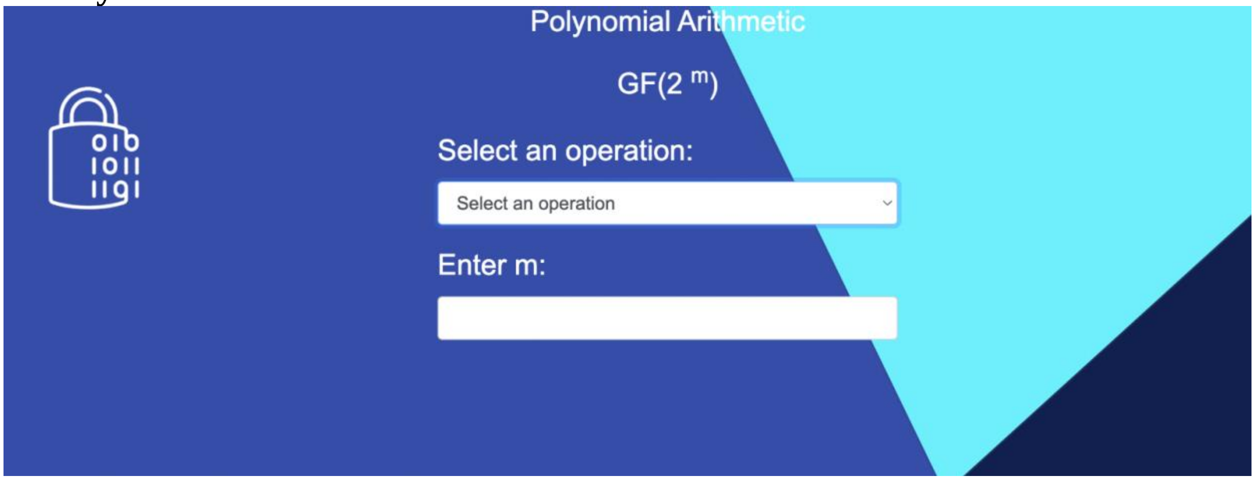


Figure 22 - Polynomial Arithmetic Page

User has choice of selecting the operations he wants to apply on input m.

