Documentation for Implementation of Elliptic Curve Operations over Prime order Field

Tanushri Sen
Mtech CrS2324

December 2, 2024

Contents

1 Introduction

Elliptic Curve Cryptography (ECC) has gained significant prominence in the field of modern cryptography due to its high security with relatively small key sizes, making it highly efficient for applications in resource-constrained environments. ECC is based on the mathematics of elliptic curves, which are equations of the form:

$$y^2 = x^3 + ax + b$$

where a and b are constants, and the curve is defined over a finite field whose characteristics $\neq 0$. The use of elliptic curves over finite fields offers efficient solutions for public-key cryptographic systems such as encryption, digital signatures, and key exchange protocols.

In this project, we aim to implement **elliptic curve operations** over a **prime order field**. A prime order field is a finite field where the number of elements is a prime number, providing robust security properties. The underlying mathematical structure of ECC is the group of points on the elliptic curve, and the challenge lies in performing operations efficiently on these points.

Key Objectives:

- **Field Definition and Operations:** Define a finite field with a prime order, which is a key requirement for the implementation of elliptic curves. Implement efficient arithmetic operations (addition, subtraction, multiplication etc.) over this prime field.
- **Elliptic Curve Definition:** Select an appropriate elliptic curve equation $y^2 = x^3 + ax + b$ over the chosen finite field. Implement point addition and point doubling operations, which form the basis of elliptic curve scalar multiplication.
- **Scalar Multiplication:** Implement scalar multiplication kP, where k is a scalar and P is a point on the elliptic curve. This operation is computationally intensive and critical for ECC's security. Focus on optimizing this operation using the double-and-add method.

1.1 Overview of the Implementation

The project is structured into several key program files, each focusing on a specific aspect of Elliptic Curve over a prime order field:

- `utilities.h`**:** Header file defining function prototypes and data structures for I/O handling, base conversions, arithmetic, and modular operations in a prime field using base 29 representation.
- `FieldArithmetic.c`**:** Source file implementing functions for I/O handling, base conversions, arithmetic operations, and modular computations in a prime field using base 29 representation.
- `Exponentiation.c`**:** Source file implementing function for exponentiation of primitive element of prime field using base 29 representation.
- `main_FieldArithmetic.c`**:** implements the core functionality by utilizing the functions declared in `utilities.h` and defined in `FieldArithmetics.c`. The file serves as the entry point for performing operations such as base conversions, field addition, subtraction, multiplication, exponentiation, and modular reduction, as well as conversions between different bases.
- `ec_utilities.h`**:** Header file containing function prototypes for different elliptic curve operations
- `ec_operations.c`**:** Contains functions for defining the elliptic curve and performing point addition and doubling, scalar multiplication.
- `main_ec.c`**:** Serves as the entry point, coordinating the other components and demonstrating the EC operations.

This modular structure ensures efficient development, testing, and maintenance of the implementation.

2 Prime Order Field Arithmetic operations

2.1 Theoretical Background

A prime order field, denoted as $\mathbb{F}_p$, is a finite field where the number of elements (or order) is a prime number p. The field consists of integers $\{0, 1, 2, \ldots, p-1\}$ and all arithmetic operations are conducted modulo p, ensuring the results stay within the finite field. The primary operations in prime order field arithmetic include:

1. Addition

Addition of two elements a and b in the field is computed as:

$$c = (a + b) \mod p$$

2. Subtraction

Subtraction of two elements a and b in the field is computed as:

$$c = (a - b) \mod p$$

3. Multiplication

Multiplication of two elements a and b is given by:

$$c = (a \cdot b) \mod p$$

4. Division (Multiplicative Inverse)

Division in finite fields is achieved by multiplying by the modular inverse. To divide a by b, we compute:

$$c = a \cdot b^{-1} \mod p$$

The multiplicative inverse of b modulo p, denoted b^{-1}, is the number such that:

$$b \cdot b^{-1} \equiv 1 \mod p$$

The inverse can be calculated using algorithms like Little Fermat's Theorem.

5. Modular Exponentiation

Exponentiation in a prime order field is performed by repeatedly multiplying the base and reducing the result modulo p as follows:

$$c = a^e \mod p$$

Relevance to ECC

Prime order field arithmetic is central to ECC because all elliptic curve operations, such as point addition, point doubling, and scalar multiplication, involve field arithmetic. For instance:

- **Point addition and doubling** on the elliptic curve involve calculating sums and differences of point coordinates, which require modular arithmetic.
- **Scalar multiplication** involves repeatedly adding a point to itself, which requires efficient multiplication and inversion in the prime field.

Efficient implementation of these arithmetic operations ensures that ECC can be applied to cryptographic systems requiring high security with smaller key sizes compared to other public-key algorithms like RSA. By performing operations modulo a large prime p, ECC offers strong security with optimized performance.

2.2 Implementation of Prime Field Arithmetic Operations

The following steps outline the implementation of efficient prime field arithmetic operations, leveraging base 29 representation for optimized computations. The process includes base conversion, arithmetic operations in base 29, and modular reduction for field operations.

Step 1: Base Conversion

To work efficiently with numbers in a prime field, the input is first converted from hexadecimal (base 16) representation to a packed representation in base 29. Later, for output purposes, numbers are converted back from base 29 to hexadecimal.

1. Hexadecimal to Base 29 Conversion

```
1 void ToBase29(uint8_t* src, uint32_t* dest, int bytes);
```

It converts a hexadecimal number (byte array) to its equivalent packed base 29 representation. The pseudocode and the logic for implementing this function is given below:

ToBase29(uint8_t src[], uint32_t dest[], int bytes)

```
begin
    mask = 0x1FFFFFFF  // Mask for 29 bits
    // Process the first 8 groups of 29 bits
    dest[0] = Extract 29 bits from src[0] to src[3] using shifts and mask
    dest[1] = Extract 29 bits from src[3] to src[7] using shifts and mask
    dest[2] = Extract 29 bits from src[7] to src[10] using shifts and mask
    dest[3] = Extract 29 bits from src[10] to src[14] using shifts and mask
    dest[4] = Extract 29 bits from src[14] to src[18] using shifts and mask
    dest[5] = Extract 29 bits from src[18] to src[21] using shifts and mask
    dest[6] = Extract 29 bits from src[21] to src[25] using shifts and mask
    dest[7] = Extract 29 bits from src[25] to src[28] using shifts and mask
    dest[8] = Extract 29 bits from src[29] to src[31] using shifts and mask

    // Handle additional bytes if present
    if (bytes == 34)
        dest[8] = extract 5 bits from src[32] using shifts, concatenate with dest[8]
                  and mask
        dest[9] = Extract 29 bits from src[32] to src[33] using shifts and mask
    end if
end
```

Process:

1. Iterate through the hexadecimal input byte array `src`.
2. Pack the digits into base 29 representation, storing them in a `uint32_t` array.
3. Use bitwise operations to align bits correctly:
   - Apply left shifts to position bits appropriately.
   - Use bitwise OR operations to combine shifted bits.
   - Apply a mask (`0x1FFFFFFF`) to retain only 29 bits per element.
4. If additional bytes are present (e.g., input size is 34 bytes):
   - Adjust the packing of the last group of digits into the final array entries.

