Elliptic Curve Cryptography: Implementation and Analysis

SK GOLAM KUDDUS

November 2024

Acknowledgment

I would like to express my heartfelt gratitude to Mr. Sabyasachi Karati for his invaluable guidance and support throughout the semester. His expertise and encouragement have been instrumental in helping me understand and explore this topic deeply.

Contents

1 Hexadecimal String to Big Integer Array Conversion **3**

2 Mathematical Intuition **6**
2.1 Polynomial Representation of Large Integers 6
2.2 Choice of 29-Bit Word Representation 6

3 Type Definitions and Array Structures **6**
3.1 Custom Types 7

4 Precomputed Constants **7**
4.1 Prime Modulus (p) Representation 7
4.2 Barrett Reduction Constant (μ) 7

5 Modular Arithmetic Operations **8**
5.1 Normalize Function 8
5.2 Right-Shift Function (`rshift1`) 8
5.3 Modular Addition Function (`add_29bit`) 9
5.4 Modular Subtraction Function (`sub_29bit`) 10
5.5 Schoolbook Multiplication Function (`mult_29bit`) 11
5.6 Comparison Function (`geq`) 12
5.7 Barrett Reduction Function (`barrett_reduction`) 12
5.8 Modular Multiplication with Barrett Reduction (`modular_multiply`) 14
5.9 Montgomery Ladder Exponentiation(`montgomery_ladder_exponentiation`) 15
5.10 Left-to-Right Modular Exponentiation 17
5.11 Right-to-Left Modular Exponentiation 18

6 Elliptic Curve Basics **20**
6.1 Elliptic Curve Groups and Subgroups 20
6.2 Choice of Parameters a and b 20
6.3 Cofactor and Its Importance 21
6.4 Elliptic Curve Point Structure 21
6.5 Arithmetic on Elliptic Curves 22
6.5.1 Point Addition ($P + Q$) 22
6.5.2 Point Doubling ($2P$) 23
6.6 Modular Inverse: Fermat's Little Theorem 25
6.7 Definition of a Generator 26
6.8 Scalar Multiplication 26

:Abstract:

This document delves into the implementation and underlying intuition behind various modular arithmetic operations essential in cryptographic algorithms. It covers key techniques such as parsing hexadecimal values, modular reduction (including Barrett reduction), and modular exponentiation methods like Montgomery ladder exponentiation (both left-to-right and right-to-left approaches). Additionally, it explores efficient multiplication of large integers represented as arrays. The implementation utilizes 29-bit words within 64-bit unsigned integers to facilitate the efficient handling of 256-bit and larger values, crucial for modern cryptographic applications. The document also highlights the arithmetic operations in elliptic curve groups using a large 256-bit prime, enabling efficient elliptic curve Diffie-Hellman (ECDH) key exchange. These modular arithmetic functions form the backbone of cryptographic protocols, particularly in elliptic-curve cryptography and finite-field arithmetic.

Introduction

We implemented modular arithmetic operations optimized for cryptographic computations, focusing on efficient handling of large integers using a 29-bit word representation within 64-bit integers. The primary features include modular addition, subtraction, multiplication, Barrett reduction, and modular exponentiation using techniques like the Montgomery ladder and binary exponentiation (left-to-right and right-to-left).

1 Hexadecimal String to Big Integer Array Conversion

The conversion of hexadecimal strings into integer arrays is a fundamental task in many cryptographic and numerical applications. This program facilitates the parsing of a 64-character hexadecimal string into a big integer array represented by unsigned 64-bit integers (`u29`). Each segment of the string is carefully processed using bitwise operations and converted into smaller chunks for efficient storage and computation.

Code: Hexadecimal Character to Integer Conversion

```
// Function to convert a hexadecimal character to an integer
u29 hextoint(char hex) {
    if (hex >= '0' && hex <= '9') {
        return hex - '0';
    } else if (hex >= 'a' && hex <= 'f') {
        return hex - 'a' + 10;
    } else if (hex >= 'A' && hex <= 'F') {
        return hex - 'A' + 10;
    }
    printf("Invalid hex char\n");  // Use printf for error messages
    exit(EXIT_FAILURE);
}
```

How It Works

- **Initialization**:
  - The function takes a single character input, `hex`.
- **Character Validation and Conversion**:
  - Checks if `hex` is a valid hexadecimal character in the range:
    - `'0'` to `'9'`: Converted by subtracting `'0'`.
    - `'a'` to `'f'`: Converted by subtracting `'a'` and adding 10.
    - `'A'` to `'F'`: Converted by subtracting `'A'` and adding 10.
  - If `hex` is not in any of these ranges, an error message is printed, and the program exits.
- **Return Result**:
  - The function returns the integer equivalent of the hexadecimal character.

Parsing Hexadecimal into a Big Integer Array

The function `parse_to_int` converts a 64-character hexadecimal string into an array of integers represented as `u29`, where each element contains exactly 29 bits. This representation ensures efficient storage and computation of large numbers in cryptographic and numerical applications.

Code: Parse Hexadecimal String to Big Integer Array

```
// Function to parse a hexadecimal string to a big integer array
void parse_to_int(char* hex, u29* bignum) {
    bignum[0] = 0;
    bignum[1] = hextoint(hex[5]) ^ hextoint(hex[4]) << 4
                ^ hextoint(hex[3]) << 8 ^ hextoint(hex[2]) << 12
                ^ hextoint(hex[1]) << 16 ^ hextoint(hex[0]) << 20;
    bignum[2] = hextoint(hex[13]) >> 3 ^ hextoint(hex[12]) << 1
                ^ hextoint(hex[11]) << 5 ^ hextoint(hex[10]) << 9
                ^ hextoint(hex[9]) << 13 ^ hextoint(hex[8]) << 17
                ^ hextoint(hex[7]) << 21 ^ (hextoint(hex[6]) & 0xf) << 25;
    bignum[3] = hextoint(hex[20]) >> 2 ^ hextoint(hex[19]) << 2
                ^ hextoint(hex[18]) << 6 ^ hextoint(hex[17]) << 10
                ^ hextoint(hex[16]) << 14 ^ hextoint(hex[15]) << 18
                ^ hextoint(hex[14]) << 22 ^ (hextoint(hex[13]) & 0x7) << 26;
    bignum[4] = hextoint(hex[27]) >> 1 ^ hextoint(hex[26]) << 3
                ^ hextoint(hex[25]) << 7 ^ hextoint(hex[24]) << 11
                ^ hextoint(hex[23]) << 15 ^ hextoint(hex[22]) << 19
                ^ hextoint(hex[21]) << 23 ^ (hextoint(hex[20]) & 0x3) << 27;
    bignum[5] = hextoint(hex[34]) ^ hextoint(hex[33]) << 4
                ^ hextoint(hex[32]) << 8 ^ hextoint(hex[31]) << 12
                ^ hextoint(hex[30]) << 16 ^ hextoint(hex[29]) << 20
                ^ hextoint(hex[28]) << 24 ^ (hextoint(hex[27]) & 0x1) << 28;
    bignum[6] = hextoint(hex[42]) >> 3 ^ hextoint(hex[41]) << 1
                ^ hextoint(hex[40]) << 5 ^ hextoint(hex[39]) << 9
                ^ hextoint(hex[38]) << 13 ^ hextoint(hex[37]) << 17
                ^ hextoint(hex[36]) << 21 ^ (hextoint(hex[35]) & 0xf) << 25;
    bignum[7] = hextoint(hex[49]) >> 2 ^ hextoint(hex[48]) << 2
                ^ hextoint(hex[47]) << 6 ^ hextoint(hex[46]) << 10
                ^ hextoint(hex[45]) << 14 ^ hextoint(hex[44]) << 18
                ^ hextoint(hex[43]) << 22 ^ (hextoint(hex[42]) & 0x7) << 26;
    bignum[8] = hextoint(hex[56]) >> 1 ^ hextoint(hex[55]) << 3
                ^ hextoint(hex[54]) << 7 ^ hextoint(hex[53]) << 11
                ^ hextoint(hex[52]) << 15 ^ hextoint(hex[51]) << 19
                ^ hextoint(hex[50]) << 23 ^ (hextoint(hex[49]) & 0x3) << 27;
    bignum[9] = hextoint(hex[63]) ^ hextoint(hex[62]) << 4
                ^ hextoint(hex[61]) << 8 ^ hextoint(hex[60]) << 12
                ^ hextoint(hex[59]) << 16 ^ hextoint(hex[58]) << 20
                ^ hextoint(hex[57]) << 24 ^ (hextoint(hex[56]) & 0x1) << 28;
}
```

