Cryptographic Engineering: Assignment 3 Prelab

In the previous assignment, you learned how to perform arithmetic operations. In this assignment, you will learn how to perform efficient reduction operations. Reduction operations are necessary in cryptography to limit the growth of numbers caused by the arithmetic operations. Combing arithmetic operations with reduction operations is known as **Modular Arithmetic**. Please read the following sections before attempting the assignment.

Modular Addition

In modular addition, the operation performed is $r = (a + b) \mod p$. Given $0 \leq a < p$ and $0 \leq b < p$, first $s = a + b$ is computed. Then, you will get $0 \leq s < 2p$. There are two cases here:

- If $0 \leq s < p$, then $r = s$
- If $p \leq s < 2p$, then $r = s - p$

There are few issues performing the operation this way. First issue is that comparing multi-precision integers s and p is an expensive operation as every array element needs to be compared. This issue can be resolved by computing first $s = a + b - p$. Then, you will get $-p \leq s < p$. The two cases become:

- If $-p \leq s < 0$, then $r = s + p$
- If $0 \leq s < p$, then $r = s$

Now the comparison is between s and 0. Comparing with 0 is very simple as negative integers have their most significant bit (MSB) 1 while positive integers have their MSB 0.

The second issue is that when $s < 0$, an extra operation (add p) is needed, while when $s \geq 0$, no extra operations are needed. This means the algorithm will not work in constant time which is necessary to protect against side channel attacks. To resolve this issue, both s and $s + p$ are computed first, then you return whichever gives the correct value in an if else statement.

The third issue is using an if statement is vulnerable to fault injection. To resolve this issue, a masking approach is used. After computing s, set the value $\texttt{mask} = 0 - \texttt{MSB}$, then compute $r = s + (\texttt{mask}\&p)$ where & is bitwise `AND` operation. The reason this work is because $\texttt{mask}\&p = 0$ when $s \geq 0$ and $\texttt{mask}\&p = p$ when $s < 0$ (here `mask` is all 1s).

Following these approaches will allow you to implement a secure modular addition. One last thing to note is that you are working with values between $-p$ and $2p$ which means all values between$-2p$ and $2p$ need to be covered. Therefore, 2 extra bits are needed when performing the additions and subtractions. For example, if you have a 256-bit modulus, then 258-bit addition and 258-bit subtraction must be used.

Modular Subtraction

In modular subtraction, the operation performed is $r = (a - b) \mod p$. Given $0 \leq a < p$ and $0 \leq b < p$, first $s = a - b$ is computed. Then, you will get $-p < s < p$. There are two cases here:

- If $-p \leq s < 0$, then $r = s + p$
- If $0 \leq s < p$, then $r = s$

With modular subtraction, the comparison is with 0. Therefore, the first issue that the modular addition had does not exist here. The masking approach that was used in modular addition can be used here to implement a secure modular subtraction. Unlike modular addition, modular subtraction works with values between $-p$ and p. Therefore, only 1 extra bit is used during addition and subtraction.

Modular Multiplication

In modular multiplication, the operation performed is $r = (a \times b) \mod p$. In the previous assignment, you performed multiplication $s = a \times b$ using three different approaches (Schoolbook, Comba, Karatsuba). Given $0 \leq a < p$ and $0 \leq b < p$, $0 < s < p^2$. The previous approaches used in modular addition and modular subtraction are not applicable as it would require a large number of checks.

In this assignment, you are going to reduce s using three different techniques. All three techniques might return a result greater than p (usually between 0 and $2p$). A subtraction by p followed by a masking approach, similar to the one used in modular addition, can be used to obtain the correct result.

Pseudo-Mersenne technique

Pseudo-Mersenne technique is the first technique to perform reduction and it was covered in Assignment 1. Pseudo-Mersenne technique is only recommended for modulus of the form $p = 2^e - c$ where c is a small value. Otherwise, it will be inefficient. After applying pseudo-Mersenne technique, the final result is going to be between 0 and $(c + 1)p$. This requires up to c checks and subtractions which is quite expensive. A better approach is to use the pseudo-Mersenne technique a second time which will reduce it between 0 and $2p$ which requires one check and subtraction.

In this assignment, the modulus is $p = 2^{255} - 19$. For the first reduction, apply pseudo-Mersenne with a modulus $2^{256} - 38$. The reason $2^{256} - 38$ is used is because 256 bits aligns perfectly with most CPUs which are 32-bit or 64-bit. For the second reduction, apply the pseudo-Mersenne technique with modulus $2^{255} - 19$. In this second reduction, the high bits will fit in **one** 32-bit integer.

Barrett Reduction technique

Barrett reduction is a general method to reduce an integer. It works with any modulus. It is based on the long division method ($a = qp + r$). Instead of computing $q = \lfloor a/p \rfloor$ which is very expensive, it is estimated by only using division by a power of 2.

The algorithm is straight forward and can be seen in Algorithm 1. k can be any value greater that the number of bits of p but is usually chosen as close as possible to p. For the if statement (Lines 3-4), it is basically setting bit $k + 1$ (the MSB) to 0. You do not need an if statement when implementing the protocol. For the while loop (Lines 5-6), the maximum possible value of r can be computed. It is usually less than $4p$. Therefore, the masking technique from modular addition can be used here twice.