Output: A `uint32_t` array containing the packed Base-29 representation of the input byte array.

2. Base 29 to Hexadecimal Conversion

```
void ToBase16(uint32_t* src, uint8_t* dest);
```

It converts a packed base 29 number back to its equivalent hexadecimal form for output. The pseudocode and the logic for implementing this function is given below:

ToBase16(uint32_t src[], uint8_t dest[])

```
begin
    // Process each 32-bit number in the src array
    dest[0] = src[0] (lower 8 bits)
    dest[1] = src[0] (bits 8 to 15)
    dest[2] = src[0] (bits 16 to 23)
    dest[3] = Combine src[0] (bits 24 to 31) with src[1] (bits 0 to 4)

    dest[4] = src[1] (bits 3 to 10)
    dest[5] = src[1] (bits 11 to 18)
    dest[6] = src[1] (bits 19 to 26)
    dest[7] = Combine src[1] (bits 27 to 31) with src[2] (bits 0 to 1)

    dest[8] = src[2] (bits 6 to 13)
    dest[9] = src[2] (bits 14 to 21)
    dest[10] = Combine src[2] (bits 22 to 31) with src[3] (bits 0 to 6)

    dest[11] = src[3] (bits 1 to 8)
    dest[12] = src[3] (bits 9 to 16)
    dest[13] = src[3] (bits 17 to 24)
    dest[14] = Combine src[3] (bits 25 to 31) with src[4] (bits 0 to 4)

    dest[15] = src[4] (bits 4 to 11)
    dest[16] = src[4] (bits 12 to 19)
    dest[17] = src[4] (bits 20 to 27)
    dest[18] = Combine src[4] (bits 28 to 31) with src[5] (bits 0 to 1)

    dest[19] = src[5] (bits 7 to 14)
    dest[20] = src[5] (bits 15 to 22)
    dest[21] = Combine src[5] (bits 23 to 31) with src[6] (bits 0 to 6)

    dest[22] = src[6] (bits 2 to 9)
    dest[23] = src[6] (bits 10 to 17)
    dest[24] = src[6] (bits 18 to 25)
    dest[25] = Combine src[6] (bits 26 to 31) with src[7] (bits 0 to 3)

    dest[26] = src[7] (bits 5 to 12)
    dest[27] = src[7] (bits 13 to 20)
    dest[28] = src[7] (bits 21 to 28)

    dest[29] = src[8] (lower 8 bits)
    dest[30] = src[8] (bits 8 to 15)
    dest[31] = src[8] (bits 16 to 23)
end
```

Process:

1. Iterate through the input 32-bit integer array `src`.
2. Convert the 32-bit numbers into a base-16 byte array `dest`.
3. Use bitwise operations to align bits correctly:
   - Extract 8-bit segments from `src` using right shifts.
   - Combine overlapping segments from adjacent `src` values using left shifts and bitwise OR.
4. Handle segments that span across two `src` values by carefully combining and aligning the bits.

Output: A `uint8_t` array `dest`, containing the unpacked Base-16 representation of the input `src` array.

These conversions ensure compatibility between human-readable input/output formats (hex) and the internal computational format (base 29).

Step 2: Arithmetic Operations in Base 29

The following functions perform arithmetic operations on numbers in base 29. These operations are essential for efficient computations in a prime field. All the functions in this step do not yet apply reduction to keep the result within the bounds of the prime field.

1. Addition: `ADD()`

The `ADD` function takes two numbers, `num1` and `num2`, represented as uint32_t arrays in packed base 29, and adds them together. The result is stored in the `sum` array.

```
void ADD(uint32_t* num1, uint32_t* num2, uint32_t* sum, int ACTIVE_COUNT);
```

The following pseudocode implements the addition of packed numbers in base 29.

ADD(uint32_t* num1, uint32_t* num2, uint32_t* sum, int ACTIVE_COUNT)

```
begin
    carry = 0
    for i = 0 to ACTIVE_COUNT - 1 do
        sum[i] = num1[i] + num2[i] + carry
        carry = sum[i] >> 29
        sum[i] = sum[i] & mask
    end for
end
```

Process:

1. Iterate through the input base-29 arrays `num1` and `num2`.
2. For each index, perform the addition of corresponding elements from `num1`, `num2`, and the carry from the previous iteration.
3. Use bitwise operations to calculate the carry:
   - The carry is calculated by shifting the sum of each element right by 29 bits.
   - The least significant 29 bits of each sum are retained using bitwise AND with a mask.
4. Repeat the addition and carry calculation for all elements defined by `ACTIVE_COUNT`.

Output: A `uint32_t` array `sum`, containing the result of the packed addition in base 29.

These operations efficiently handle base-29 addition, ensuring correct carry propagation and element-wise summation in a prime field.

2. Subtraction: `SUB()`

The `SUB` function performs subtraction on two packed base-29 numbers using 2's complement. It computes the difference between `num1` and `num2`, storing the result in the `result` array.

```
void SUB(uint32_t* num1, uint32_t* num2, uint32_t* result, int ACTIVE_COUNT);
```

The following pseudocode implements the subtraction of packed numbers in base 29.

SUB(uint32_t* num1, uint32_t* num2, uint32_t* result, int ACTIVE_COUNT)

```
begin
    carry = 1
    for i = 0 to ACTIVE_COUNT - 1 do
        result[i] = num1[i] + (num2[i] XOR mask) + carry
        carry = result[i] >> 29
        result[i] = result[i] & mask
    end for
end
```

Process:

1. Iterate through the input base-29 arrays `num1` and `num2`.
2. Convert the second number, `num2`, to its 2's complement form by XOR'ing each element with a mask.
3. Add the corresponding elements from `num1`, the 2's complement of `num2`, and the carry from the previous iteration.
4. Use bitwise operations to calculate the carry:
   - The carry is calculated by shifting the sum of each element right by 29 bits.
   - The least significant 29 bits of each result are retained using bitwise AND with a mask.
5. Repeat the subtraction and carry calculation for all elements defined by `ACTIVE_COUNT`.

