Implementation of Diffie-Hellman Key Exchange Protocol

Bibek Giri
Roll No.: CRS2305

December 2, 2024

Contents

Chapter 1

Introduction

Description:

The **Diffie-Hellman (DH) key exchange** is a pioneering cryptographic protocol that allows two parties to securely generate a shared secret key over an insecure public channel. It was introduced in 1976 by Whitfield Diffie and Martin Hellman, with conceptual contributions from Ralph Merkle, making it one of the earliest implementations of public-key cryptography. This shared key can then be used for symmetric encryption, securing subsequent communications.

How It Works:

Initial Setup

Both parties agree on a large prime number p and a base g, which are public parameters.

Private Key Selection

Each party selects a private key (a for Alice, b for Bob) that remains secret.

Public Key Computation

- Alice computes $A = g^a \mod p$ and sends A to Bob.
- Bob computes $B = g^b \mod p$ and sends B to Alice.

Shared Secret Calculation

- Alice computes $S = B^a \mod p$.
- Bob computes $S = A^b \mod p$.

Both derive the same shared secret $S = g^{ab} \mod p$.

Applications:

- **Internet Security:** DH is used in protocols like **TLS** (Transport Layer Security) for secure web communications.
- **VPNs and Messaging:** It's used in securing **Virtual Private Networks (VPNs)** and **end-to-end encrypted messaging systems**.
- **Forward Secrecy:** DH supports ephemeral key exchanges (DHE/EDH) that enhance security by ensuring that past communications cannot be decrypted even if future keys are compromised.

Chapter 2

Field Arithmetic

2.1 Key Notes:

The whole implementation process is based over a finite field $\mathbb{Z}_p$, p being a prime of 256 bit long. Let's assume **PRM** = 1054706150724240074647770570060171135350368668270824682 63120632948849084329973 and

$$\langle 2 \rangle = \mathbb{Z}_p^*$$

Array of length 9, having each element of 64 bits has been considered to store a 256 bit number. Each element of such array contains 29 bits except the last one that contains 24 bits. A 256-bit unsigned integer a can be represented as a polynomial with coefficients $a_0, a_1, \ldots, a_8$, where:

$$a = a_0 + a_1\theta + a_2\theta^2 + \cdots + a_8\theta^8, \quad \theta = 2^{29}$$

Here, a_i are the elements of the array representing a in base θ. For example,

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

2.2 Subtraction:

Goal of the function:

The goal of the Subtraction function is to perform chunk-based subtraction of two large integers, represented as arrays of fixed-width (29-bit) segments. It handles borrowing management during the subtraction process, ensuring that when a chunk results in a negative difference, the necessary adjustments are made by borrowing from the next chunk. The function returns the result of the subtraction as a new array, with each element constrained to fit within 29 bits. This approach allows efficient subtraction of large integers, even beyond typical 64-bit limits.

Steps of the Algorithm

1. Memory Allocation for the Result:

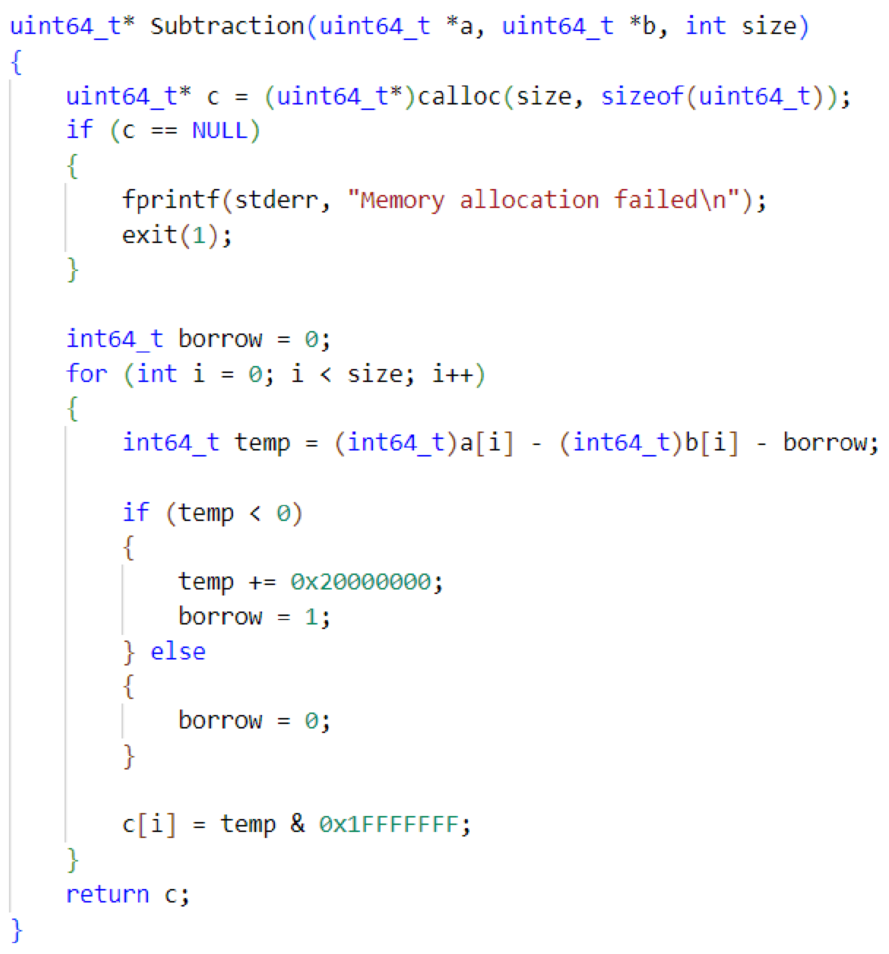
- Allocate a new array c of the same size as the input arrays a and b to store the subtraction result.
- Initialize all elements of c to zero.
- If memory allocation fails, terminate the process with an error.

2. Initialize Borrow:

- Set the borrow variable to 0. This variable will track whether a borrow occurs during subtraction.

3. Iterative Subtraction by Chunk:

- Loop through each index i of the input arrays (from 0 to size $-$ 1):



```c
uint64_t* Subtraction(uint64_t *a, uint64_t *b, int size)
{
    uint64_t* c = (uint64_t*)calloc(size, sizeof(uint64_t));
    if (c == NULL)
    {
        fprintf(stderr, "Memory allocation failed\n");
        exit(1);
    }

    int64_t borrow = 0;
    for (int i = 0; i < size; i++)
    {
        int64_t temp = (int64_t)a[i] - (int64_t)b[i] - borrow;

        if (temp < 0)
        {
            temp += 0x20000000;
            borrow = 1;
        } else
        {
            borrow = 0;
        }

        c[i] = temp & 0x1FFFFFFF;
    }
    return c;
}
```

Figure 2.1: Snapshot of the Subtraction Algorithm

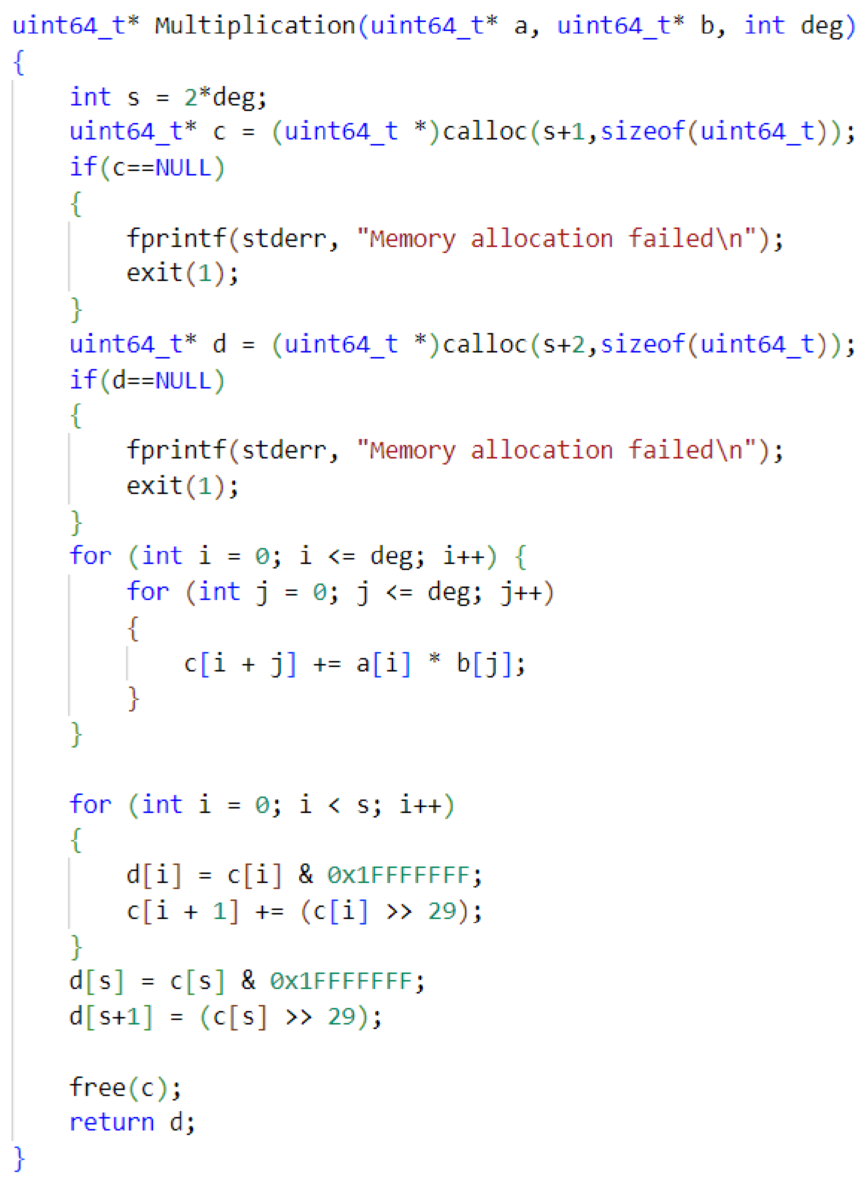
- **Compute Temporary Result:**
  - Subtract the i-th element of b from the i-th element of a, considering the borrow from the previous chunk.
  - Use signed arithmetic to handle potential negative results.
- **Check for Negative Result:**
  - If the temporary result is negative:
    - Add 2^{29} (since chunks are 29 bits wide) to make the result positive.
    - Set borrow to 1 for the next iteration.
  - Otherwise:
    - Set borrow to 0 (no borrow needed for the next chunk).
- **Mask the Result:**
  - Apply a bitwise mask of $2^{29} - 1$ to ensure the result fits within 29 bits.
  - Store the masked result in the corresponding position in the result array c.