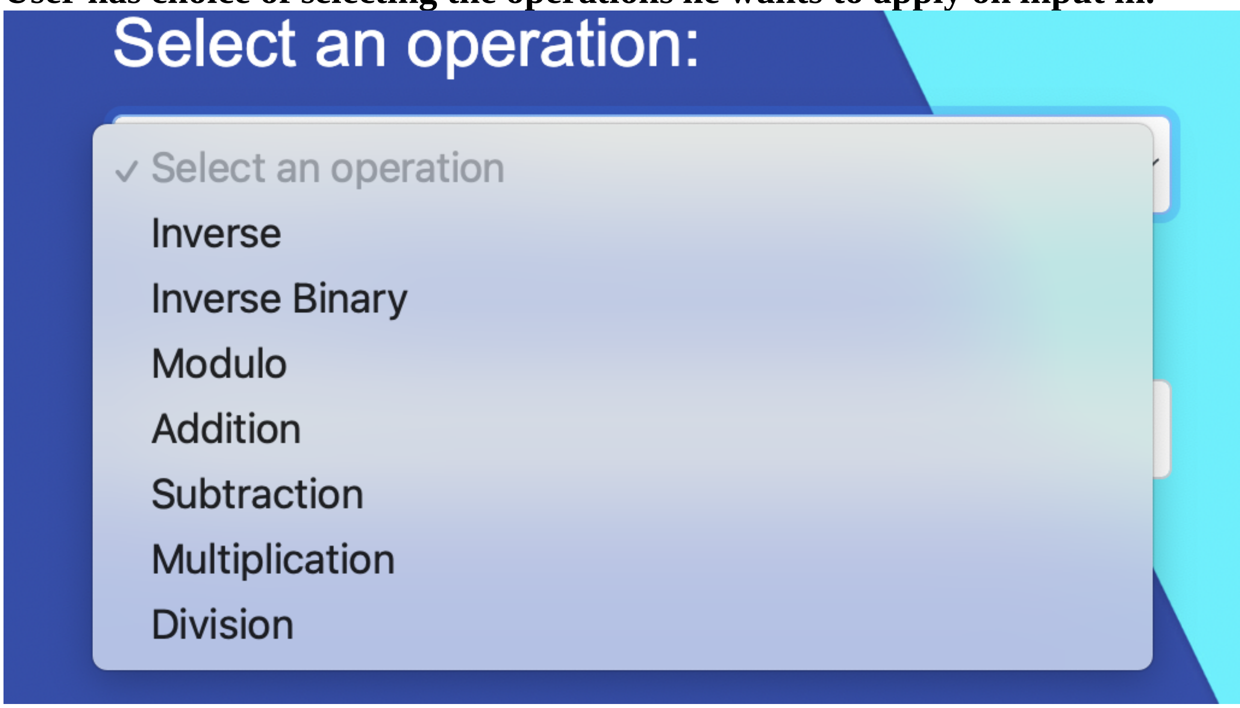


Figure 23 - PA Operation Options

--> Operation: Inverse <--

--> Test Case: m =3, poly = 0x7

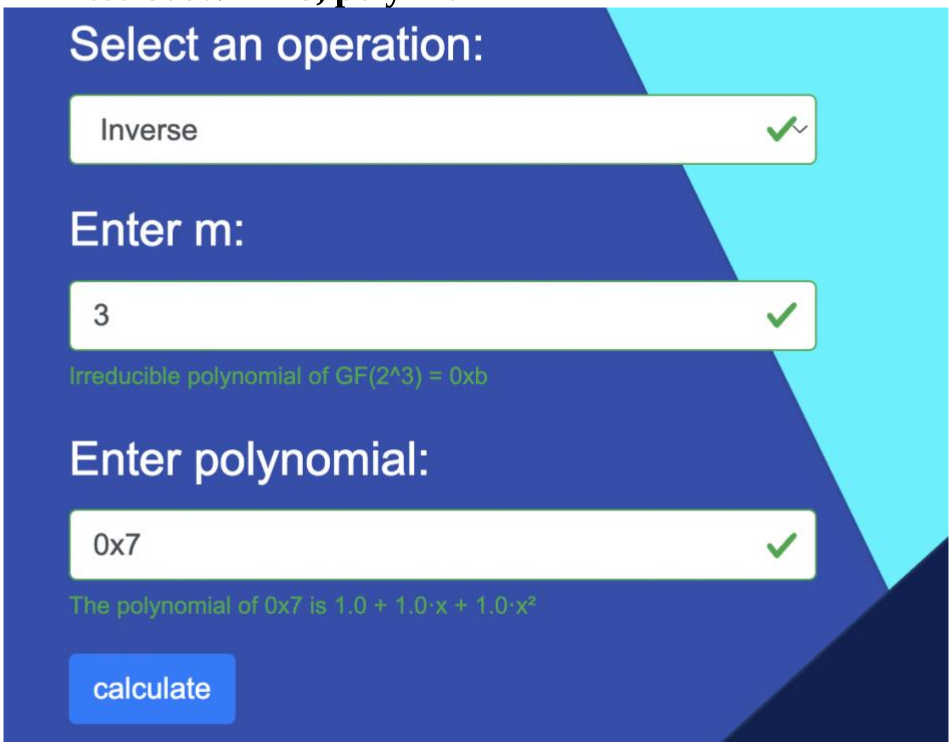
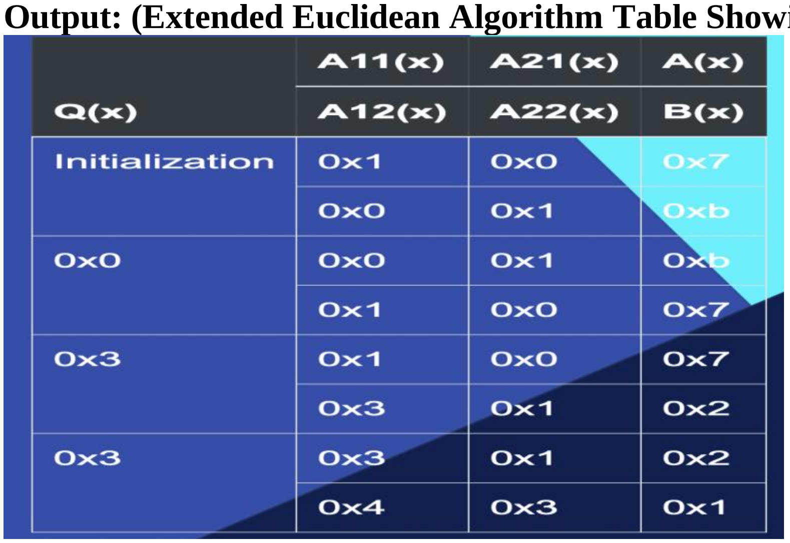