Output: A `uint32_t` array `result`, containing the result of the packed subtraction in base 29.

These operations ensure the correct subtraction of two packed numbers in base 29 using 2's complement, managing carry propagation and bitwise manipulation efficiently.

3. Multiplication: `Mult()`

The `Mult` function multiplies two packed base-29 numbers, `num1` and `num2`, and stores the result in the `result` array. The function does not yet apply reduction to keep the result within the bounds of the prime field.

```
void Mult(uint32_t* num1, uint32_t* num2, uint32_t* result, int ACTIVE_COUNT);
```

The multiplication is performed efficiently by first calculating the products of all pairs of elements from `num1` and `num2`, and then handling the carry and base-29 conversion to ensure proper result representation.

Mult(uint32_t* num1, uint32_t* num2, uint32_t* result, int ACTIVE_COUNT)

```
begin
    mult[0...2*(ACTIVE_COUNT - 1)] = 0

    for i = 0 to ACTIVE_COUNT - 1 do
        for j = 0 to ACTIVE_COUNT - 1 do
            mult[i+j] = mult[i+j] + (num1[i] * num2[j])
        end for
    end for

    carry = 0
    for i = 0 to 2*(ACTIVE_COUNT - 1) do
        mult[i] = mult[i] + carry
        carry = mult[i] >> 29
        result[i] = mult[i] AND mask
    end for

    result[2*ACTIVE_COUNT - 1] = carry
end
```

Process:

1. Initialize an array `mult` to store partial multiplication results.
2. Perform multiplication of each pair of elements from `num1` and `num2`.
3. Store the results of these multiplications in the `mult` array at the corresponding indices.
4. Convert the resulting products to base-29 by iterating through the `mult` array:
   - Add the carry from the previous iteration to the current element.
   - Calculate the new carry by shifting the element right by 29 bits.
   - Store the least significant 29 bits of each result using bitwise AND with a mask.
5. Store the final carry at the last index of the result array.

Output: A `uint32_t` array `result`, containing the result of the packed multiplication in base 29.

In the next step, we'll apply reduction to keep the results within the prime field, ensuring efficient operations.

Step 3: Field Arithmetics with Reduction

After performing base conversion and basic arithmetic, the next step is to handle arithmetic within a finite field. This step involves performing operations like addition, subtraction, multiplication, inversion, and division under modulo of a prime field, using base 29 representation with reduction.

3.1 Field Operations Pre-requisites

Before performing any other field operations, two crucial steps are necessary to ensure that the numbers stay within the defined field and are properly reduced:

• **IsGreater()** This function checks if one number is greater than another in base 29. It is crucial for comparison operations before performing other arithmetic in the field.

```
1 int IsGreater(uint32_t* num1, uint32_t* num2);
```

It compares the elements of both numbers and returns 1 if `num1` is greater than `num2`, otherwise returns 0.

IsGreater(uint32_t* num1, uint32_t* num2)

```
begin
    IsGreater = 0

    for i = 9 downto 0 do
        if num1[i] > num2[i] then
            IsGreater = 1
            break
        else if num1[i] == num2[i] then
            continue
        else
            break
    end for
    return IsGreater
end
```

Process:

1. The function compares the two input arrays `num1` and `num2` element by element starting from the most significant element (index 9).
2. If an element of `num1` is greater than the corresponding element in `num2`, the function sets `IsGreater` to 1 and exits the loop.
3. If the elements are equal, the loop continues to the next pair of elements.
4. If an element of `num1` is smaller than the corresponding element in `num2`, the loop exits, and `IsGreater` remains 0.

Output: Returns 1 if `num1` is greater than `num2`, otherwise returns 0.

• **Barrett_Red()** Barrett reduction is used to reduce larger numbers into valid field elements. This method ensures that numbers larger than the field size are reduced modulo a prime `p`.

```
1 void Barrett_Red(uint32_t* num, uint32_t* p, uint32_t* result);
```

The algorithm mirrors the core principles of the Barrett reduction algorithm, which aims to efficiently reduce large numbers modulo p by utilizing precomputed values like $T = \lfloor \theta^{2L}/p \rfloor$ where $\theta = 2^{29}$ and $L = 9$ in our case.

Process:

1. Initialize the temporary arrays `q2` and `temp`, and set `r1` to `num` and `r2` to `temp`.
2. Perform multiplication on the input $\lfloor \texttt{num}/\theta^{L-1} \rfloor$ with a constant `T` to get `q2`. Here the need for complex division is avoided by computing `num + 8`.
3. Perform another multiplication of $\lfloor \texttt{q2}/\theta^{L+1} \rfloor$ with the prime `p` to get `temp`. Again `q2 + 10` is computed to avoid complex division.
4. Now, `r1 = num mod` θ^{L+1} and `r2 = temp mod` θ^{L+1}
5. Subtract `r2` from `r1` to get the result.
6. If the result is greater than `p`, subtract `p` from the result twice to ensure the result is within the field.

Output: The result of the reduction of `num` modulo `p`, which is stored in `result`.

Barrett_Red(uint32_t* num, uint32_t* p, uint32_t* result)

```
begin
    q2 = {0}
    temp = {0}
    r1 = num
    r2 = temp

    Mult(num[8], T, q2, 10)
    Mult(q2[10], p, temp, 10)

    SUB(r1, r2, result, 10)

    if IsGreater(result, p) then
        SUB(result, p, result, 10)
    end if
    if IsGreater(result, p) then
        SUB(result, p, result, 10)
    end if
end
```

This function helps maintain the integrity of the field by reducing larger numbers into valid field elements.

3.2 Field Operations

These operations work with elements in a finite field, reducing results after each operation to ensure they stay within the bounds of the field. The following functions implement these field arithmetic operations:

1. FieldAddition() This function performs the addition of two numbers in the field. The addition is done element-wise in base 29, and the result is reduced to the field size after the operation.

```
void FieldAddition(uint32_t* num1, uint32_t* num2, uint32_t* result);
```

Below the pseudocode and the logic for implementing this function is given:

FieldAddition(uint32_t num1[], uint32_t num2[], uint32_t sum[])

```
begin
    ADD(num1, num2, sum, 9)

    if (IsGreater(sum, p) == 1)
        SUB(sum, p, sum, 9)
    end if
end
```

Process:

1. Perform addition of the two input numbers, `num1` and `num2`, using base 29 addition through the `ADD` function.
2. After the addition, check if the result (`sum`) exceeds the field size `p` by using the `IsGreater` function.
3. If `sum` is greater than `p`, subtract `p` from `sum` using the `SUB` function to ensure the result remains within the prime field.

