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

Contents

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 $O(n^{\log_2 3})$.
    - **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:**

```
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:**

```
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")
    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(0x592344231223244412312312421312312421)
    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:**

```
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-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.