How It Works

- **Initialization**:
  - The function initializes the first element of the big integer array (`bignum[0]`) to zero.
- **Hexadecimal Parsing and Conversion**:
  - Iteratively parses segments of the hexadecimal string using the `hextoint` function.
  - Each segment is shifted and combined using bitwise operations (`>>`, `<<`, `^`).

- ☐ Segments are stored into the corresponding indices of the `bignum` array.

☐ **Output Storage**:

- ☐ The parsed integer values are stored in the array `bignum`, representing the hexadecimal string in numerical form.

Example Usage

☐ **Input**: [1]

```
Enter a 64-character hexadecimal string:
e92e40ad6f281c8a082afdc49e1372659455bec8ceea043a614c835b7fe9eff5
```

☐ **Output**:

```
Converted big integer array:
bignum[0] = 0
bignum[1] = 15281728
bignum[2] = 363717891
bignum[3] = 304619691
bignum[4] = 518147849
bignum[5] = 388389189
bignum[6] = 192778653
bignum[7] = 444665496
bignum[8] = 174332635
bignum[9] = 535425013
```

[1]The given prime is 105470615072424007464777057006017113535036866827082468263120632948849084329973, and its hexadecimal representation is `0xe92e40ad6f281c8a082afdc49e1372659455bec8ceea043a614c835b7fe9eff5`.

Modular Arithmetic

2 Mathematical Intuition

2.1 Polynomial Representation of Large Integers

To efficiently handle 256-bit primes in cryptographic computations, we represent large integers as polynomials in a chosen base. A 256-bit integer N is expressed in base 2^{29} as follows:

$$N = c_0 + c_1 \cdot 2^{29} + c_2 \cdot (2^{29})^2 + \cdots + c_{m-1} \cdot (2^{29})^{m-1}$$

where:

- c_i are the coefficients of the polynomial, each a 29-bit integer.
- m is the number of 29-bit words required to represent N, calculated as $m = \lceil \frac{\text{bit length of } N}{29} \rceil$. For a 256-bit prime, $m = \lceil \frac{256}{29} \rceil = 9$.
- 2^{29} is the chosen base, facilitating efficient arithmetic operations.

This polynomial representation decomposes a large integer into smaller segments, making it easier to perform operations such as addition, subtraction, multiplication, and modular reduction.

For instance, a 256-bit prime P is stored as an array of 9 words, represented as:

$$P = \text{word}[0] + \text{word}[1] \cdot 2^{29} + \text{word}[2] \cdot (2^{29})^2 + \cdots + \text{word}[8] \cdot (2^{29})^8$$

2.2 Choice of 29-Bit Word Representation

Why Choose 29-Bit Words? The decision to use 29-bit words for representing large integers is motivated by the following considerations:

- **Optimized Use of 64-Bit Architecture:** Modern processors are optimized for 64-bit operations. With a 29-bit word size, two 29-bit coefficients fit into a single 64-bit register, leaving 6 bits for carry operations. This maximizes the utilization of available bits while avoiding overflow during arithmetic computations.
- **Minimizing Overflow in Arithmetic:** Carry propagation is critical in multi-word arithmetic. Using 29 bits per word ensures that intermediate results during addition or multiplication remain within bounds, reducing the need for frequent carry adjustments and improving computational efficiency.

3 Type Definitions and Array Structures

To efficiently handle large integer arithmetic required in cryptographic algorithms, several custom types and arrays are defined to support modular arithmetic operations.

3.1 Custom Types

- `u29`: A custom-defined type representing a 29-bit segment of a large integer, stored within a 64-bit unsigned integer. This type enables efficient handling of arithmetic operations while leveraging a modern 64-bit architecture.
- **Array Length:** The length of the `WORDS` array is set to 10, even though a 256-bit integer requires only 9 words ($\lceil \frac{256}{29} \rceil = 9$).The extra word provides a buffer for carry propagation during arithmetic operations such as addition, multiplication, or modular reduction. It ensures safe handling of edge cases, especially in cryptographic computations, where intermediate results might temporarily exceed the expected bit length.
- **BASE:** The macro `#define BASE (1UL « 29)` defines the base for our 29-bit word system as 2^{29}. Ensures normalization during operations like addition or multiplication by retaining values within the $[0, 2^{29} - 1]$ range.

4 Precomputed Constants

Throughout the documentation and related code, we use precomputed constants represented as arrays of 10 words, where each word consists of 29 bits.These constants include a 256-bit given prime and a Barrett constant.

4.1 Prime Modulus (p) Representation

The given prime is:

$$p = 105470615072424007464777057006017113535036866827082468263120632948849084329973$$

It is represented as an array of 10 words, each consisting of 29 bits:

$$\text{mod} = \{535425013, 174332635, 444665496, 192778653, 388389189, 518147849, 304619691, 363717891, 15281728, 0\}$$

As

$$p = 535425013 + 174332635 \cdot 2^{29} + 444665496 \cdot (2^{29})^2 + \cdots + 15281728 \cdot (2^{29})^8$$

In our code we consider p as mod(modulus).

4.2 Barrett Reduction Constant (μ)

The Barrett constant μ is precomputed to facilitate efficient modular reduction and is defined as:

$$\mu = \left\lfloor \frac{2^{29 \cdot 18}}{p} \right\rfloor = \left\lfloor \frac{2^{522}}{p} \right\rfloor$$

where p is the 256-bit prime modulus. The computed value of μ is:

$$\mu = 450887704 + 490307913 \cdot 2^{29} + 387807083 \cdot (2^{29})^2 + \cdots + 35 \cdot (2^{29})^9$$

In its compact array form:

$$\text{mu} = \{450887704, 490307913, 387807083, 403879883, 291135210, 307268612, 110539282, 24605042, 70628772, 35\}$$

5 Modular Arithmetic Operations

In this section, we will explore the various modular arithmetic operations implemented in our code. Each operation is accompanied by a detailed explanation of its purpose, functionality, and role within the system. These operations are crucial for ensuring efficient computations in a 29-bit word-based modular arithmetic framework, supporting cryptographic and numerical applications.