Output: A `uint32_t` array `sum`, containing the result of the field addition, reduced if necessary.

2. FieldSubtraction() This function performs subtraction between two numbers in the field. The subtraction is done element-wise in base 29, with proper handling of borrow and carry bits.

```
void FieldSubtraction(uint32_t* num1, uint32_t* num2, uint32_t* sub);
```

Below the pseudocode and the logic for implementing this function is given:

FieldSubtraction(uint32_t num1[], uint32_t num2[], uint32_t sub[])

```
begin
    uint32_t result[20] = {0}

    SUB(num1, num2, result, 9)
    ADD(result, p, result, 9)
    Barrett_Red(result, p, result)

    for i = 0 to 8 do
        sub[i] = result[i]
    end for
end
```

Process:

1. Perform modular subtraction between `num1` and `num2` using the `SUB` function.
2. Add the prime modulus `p` to the result to ensure a non-negative outcome.
3. Perform Barrett reduction to ensure that the result is within the valid field range, i.e., less than `p`.
4. Copy the final result into the `sub` array for the output.

Output: A `uint32_t` array `sub`, containing the result of the field subtraction, reduced to stay within the field.

3. FieldMult() This function multiplies two numbers in the field and applies the Barrett reduction to keep the result within the field size. The multiplication is done in base 29, and the final result is reduced using Barrett's method.

```
void FieldMult(uint32_t* num1, uint32_t* num2, uint32_t* result);
```

Below the pseudocode and the logic for implementing this function is given:

FieldMult(uint32_t num1[], uint32_t num2[], uint32_t result[])

```
begin
    uint32_t temp[20] = {0}
    Mult(num1, num2, temp, 9)

    uint32_t temp1[9]={0}
    Barrett_Red(temp, p, result)
end
```

Process:

1. Multiply `num1` and `num2` using the `Mult` function, storing the intermediate result in the `temp` array.

$$\texttt{temp} = \texttt{num1} * \texttt{num2}$$

2. Perform Barrett reduction on the result to ensure it is reduced to fit within the field, using the prime modulus `p`.

$$\texttt{temp} = \texttt{temp mod p}$$

3. Store the final reduced result in the `result` array.

Output: A `uint32_t` array `result`, containing the product of `num1` and `num2` reduced to the prime field.

4. FieldInverse() This function computes the modular inverse of a number in the prime field using Fermat's Little Theorem. It is used for division operations and ensures the result is within the field.

```
void FieldInverse(uint32_t* num, uint32_t* result);
```

Fermat's Little Theorem states that if p is a prime number and a is an integer not divisible by p, then:

$$a^{p-1} \equiv 1 \pmod{p}$$

Alternatively:

$$a^{-1} \equiv a^{p-2} \pmod{p}$$

Below the pseudocode and the logic for implementing this function is given:

FieldInverse(uint32_t num[], uint32_t result[])

```
begin
    exp = p-2
    FieldExp_Montgomery_noBranching(num, exp, result)
end
```

Process:

1. Initialize an array `exp` to hold the value of the prime modulus `p-2`.
2. Use the `FieldExp_Montgomery_noBranching` function (will be discussed in next section) to compute $\texttt{num}^{p-2}$ `mod p`, which is the modular inverse of `num`.

$$\texttt{num}^{-1} \equiv \texttt{num}^{p-2} \pmod{p}$$

Output: A `uint32_t` array `result`, containing the modular inverse of `num` in the prime field.

5. FieldDivision() This function divides one number by another in the field. It achieves this by multiplying `num1` with the modular inverse of `num2` (calculated using the `FieldInverse` function) and reducing the result modulo `p`.

```
void FieldDivision(uint32_t* num1, uint32_t* num2, uint32_t* result);
```

Below the pseudocode and the logic for implementing this function is given:

FieldDivision(uint32_t num1[], uint32_t num2[], uint32_t result[])

```
begin
    uint32_t inverse[10] = {0}
    FieldInverse(num2, inverse)

    FieldMult(num1, inverse, result)
end
```

Process:

1. Use the `FieldInverse` function to compute the modular inverse of `num2`.
2. Multiply `num1` with the computed inverse of `num2` using the `FieldMult` function.
3. The result of this operation is equivalent to $(\texttt{num1} \times \texttt{num2}^{-1}) \mod \texttt{p}$.

Output: A `uint32_t` array `result`, containing the quotient $(\texttt{num1}/\texttt{num2}) \mod \texttt{p}$.

6. Field_ConstMult() This function performs multiplication of a field element by a constant scalar of size 32 bits (maximum). The result is reduced to the field size after the operation.

```
void Field_ConstMult(uint32_t* num, uint32_t constant, uint32_t* result);
```

This function will be used to improve the efficiency of the `dbl` operation of elliptic curve. Below the pseudocode and the logic for implementing this function is given:

Field_ConstMult(uint32_t num[], uint32_t constant, uint32_t result[])

```
begin
    for i = 0 to 8 do
        mult[i] = num[i] * constant
    end for

    carry = 0
    for i = 0 to 8 do
        mult[i] = mult[i] + carry
        carry = mult[i] >> 29
        temp[i] = mult[i] & mask
    end for

    Barrett_Red(temp, p, result)
end
```

Process:

1. Multiply each 29-bit digit of the input `num` by the small `constant`., which takes only 9 multiplications, much lesser than the number of multiplications needed in `FieldMult()` [2.2].
2. Handle carry propagation and convert the result into 29-bit packed representation.
3. Apply the `Barrett_Red` function to ensure the result is a valid field element ($< p$).

Output: A `uint32_t` array `result`, containing the product $(\texttt{num} \times \texttt{constant}) \mod \texttt{p}$.

These field arithmetic functions are essential for efficiently performing operations in the prime field, with each function ensuring the correct result by applying base 29 arithmetic and reduction methods such as Barrett's reduction and Fermat's Little Theorem for inverses.