4. Return the Result:
After processing all chunks, return the result array c.

2.3 Multiplication:

Goal of the function:

The goal of the provided multiplication function is to compute the product of two large integers, represented as arrays of 29-bit chunks, using polynomial-like multiplication. The function performs pairwise multiplication of corresponding elements from the two arrays, handles carries during the intermediate results, and stores the final product in a new array. The result is also represented as an array of 29-bit chunks.



```c
uint64_t* Multiplication(uint64_t* a, uint64_t* b, int deg)
{
    int s = 2*deg;
    uint64_t* c = (uint64_t *)calloc(s+1,sizeof(uint64_t));
    if(c==NULL)
    {
        fprintf(stderr, "Memory allocation failed\n");
        exit(1);
    }
    uint64_t* d = (uint64_t *)calloc(s+2,sizeof(uint64_t));
    if(d==NULL)
    {
        fprintf(stderr, "Memory allocation failed\n");
        exit(1);
    }
    for (int i = 0; i <= deg; i++) {
        for (int j = 0; j <= deg; j++)
        {
            c[i + j] += a[i] * b[j];
        }
    }

    for (int i = 0; i < s; i++)
    {
        d[i] = c[i] & 0x1FFFFFFF;
        c[i + 1] += (c[i] >> 29);
    }
    d[s] = c[s] & 0x1FFFFFFF;
    d[s+1] = (c[s] >> 29);

    free(c);
    return d;
}
```

Figure 2.2: Snapshot of the Multiplication Algorithm

Steps of the Algorithm

Initialization

Given two arrays a and b, each of size $\deg + 1$ (where deg is the degree of the polynomial representation of the numbers), the algorithm computes the product of these two numbers. Two arrays c and d are allocated to store the intermediate and final results respectively. The size of c is $2 \times \deg + 1$, and the size of d is $2 \times \deg + 2$.

Memory Allocation

- **Array** c**:** This array will store the intermediate multiplication results. It has a size of $2 \times \text{deg} + 1$ because the product of two numbers of degree deg can have at most $2 \times \text{deg}$ terms.
- **Array** d**:** This array will store the final result after carry propagation. It has a size of $2 \times \text{deg} + 2$ to accommodate any overflow from the carry propagation process.

Pairwise Multiplication

The algorithm performs a nested loop over each pair of elements from arrays a and b. For each pair of indices i and j, the corresponding elements $a[i]$ and $b[j]$ are multiplied, and the result is added to the appropriate position in the array c:

- The product $a[i] \times b[j]$ is added to the element $c[i + j]$ in the intermediate result array.
- This step performs the pairwise multiplication, similar to multiplying polynomials term-by-term.

Carry Propagation

After the pairwise multiplication, the algorithm needs to ensure that the intermediate result fits within the 29-bit chunk size. To achieve this:

- Each element in the intermediate result array c is masked with 0x1FFFFFFF (which is $2^{29} - 1$) to ensure it fits within 29 bits.
- If any element in c exceeds 29 bits, the overflow is carried over to the next element. Specifically, the carry is propagated by shifting the excess value to the next higher bit position.
- This process ensures that all values in c are constrained to 29 bits and any overflow is correctly handled.

Final Result Construction

After the carry propagation, the algorithm stores the results in the final result array d. The final value at each position in d is taken from c, with the appropriate carry propagated.

- The result array d is constructed such that it contains the product of the two large numbers, stored in 29-bit chunks.

Free Memory

The intermediate result array c is no longer needed after the final result has been constructed in array d, so it is deallocated to free up memory.

Return Result

Finally, the algorithm returns the result stored in the array d, which now contains the product of the two input numbers, represented as an array of 29 bit fragments.

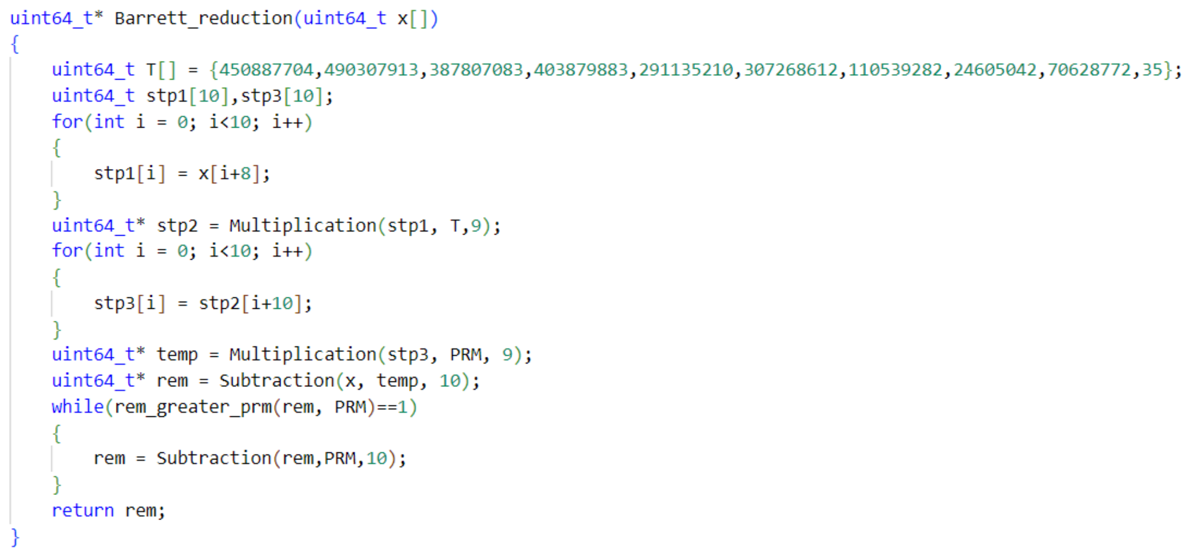
2.4 Barrett Reduction

Barrett Reduction is an efficient algorithm used to compute $z \mod p$ for given positive integers z and p. Unlike traditional modular reduction algorithms that may exploit specific properties of p, Barrett reduction works with any general modulus. It avoids repeated division operations by estimating the quotient $\lfloor z/p \rfloor$ using simpler calculations based on a power of a chosen base θ (commonly $\theta = 2^L$, where L is dependent on p but not on z).

The algorithm involves precomputing a modulus-dependent value $\lfloor \theta^{2k}/p \rfloor$, which significantly improves efficiency when multiple reductions are performed with the same modulus p.

Algorithm 1 Barrett Reduction

```
Require: $p, b \geq 3,\ k = \lfloor \log_b p \rfloor + 1,\ 0 \leq z < b^{2k},\ \mu = \left\lfloor \frac{b^{2k}}{p} \right\rfloor$
Ensure: $z \mod p$
1: $q \leftarrow \left\lfloor \frac{\lfloor z/b^{k-1} \rfloor \cdot \mu}{b^{k+1}} \right\rfloor$
2: $r \leftarrow (z \mod b^{k+1}) - (q \cdot p \mod b^{k+1})$
3: if $r < 0$ then
4:     $r \leftarrow r + b^{k+1}$
5: end if
6: while $r \geq p$ do
7:     $r \leftarrow r - p$
8: end while
9: return $r$
```



```
uint64_t* Barrett_reduction(uint64_t x[])
{
    uint64_t T[] = {450887704,490307913,387807083,403879883,291135210,307268612,110539282,24605042,70628772,35};
    uint64_t stp1[10],stp3[10];
    for(int i = 0; i<10; i++)
    {
        stp1[i] = x[i+8];
    }
    uint64_t* stp2 = Multiplication(stp1, T,9);
    for(int i = 0; i<10; i++)
    {
        stp3[i] = stp2[i+10];
    }
    uint64_t* temp = Multiplication(stp3, PRM, 9);
    uint64_t* rem = Subtraction(x, temp, 10);
    while(rem_greater_prm(rem, PRM)==1)
    {
        rem = Subtraction(rem,PRM,10);
    }
    return rem;
}
```

Figure 2.3: Snapshot of the Barrett_Reduction Algorithm

Steps of the Algorithm

Barrett Reduction Function (`Barrett_reduction`)

- PRM is the modulus array representing the large prime number for which we want to reduce the input.
- The function begins by defining a constant $T = \lfloor \frac{\theta^{2k}}{PRM} \rfloor$, where $\theta = 2^{29}$ and $k = \lfloor \log_\theta PRM \rfloor + 1 = 9$.
- A portion of the input array x (from index 8 to 17) is extracted into the array `stp1` $= \lfloor \frac{x}{\theta^{k-1}} \rfloor$.
- The array `stp1` is multiplied by the precomputed constant T.
- From the array `stp2`, the lower 10 elements are extracted into the array `stp3` $= \lfloor \frac{stp2}{\theta^{k-1}} \rfloor$.
- A `while` loop checks if the remainder `rem` is greater than PRM using the helper function `rem_greater_prm`.
- If `rem > PRM`, the remainder is further reduced by subtracting PRM from it. This step ensures that the final remainder is less than PRM and is congruent to x modulo PRM.