5.1 Normalize Function

This function ensures that numbers represented in the 29-bit base system are properly normalized by handling carries across words.

Pseudo-code

Normalize Function

```
def normalize(number, length):
    carry = 0
    for i in range(length):
        value = number[i] + carry
        number[i] = value % BASE
        carry = value // BASE
```

Purpose of the Code

The purpose of the ‘normalize‘ function is to:

- ☐ Ensure all words in the array respect the 29-bit constraint.
- ☐ Propagate any carry values from one word to the next, maintaining consistency in the representation.
- ☐ Prevent invalid results in modular arithmetic operations by keeping numbers within the valid range.

5.2 Right-Shift Function (`rshift1`)

This function performs a right shift by 1 bit on a multi-word number represented using 29-bit words.

Code: Right-Shift Function

```
// Helper function to perform right shift by 1 bit
void rshift1(u29* num, int length) {
    for (int i = 0; i < length - 1; i++) {
        num[i] = (num[i] >> 1) | ((num[i + 1] & 1) << 28);
    }
    num[length - 1] >>= 1;
}
```

Purpose of the Code

The purpose of the ‘rshift1‘ function is to:

- ☐ Perform a logical right shift by 1 bit on a multi-word number.

- ☐ Propagate the least significant bit of one word to the most significant bit of the next word to maintain continuity in the 29-bit representation.
- ☐ Support operations that require bit-level manipulation, such as binary exponentiation or modular division.

How It Works

- ☐ The function processes all but the last word of the number, iterating through each word.
- ☐ For each word:
  - The current word is right-shifted by 1 bit.
  - The least significant bit of the next word is propagated to the most significant bit of the current word using the bitwise OR operation with $(\texttt{num[i + 1]}\&1) \ll 28$.
- ☐ The last word is right-shifted by 1 bit, as it does not have a next word to propagate from.

5.3 Modular Addition Function (`add_29bit`)

This function performs modular addition of two multi-word numbers represented in a 29-bit base system.

Code: Modular Addition Function

```
// Modular addition
void add_29bit(const u29* a, const u29* b, u29* result, int length) {
    uint64_t carry = 0;
    for (int i = 0; i < length; i++) {
        uint64_t sum = (uint64_t)a[i] + b[i] + carry;
        result[i] = sum % BASE;
        carry = sum / BASE;
    }
    normalize(result, length);
}
```

Purpose of the Code

The purpose of the function is to:

- ☐ Compute the sum of two multi-word numbers represented in 29-bit words.
- ☐ Handle carries during addition to ensure correctness across the multi-word representation.
- ☐ Ensure the result respects the 29-bit base system by calling the `normalize` function.

How It Works

- ☐ The function iterates through each word of the input arrays `a` and `b`.
- ☐ For each word:
  - Adds the corresponding words from `a` and `b`, along with any carry from the previous word.
  - Stores the remainder of the sum divided by `BASE` into the `result` array to retain the 29-bit constraint.

- Updates the carry as the integer division of the sum by `BASE`.

☐ After processing all words, the function calls `normalize` to ensure the result conforms to the 29-bit word system.

5.4 Modular Subtraction Function (`sub_29bit`)

This function performs modular subtraction of two multi-word numbers represented in a 29-bit base system.

Code: Modular Subtraction Function

```
void sub_29bit(const u29* a, const u29* b, u29* result, int length) {
    uint64_t carry = 1;  // Start with carry = 1 for the 2's complement of b

    for (int i = 0; i < length; i++) {
        uint64_t diff = (uint64_t)a[i] + (~b[i] & BASE29_MASK) + carry;
        result[i] = diff % BASE;    // Keep only the lowest 29 bits
        carry = diff / BASE;        // Set carry for the next word
    }

    // Check if result is negative (borrow occurred)
    if (carry == 0) { // Borrow indicates a negative result
        // Add the modulus to ensure the result is positive
        add_29bit(result, mod, result, length);
    }
}
```

Purpose of the Code

The purpose of the `sub_29bit` function is to:

☐ Compute the modular subtraction $a - b$ for two multi-word numbers in 29-bit representation.

☐ Handle borrowing during subtraction to ensure correctness across multi-word arithmetic.

☐ Ensure the result is non-negative by adding the modulus if borrowing occurs.

How It Works

☐ The function initializes `carry` to 1 to account for the 2's complement of `b`.

☐ It iterates through each word of the input arrays `a` and `b`.

☐ For each word:

- Computes the difference using the formula:

$$\texttt{diff} = \texttt{a[i]} + \sim \texttt{b[i]} + \texttt{carry}$$

  where $\sim$ `b[i]` is the bitwise complement of `b[i]`.
- Retains only the lowest 29 bits of the result using modulo `BASE`.
- Updates the carry as the integer division of `diff` by `BASE`.

☐ After processing all words:

- If `carry` is 0, it indicates that borrowing occurred, and the result is negative.
- To correct this, the modulus is added to the result using `add_29bit`.

5.5 Schoolbook Multiplication Function (`mult_29bit`)

This function performs schoolbook multiplication of two multi-word numbers represented in a 29-bit base system.

Code: Schoolbook Multiplication Function

```
// Schoolbook multiplication
void mult_29bit(const u29* a, const u29* b, u29* result) {
    memset(result, 0, 2 * WORDS * sizeof(u29)); // Initialize result to zero
    for (int i = 0; i < WORDS; i++) {
        uint64_t carry = 0;
        for (int j = 0; j < WORDS; j++) {
            uint64_t product = (uint64_t)a[i] * b[j] + result[i + j] + carry;
            result[i + j] = product % BASE;
            carry = product / BASE;
        }
        result[i + WORDS] += carry;
    }
    normalize(result, 2 * WORDS);
}
```

Purpose of the Code

The purpose of the `mult_29bit` function is to:

- ☐ Compute the product of two multi-word numbers in a 29-bit base system using the schoolbook multiplication algorithm.
- ☐ Handle carries during multiplication to ensure correctness across multiple words.
- ☐ Normalize the result to conform to the 29-bit representation.

How It Works

- ☐ The function initializes the `result` array to zero using `memset`.
- ☐ Iterates through each word of `a`:
  - Multiplies the current word of `a` with each word of `b`.
  - Adds the product to the corresponding position in the `result` array, along with any carry from the previous calculation.
  - Retains only the lowest 29 bits of the result using modulo `BASE`.
  - Updates the carry as the integer division of the product by `BASE`.
- ☐ After processing all words of `b` for a specific word of `a`, any leftover carry is added to the next word in the `result` array.
- ☐ Calls `normalize` to ensure the final result conforms to the 29-bit representation.

5.6 Comparison Function (geq)

This function checks if one multi-word number (a) is greater than or equal to another multi-word number (b) in a 29-bit base system.