2.3 Implementation of Exponentiation of a Primitive Element in a Prime Field

In the context of finite fields, a **primitive element** is an element that generates all non-zero elements of the field under multiplication. In other words, every non-zero element of the field can be written as a power of the primitive element.

Goal: The objective is to compute the exponentiation of the primitive element in the field, which is essential for many cryptographic operations, including modular exponentiation and elliptic curve cryptography.

The following algorithms are used to compute the exponentiation of a primitive element in a finite field:

1. **Square-and-Multiply:** a classic method used to compute modular exponentiation. It works by representing the exponent in binary and performing a series of squaring and multiplying operations.
   - Left-to-Right Square-and-Multiply
   - Right-to-Left Square-and-Multiply
2. **Montgomery Ladder:** is an efficient method for modular exponentiation that is commonly used in cryptographic applications, especially in elliptic curve cryptography. It operates in constant time, meaning that the number of operations does not depend on the specific bits of the exponent, preventing timing attacks.

Before proceed to the implementation explanation of above algorithms, first discuss the pre-requisite functions to implement them.

• **IsZero():** This function checks whether a number in base 29 is zero.

```
int IsZero(uint32_t* num, int length);
```

It iterates through all the base 29 digits and confirms if they are all zero.

IsZero(uint32_t num[], int length)

```
begin
    for i = 0 to length - 1 do
        if num[i] != 0
            return 0
        end if
    end for

    return 1
end
```

Process:

1. Iterate through all digits of the input array `num`.
2. If any digit is non-zero, return `0`, indicating the number is not zero.
3. If all digits are zero, return `1`, indicating the number is zero.

Output: Returns `1` if the number is zero, otherwise returns `0`.

This step ensures that the input is valid before performing exponentiation.

• **BitLength():** This function computes the number of bits needed to represent a given exponent in binary.

```
int BitLength(uint32_t* exp);
```

Takes exponent `exp` as input and computes number of bits in `exp`.

BitLength(uint32_t exp[])

```
begin
    for i = 8 down to 0 do
        if exp[i] != 0
            for j = 28 down to 0 do
                if (exp[i] >> j) & 1 != 0
                    return (i * 29 + j + 1)
                end if
            end for
        end if
    end for

    return 0
end
```

Process:

1. Traverse the exponent array `exp` from the most significant 29-bit chunk.
2. Check each bit in the chunk. If a bit is set to 1, compute and return the bit length.
3. The bit length is calculated as the position of the most significant set bit.
4. If all chunks are zero, return 0, indicating that the exponent is zero.

Output: The bit length of the exponent `exp`, or `0` if the exponent is zero.

• **IsCompatible():** This function checks whether the exponent is within the valid range, ensuring that the exponent satisfies the condition $2 \leq \exp \leq p - 2$.

```
int IsCompatible(uint32_t* exp);
```

This is crucial during the Diffie-Hellman Key Exchange to prevent the secret key from becoming equal to the primitive element, ensuring secure key generation.

IsCompatible(uint32_t exp[])

```
begin
    one = 0x1
    p_1 = p - 1

    flag = IsGreater(exp, one) && IsGreater(p_1, exp)
    return flag
end
```

Process:

1. Initialize an array `one` to represent the value 1.
2. Compute `p_1`, which is `p-1`, by subtracting 1 from the first element of the prime `p` and copying the remaining elements.
3. Check if the exponent `exp` is greater than 1 (i.e., `exp > 1`) and less than `p-1` (i.e., `exp < p-1`).
4. The function returns 1 if the condition is met, indicating that `exp` is in the valid range; otherwise, it returns 0.

Output: 1 if `exp` is between 2 and `p-2`, inclusive; otherwise, 0.

Finding exponentiation of primitive element g with different algorithms mentioned above:

2.3.1 Left-to-Right Square-and-Multiply

This algorithm performs modular exponentiation. It iterates over the bits of the exponent from left to right, repeatedly squaring the base and multiplying the result by the base when the corresponding bit of the exponent is 1. This method is efficient and widely used in cryptographic applications.

```
void FieldExp_left2right(uint32_t* base, uint32_t* exponent, uint32_t* result);
```

Pseudocode and logic to implementing the function is given below:

FieldExp_left2right(uint32_t base[], uint32_t exp[], uint32_t result[])

```
begin
    result = 1
    expBitLength = BitLength(exp)

    for i = expBitLength - 1 to 0
        FieldMult(result, result, result)
        if ((exp[i / 29] >> (i % 29)) & 1)
            FieldMult(result, base, result)

    return result
end
```

Process:

1. Initialize `result` to 1 in base-29 format (the identity element in modular arithmetic).
2. Calculate the bit length of the exponent `exp` using the `BitLength` function.
3. Iterate over each bit of the exponent starting from the most significant bit.
4. For each bit, square the current `result` (i.e., `result = result`^{2} `mod p`).
5. If the current bit is set (i.e., `1`), multiply the `result` by the `base` (i.e., `result = result * base mod p`).
6. After completing the loop, return the final `result`.

Output: The modular exponentiation `base`$^{\texttt{exp}}$ `mod p`.

2.3.2 Right-to-Left Square-and-Multiply

In contrast to the left-to-right method, this algorithm performs modular exponentiation by processing the exponent from right to left. It uses the same square-and-multiply approach but in reverse order, which may offer different performance characteristics depending on the hardware or implementation.

```
void FieldExp_right2left(uint32_t* base, uint32_t* exponent, uint32_t* result);
```

Pseudocode and logic to implementing the function is given below:

FieldExp_right2left(uint32_t base[], uint32_t exp[], uint32_t result[])

```
begin
    b = base
    result = 1
    expBitLength = BitLength(exp)

    for i = 0 to expBitLength - 1
        if ((exp[i / 29] >> (i % 29)) & 1)
            FieldMult(result, b, result)
        endif
        FieldMult(b, b, b)

    return result
end
```

Process:

1. Copy the `base` into a temporary array `b` so that during execution the original `base` doesn't get changed.
2. Initialize `result` to 1 in base-29 format (the identity element in modular arithmetic).
3. Calculate the bit length of the exponent `exp` using the `BitLength` function.
4. Iterate over each bit of the exponent starting from the least significant bit.
5. For each bit, if the current bit is set (i.e., `1`), multiply the `result` by the base `b`.
6. After each iteration, square `b` (i.e., `b = b`^{2} `mod p`).
7. After completing the loop, return the final `result`.

Output: The modular exponentiation `base`$^{\texttt{exp}}$ `mod p`.