Algorithm 1: Barrett Reduction.

```
  Input  : $x$, $p < 2^k$ and $\mu = \lfloor 2^{2k}/p \rfloor$
  Output: $r = x \mod p$
1 $q_1 \leftarrow \lfloor x/2^{k-1} \rfloor$, $q_2 \leftarrow q_1 \cdot \mu$, $q_3 \leftarrow \lfloor q_2/2^{k+1} \rfloor$
2 $r_1 \leftarrow x \mod 2^{k+1}$, $r_2 \leftarrow q_3 \cdot p \mod 2^{k+1}$, $r \leftarrow r_1 - r_2$
3 if $r < 0$ then
4     $r \leftarrow r + 2^{k+1}$
5 while $r \geq p$ do
6     $r \leftarrow r - p$
7 return $r$
```

Montgomery Reduction

Montgomery Reduction is also a general method to reduce an integer. It is usually more efficient than Barrett reduction. Montgomery reduction does not compute $s \mod p$ but computes $s \cdot R^{-1} \mod p$

instead where $R = 2^k$. k is chosen similar to Barrett reduction (greater than number of bits of p but as close as possible to p). Combining the multiplication of two inputs followed by Montgomery reduction is called Montgomery multiplication.

To obtain the correct results, all inputs need to be multiplied by $R^2 \mod p$ and reduced by the Montgomery reduction method once at the beginning. This will convert them to Montgomery domain. All arithmetic operations (modular addition, modular subtraction, and Montgomery multiplication (replacing modular multiplication) are carried out as normal. Once all arithmetic operations are completed, the final output (still in Montgomery domain) is multiplied by 1 and reduced by Montgomery reduction to convert it back to the original domain which will then be the final output.

The Montgomery reduction algorithm is straightforward and can be seen in Algorithm 2. The algorithm first computes $q = (x \times \mu) \mod R$ (note that only the first loop of a Comba multiplication is needed here). Then, $t = x + qp$ is computed which has 0s in the k least significant bits. Afterwards, all these 0s are removed (shift right) and that will be the output (Line 3). The result is usually between 0 and $2p$. Therefore, one subtraction check (line 4-5) is needed.

Since we are dealing with large integers, computing q which 0s k bits of the result in one go is very expensive. Instead, we compute q that only 0s one least significant digit of the result at a time and perform it the total number of digits times. This can be seen in Algorithm 3. As an example, if the modulus is 16 digits, Algorithm 2 requires 136 multiplications to compute q while Algorithm 3 requires 1 multiplication for computing 16 different q for a total of 16 multiplications.

Furthermore, some moduli can have $\mu = 1$ and thus $q = x_0$ eliminating the need to do the multiplication. For example, Modulus of the form $f \cdot 2^e - 1$ where $e > w$ have $\mu = 1$. This modulus form is used in Supersingular Isogeny Key Encapsulation (SIKE), a public-key cryptography protocol proposed for post-quantum cryptography (sike.org).

Algorithm 2: Montgomery Reduction

```
   Input  : $x$, $p < 2^k$, $R = 2^k$ and $\mu = -p^{-1} \mod R$
   Output: $t = xR^{-1} \mod p$
1  $q \leftarrow (x \times \mu) \mod R$
2  $t \leftarrow x + qp$
3  $t \leftarrow t/R$
4  if $t \geq p$ then
5      $t \leftarrow t - p$
6  return $t$
```

Algorithm 3: Montgomery Reduction digit approach

```
   Input  : $w$ : digit size, $s$: number of digits, $x = (x_{2s-1} \dots x_1 x_0)$, $p = (p_{s-1} \dots p_1 p_0)$, $r = 2^w$,
            $R = 2^{sw}$ and $\mu = -p^{-1} \mod r$
   Output: $t = xR^{-1} \mod p$
1  for $j \leftarrow 0$ to $s - 1$ do
2      $q \leftarrow (x_0 \times \mu) \mod r$
3      $t \leftarrow x + qp$
4      $t \leftarrow t/r$
5  if $t \geq p$ then
6      $t \leftarrow t - p$
7  return $t$
```

Modular Inversion

Modular inversion is required in some cryptography protocols. Modular inversion computes $r = a^{-1} \mod p$. It is a very expensive operation and requires an algorithm to compute. There are two methods you applied in assignment 1 for modular inversion; Euclidean Extended Algorithm (EEA) and Fermat's Little Theorem (FLT). While EEA is the faster method, it is not constant time and therefore not secure. Therefore, FLT is used.

FLT is basically exponentiation and you are going to apply the square-and-multiply algorithm to compute it. Algorithm 4 shows the implementation. $(e_{k-1} \dots e_1 e_0)$ is binary representation of e. Here, the exponent $e = p - 2$ and is known. This means that the if statement (Line 4) is not needed because the condition in each iteration is known.

Note that in crypto algorithms that require inversion, the modulus is chosen as prime since otherwise not all inputs will have inverses.

Algorithm 4: Modular exponentiation using square-and-multiply algorithm.

```
   Input  : $a$, $p$, $e = (e_{k-1} \dots e_1 e_0)$ with $e_{k-1} = 1$
   Output: $b = a^e \mod p$
1  $b \leftarrow a$
2  for $i \leftarrow k-2$ downto $0$ do
3      $b \leftarrow b^2 \mod p$
4      if $e_i = 1$ then
5          $b \leftarrow b \cdot a \mod p$
6  return $b$
```