Code: Comparison Function

```
// Function to check if a >= b for multi-word numbers
int geq(const u29* a, const u29* b) {
    for (int i = WORDS - 1; i >= 0; i--) {
        if (a[i] > b[i]) return 1;
        if (a[i] < b[i]) return 0;
    }
    return 1; // a == b
}
```

Purpose of the Code

The purpose of the geq function is to:

- ☐ Determine if a multi-word number a is greater than or equal to another multi-word number b.
- ☐ Support modular arithmetic operations, such as comparisons during reductions or condition checks.

How It Works

- ☐ The function iterates through the words of a and b, starting from the most significant word.
- ☐ For each word:
  - If the current word of a is greater than the corresponding word of b, the function immediately returns 1 (true).
  - If the current word of a is less than the corresponding word of b, the function immediately returns 0 (false).
- ☐ If all corresponding words are equal, the function concludes that a is equal to b and returns 1.

5.7 Barrett Reduction Function (barrett_reduction)

Given two integer numbers z and p, Barrett's reduction algorithm computes the remainder $r = z \mod p$, where $z = qp + r$, in an efficient way. The main idea of Barrett reduction is to find an approximation for the quotient q using elementary operations, and arrive at the final result r by subtractions of the modulo p. Barrett reduction exploits a precomputed quantity

$$\mu = \left\lfloor \frac{\beta^{2k}}{p} \right\rfloor, \quad \beta > 3$$

To reduce the complexity of the modulo reduction, where k denotes the number of words required to represent the 256-bit prime p, the radix β is typically chosen as 2^{29}. For a 256-bit prime p, $k = 9$, as it requires 9 words to represent p in a radix 2^{29}. This choice of radix aligns with the word size of the processor to optimize arithmetic operations by leveraging efficient digit shifts and truncations.

Algorithm 1 Barrett Reduction Alg.

Require: Two positive integer numbers z and m, $\mu = \lfloor \beta^{2k}/m \rfloor$, $\beta > 3$
Ensure: Integer number $r = z \mod m$

1: $q_1 \leftarrow \lfloor z/\beta^{k-1} \rfloor$
2: $q_2 \leftarrow q_1 \mu$
3: $q_3 \leftarrow \lfloor q_2/\beta^{k+1} \rfloor$
4: $r_1 \leftarrow z \bmod \beta^{k+1}$
5: $r_2 \leftarrow q_3 \cdot m \mod \beta^{k+1}$
6: $r' \leftarrow r_1 - r_2$
7: **if** $r' < 0$ **then**
8: $\quad r' \leftarrow r' + \beta^{k+1}$
9: **end if**
10: **while** $r' \geq m$ **do**
11: $\quad r' \leftarrow r' - m$
12: **end while**
13: $r \leftarrow r'$
14: **return** r

Code: Barrett Reduction Function

```
// Barrett reduction
void barrett_reduction(u29* a, u29* mod, u29* mu, u29* result) {
    u29 q1[WORDS] = {0};
    u29 q2[2 * WORDS] = {0};
    u29 q3[WORDS] = {0};
    u29 temp[2 * WORDS] = {0};
    u29 rem[WORDS + 1] = {0};

    // Step 1: Initialize q1 with a[8] to a[17]
    memcpy(q1, a + 8, (WORDS) * sizeof(u29));

    // Step 2: q2 = q1 * mu
    mult_29bit(q1, mu, q2);

    // Step 3: Populate q3 with q2[10] to q2[19]
    memcpy(q3, q2 + WORDS, WORDS * sizeof(u29));

    // Step 4: temp = q3 * mod
    mult_29bit(q3, mod, temp);

    // Step 5: rem = a - temp
    sub_29bit(a, temp, rem, WORDS + 1);

    // Step 6: Adjust remainder to ensure rem < mod
    while (geq(rem, mod)) {
        sub_29bit(rem, mod, rem, WORDS + 1);
    }

    // Store result in output parameter
    memcpy(result, rem, WORDS * sizeof(u29));
}
```

Purpose of the Code

The purpose of the `barrett_reduction` function is to:

- ☐ Efficiently compute $a \mod \texttt{mod}$ for large integers a using precomputed constants.
- ☐ Avoid expensive division operations by leveraging the Barrett constant (`mu`).
- ☐ Provide a modular reduction mechanism optimized for cryptographic and numerical applications.

How It Works

- ☐ The function uses the Barrett reduction method, which involves precomputing a constant `mu` to replace division with multiplication and shifts.
- ☐ Step-by-step explanation:
  - Step 1: Extract the higher-order words of a (from $a[8]$ to $a[17]$) into `q1`.
  - Step 2: Multiply `q1` with `mu` to compute `q2`.
  - Step 3: Extract the higher-order words of `q2` (from $q2[10]$ to $q2[19]$) into `q3`.
  - Step 4: Multiply `q3` with the modulus `mod` to compute the temporary product `temp`.
  - Step 5: Subtract `temp` from a to compute the remainder (`rem`).
  - Step 6: If `rem` is greater than or equal to `mod`, iteratively subtract `mod` until `rem` is less than `mod`.
- ☐ The final remainder is stored in the `result` array.

5.8 Modular Multiplication with Barrett Reduction (`modular_multiply`)

This function performs modular multiplication using the Barrett reduction technique. It computes `result` $= (a \cdot b) \mod \texttt{mod}$ efficiently by reducing the intermediate product using a precomputed Barrett constant μ.

Code: Modular Multiplication with Barrett Reduction

```
// Modular multiplication with Barrett reduction
void modular_multiply(u29* a, u29* b, u29* result, u29* mod, u29* mu) {
    u29 temp[2 * WORDS] = {0}; // Temporary variable to hold multiplication result
    mult_29bit(a, b, temp);
    barrett_reduction(temp, mod, mu, result);
}
```

How It Works

- ☐ Compute the intermediate product `temp` $= a \cdot b$ using the `mult_29bit` function, which supports multi-word multiplication in a 29-bit base system.
- ☐ Reduce the intermediate product modulo `mod` using the `barrett_reduction` function, which efficiently reduces the large product using the precomputed constant μ.
- ☐ Store the final result in the `result` array, ensuring it is in the valid range $[0, \texttt{mod} - 1]$.

5.9 Montgomery Ladder Exponentiation(`montgomery_ladder_exponentiation`)

This function implements the Montgomery Ladder algorithm for modular exponentiation, which is optimized for cryptographic applications. It computes `result` = `base`$^{\texttt{exponent}}$ mod `mod` efficiently using Barrett reduction for modular operations.2 is a generator of $\mathbb{Z}_p^*$, so we can use $g = 2$ as the base.