2.3.3 Montgomery Ladder

This algorithm uses the Montgomery ladder method, which is a more efficient approach for modular exponentiation. It avoids conditional branches, making it resistant to timing attacks and more suitable for hardware implementations. It is commonly used in cryptographic protocols where security is a priority.

• **FieldExp_Montgomery_noBranching():** This version of the Montgomery ladder method eliminates branching entirely, offering further optimizations. By removing conditional branching, it ensures that the execution time is independent of the exponent, thus enhancing security and performance.

```
1 void FieldExp_Montgomery_noBranching(uint32_t* base, uint32_t* exp, uint32_t*
    result);
```

Pseudocode and logic to implementing the function is given below:

Process:

1. If the exponent is zero, initialize the result as 1 and return it.
2. Copy the `base` into `S` and compute `base`^{2} into `R`.
3. Calculate the bit length of the exponent `exp` using the `BitLength` function.
4. Iterate over each bit of the exponent starting from the second most significant bit.
5. For each bit, if the bit is zero, swap `S` and `R`.
6. Compute `S * R` and `R * R`, then update `S` and `R` without branching based on the current bit.

7. After completing the loop, return the final `result` stored in `S`.

Output: The modular exponentiation $\texttt{base}^{\texttt{exp}} \bmod \texttt{p}$.

FieldExp_Montgomery_noBranching(uint32_t base[], uint32_t exp[], uint32_t result[])

```
begin
    if IsZero(exp, 10)
        result = 1
        return result
    endif

    S = base
    R = base^2
    expBitLength = BitLength(exp)

    for i = expBitLength - 2 downto 0
        bit = (exp[i / 29] >> (i % 29)) & 1

        if (bit == 0)  swap(S, R)

        FieldMult(S, R, tempSR)
        FieldMult(R, R, tempR)

        for j = 0 to 9
            S[j] = (bit * tempSR[j]) + ((1 - bit) * tempR[j])
            R[j] = (bit * tempR[j]) + ((1 - bit) * tempSR[j])
        endfor
    endfor

    result = S
    return result
end
```

These algorithms allow efficient computation of the exponentiation of the primitive element, which is fundamental for performing operations such as key exchange in cryptographic systems.

2.4 Testing and Verification

Refer to the code provided in the **Code Listing** section for example usage of these functions in the main body of the code. To verify the correctness of the implementation, use the following test vectors:

- **Given prime -**

  `p = 0xe92e40ad6f281c8a082afdc49e1372659455bec8ceea043a614c835b7fe9eff5`

For Field Arithmetic operations

- **Given two numbers -**

  `a = 0x78ae6fbfa1e4a734e9c9b1a55d826707107f8784e4accafd50f726417da9a84d`

  `b = 0xc43a291c2502b6e002c9ba5efaad8a4e444ad6c2c6bb0ad75bf9d4d3922ac366`

- **Expected Outputs -**

  `a+b mod p = 0x53ba582e57bf418ae4686e3fba1c7eefc0749f7edc7dd19a4ba477b98fea7bbe`

`a-b mod p = 0x9da28750ec0a0cdeef2af50b00e84f1e608a6f8aecdbc4605649d4c96b68d4dc`

`a*b mod p = 0x3b7ff86865616aff57b4d2de0f8e68950bee21326729b4dd487db6a8c6f0a5fa`

a^{-1} `mod p = 0x1cf6b4fe39eb908d6798a63f1ceeab8f596ec25d39fe9f114642cb618c901544`

`a/b mod p = 0x276463815a64715fa0ef1f9b6bb65afbbde2946f63e4447bbbdb5584c64744cc`

For Exponentiation of primitive element

- **Sample Inputs:**

  Primitive element of the field $\mathbb{F}_p$, `g = 2`

  Exponent, `exp = 0x1b50014e06c430d5f5436db1aff38f2d9996c86afaf66dfc585f6ce9793af55b`

- **Expected output:**

  g^{exp} `mod p = 0xc07e4038b87688946eec1e605f9f0f59dc3f4f95c0bd8f41d39efdc0cb029580`

3 Elliptic Curve Operations

3.1 Theoretical Background

Elliptic Curve: The equation of an elliptic curve over a finite field is given by:

$$y^2 = x^3 + ax + b \mod p$$

where a, b, and p are constants, and p is a prime number defining the finite field where $p \neq 2, 3$. The main operations on elliptic curves are:

- **Point Addition:** Adding two points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ on the curve.
- **Point Doubling:** Doubling a point $P = (x_1, y_1)$ to get $2P = (x_3, y_3)$.
- **Scalar Multiplication:** Multiplying a point P by a scalar k (an integer), resulting in a new point kP.

Point Addition

Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be two distinct points on the elliptic curve. The point addition formula is:

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1} \mod p$$

$$x_3 = \lambda^2 - x_1 - x_2 \mod p$$

$$y_3 = \lambda(x_1 - x_3) - y_1 \mod p$$

where λ is the slope of the line joining P and Q.

Point Doubling

If $P = Q$, the point doubling formula is used:

$$\lambda = \frac{3x_1^2 + a}{2y_1} \mod p$$

$$x_3 = \lambda^2 - 2x_1 \mod p$$

$$y_3 = \lambda(x_1 - x_3) - y_1 \mod p$$

where λ is the slope of the tangent at P.

Scalar Multiplication

Scalar multiplication of a point P by a scalar k is computed using the double-and-add method:

$$kP = P + P + \cdots + P \quad \text{(k times)}$$

This can be optimized using efficient algorithms like the double-and-add.

3.2 Implementation of Elliptic Curve Operations

The implementation of elliptic curve operations involves the following steps:

Step 1: Define the Curve Parameters

The elliptic curve is defined by selecting parameters a and b such that the discriminant $\Delta = -16(4a^3+27b^2) \neq 0$, ensuring a valid curve. Tools like SageMath can be used to select these parameters efficiently. For this implementation,

$$a = 0\text{x}1 \text{ and } b = 0\text{x}71$$

is selected to simplify computations.

Step 2: Optimize Curve Properties

Choose parameters that minimize the cofactor h, making the curve more efficient for cryptographic operations. For this implementation, a, b are chosen so that $h = 1$.

Step 3: Validate Points on the Curve

Input points (x, y) are verified to ensure they satisfy the curve equation using the `IsPointOnCurve()` function. This ensures that only valid points are processed in subsequent operations.

