Cost Evaluation of the Asymmetric Cryptographic Algorithms

Andrej Hadži-Đorđević

August 2025

Summary

This project presents the implementation and results of evaluating libraries used for mathematical operations in asymmetric encryption schemes.

Keywords: Elliptic Curves, Prime-ordered Groups, NTL, GMP, Charm,
GitHub repository: available here.

Contents

1 Libraries and Environment

Before discussing the results and the implementation, we should first define the test environment and the libraries which are prerequisite for the tests. For more information on installing, refer to the listed references, where appropriate hyperlinks are provided.

Prerequisite software:

- GMP - GNU Multiple Precision Library for (C/C++)
- NTL - A Library for doing Number Theory (C/C++)
- Charm - Framework for rapidly prototyping cryptosystems (Python)

Output of neofetch tool used for retrieving host system information:

```
OS: Kali GNU/Linux Rolling x86_64
Host: ROG Strix G513RM_G513RM 1.0
Kernel: 6.11.2-amd64
Uptime: 2 hours, 58 mins
Packages: 4562 (dpkg), 13 (flatpak)
Shell: zsh 5.9
Resolution: 2560x1440
DE: GNOME 46.3.1
Theme: Kali-Dark [GTK2/3]
Icons: Adwaita+Flat-Remix-Blue [GTK2/3]
Terminal: vscode
CPU: AMD Ryzen 7 6800H with Radeon Graphics (16) @ 4.785GHz
GPU: AMD ATI Radeon 680M
GPU: NVIDIA GeForce RTX 3060 Mobile / Max-Q
Memory: 10445MiB / 15249MiB
```

2 Cost of Prime Ordered Group Operations

2.1 Implementation

This section outlines the evaluation of the cost of multiplication and exponentiation in prime-ordered groups, given a modulus and group order. First of all, it is important for us to be able to construct a prime ordered group; for this, we will be using NTL library, which in its implementation uses GMP library. In the following is an algorithm that makes it possible for us to do so.

ALGORITHM 8.65
A group-generation algorithm $\mathcal{G}$

Input: Security parameter 1^n, parameter $\ell = \ell(n)$
Output: Cyclic group $\mathbb{G}$, its (prime) order q, and a generator g

generate a uniform n-bit prime q
generate an ℓ-bit prime p such that $q \mid (p-1)$
// we omit the details of how this is done
choose a uniform $h \in \mathbb{Z}_p^*$ with $h \neq 1$
set $g := [h^{(p-1)/q} \bmod p]$
return p, q, g // $\mathbb{G}$ is the order-q subgroup of $\mathbb{Z}_p^*$

Figure 1: Group generation algorithm taken from INTRODUCTION TO MODERN CRYPTOGRAPHY

Now we examine each part of implementing this algorithm. Below is the function used to find a generator of a cyclic group g, given a modulus p and order q.

```
bool find_generator(const ZZ& p, const ZZ& q, ZZ& g, long max_attempts = 1000) {
    ZZ exponent = (p - 1) / q;

    for (long attempt = 0; attempt < max_attempts; ++attempt) {
        ZZ h = RandomBnd(p);
        if (h == 1 || h==0) continue;
        PowerMod(g, h, exponent, p);
        if (g != 1) {
            return true;
        }
    }
    return false;
}
```

It is interesting to pay attention to the GMP-like style of implementing functions, where values are assigned to variables through references and not returned from the function.

Now we take a look at the function that constructs a prime ordered group. It first generates prime p and q such that p = k * q + 1 and then generates a generator g using the previous function. It is important to realize that this algorithm makes k such that the p has most significant bit set, so that its bit representation is actually modulus bits long and not less.

```
bool generate_prime_order_group(ZZ& p, ZZ& q, ZZ& g,
    unsigned int modulus_bits, unsigned int order_bits,
    unsigned int max_attempts = 10000) {
    q = GenPrime_ZZ(order_bits);
    for (int attempt = 0; attempt < max_attempts;
    attempt++) {
        ZZ lower_bound = ZZ(1) << (modulus_bits - 1);
        ZZ rem;
        DivRem(lower_bound, rem, lower_bound, q);
        if (rem != 0) lower_bound++;
        unsigned int k_bits = modulus_bits - NumBits(q);
        if (k_bits <= 0) return false;
        ZZ k = RandomLen_ZZ(k_bits);
        if (k < lower_bound)
            k = lower_bound + RandomLen_ZZ(k_bits / 2);

        p = k * q + 1;
        if (NumBits(p) != modulus_bits)
            continue;
        if (ProbPrime(p, 25)){
            if (find_generator(p, q, g)) return true;
            else return false;
        }
    }
    return false;
}
```