Algorithm 2 Montgomery Ladder Algorithm

Require: g (base), $k = (k_{\ell-1}, \ldots, k_0)_2$ (binary representation of the exponent)
Ensure: $y = g^k \mod m$

$R_0 \leftarrow 1$
$R_1 \leftarrow g$
for $j = \ell - 1$ **downto** 0 **do**
    if $k_j = 0$ **then**
        $R_1 \leftarrow (R_0 \cdot R_1) \mod m$
        $R_0 \leftarrow (R_0^2) \mod m$
    else
        $R_0 \leftarrow (R_0 \cdot R_1) \mod m$
        $R_1 \leftarrow (R_1^2) \mod m$
    end if
end for
return R_0

Purpose of the Code

The purpose of the `montgomery_ladder_exponentiation` function is to:

- ☐ Compute modular exponentiation `base`$^{\texttt{exponent}}$ mod `mod` in a secure and efficient manner.
- ☐ Use the Montgomery Ladder algorithm to prevent side-channel attacks by ensuring a consistent computational pattern regardless of the bit values of the exponent.
- ☐ Utilize Barrett reduction to optimize modular multiplication operations.

Code: Montgomery Ladder Exponentiation

```
// Montgomery Ladder for Exponentiation with Barrett Reduction
void montgomery_ladder_exponentiation(u29* base, u29* exponent, u29* mod,
u29* mu, u29* result) {
    u29 one[WORDS] = {0};
    one[0] = 1;  // Montgomery representation of 1

    u29 r0[WORDS];  // Initialize R0 = 1 (Montgomery form)
    u29 r1[WORDS];  // Initialize R1 = base (Montgomery form)
    memcpy(r0, one, sizeof(one));
    memcpy(r1, base, sizeof(u29) * WORDS);

    int total_bits = WORDS * 29;

    // Montgomery ladder loop
    for (int j = total_bits - 1; j >= 0; j--) {
        // Extract the j-th bit of the exponent
        u29 bit = (exponent[j / 29] >> (j % 29)) & 1;

        if (bit == 0) {
            modular_multiply(r1, r0, r1, mod, mu);  // R1 = R0 * R1
            modular_multiply(r0, r0, r0, mod, mu);  // R0 = (R0)^2
        } else {
            modular_multiply(r0, r1, r0, mod, mu);  // R0 = R0 * R1
            modular_multiply(r1, r1, r1, mod, mu);  // R1 = (R1)^2
        }
    }

    // The result is stored in R0
    memcpy(result, r0, sizeof(u29) * WORDS);
}
```

How It Works

- ☐ Initialization:
  - Set $R_0 = 1$ (identity element for multiplication).
  - Set $R_1 =$ `base` (the initial value of the base).
- ☐ Bitwise Exponentiation:
  - Loop through all bits of the exponent, starting from the most significant bit.
  - For each bit:
    - If the bit is 0: Update $R_1 = R_0 \cdot R_1 \mod$ `mod` and $R_0 = R_0^2 \mod$ `mod`.
    - If the bit is 1: Update $R_0 = R_0 \cdot R_1 \mod$ `mod` and $R_1 = R_1^2 \mod$ `mod`.
- ☐ Final Result:
  - After the loop, R_0 contains the final result $\texttt{base}^{\texttt{exponent}} \mod \texttt{mod}$.

5.10 Left-to-Right Modular Exponentiation

The Left-to-Right Method for modular exponentiation is a binary approach where the bits of the exponent are processed starting from the most significant bit (MSB) to the least significant bit (LSB). . It computes $\texttt{result} = \texttt{base}^{\texttt{exponent}} \mod \texttt{mod}$, leveraging modular multiplication with Barrett reduction for efficiency.

Algorithm 3 Left-to-Right Binary Exponentiation

Require: Base g, exponent $k = (k_{\ell-1}, \ldots, k_0)_2$
Ensure: Result $y = g^k \mod m$
1: Initialize $R_0 \leftarrow 1$, $R_1 \leftarrow g$
2: **for** $j = \ell - 1$ **downto** 0 **do**
3: $\quad R_0 \leftarrow (R_0)^2$
4: $\quad$ **if** $k_j = 1$ **then**
5: $\quad\quad R_0 \leftarrow R_0 \cdot R_1$
6: $\quad$ **end if**
7: **end for**
8: **return** R_0

Code: Left-to-Right Modular Exponentiation Function

```
// Left-to-right modular exponentiation
void left_to_right_binary_exponentiation(u29* base, u29* exponent, u29* mod,
u29* mu, u29* result) {
    u29 r0[WORDS];  // R0 = 1
    u29 r1[WORDS];  // R1 = base
    u29 one[WORDS] = {0};
    one[0] = 1;

    memcpy(r0, one, sizeof(one));          // Initialize R0 = 1
    memcpy(r1, base, sizeof(u29) * WORDS); // Initialize R1 = base

    int total_bits = WORDS * 29;

    for (int j = total_bits - 1; j >= 0; j--) {
        modular_multiply(r0, r0, r0, mod, mu); // R0 = (R0)^2

        // Check the j-th bit of the exponent
        u29 bit = (exponent[j / 29] >> (j % 29)) & 1;
        if (bit == 1) {
            modular_multiply(r0, r1, r0, mod, mu); // R0 = R0 * R1
        }
    }

    memcpy(result, r0, sizeof(u29) * WORDS); // Return R0
}
```

How It Works

- ☐ **Initialization**:
  - Set $R_0 = 1$, the identity element for multiplication.
  - Set $R_1 =$ `base`, the initial value of the base.
- ☐ **Bitwise Processing**:
  - Calculate the total number of bits in the exponent as `WORDS` $\times 29$.
  - Iterate through the bits of the exponent from the most significant to the least significant (left-to-right approach).
  - For each bit:
    - Square R_0: $R_0 = R_0^2 \mod$ `mod`.
    - If the current bit is 1, multiply R_0 by R_1: $R_0 = R_0 \cdot R_1 \mod$ `mod`.
- ☐ **Finalization**:
  - After processing all bits of the exponent, R_0 contains the final result.
  - Copy R_0 to the `result` array.

5.11 Right-to-Left Modular Exponentiation

This function implements the right-to-left binary method for modular exponentiation. It efficiently computes $\text{base}^{\text{exponent}} \mod \text{mod}$ by processing the bits of the exponent starting from the least significant bit (LSB) to the most significant bit (MSB).

Algorithm 4 Right-to-Left Binary Exponentiation

Require: Base g, exponent $k = (k_0, \ldots, k_{\ell-1})_2$, modulus m
Ensure: Result $y = g^k \mod m$

```
Initialize R_0 ← 1, R_1 ← g
for j = 0 to ℓ − 1 do
    if k_j = 1 then
        R_0 ← (R_0 · R_1) mod m
    end if
    R_1 ← (R_1 · R_1) mod m
end for
return R_0
```

Code: Right-to-Left Binary Exponentiation Function

```
// Right-to-left modular exponentiation
void right_to_left_binary_exponentiation(u29* base, u29* exponent, u29* mod,
u29* mu, u29* result) {
    u29 r0[WORDS];  // R0 = 1
    u29 r1[WORDS];  // R1 = base
    u29 one[WORDS] = {0};
    one[0] = 1;

    memcpy(r0, one, sizeof(one));          // Initialize R0 = 1
    memcpy(r1, base, sizeof(u29) * WORDS); // Initialize R1 = base

    int total_bits = WORDS * 29;

    for (int j = 0; j < total_bits; j++) {
        // Check the j-th bit of the exponent
        u29 bit = (exponent[j / 29] >> (j % 29)) & 1;

        if (bit == 1) {
            modular_multiply(r0, r1, r0, mod, mu); // R0 = R0 * R1
        }
        modular_multiply(r1, r1, r1, mod, mu); // R1 = (R1)^2
    }

    memcpy(result, r0, sizeof(u29) * WORDS); // Return R0
}
```

How It Works

- ☐ **Initialization**:
  - ☐ R_0: Set to 1 (identity for multiplication).
  - ☐ R_1: Set to the base value.
- ☐ **Processing the Bits of the Exponent**:
  - ☐ Iterate through each bit of the exponent, starting from the least significant bit.
  - ☐ For each bit:
    - If the bit is 1, multiply R_0 by R_1 modulo mod.
    - Square R_1 modulo mod to prepare for the next bit.
- ☐ **Return Result**:
  - ☐ The result of $\text{base}^{\text{exponent}}$ mod mod is stored in R_0.