• **IsPointOnCurve():** This function checks whether a given point (x, y) lies on the elliptic curve.

```
bool IsPointOnCurve(uint32_t* x, uint32_t* y);
```

Pseudocode for this function is given below:

IsPointOnCurve(uint32_t x[], uint32_t y[])

```
begin
    y_squared = 0, x_cubed = 0
    ax = 0, rhs[10] = {0}

    FieldMult(y, y, y_squared)
    FieldMult(x, x, x_cubed)
    FieldMult(x_cubed, x, x_cubed)
    FieldMult(x, a, ax)
    FieldAddition(x_cubed, ax, rhs)
    FieldAddition(rhs, b, rhs)

    for i = 0 to 9
        if y_squared[i] != rhs[i]
            return false
        endif
    endfor
    return true
end
```

Process:

1. Compute $y^2 \mod p$ and store the result in `y_squared`.
2. Compute $x^3 \mod p$ and store the result in `x_cubed`.
3. Compute $ax \mod p$ using the coefficient a and store the result in `ax`.
4. Compute the right-hand side (RHS) of the elliptic curve equation:
$$\text{RHS} = x^3 + ax + b \mod p$$
5. Compare y^2 with RHS chunk-by-chunk. If any chunk differs, return `false`.
6. If all chunks are equal, return `true`.

Output: A boolean value indicating whether the point (x, y) lies on the elliptic curve.

Step 4: Implement Elliptic Curve Operations

Below the functions are defined to perform key elliptic curve operations. These operations are the foundation for elliptic curve cryptographic protocols.

3.2.1 Point Addition (add)

Performs the addition of two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ to compute $P_1 + P_2 = (x_3, y_3)$ using the formula mentioned in above section 3.1.

```
void add(uint32_t* x1, uint32_t* y1, uint32_t* x2, uint32_t* y2, uint32_t* x3,
    uint32_t* y3);
```

Pseudocode and the explanation given below:

add(uint32_t x1[], uint32_t y1[], uint32_t x2[], uint32_t y2[], uint32_t x3[], uint32_t y3[])

```
begin
    lambda = 0, temp1 = 0, temp2 = 0, temp3 = 0

    FieldSubtraction(y2, y1, temp1)
    FieldSubtraction(x2, x1, temp2)
    FieldDivision(temp1, temp2, lambda)

    FieldMult(lambda, lambda, temp2)
    FieldSubtraction(temp2, x1, temp2)
    FieldSubtraction(temp2, x2, x3)

    FieldSubtraction(x1, x3, temp1)
    FieldMult(lambda, temp1, temp3)
    FieldSubtraction(temp3, y1, y3)
end
```

Process:

1. Compute `lambda` $= \frac{y_2 - y_1}{x_2 - x_1} \mod p$ using field subtraction for the numerator $(y_2 - y_1)$ and denominator $(x_2 - x_1)$, followed by field division.
2. Compute $x_3 = \texttt{lambda}^2 - x_1 - x_2 \mod p$ using field multiplication for $\texttt{lambda}^2$, then field subtraction to adjust for x_1 and x_2.
3. Compute $y_3 = \texttt{lambda} \cdot (x_1 - x_3) - y_1 \mod p$ using field subtraction for $x_1 - x_3$, followed by field multiplication and another field subtraction for the final result.

Output: The coordinates (x_3, y_3), representing the sum of the two input points (x_1, y_1) and (x_2, y_2) on the elliptic curve.

3.2.2 Point Doubling (dbl)

Performs the doubling of a point $P = (x_1, y_1)$ to compute $2P = (x_3, y_3)$ using the formula mentioned in above section 3.1.

```
void dbl(uint32_t* x1, uint32_t* y1, uint32_t* x3, uint32_t* y3);
```

Pseudocode and the explanation given below:

dbl(uint32_t x1[], uint32_t y1[], uint32_t x4[], uint32_t y4[])

```
begin
    lambda = 0, temp1 = 0, temp2 = 0

    FieldMult(x1, x1, temp1)
    Field_ConstMult(temp1, 3, temp1)
    FieldAddition(temp1, a, temp1)
    FieldAddition(y1, y1, lambda)
    FieldDivision(temp1, lambda, lambda)

    FieldMult(lambda, lambda, temp2)
    FieldAddition(x1, x1, temp1)
    FieldSubtraction(temp2, temp1, x4)

    FieldSubtraction(x1, x4, temp1)
    FieldMult(lambda, temp1, temp2)
    FieldSubtraction(temp2, y1, y4)
end
```

Process:

1. Compute `lambda` $= \frac{3x_1^2+a}{2y_1} \mod p$:
   - Square x_1 and multiply by 3 using `FieldMult` and `Field_ConstMult`, here using `Field_ConstMult` improves efficiency.
   - Add the curve parameter a to compute the numerator.
   - Double y_1 to compute the denominator and perform modular division.
2. Compute $x_4 =$ `lambda`$^2 - 2x_1 \mod p$:
   - Square `lambda`, subtract $2x_1$, and store the result in x_4.
3. Compute $y_4 =$ `lambda`$\cdot (x_1 - x_4) - y_1 \mod p$:
   - Subtract x_4 from x_1, multiply the result by `lambda`, and subtract y_1.

Output: The coordinates (x_4, y_4), representing the doubled point $2(x_1, y_1)$ on the elliptic curve.

3.2.3 Scalar Multiplication

Performs scalar multiplication of a point P by a scalar k (i.e., kP) using the double-and-add method. Double-and-Add algorithm involves iteratively doubling and adding points based on the binary representation of the scalar k. If k has n bits:

$$k = (k_{n-1}k_{n-2}\dots k_0)_2$$

• **Double-and-Add (left-to-right):** The algorithm processes the bits of k from the most significant bit (MSB) to the least significant bit (LSB).

```
void ScalarMult_left2right(uint32_t* xP, uint32_t* yP, uint32_t* scalar, uint32_t*
    xR, uint32_t* yR);
```

Pseudocode and explanation is given below:

ScalarMult_left2right(uint32_t x[], uint32_t y[], uint32_t scalar[], uint32_t xR[], uint32_t yR[])

```
begin
    (xQ,yQ) = (0,0)
    scalarBitLength = BitLength(scalar)

    for i = scalarBitLength - 1 down to 0
        if (xQ != 0 or yQ != 0)
            dbl(xQ, yQ, xQ, yQ)

        flag = (scalar[i / 29] >> (i % 29)) & 1

        if (flag)
            if (xQ == 0 and yQ == 0)
                xQ = x, yQ = y
            else
                add(xQ, yQ, x, y, xQ, yQ)
    end
    xR = xQ, yR = yQ
end
```

Process:

1. Initialize Q as the point at infinity (0, 0).
2. Compute the bit length of the scalar using the `BitLength` function.
3. For each bit of the scalar, starting from the most significant bit:
   - Perform point doubling on Q (i.e., $R = 2R$).
   - If the current bit of the scalar is 1, perform point addition $Q = Q + P$.
   - If Q is the point at infinity, set $Q = P$.
4. After processing all the bits, store the final result in Q, which is the result of scalar multiplication kP.