Figure 24 - Test Case 8

Output: (Extended Euclidean Algorithm Table Showing the inverse 0x3)



	A11(x)	A21(x)	A(x)
Q(x)	A12(x)	A22(x)	B(x)
Initialization	0x1	0x0	0x7
	0x0	0x1	0xb
0x0	0x0	0x1	0xb
	0x1	0x0	0x7
0x3	0x1	0x0	0x7
	0x3	0x1	0x2
0x3	0x3	0x1	0x2
	0x4	0x3	0x1

Figure 25 - Output of Test Case 8

--> Operation: Addition<---

--> Test Case: m =3, poly1 = 0x7, poly2 = 0x5

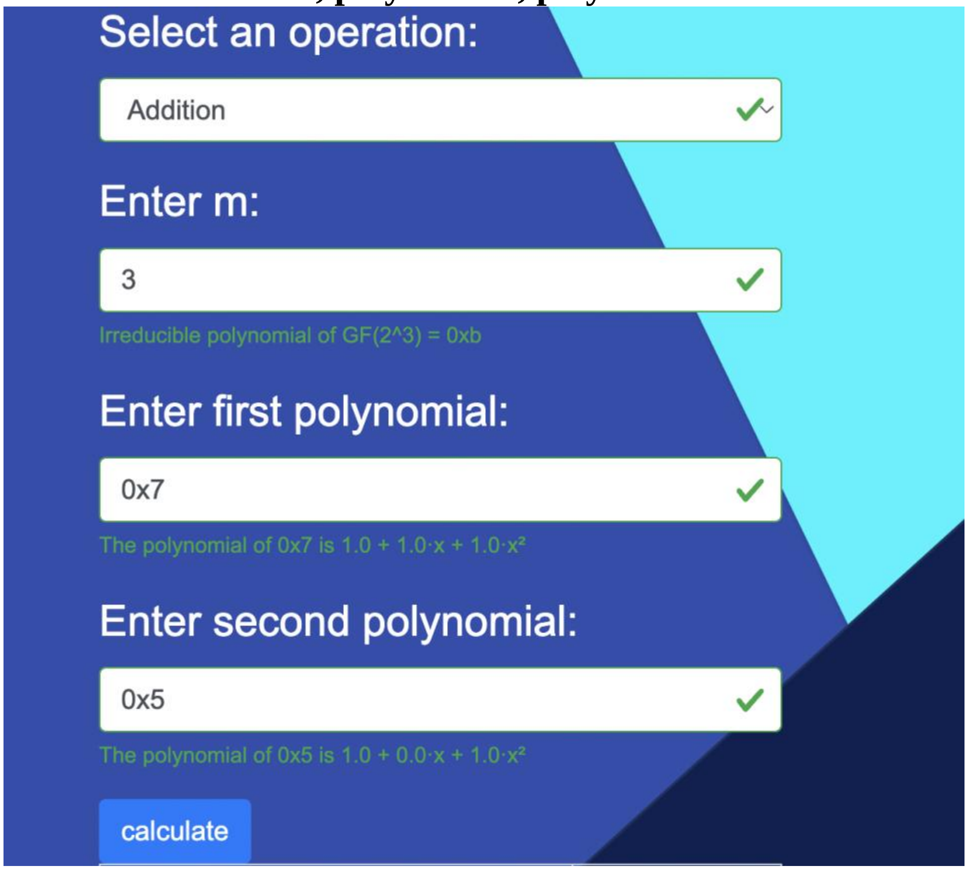


Figure 26 - Test Case 9

Output:

Figure 27 - Output of Test Case 9

--> Operation: Subtraction<--
--> Test Case: m =3, poly1 = 0x7, poly2 = 0x5

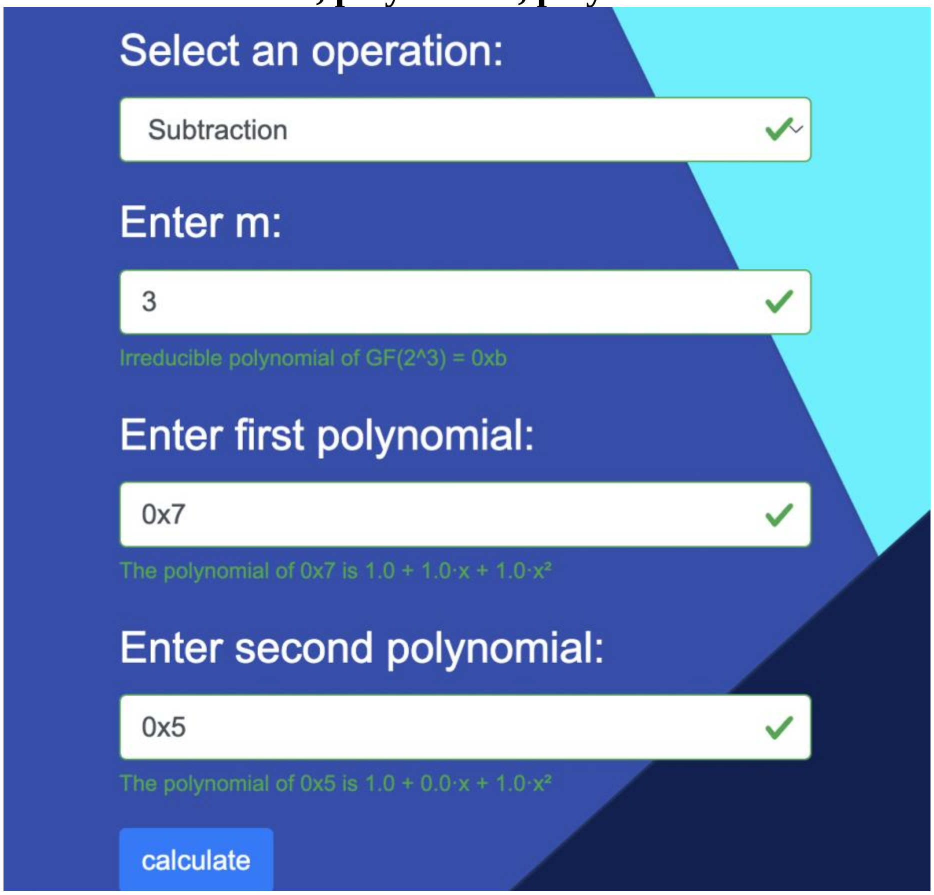
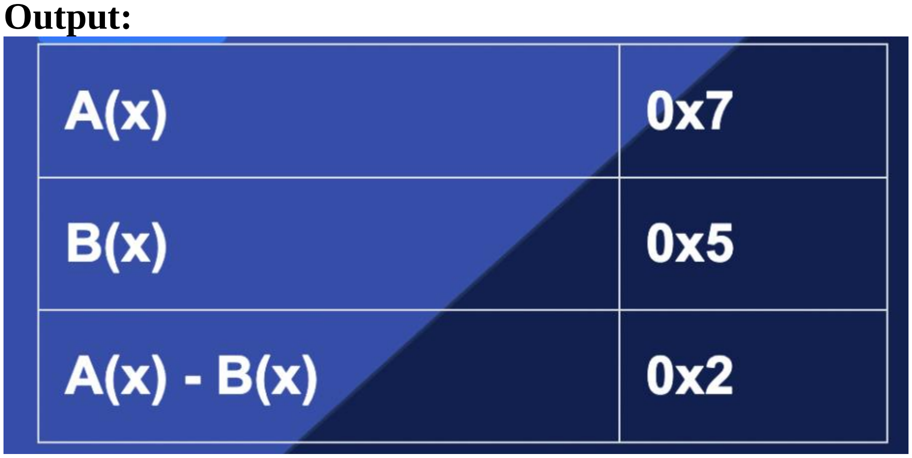


Figure 28 - Test Case 10

Output:



A(x)	0x7
B(x)	0x5
A(x) - B(x)	0x2

Figure 29 - Output of Test Case 10

--> Operation: Multiplication<--

--> Test Case: m =3, poly1 = 0x7, poly2 = 0x5

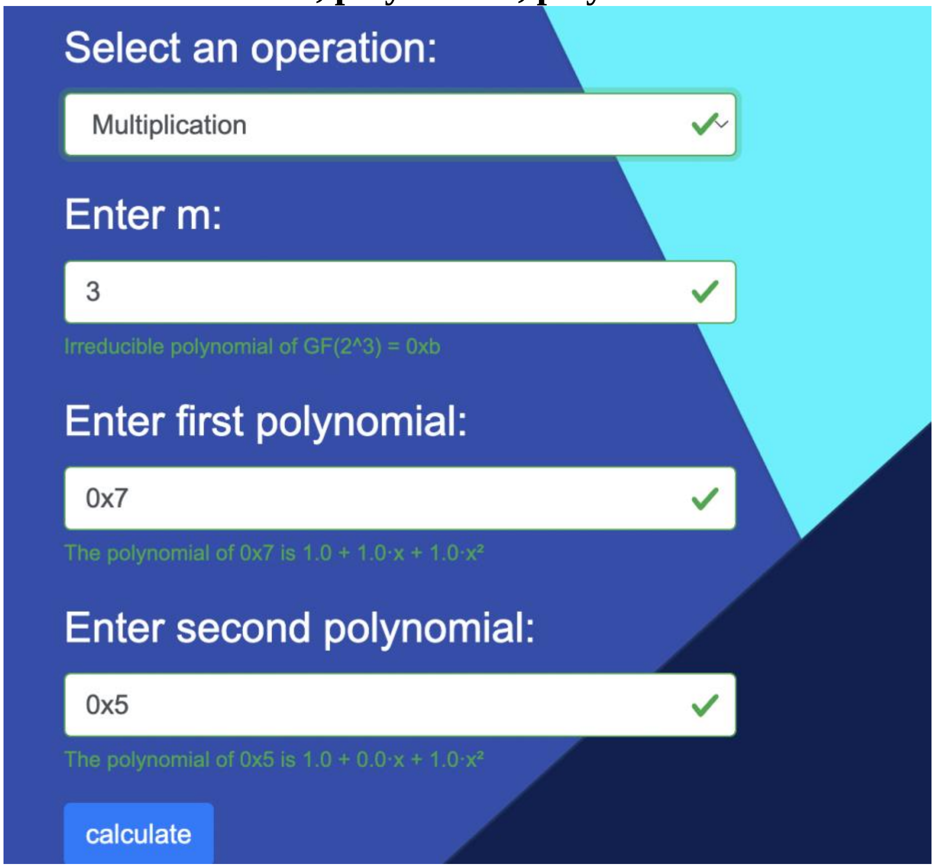
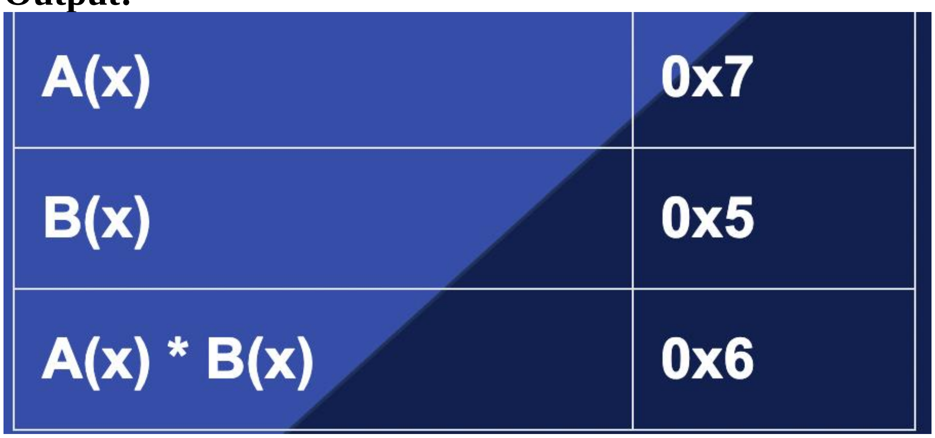


Figure 30 - Test Case 11

Output:



A(x)	0x7
B(x)	0x5
A(x) * B(x)	0x6

Figure 31 - Output of Test Case 11

--> Operation: Division<---
--> Test Case: m =3, poly1 = 0x7, poly2 = 0x5

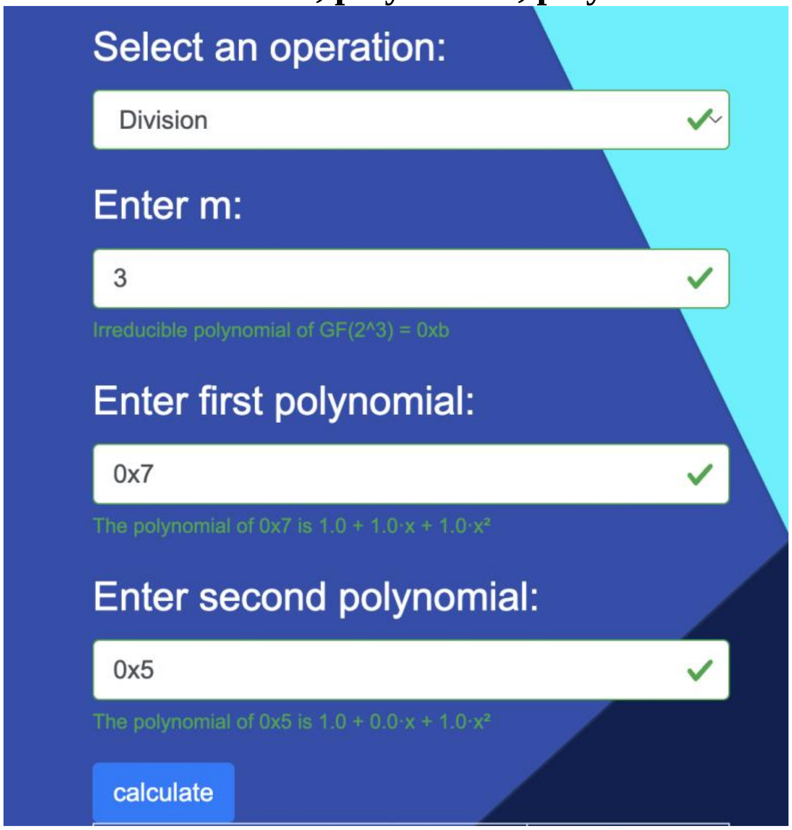
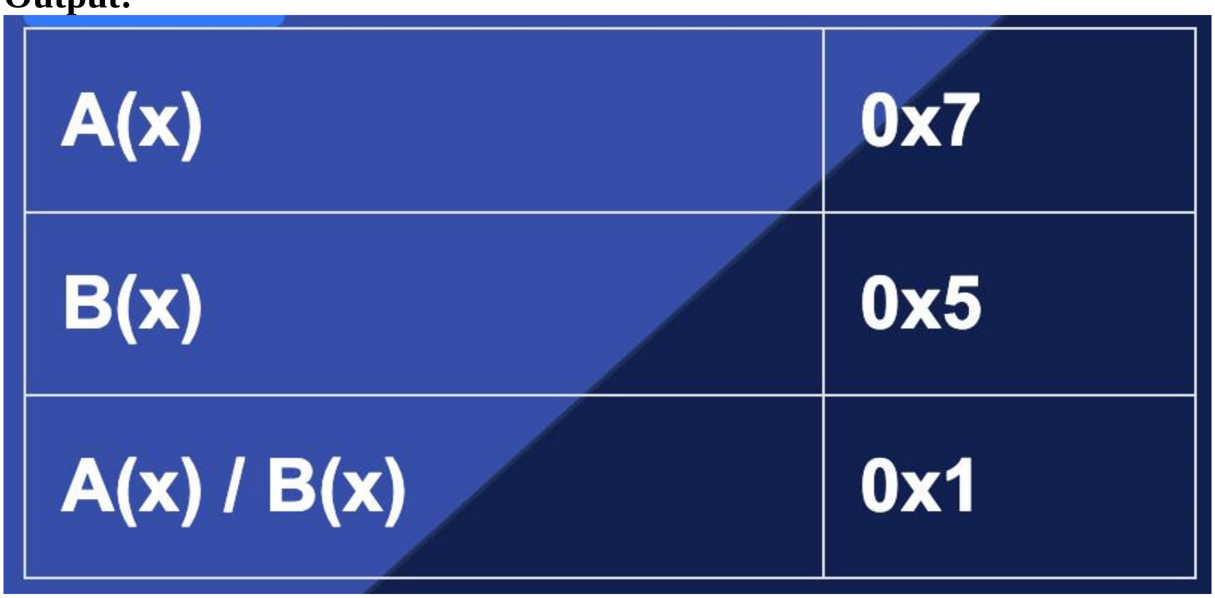


Figure 32 - Test Case 12

Output:



A(x)	0x7
B(x)	0x5
A(x) / B(x)	0x1

Figure 33 - Output of Test Case 12

--> Operation: Inverse Binary<---

--> Test Case: m =3, poly1 = 0x7, poly2 = 0x5

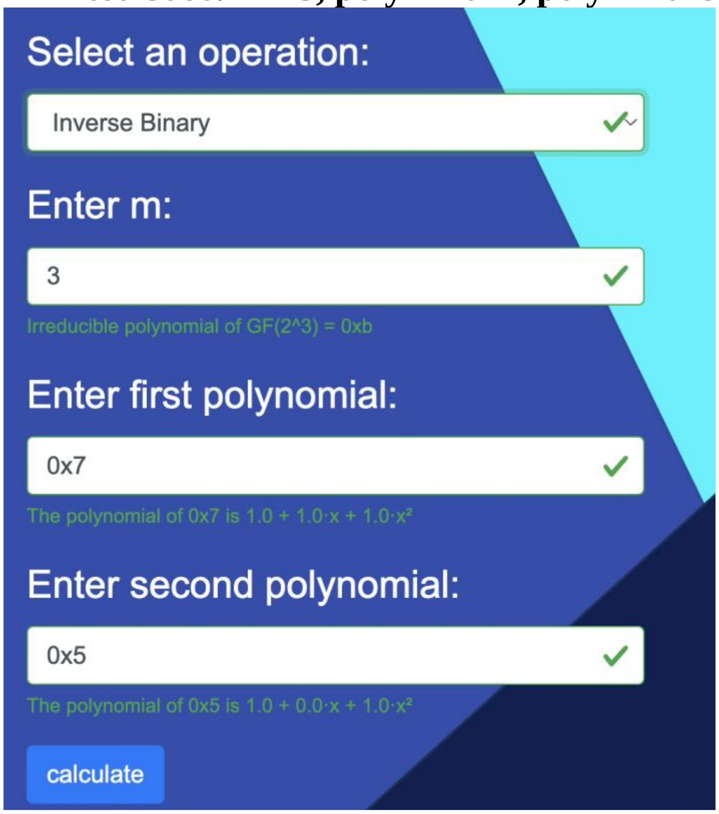
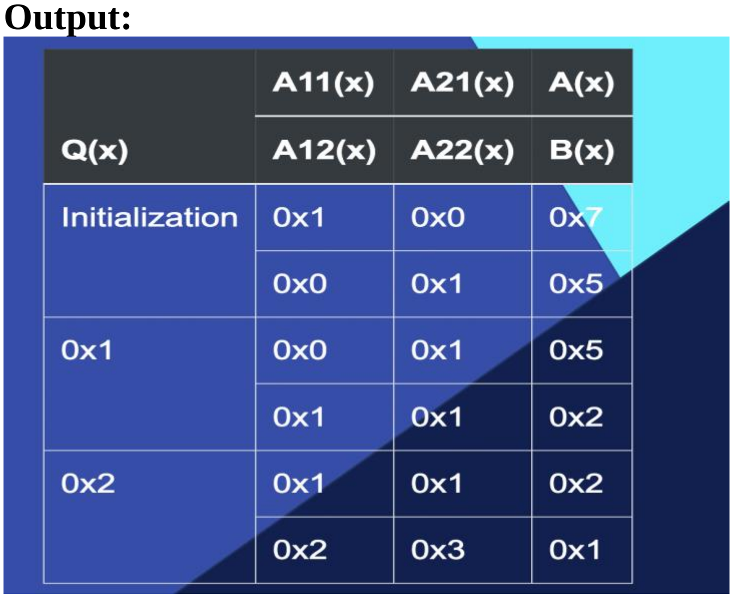


Figure 34 - Test Case 13

Output:



	A11(x)	A21(x)	A(x)
Q(x)	A12(x)	A22(x)	B(x)
Initialization	0x1	0x0	0x7
	0x0	0x1	0x5
0x1	0x0	0x1	0x5
	0x1	0x1	0x2
0x2	0x1	0x1	0x2
	0x2	0x3	0x1

Figure 35 - Output of Test Case 13

8- Conclusion

In conclusion, our encryption project represents the synergy of robust algorithms, mathematical precision, and user-centric design, culminating in a cybersecurity platform that seamlessly balances advanced security measures with simplicity. As we navigate the intricacies of classical and modern ciphers, the commitment to eliminating user intervention underscores our dedication to providing a reliable and hassle-free encryption experience. This report is not just a testament to our technical proficiency but also a reaffirmation of our mission to redefine cybersecurity by making cutting-edge protection accessible and user-friendly.