Elliptic curve operations

6 Elliptic Curve Basics

An **elliptic curve** over a finite field $\mathbb{F}_p$ (where p is a prime) is defined by an equation of the form:

$$y^2 = x^3 + ax + b \pmod{p} \tag{1}$$

where a and b are constants in $\mathbb{F}_p$ such that the curve is non-singular. This non-singularity condition is essential to avoid cusps or self-intersections on the curve, which would make it unsuitable for cryptographic applications. The non-singularity condition is mathematically defined as:

$$4a^3 + 27b^2 \neq 0 \pmod{p}. \tag{2}$$

The set of points (x, y) satisfying the elliptic curve equation, along with a special point at infinity $\mathcal{O}$, form an **abelian group** under a defined addition operation.

6.1 Elliptic Curve Groups and Subgroups

Elliptic curves are fundamental in cryptography because they form a cyclic group suitable for secure cryptographic operations. The group consists of:

- The **points** $P = (x, y)$ on the curve that satisfy the curve equation.
- The **point at infinity** $\mathcal{O}$, which acts as the identity element.

Each elliptic curve over $\mathbb{F}_p$ has a finite number of points, known as the **order** of the elliptic curve, denoted by $\#E(\mathbb{F}_p)$. The order can be computed as the total count of all points on the curve, including $\mathcal{O}$.

A **subgroup** of an elliptic curve group is a subset of points that form a group under the same addition operation. A cyclic subgroup can be generated by repeatedly adding a single point G, known as the **generator point**, to itself.

6.2 Choice of Parameters a and b

Choosing appropriate values for a and b is crucial to ensure security:

- The values of a and b must satisfy the non-singularity condition $4a^3 + 27b^2 \neq 0 \pmod{p}$.
- In cryptographic applications, a and b are often chosen from well-established and standardized curves to guarantee security. For example, popular curves like secp256k1 use $a = 0$ and $b = 7$, while secp256r1 uses specific values for a and b to ensure robust cryptographic properties.

Randomly chosen a and b values may result in a weak curve. Therefore, standardized curves are commonly recommended in cryptographic applications. In our computation we use $a = 0$ and $b = 7$.

6.3 Cofactor and Its Importance

The **cofactor** h is an integer that represents the ratio of the elliptic curve's order to the order of the subgroup generated by a point G. Mathematically, it is defined as:

$$h = \frac{\#E(\mathbb{F}_p)}{\text{Order of } G}. \tag{3}$$

A cofactor of $h = 1$ is ideal, as it implies that the point G generates the entire group of the elliptic curve, maximizing the security of cryptographic operations. Higher values of h may expose the curve to security vulnerabilities such as small subgroup attacks, so secure elliptic curves typically have small cofactors, with $h = 1$ or $h = 2$ being common.

6.4 Elliptic Curve Point Structure

In elliptic curve cryptography (ECC), a point on the elliptic curve is represented as a pair of coordinates (x, y) over a finite field, along with an indicator to determine whether the point is at infinity. The following structure defines an elliptic curve point:

Code: Elliptic Curve Point Structure

```
// Define the elliptic curve point structure
typedef struct {
    u29 x[WORDS];          // x-coordinate of the point
    u29 y[WORDS];          // y-coordinate of the point
    int is_infinity;       // 1 if the point is at infinity, 0 otherwise
} ECPoint;
```

Explanation

The structure `ECPoint` consists of the following components:

- `x[WORDS]`: Represents the x-coordinate of the elliptic curve point. The coordinate is stored as an array of 29-bit words (`u29`).
- `y[WORDS]`: Represents the y-coordinate of the elliptic curve point. Similar to the x-coordinate, it is stored as an array of 29-bit words (`u29`).
- `is_infinity`: A flag that indicates whether the point is the point at infinity:
  - `1`: The point is at infinity, representing the identity element in the elliptic curve group.
  - `0`: The point is a valid finite point on the elliptic curve.

Use Case

The `ECPoint` structure is used in various elliptic curve operations, such as:

- **Point Addition**: Adding two points on the elliptic curve.
- **Point Doubling**: Doubling a single point.
- **Scalar Multiplication**: Multiplying a point by a scalar to compute a new point.

This representation ensures flexibility and efficiency in handling large numbers and elliptic curve computations within cryptographic algorithms.

6.5 Arithmetic on Elliptic Curves

The group operation on elliptic curves is defined as **point addition**. Given two points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ on the curve, the addition $R = P + Q = (x_3, y_3)$ is defined by the following rules:

6.5.1 Point Addition ($P + Q$)

For $P \neq Q$, the addition of two distinct points P and Q is given by:

$$\begin{aligned} m &= \frac{y_2 - y_1}{x_2 - x_1} \pmod{p} \quad \text{(slope)} \\ x_3 &= m^2 - x_1 - x_2 \pmod{p} \\ y_3 &= m(x_1 - x_3) - y_1 \pmod{p} \end{aligned}$$

Special cases include:

- If $P = \mathcal{O}$ (point at infinity), then $P + Q = Q$.
- If $Q = \mathcal{O}$, then $P + Q = P$.
- If $P = -Q$, where $-Q = (x_2, -y_2)$, then $P + Q = \mathcal{O}$.

How It Works

☐ **Initialization**:

- ☐ Check if either point is the point at infinity. If so, return the other point.

☐ **Compute Slope (λ)**:

- ☐ Calculate the numerator: $y_2 - y_1$
- ☐ Calculate the denominator: $x_2 - x_1$
- ☐ Compute the modular inverse of the denominator.
- ☐ Multiply numerator by the modular inverse to compute λ modulo m.

☐ **Compute Resultant Point (x_3, y_3)**:

- ☐ Compute $x_3 \leftarrow \lambda^2 - x_1 - x_2 \mod m$
- ☐ Compute $y_3 \leftarrow \lambda \cdot (x_1 - x_3) - y_1 \mod m$

☐ **Return Result**:

- ☐ Store x_3 and y_3 in the result and set it as a valid elliptic curve point.