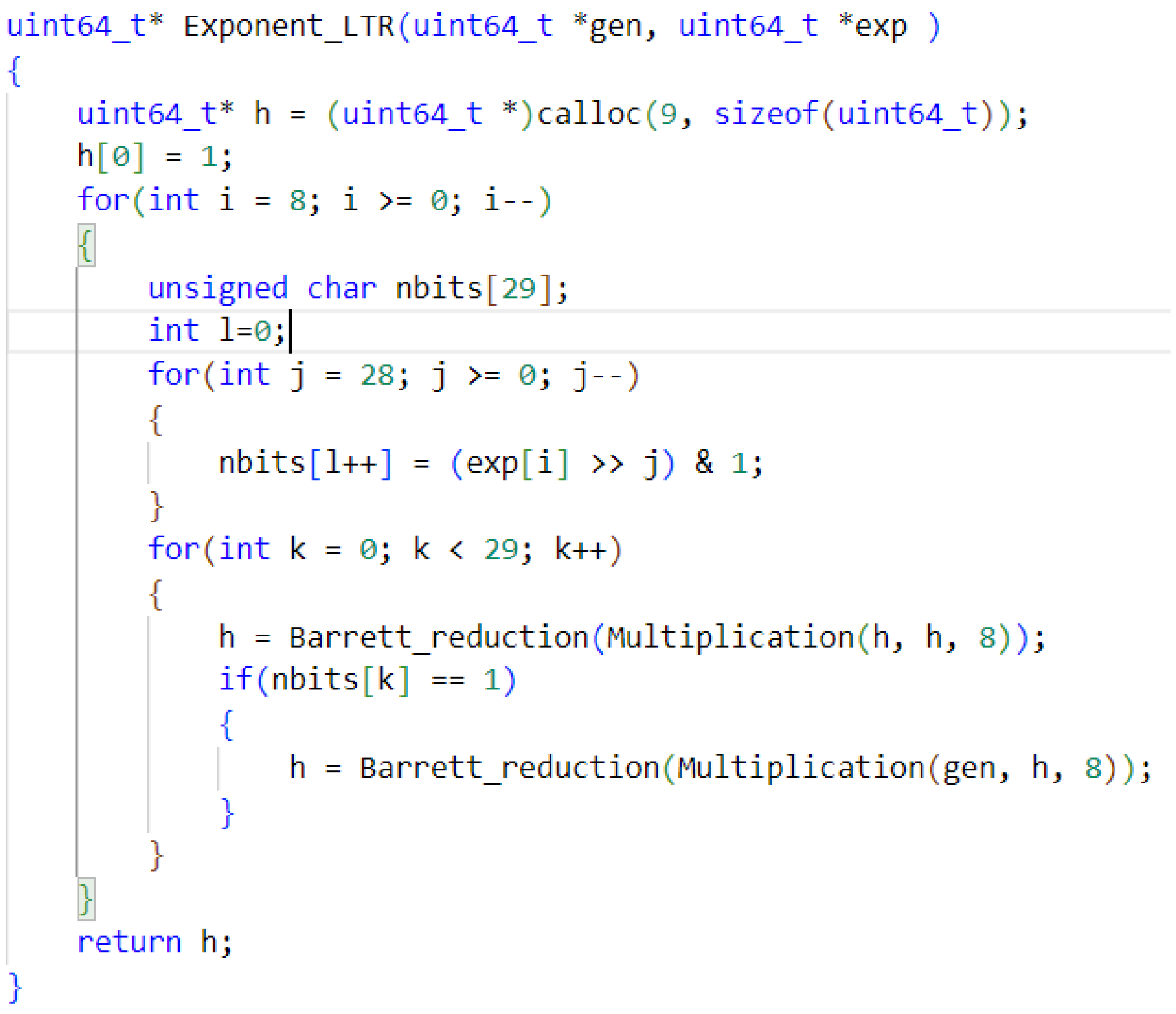
Final Result:

- Once the remainder is smaller than PRM, the loop exits, and the final reduced value (`rem`) is returned as the result of the Barrett reduction.

2.5 Exponentiation from Left to Right

Goal of the Function

The primary goal of the `Exponent_LTR` function is to perform modular exponentiation using the Left-to-Right (LTR) binary method. The function efficiently computes the result by leveraging the binary representation of the exponent and applying repeated squaring and modular multiplication. The result is kept within bounds using Barrett reduction.



```c
uint64_t* Exponent_LTR(uint64_t *gen, uint64_t *exp )
{
    uint64_t* h = (uint64_t *)calloc(9, sizeof(uint64_t));
    h[0] = 1;
    for(int i = 8; i >= 0; i--)
    {
        unsigned char nbits[29];
        int l=0;
        for(int j = 28; j >= 0; j--)
        {
            nbits[l++] = (exp[i] >> j) & 1;
        }
        for(int k = 0; k < 29; k++)
        {
            h = Barrett_reduction(Multiplication(h, h, 8));
            if(nbits[k] == 1)
            {
                h = Barrett_reduction(Multiplication(gen, h, 8));
            }
        }
    }
    return h;
}
```

Figure 2.4: Snapshot of the Exponentiation(LTR) Algorithm

Steps of the Algorithm

1. **Initialization**: An array `h` of 9 `uint64_t` elements is initialized, with the first element set to 1 (the identity for multiplication) and the rest set to 0. This array will hold the intermediate results during the computation.

2. **Exponent Decomposition**: The exponent array `exp` is processed starting from the most significant bit. Each element in the array is further decomposed into its 29 most significant bits, representing the binary form of the exponent.

3. **Left-to-Right Exponentiation**: For each bit of the exponent:

   - The current value of `h` is squared.

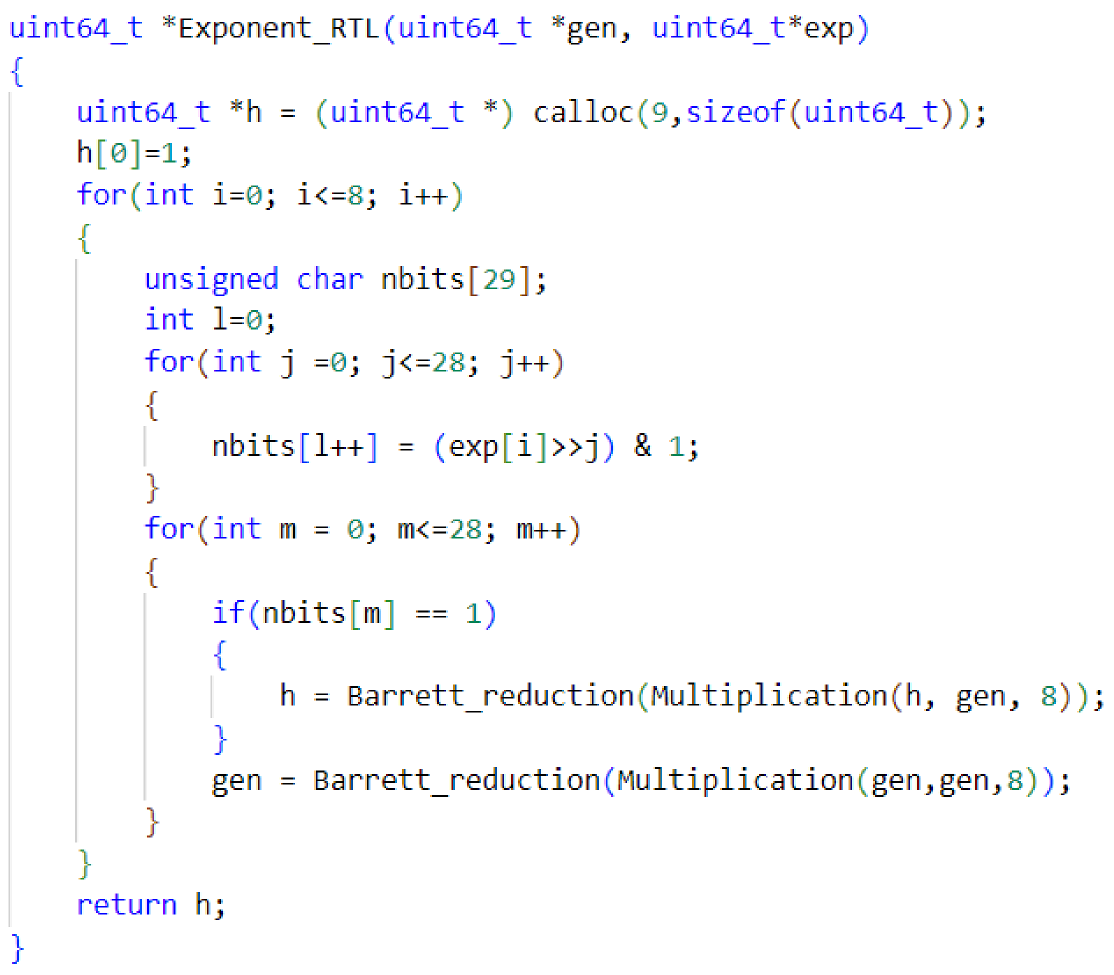
- If the corresponding bit of the exponent is 1, `h` is multiplied by the base `gen`.
- After every multiplication, Barrett reduction is applied to ensure that the result remains within modular limits.