Output: The coordinates (x_R, y_R), representing the result of scalar multiplication kP on the elliptic curve.

• **Double-and-Add (right-to-left):** In contrast with left-to-right, this algorithm processes the bits of k from the least significant bit (LSB) to the most significant bit (MSB).

```
void ScalarMult_right2left(uint32_t* xP, uint32_t* yP, uint32_t* scalar, uint32_t*
    xR, uint32_t* yR);
```

Pseudocode and explanation is given below:

ScalarMult_right2left(uint32_t x[],uint32_t y[],uint32_t scalar[], uint32_t xR[], uint32_t yR[])

```
begin
    (xQ,yQ) = (0,0)
    (xR, yR) = (x, y)
    scalarBitLength = BitLength(scalar)

    for i = 0 to scalarBitLength - 1
        flag = (scalar[i / 29] >> (i % 29)) & 1

        if flag == 1
            if xQ == 0 and yQ == 0
                xQ = xR and yQ = yR
            else
                add(xQ, yQ, xR, yR, xQ, yQ)
            end if
        end if

        dbl(xR, yR, xR, yR)
    end for
    xR = xQ, yR = yQ
end
```

Process:

1. Initialize Q as the point at infinity (0, 0) and $R = P$.
2. Compute the bit length of the scalar using the `BitLength` function.
3. For each bit of the scalar, starting from the least significant bit:
   - If the current bit is 1, perform $Q = Q + R$ using point addition.
   - If Q is the point at infinity, set $Q = R$.
   - Perform point doubling: $R = 2R$.
4. After processing all the bits, store the final result in Q, which is the result of scalar multiplication kP.

Output: The coordinates (x_R, y_R), representing the result of scalar multiplication kP on the elliptic curve.

3.3 Testing and Verification

Refer to the code provided in the **Code Listing** section for example usage of these functions in the main body of the code. To verify the correctness of the implementation, use the following test vectors for the curve

$$y^2 = x^3 + ax + b \mod p$$

where

$$a = 0\text{x}1 \text{ and } b = 0\text{x}71$$

and

```
p = 0xe92e40ad6f281c8a082afdc49e1372659455bec8ceea043a614c835b7fe9eff5
```

- **Two sample points on Curve-**

  `(x1,y1)` where

x1 = 0x1cdb0c0f208404adbb49e2032a0d43ee4f62ca4c0776cc61bb60adaa1e4cd724

y1 = 0x87112344fb9b053f122f2c7f58b750f07e6ac40ffb5d48c1757c0e599f1ffb8d

(x2,y2) where

x2 = 0xe4e95f48f4d68cf9ec1f627b184e31e7d1d6b7d5d432f1a0ff862d8c13d7060e

y2 = 0x3f66bc52b330667a6b92d644f41ddccdef2534f233f1aba932cda319d6774d54

- **scalar** k = 0x6bac8b56bfbf8751448c650eb8045825411ee77eda0ba9021ffb465936c446a6
- **Expected outputs-**

  addition (x1,y1) + (x2,y2) = (x3,y3) where

  x3 = 0xa0f2bc8cffb991e577a4bcc4a1b8d2f2c41b8254acba27a48bdd8571a4abc0c4

  y3 = 0x072fb278dfe7846bb0886d6faef903e895d979e9fbebb2b173247280d7c170e6

  doubling $(x1,y1)^2$ = (x4,y4) where

  x4 = 0x16c77ce78f5b665b13f1ef21d7076621d951e603098dffee2cb0ea2f5fa26b98

  y4 = 0xd976271985f2184f2e36d4db68dd5fcdf883a63624104a0fca3750eb31a4c296

  scalar multiplication k(x1,y1) = (x5,y5) where

  x5 = 0xc46f7bca9f2b0aced005f4aaba2ff40b8d33aa6a27acf1063442cb34c2be61dc

  y5 = 0xe8538dbda36fc2ec29d8cf29335ce23a19523e06f43750fb43487bd145220f8c

4 Instructions for Running the Code

1. Prerequisites

GCC compiler or any C compiler installed on your device.

2. Compiling and Running

Use the following bash commands to compile and execute:

Listing 1: Compilation Instructions for Prime Field Arithmetic operations

```
# Compile the files
gcc -o main main_FieldArithmetic.c FieldArithmetic.c Exponentiation.c

# Run the executables
./main
```

Listing 2: Compilation Instructions for Elliptic Curve operations

```
# Compile the files
gcc -o main main_ec.c FieldArithmetic.c Exponentiation.c ec_operations.c

# Run the executables
./main
```

5 Conclusion

In conclusion, this project successfully implements elliptic curve operations, including point addition, doubling, and scalar multiplication, using efficient algorithms tailored for cryptographic applications. By utilizing modular arithmetic and optimized techniques such as the left-to-right double-and-add method, the implementation ensures high performance and scalability for large elliptic curve computations. The integration of these operations in the context of cryptography contributes to secure key exchanges and digital signatures, forming the backbone of modern cryptographic systems.

6 Acknowledgement

I would like to express my sincere gratitude to my course instructor, Dr. Sabyasachi Karati, for his invaluable guidance and support throughout this project. The insights and feedback were crucial in shaping the implementation and ensuring its accuracy. I also appre- ciate the encouragement and resources provided, which greatly contributed to the successful completion of this work. Thank you for your dedication and mentorship.

7 Appendix

7.1 Reference Material

1. Wikipedia: Prime Field
2. Hankerson, D., Menezes, A. J., & Vanstone, S. A. (2004). *Guide to Elliptic Curve Cryptography*. Springer-Verlag.
3. Wikipedia: Elliptic Curve.

7.2 Code Listing

For completeness, the codes are provided in attached link: Click here

- This repository includes all the files used in the project, where in this documentation I have provided a thorough explanation of each function. Additionally, instructions for running the code are included.
- The Sagemath code for generating curve parameters and verifying the results is also provided in the repository to ensure accurate implementation and testing of the elliptic curve operations.