Code: Elliptic Curve Point Addition

```
// Elliptic curve point addition
void ecc_add(ECPoint* P, ECPoint* Q, ECPoint* result, u29* mod, u29* mu) {
    if (P->is_infinity) {
        memcpy(result, Q, sizeof(ECPoint));
        return;
    }
    if (Q->is_infinity) {
        memcpy(result, P, sizeof(ECPoint));
        return;
    }

    u29 lambda[WORDS] = {0};
    u29 numerator[WORDS] = {0};
    u29 denominator[WORDS] = {0};
    u29 denominator_inv[WORDS] = {0};

    // lambda = (y2 - y1) / (x2 - x1)
    sub_29bit(Q->y, P->y, numerator, WORDS);
    sub_29bit(Q->x, P->x, denominator, WORDS);
    mod_inverse(denominator, mod, mu, denominator_inv);
    modular_multiply(numerator, denominator_inv, lambda, mod, mu);

    // x3 = lambda^2 - x1 - x2
    u29 lambda_squared[WORDS] = {0};
    modular_multiply(lambda, lambda, lambda_squared, mod, mu);

    sub_29bit(lambda_squared, P->x, result->x, WORDS);
    sub_29bit(result->x, Q->x, result->x, WORDS);
    normalize(result->x, WORDS);

    // y3 = lambda * (x1 - x3) - y1
    u29 temp[WORDS] = {0};
    sub_29bit(P->x, result->x, temp, WORDS);
    modular_multiply(lambda, temp, result->y, mod, mu);
    sub_29bit(result->y, P->y, result->y, WORDS);

    result->is_infinity = 0;
}
```

6.5.2 Point Doubling ($2P$)

When $P = Q$, the addition $P + P = 2P$ is known as **point doubling**. The formulas for point doubling are:

$$\begin{aligned} m &= \frac{3x_1^2 + a}{2y_1} \pmod{p} \quad \text{(slope)} \\ x_3 &= m^2 - 2x_1 \pmod{p} \\ y_3 &= m(x_1 - x_3) - y_1 \pmod{p} \end{aligned}$$

Point doubling is essential for efficient elliptic curve arithmetic, as it is a core part of scalar multiplication.

Code: Elliptic Curve Point Addition

```
// Elliptic curve point doubling
void ecc_double(ECPoint* P, ECPoint* result,  u29* mod,  u29* mu) {
    if (P->is_infinity || P->y == 0) {
        result->is_infinity = 1;
        return;
    }

    u29 lambda[WORDS] = {0};
    u29 numerator[WORDS] = {0};
    u29 denominator[WORDS] = {0};
    u29 temp1[WORDS] = {0};
    u29 temp2[WORDS]={0};
    u29 denominator_inv[WORDS] = {0};

    // lambda = (3 * x1^2 + a) / (2 * y1)
    modular_multiply(P->x, P->x, temp1, mod, mu); // x1^2
    add_29bit(temp1, temp1, temp2, WORDS); // 2 * x1^2
    add_29bit(temp2, temp1, numerator, WORDS);      // 3 * x1^2 (a = 0)
    ////// numerator = 3 * x1^2 + a(this line of code is useful when a non zero)///
    //add_29bit(numerator, curve_param_a, numerator, WORDS);

    add_29bit(P->y, P->y, denominator, WORDS);        // 2 * y1
    mod_inverse(denominator, mod, mu, denominator_inv);
    modular_multiply(numerator, denominator_inv, lambda, mod, mu);

    // x3 = lambda^2 - 2 * x1
    u29 lambda_squared[WORDS] = {0};
    modular_multiply(lambda, lambda, lambda_squared, mod, mu);

    sub_29bit(lambda_squared, P->x, result->x, WORDS);
    sub_29bit(result->x, P->x, result->x, WORDS);

    // y3 = lambda * (x1 - x3) - y1
    u29 temp[WORDS] = {0};
    sub_29bit(P->x, result->x, temp, WORDS);
    modular_multiply(lambda, temp, result->y, mod, mu);
    sub_29bit(result->y, P->y, result->y, WORDS);

    result->is_infinity = 0;
}
```

How It Works

- ☐ Check if P is at infinity or if $y = 0$. If true, the result is set to the point at infinity.
- ☐ Compute the slope λ using the formula:

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

- For curves with $a = 0$, the numerator simplifies to $3x_1^2$.

☐ Compute x_3 using:

$$x_3 = \lambda^2 - 2x_1 \pmod{p}$$

☐ Compute y_3 using:

$$y_3 = \lambda(x_1 - x_3) - y_1 \pmod{p}$$

Notes

- If $a \neq 0$, uncomment the line:

```
add_29bit(numerator, curve_param_a, numerator, WORDS);
```

- Handles edge cases such as:
  - $P = \infty$: The result is ∞.
  - $y = 0$: The tangent at P is vertical, so $2P = \infty$.

6.6 Modular Inverse: Fermat's Little Theorem

The **Modular Inverse** computes $x^{-1} \mod p$ using Fermat's Little Theorem:

$$x^{-1} \equiv x^{p-2} \pmod{p},$$

valid when p is a prime number. This method leverages modular exponentiation with the exponent $p - 2$.

Code: Modular Inverse

```
// Modular inverse using Fermat's Little Theorem
void mod_inverse(u29* x, u29* p, u29* mu, u29* result) {
    u29 exponent[WORDS] = {0};
    memcpy(exponent, p, WORDS * sizeof(u29)); // Copy modulus p to exponent
    exponent[0] -= 2; // Compute p-2 for Fermat's theorem

    // Perform modular exponentiation using Montgomery ladder
    montgomery_ladder_exponentiation(x, exponent, p, mu, result);
}
```

How It Works

☐ Compute $p - 2$ to determine the exponent for Fermat's theorem.

☐ Use modular exponentiation to calculate $x^{p-2} \mod p$.

☐ The result is $x^{-1} \mod p$, the modular inverse of x.

Notes

- p must be a prime number for Fermat's theorem to be applicable.
- The 'montgomery_ladder_exponentiation' function is used for efficient and secure modular exponentiation.
- Ensure $x \neq 0 \mod p$; otherwise, the modular inverse does not exist.

6.7 Definition of a Generator

A generator G of an elliptic curve group satisfies:

$$\mathcal{E}(F_p) = \{\infty, G, 2G, 3G, \ldots, (n-1)G\},$$

where n is the **order of the group**, and:

$$nG = 0,$$

where 0 is the point at infinity.

- n is the smallest positive integer such that $nG = 0$, and it is called the **order of** G.
- If G is a generator, the subgroup generated by G contains n distinct points, including ∞.
- The order n divides the total number of points on the elliptic curve (known as the **curve order**), as defined by Hasse's theorem.

Generated Generator Point

The generator point G for the elliptic curve is:

$$G = (x, y),$$

where:

$$x = 95671132857509526213231496207533756774784754527969429833795469349725490669018$$

$$y = 29910388395202366146648456229462384962407762819558613063906828192588010312938$$

The 29-bit representation of the coordinates is:

$$x = [511206874, 254855740, 285194860, 443465366, 268068044, 451848614, 325285346, 118243884, 13861873, 0]$$

$$y = [473912554, 460180803, 198869900, 64737929, 437536564, 372033891, 274885473, 29463607, 4333742, 0]$$

Key Properties of the Generator

- The generator G is used to form a cyclic subgroup of the elliptic curve group.
- Every point in the subgroup is of the form kG, where k is an integer $0 \leq k < n$.
- The security of cryptographic schemes such as ECDH and ECDSA relies on the generator G and the difficulty of solving the discrete logarithm problem on elliptic curves.