4. **Repeat the process** until all bits of the exponent have been processed.

2.6 Exponentiation from Right to Left

Goal of the Function

The `Exponent_RTL` function performs modular exponentiation using the Right-to-Left (RTL) binary method. It calculates $\text{gen}^{\text{exp}} \mod N$, where `gen` is the base and `exp` is the exponent. This method processes the exponent from its least significant bit to the most significant bit.



```
uint64_t *Exponent_RTL(uint64_t *gen, uint64_t*exp)
{
    uint64_t *h = (uint64_t *) calloc(9,sizeof(uint64_t));
    h[0]=1;
    for(int i=0; i<=8; i++)
    {
        unsigned char nbits[29];
        int l=0;
        for(int j =0; j<=28; j++)
        {
            nbits[l++] = (exp[i]>>j) & 1;
        }
        for(int m = 0; m<=28; m++)
        {
            if(nbits[m] == 1)
            {
                h = Barrett_reduction(Multiplication(h, gen, 8));
            }
            gen = Barrett_reduction(Multiplication(gen,gen,8));
        }
    }
    return h;
}
```

Figure 2.5: Snapshot of the Exponentiation(RTL) Algorithm

Steps of the Algorithm

1. **Initialization**: Allocate a result array `h` of size 9, initialized to zero, with `h[0] = 1` as the identity element for multiplication.

2. **Exponent Processing**: For each `exp[i]` (from 0 to 8):

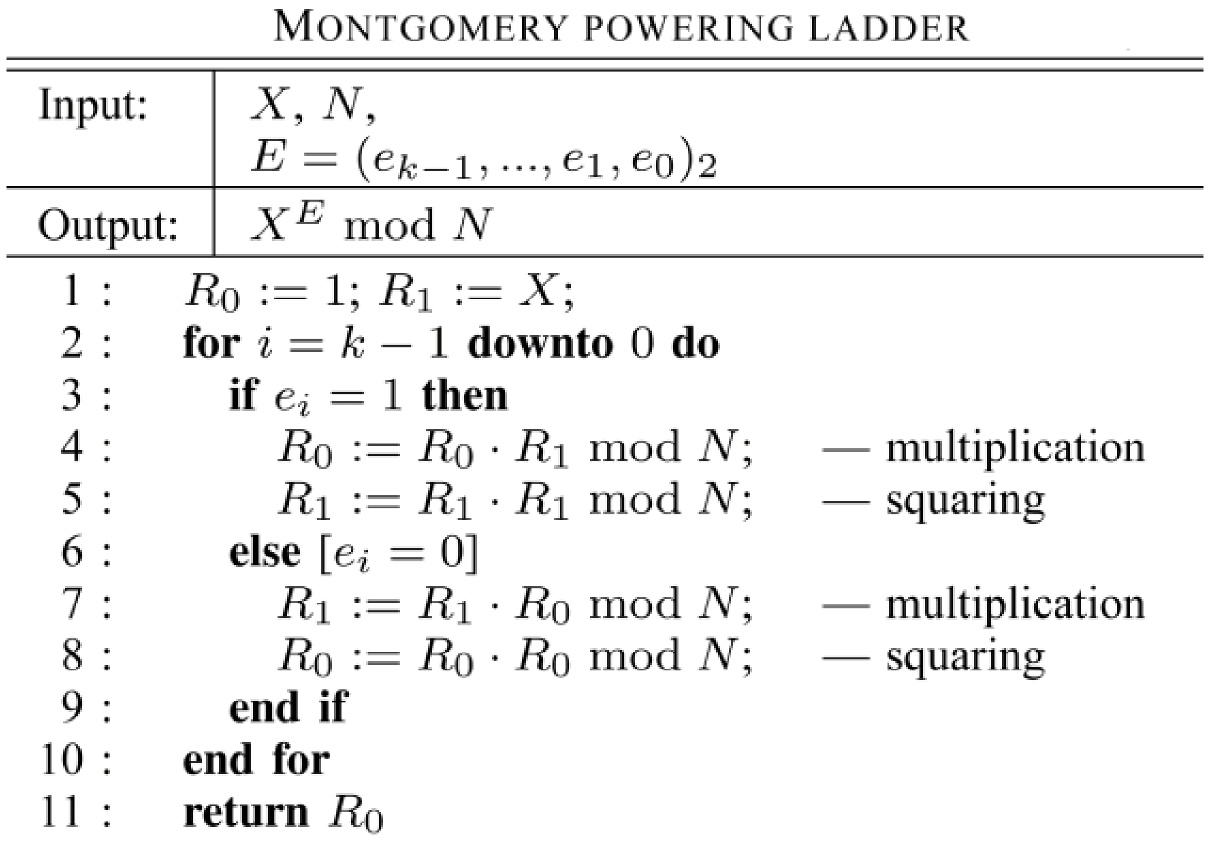
   - Decompose `exp[i]` into 29 bits and store in `nbits`.
   - For each bit `nbits[m]` (from 0 to 28):
     - If `nbits[m] == 1`, multiply `h` by `gen` and apply Barrett reduction.
     - Square `gen` and apply Barrett reduction after each step.

3. **Return the Result**: The final value of `h` represents $\text{gen}^{\text{exp}} \mod N$. Return this array to the caller.

2.7 Exponentiation Using Montgomery Ladder

Goal of the Function

The `Exponent_Montgomery` function performs modular exponentiation using the Montgomery Ladder technique, which is a secure and efficient method for computing $\text{gen}^{\text{exp}} \mod N$. This method offers protection against side-channel attacks by ensuring that the same sequence of operations is performed regardless of the exponent's bit values.

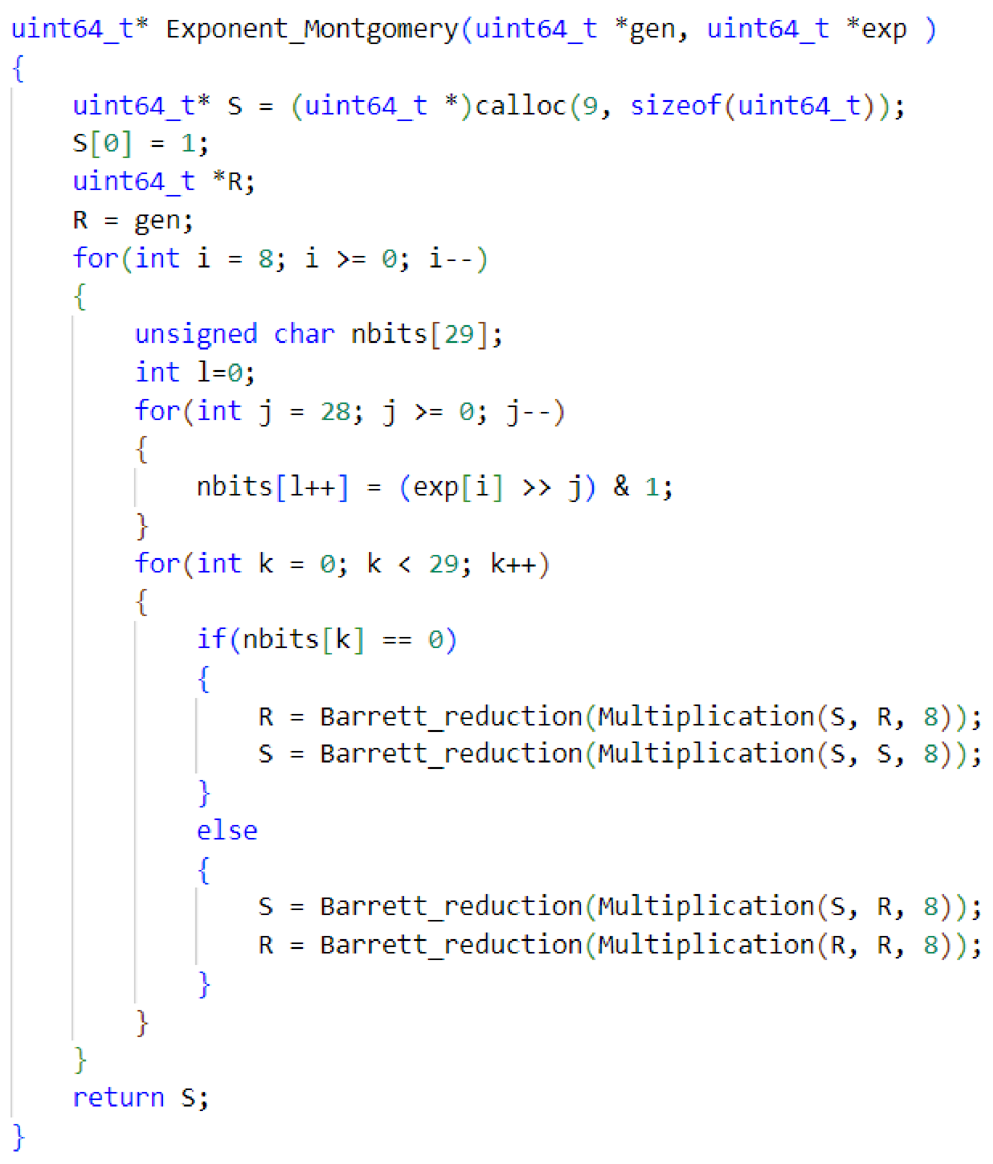


MONTGOMERY POWERING LADDER

Input:	X, N, $E = (e_{k-1}, ..., e_1, e_0)_2$
Output:	$X^E \bmod N$

```
1 :  $R_0 := 1$; $R_1 := X$;
2 :  for $i = k - 1$ downto $0$ do
3 :     if $e_i = 1$ then
4 :        $R_0 := R_0 \cdot R_1 \bmod N$;     — multiplication
5 :        $R_1 := R_1 \cdot R_1 \bmod N$;     — squaring
6 :     else $[e_i = 0]$
7 :        $R_1 := R_1 \cdot R_0 \bmod N$;     — multiplication
8 :        $R_0 := R_0 \cdot R_0 \bmod N$;     — squaring
9 :     end if
10 : end for
11 : return $R_0$
```

Figure 2.6: Algorithm of the Exponentiation(Montgomery Ladder)



```c
uint64_t* Exponent_Montgomery(uint64_t *gen, uint64_t *exp )
{
    uint64_t* S = (uint64_t *)calloc(9, sizeof(uint64_t));
    S[0] = 1;
    uint64_t *R;
    R = gen;
    for(int i = 8; i >= 0; i--)
    {
        unsigned char nbits[29];
        int l=0;
        for(int j = 28; j >= 0; j--)
        {
            nbits[l++] = (exp[i] >> j) & 1;
        }
        for(int k = 0; k < 29; k++)
        {
            if(nbits[k] == 0)
            {
                R = Barrett_reduction(Multiplication(S, R, 8));
                S = Barrett_reduction(Multiplication(S, S, 8));
            }
            else
            {
                S = Barrett_reduction(Multiplication(S, R, 8));
                R = Barrett_reduction(Multiplication(R, R, 8));
            }
        }
    }
    return S;
}
```

Figure 2.7: Snapshot of the Exponentiation(using Montgomery Ladder) Algorithm

Steps of the Algorithm

1. **Initialization**:
   - Allocate an array `S` of size 9, initialized to zero, and set `S[0] = 1` (identity element for multiplication).
   - Set `R` to point to `gen`, the base of the exponentiation.
2. **Process the Exponent**:
   - For each element `exp[i]` from 8 down to 0:
     - Decompose `exp[i]` into its 29-bit binary representation and store the bits in the array `nbits`.
     - For each bit `k` from 0 to 28:
       * If `nbits[k] == 0`:
         · Update `R` by computing $R = S \times R \mod N$ using `Multiplication` and `Barrett_reduction`.
         · Update `S` by computing $S = S \times S \mod N$.
       * If `nbits[k] == 1`:

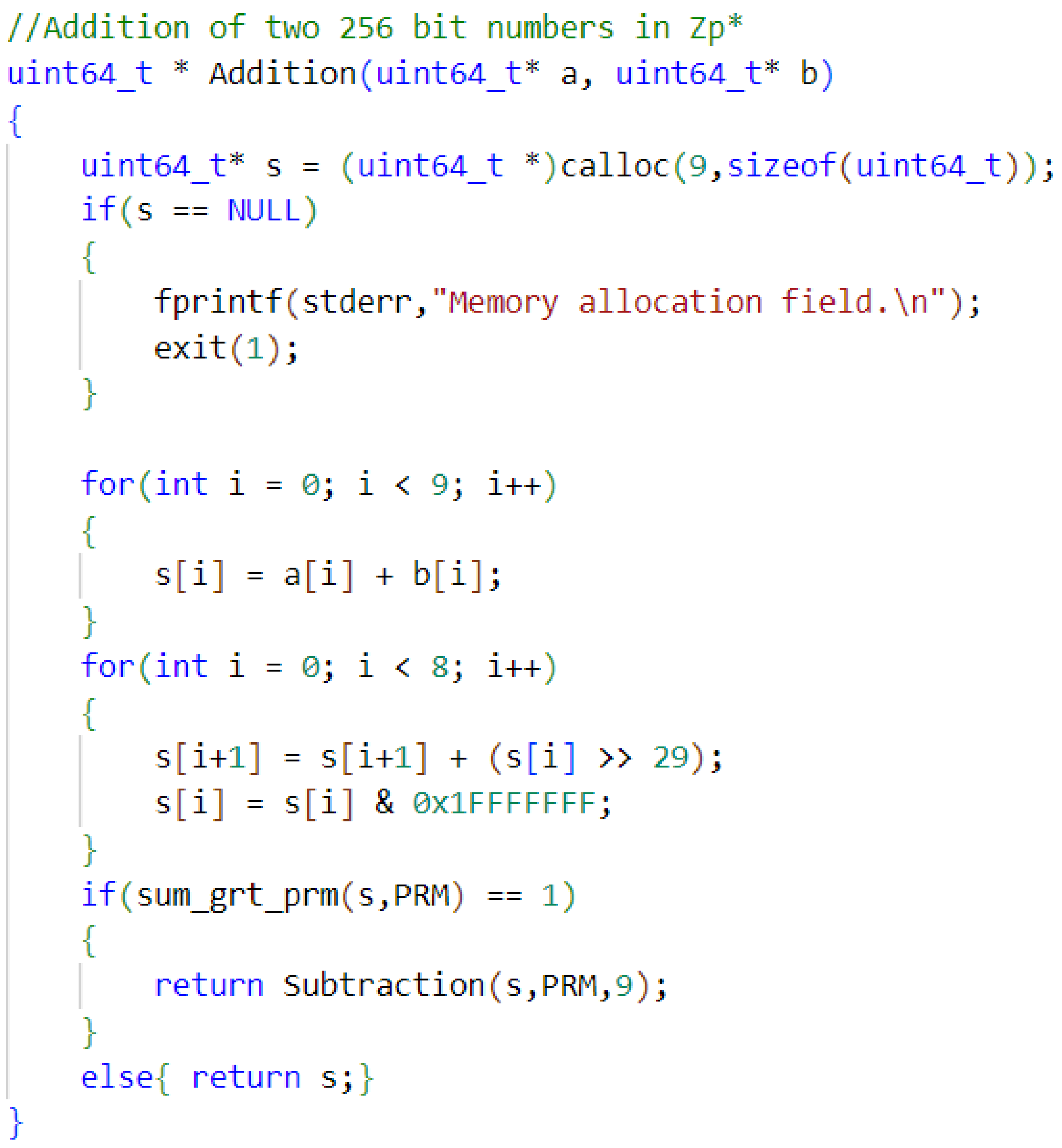
- Update `S` by computing $S = S \times R \mod N$.
- Update `R` by computing $R = R \times R \mod N$.

3. **Return the Result**: Return the array `S`, which contains the result of $\text{gen}^{\text{exp}} \mod N$.

2.8 Addition in $\mathbb{Z}_p$

Goal of the Function

The provided functions implement the addition of two 256-bit numbers in the finite field $\mathbb{Z}_p^*$. The field $\mathbb{Z}_p$ represents the set of integers modulo a prime p, ensuring that the result always stays within this field. The functions handle large number addition, carry propagation, and modular reduction when the sum exceeds the prime p.



```
//Addition of two 256 bit numbers in Zp*
uint64_t * Addition(uint64_t* a, uint64_t* b)
{
    uint64_t* s = (uint64_t *)calloc(9,sizeof(uint64_t));
    if(s == NULL)
    {
        fprintf(stderr,"Memory allocation field.\n");
        exit(1);
    }

    for(int i = 0; i < 9; i++)
    {
        s[i] = a[i] + b[i];
    }
    for(int i = 0; i < 8; i++)
    {
        s[i+1] = s[i+1] + (s[i] >> 29);
        s[i] = s[i] & 0x1FFFFFFF;
    }
    if(sum_grt_prm(s,PRM) == 1)
    {
        return Subtraction(s,PRM,9);
    }
    else{ return s;}
}
```

Figure 2.8: Snapshot of the Addition function.

Steps of the Algorithm

1. Function: `sum_grt_prm`

- Compares two 256-bit numbers, `sum` and `prm`, to determine if `sum` is greater than `prm`.
- Iterates from the most significant part (index 8) down to the least significant part:

  - Returns `1` if `sum` is greater than `prm`.
  - Returns `0` if `sum` is less than or equal to `prm`.

2. Function: `Addition`

- **Memory Allocation**: Allocates memory for a 9-element array `s` to store the sum. If memory allocation fails, the program exits with an error message.
- **Addition and Carry Propagation**:
  - Adds corresponding elements of arrays `a` and `b`.
  - Propagates carries by shifting 29 bits and masking lower 29 bits.
- **Modular Reduction**:
  - Uses `sum_grt_prm` to check if the result `s` exceeds the prime `PRM`.
  - If `s > PRM`, performs modular reduction using the `Subtraction` function.
  - Otherwise, returns `s` as the result.