6.8 Scalar Multiplication

Scalar multiplication is a fundamental operation in elliptic curve cryptography (ECC), where a scalar k is multiplied by a point P on the elliptic curve to compute $R = k \cdot P$. Efficient implementation of this operation is crucial for the performance of ECC-based systems. Two common methods to perform scalar multiplication are the **Left-to-Right Binary Method** and the **Right-to-Left Binary Method**.

Algorithm 5 Left-to-Right Binary Method for Point Multiplication

Require: Scalar $k = (k_{t-1}, \ldots, k_1, k_0)_2$, Point $P \in E(\mathbb{F}_q)$
Ensure: $Q = kP$
1: $Q \leftarrow \infty$
2: **for** $i \leftarrow t-1$ to 0 **do**
3: $\quad Q \leftarrow 2Q$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \leftarrow Q + P$
6: $\quad$ **end if**
7: **end for**
8: **return** Q

Left-to-Right Binary Method :

The **Left-to-Right Binary Method** processes the scalar k starting from the most significant bit (MSB) to the least significant bit (LSB). At each step, the intermediate result is *doubled* to account for the positional weight of the current bit. If the bit is 1, the point P is added to the intermediate result.

Code: Scalar Multiplication (Left-to-Right Binary Method)

```
// Scalar multiplication (left-to-right binary method)
void ecc_scalar_mult_l2r(u29* scalar, ECPoint* P, ECPoint* result,
u29* mod, u29* mu) {
    ECPoint Q = {0}; // Initialize Q as the point at infinity
    Q.is_infinity = 1;
    // Process scalar bits from MSB to LSB
    for (int i = WORDS * 29 - 1; i >= 0; i--) {
        // Debug: Show current bit and Q before doubling
        int bit = (scalar[i / 29] >> (i % 29)) & 1;
        // Double Q
        ECPoint temp_double = {0};
        ecc_double(&Q, &temp_double, mod, mu);
        memcpy(&Q, &temp_double, sizeof(ECPoint));

        // If the current scalar bit is 1, add P to Q
        if (bit) {
            ECPoint temp_add = {0};
            ecc_add(&Q, P, &temp_add, mod, mu);
            memcpy(&Q, &temp_add, sizeof(ECPoint));
        }
    }

    // Copy the final result to output
    memcpy(result, &Q, sizeof(ECPoint));
}
```

How It Works

- ☐ Process scalar bits from the most significant bit (MSB) to the least significant bit (LSB).
- ☐ Initialize the intermediate result Q as the point at infinity.

☐ At each iteration:

- Double the intermediate result Q.
- If the current scalar bit is 1, add the point P to Q.

☐ The resultant point after scalar multiplication is stored in Q.

☐ Finally, copy the value of Q to the result.

Right-to-Left Binary Method :

The **Right-to-Left Binary Method** processes the scalar k starting from the least significant bit (LSB) to the most significant bit (MSB). It maintains two intermediate points:

- R_0: Accumulates the result, initialized as the point at infinity.
- R_1: Holds the current power of P, initialized to P.

For each bit of k:

1. If the bit is 1, R_1 is added to R_0.
2. R_1 is doubled unconditionally to prepare for the next bit.

Algorithm 6 Right-to-Left Binary Method for Point Multiplication

Require: Scalar $k = (k_{t-1}, \ldots, k_1, k_0)_2$, Point $P \in E(\mathbb{F}_q)$
Ensure: $Q = kP$

```
Q ← ∞
for i ← 0 to t − 1 do
    if k_i = 1 then
        Q ← Q + P
    end if
    P ← 2P
end for
return Q
```

Code: Scalar Multiplication (Right-to-Left Binary Method)

```
// Scalar multiplication (right-to-left binary method)
void ecc_scalar_mult_r2l(u29* scalar, ECPoint* P, ECPoint* result,
u29* mod, u29* mu) {
    ECPoint R0 = {0}; // R0 starts as point at infinity
    R0.is_infinity = 1;

    ECPoint R1 = {0}; // R1 starts as P
    memcpy(&R1, P, sizeof(ECPoint));

    // Process scalar bits from LSB to MSB
    for (int i = 0; i < WORDS * 29; i++) {
        if ((scalar[i / 29] >> (i % 29)) & 1) { // If the current bit is 1
            ECPoint temp = {0};
            ecc_add(&R0, &R1, &temp, mod, mu); // Add R1 to R0, store in temp
            memcpy(&R0, &temp, sizeof(ECPoint)); // Update R0 with temp
        }

        ECPoint temp = {0};
        ecc_double(&R1, &temp, mod, mu); // Double R1, store in temp
        memcpy(&R1, &temp, sizeof(ECPoint)); // Update R1 with temp
    }

    memcpy(result, &R0, sizeof(ECPoint)); // Copy final result
}
```

How It Works

☐ Process scalar bits from the least significant bit (LSB) to the most significant bit (MSB).

☐ Initialize two intermediate points:

- R_0: Starts as the point at infinity and accumulates the result.
- R_1: Starts as P and holds the current multiple of P.

☐ At each iteration:

- If the current scalar bit is 1, add R_1 to R_0.
- Always double R_1 for the next bit.

☐ The final result is stored in R_0.

Conclusion

Elliptic curves provide a secure and efficient framework for modern cryptography. By selecting appropriate parameters a, b, and a generator point G, and by ensuring a minimal cofactor, elliptic curves enable secure protocols such as **Elliptic Curve Diffie-Hellman Key Exchange (ECDH)**.

ECDH is achieved by allowing two parties, each with their private scalar, to compute a shared secret using their respective scalar multiplications on the generator point G. The shared secret is derived by combining the private scalar of one party with the public point of the other, leveraging the mathematical properties of elliptic curves. This process ensures that the shared secret remains secure even if an eavesdropper has access to the public parameters, thanks to the hardness of the *Elliptic Curve Discrete Logarithm Problem (ECDLP)*.

The operations of point addition and doubling form the backbone of elliptic curve arithmetic, making elliptic curve cryptography a powerful tool for secure communication. Efficient and secure implementations of these operations require careful memory management. For instance, `memcpy` is widely used to copy intermediate results safely during cryptographic operations. It ensures that data integrity is preserved without unintended overwrites or leaks during point calculations. However, it is essential to use `memcpy` cautiously in cryptographic contexts to avoid vulnerabilities such as side-channel attacks. Proper use of temporary variables and constant-time memory operations further enhances the security of elliptic curve implementations.

Elliptic curve cryptography thus combines strong theoretical guarantees with practical efficiency, making it a cornerstone of modern secure communication systems.

References

[1] Darrel Hankerson, Alfred Menezes, Scott Vanstone, *Guide to Elliptic Curve Cryptography*, Springer.

[2] Malek Safieh, Andreas Furch, Fabrizio De Santis, "An Efficient Barrett Reduction Algorithm for Gaussian Integer Moduli".