2.9 Inverse in $\mathbb{Z}_p^*$

Goal of the Function

The `Inverse` function computes the modular multiplicative inverse of a given 256-bit number z in the finite field $\mathbb{Z}_p^*$, where p is a prime. The result satisfies the condition:

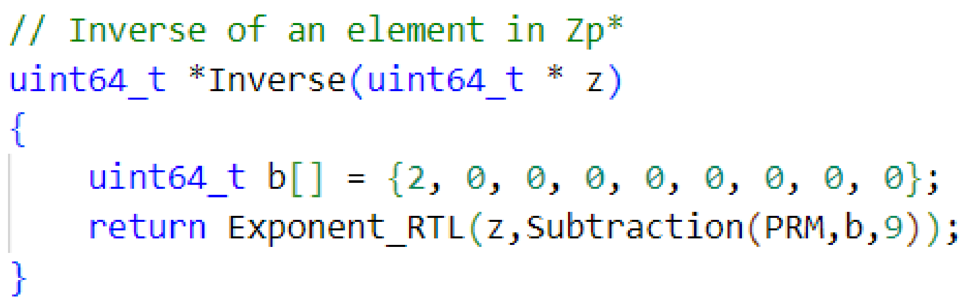
$$z \cdot z^{-1} \equiv 1 \pmod{p}$$

This function uses Fermat's Little Theorem, which states that for any integer z in $\mathbb{Z}_p^*$:

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

Thus, the inverse is computed as:

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



```
// Inverse of an element in Zp*
uint64_t *Inverse(uint64_t * z)
{
    uint64_t b[] = {2, 0, 0, 0, 0, 0, 0, 0, 0};
    return Exponent_RTL(z,Subtraction(PRM,b,9));
}
```

Figure 2.9: Snapshot of the Inverse function.

Steps of the Algorithm

1. **Initialize the Exponent**: The array `b` represents the constant value 2, i.e., $[2, 0, 0, \ldots, 0]$. The exponent for the modular inverse is calculated as $p - 2$ using the function `Subtraction(PRM, b, 9)`, where `PRM` is the prime modulus.
2. **Exponentiation**: The `Exponent_RTL` function is used to compute $z^{p-2} \mod p$ using the Right-to-Left binary exponentiation method.
3. **Return the Inverse**: The result of z^{p-2} is returned, which represents the modular inverse of z.

Chapter 3

Elliptic Curve Diffie-Hellman Key Exchange

3.1 Elliptic Curve over Finite Field

Elliptic Curve Cryptography (ECC) is a modern approach to public-key cryptography that provides strong security with smaller key sizes compared to traditional methods like RSA and Diffie-Hellman. ECC is based on the algebraic structure of elliptic curves over finite fields, typically defined over the finite field $\mathbb{Z}_p$, where p is a prime number. This finite field ensures that the set of points on the elliptic curve forms a finite, well-defined group.

Weierstrass Equation

The most commonly used form of an elliptic curve in ECC, defined over the finite field $\mathbb{Z}_p$, is given by the Weierstrass equation:

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

where:

- x and y are variables representing points on the curve, and these coordinates are elements of the finite field $\mathbb{Z}_p$.
- a and b are constants that define the curve, subject to the condition:

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

This condition ensures that the curve is *non-singular*, meaning it has no sharp corners or self-intersections. The set of points on an elliptic curve over a field $\mathbb{Z}_p$ is denoted by $E(\mathbb{Z}_p)$ and is given as:
$E(\mathbb{Z}_p) = \{\infty\} \cup \{(x, y) \in \mathbb{Z}_p \times \mathbb{Z}_p \mid y^2 = x^3 + Ax + B\}$.
In this equation:

- ∞ is the point at infinity, which is included by definition as the identity element in the elliptic curve group.
- (x, y) represents a point on the elliptic curve, where x and y are coordinates in the field($\mathbb{Z}_p$), and the curve is defined by the Weierstrass equation $y^2 = x^3 + Ax + B$.
- A and B are constants that define the specific elliptic curve.

The point at infinity ∞ serves as the identity element for the group of points on the elliptic curve, enabling the structure of an abelian group under the operation of point addition.

Elliptic Curve Operations

ECC uses points on the curve to perform cryptographic operations, with the key operations being:

- **Point Addition**: Adding two distinct points P and Q on the curve to get another point $R = P + Q$.

- **Point Doubling**: Adding a point P to itself to get $R = 2P$.
- **Scalar Multiplication**: Multiplying a point P by an integer k, denoted as kP, which is the core operation in ECC.

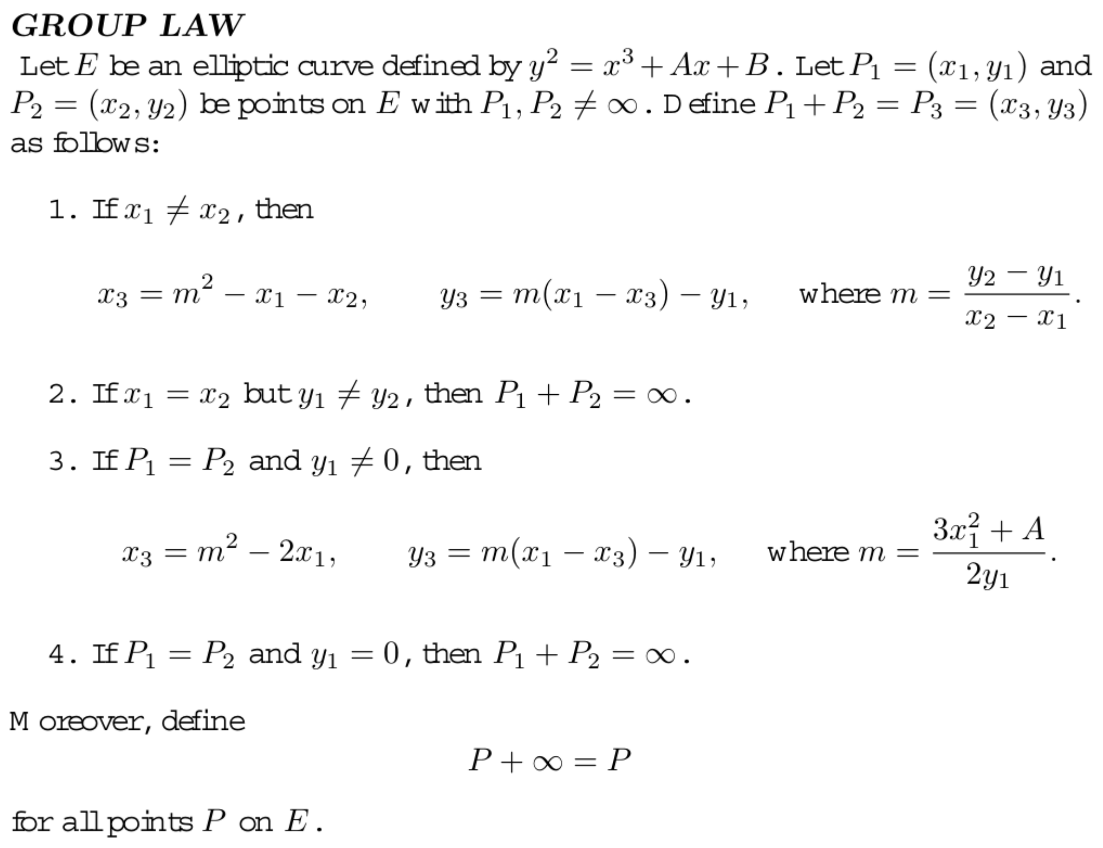
Advantages of ECC

- **Smaller Key Sizes**: ECC achieves the same level of security as RSA with much smaller keys. For example, a 256-bit ECC key offers comparable security to a 3072-bit RSA key.
- **Efficiency**: Faster computations and lower power consumption, making ECC ideal for devices with limited resources such as IoT devices and mobile phones.
- **Strong Security**: The hardness of the Elliptic Curve Discrete Logarithm Problem (ECDLP) underpins ECC's security, making it resistant to known cryptographic attacks.

3.2 Point Addition on Elliptic Curve: `Elliptic_add`

Goal of the Function

The goal of the `Elliptic_add` function is to perform point addition on an elliptic curve. It takes two points, a and b, and returns their sum. The function handles special cases like adding the point at infinity, adding points with the same x-coordinate (point doubling), and adding distinct points. This operation follows the elliptic curve point addition law over a finite field.

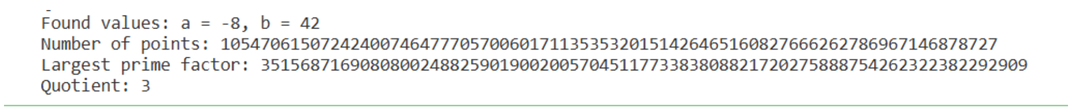


GROUP LAW

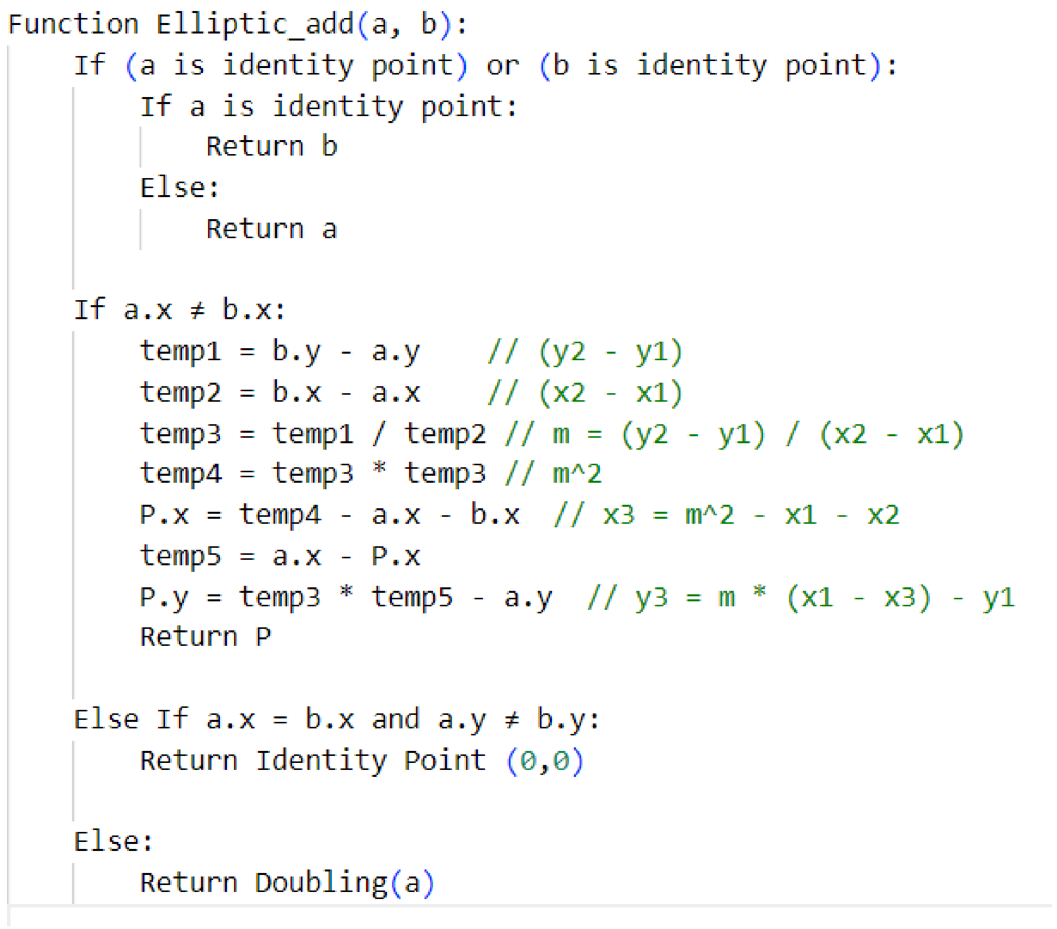
Let E be an elliptic curve defined by $y^2 = x^3 + Ax + B$. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be points on E with $P_1, P_2 \neq \infty$. Define $P_1 + P_2 = P_3 = (x_3, y_3)$ as follows:

1. If $x_1 \neq x_2$, then
$$x_3 = m^2 - x_1 - x_2, \qquad y_3 = m(x_1 - x_3) - y_1, \qquad \text{where } m = \frac{y_2 - y_1}{x_2 - x_1}.$$
2. If $x_1 = x_2$ but $y_1 \neq y_2$, then $P_1 + P_2 = \infty$.
3. If $P_1 = P_2$ and $y_1 \neq 0$, then
$$x_3 = m^2 - 2x_1, \qquad y_3 = m(x_1 - x_3) - y_1, \qquad \text{where } m = \frac{3x_1^2 + A}{2y_1}.$$
4. If $P_1 = P_2$ and $y_1 = 0$, then $P_1 + P_2 = \infty$.

Moreover, define
$$P + \infty = P$$
for all points P on E.



```
Found values: a = -8, b = 42
Number of points: 105470615072424007464777057006017113535320151426465160827666262786967146878727
Largest prime factor: 35156871690808002488259019002005704511773383808821720275888754262322382292909
Quotient: 3
```



```
Function Elliptic_add(a, b):
    If (a is identity point) or (b is identity point):
        If a is identity point:
            Return b
        Else:
            Return a

    If a.x ≠ b.x:
        temp1 = b.y - a.y    // (y2 - y1)
        temp2 = b.x - a.x    // (x2 - x1)
        temp3 = temp1 / temp2 // m = (y2 - y1) / (x2 - x1)
        temp4 = temp3 * temp3 // m^2
        P.x = temp4 - a.x - b.x  // x3 = m^2 - x1 - x2
        temp5 = a.x - P.x
        P.y = temp3 * temp5 - a.y  // y3 = m * (x1 - x3) - y1
        Return P

    Else If a.x = b.x and a.y ≠ b.y:
        Return Identity Point (0,0)

    Else:
        Return Doubling(a)
```

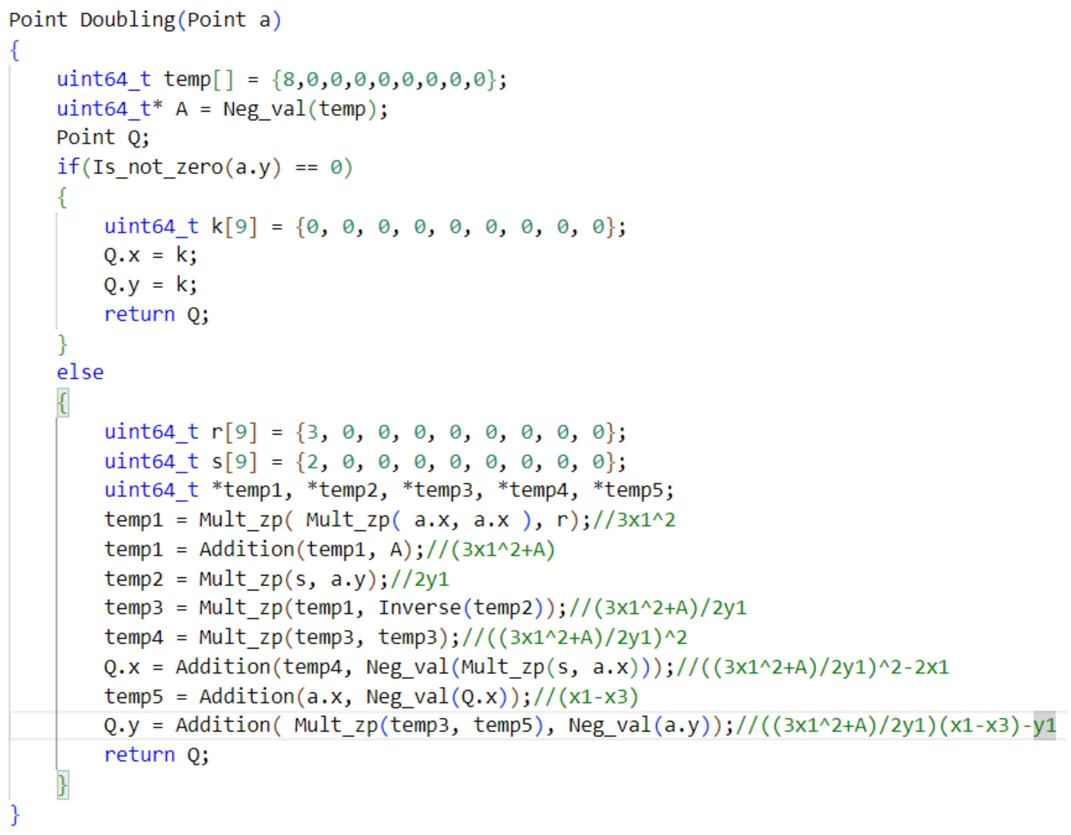
3.3 Point Doubling

The Point Doubling function computes the result of doubling a point on an elliptic curve. Let the point $P = (x_1, y_1)$ be a point on the elliptic curve defined by the equation:

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

where a and b are constants defining the curve.

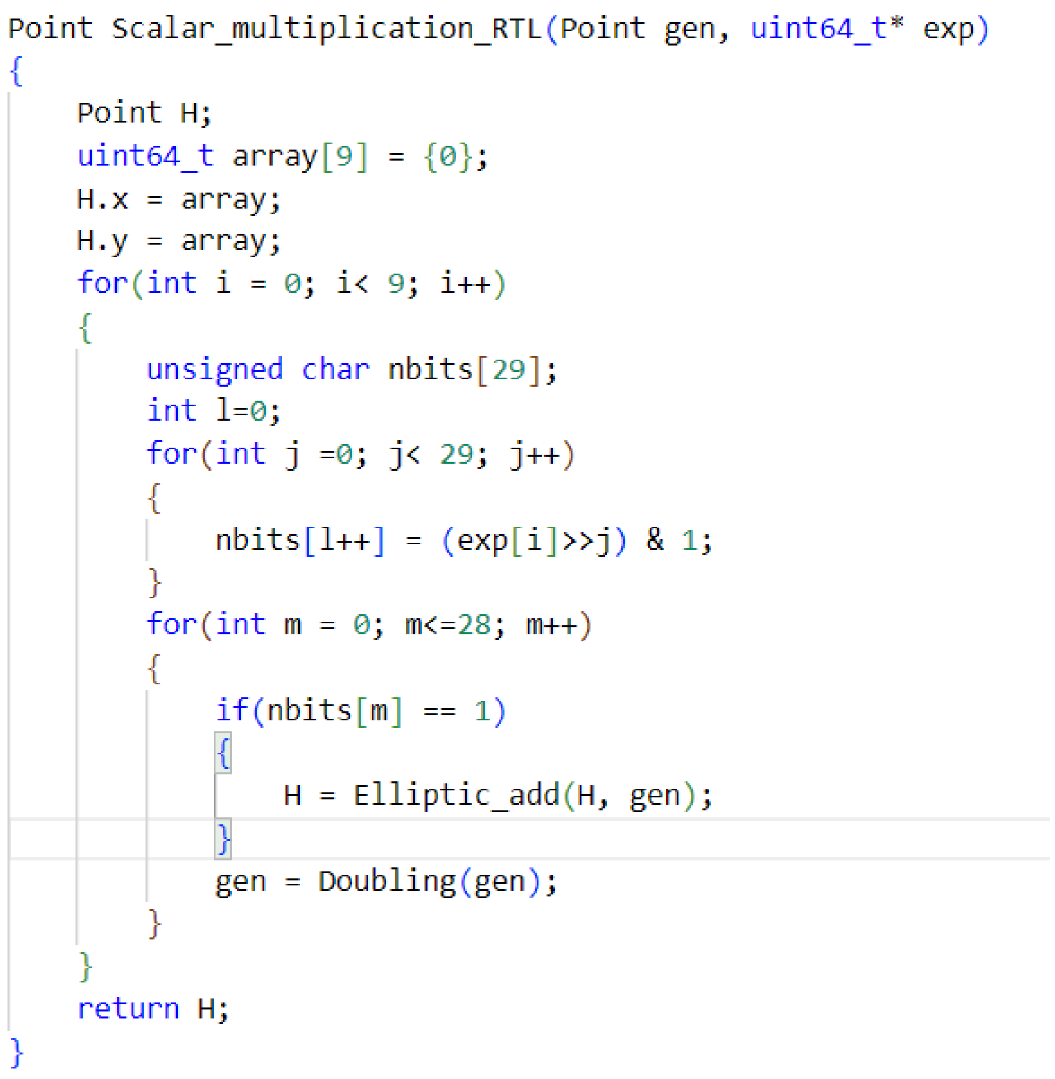
Given a point $P = (x_1, y_1)$, the goal of the point doubling operation is to compute $2P = (x_2, y_2)$, where x_2 and y_2 are the coordinates of the doubled point.



```
Point Doubling(Point a)
{
    uint64_t temp[] = {8,0,0,0,0,0,0,0,0};
    uint64_t* A = Neg_val(temp);
    Point Q;
    if(Is_not_zero(a.y) == 0)
    {
        uint64_t k[9] = {0, 0, 0, 0, 0, 0, 0, 0, 0};
        Q.x = k;
        Q.y = k;
        return Q;
    }
    else
    {
        uint64_t r[9] = {3, 0, 0, 0, 0, 0, 0, 0, 0};
        uint64_t s[9] = {2, 0, 0, 0, 0, 0, 0, 0, 0};
        uint64_t *temp1, *temp2, *temp3, *temp4, *temp5;
        temp1 = Mult_zp( Mult_zp( a.x, a.x ), r);//3x1^2
        temp1 = Addition(temp1, A);//(3x1^2+A)
        temp2 = Mult_zp(s, a.y);//2y1
        temp3 = Mult_zp(temp1, Inverse(temp2));//(3x1^2+A)/2y1
        temp4 = Mult_zp(temp3, temp3);//((3x1^2+A)/2y1)^2
        Q.x = Addition(temp4, Neg_val(Mult_zp(s, a.x)));//((3x1^2+A)/2y1)^2-2x1
        temp5 = Addition(a.x, Neg_val(Q.x));//(x1-x3)
        Q.y = Addition( Mult_zp(temp3, temp5), Neg_val(a.y));//((3x1^2+A)/2y1)(x1-x3)-y1
        return Q;
    }
}
```

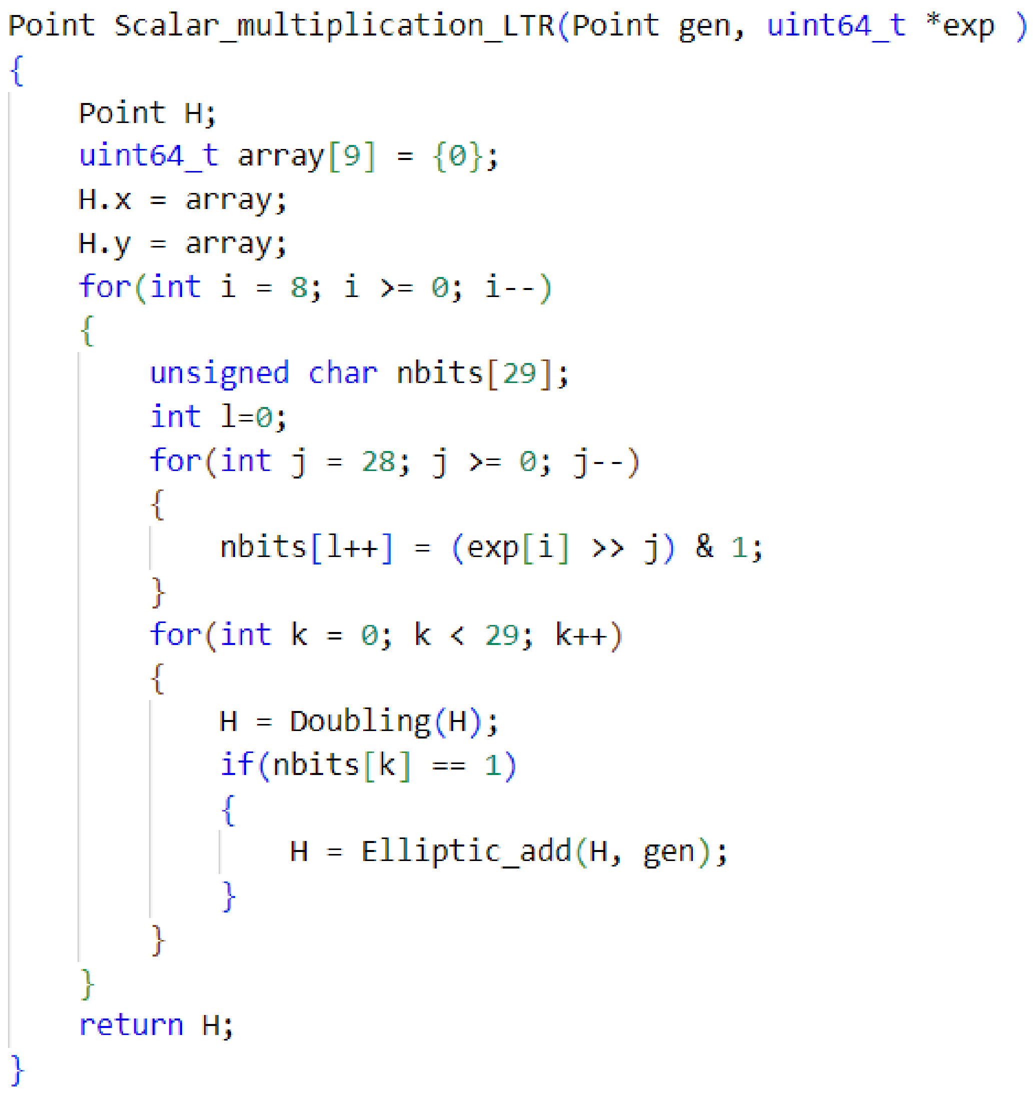
3.4 Scalar Multiplication

Two functions, Scalar_multiplication_RTL and Scalar_multiplication_LTR, implement scalar multiplication on an elliptic curve using different traversal approaches of the scalar's bits. Scalar multiplication is a core operation in elliptic curve cryptography (ECC), where a point on the curve is multiplied by a scalar (a large integer) to produce another point on the curve.



```
Point Scalar_multiplication_RTL(Point gen, uint64_t* exp)
{
    Point H;
    uint64_t array[9] = {0};
    H.x = array;
    H.y = array;
    for(int i = 0; i< 9; i++)
    {
        unsigned char nbits[29];
        int l=0;
        for(int j =0; j< 29; j++)
        {
            nbits[l++] = (exp[i]>>j) & 1;
        }
        for(int m = 0; m<=28; m++)
        {
            if(nbits[m] == 1)
            {
                H = Elliptic_add(H, gen);
            }
            gen = Doubling(gen);
        }
    }
    return H;
}
```

Figure 3.1: Algorithm of Scalar_multiplication_RTL



```
Point Scalar_multiplication_LTR(Point gen, uint64_t *exp )
{
    Point H;
    uint64_t array[9] = {0};
    H.x = array;
    H.y = array;
    for(int i = 8; i >= 0; i--)
    {
        unsigned char nbits[29];
        int l=0;
        for(int j = 28; j >= 0; j--)
        {
            nbits[l++] = (exp[i] >> j) & 1;
        }
        for(int k = 0; k < 29; k++)
        {
            H = Doubling(H);
            if(nbits[k] == 1)
            {
                H = Elliptic_add(H, gen);
            }
        }
    }
    return H;
}
```

Figure 3.2: Algorithm of Scalar_multiplication_LTR