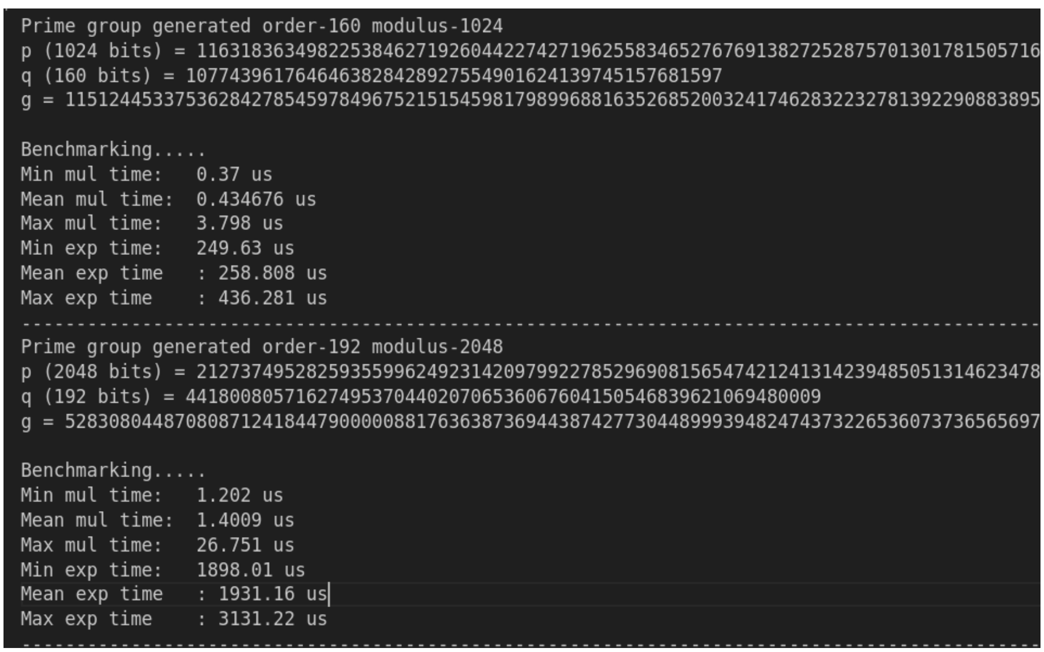
Rest of the implementation just calls this function to create prime order groups of different security levels and then iterates creating different elements of these groups and captures times of operations and finds minimum, maximum and mean cpu times. Whole code is available in GitHub repository.

2.2 Results

For running and compiling next two commands are needed

```
//for compiling and linking with ntl and gmp libraries
g++ -o benchmark benchmark.cpp -lntl -lgmp
//for running
./benchmark
```

The test was done for 10000 operations for each prime ordered group and results are provided below. The shortened output example of results displays only few digits of the parameters, the full version is in GitHub repository:



```
Prime group generated order-160 modulus-1024
p (1024 bits) = 116318363498225384627192604422742719625583465276769138272528757013017815057161
q (160 bits) = 1077439617646463828428927554901624139745157681597
g = 115124453375362842785459784967521515459817989968816352685200324174628322327813922908838951

Benchmarking.....
Min mul time:   0.37 us
Mean mul time:  0.434676 us
Max mul time:   3.798 us
Min exp time:   249.63 us
Mean exp time   : 258.808 us
Max exp time    : 436.281 us
-----------------------------------------------------------------------------------------
Prime group generated order-192 modulus-2048
p (2048 bits) = 212737495282593559962492314209799227852969081565474212413142394850513146234781
q (192 bits) = 441800805716274953704402070653606760415054683962106948000
g = 528308044870808712418447900000881763638736944387427730448999394824743732265360737365656971

Benchmarking.....
Min mul time:   1.202 us
Mean mul time:  1.4009 us
Max mul time:   26.751 us
Min exp time:   1898.01 us
Mean exp time   : 1931.16 us
Max exp time    : 3131.22 us
-----------------------------------------------------------------------------------------
```

Figure 2: Results first part

```
------------------------------------------------------------------------------------------
Prime group generated order-256 modulus-2048
p (2048 bits) = 2323701764181823911238787950422671187952829887807383287218649431084723419135
q (256 bits) = 86385302385479505956956475220737044520099656377543168484581674028595967576189
g = 17663780216129301843353775025951714191519991334476043527822914723246006606980868305129389

Benchmarking.....
Min mul time:   1.202 us
Mean mul time:  1.3468 us
Max mul time:   14.697 us
Min exp time:   1897.7 us
Mean exp time   : 1926.77 us
Max exp time    : 2676.98 us
------------------------------------------------------------------------------------------
Prime group generated order-256 modulus-4096
p (4096 bits) = 5221944407065762533458763553583121912899821245236918901921167416419769539857
q (256 bits) = 60744965116500588958947138331944108075343697215080085214082957320820266784197
g = 36737956789361523405260492951346961297752716246711329282883709834468266042916794479572928

Benchmarking.....
Min mul time:   3.727 us
Mean mul time:  4.39582 us
Max mul time:   50.385 us
Min exp time:   14146.7 us
Mean exp time   : 14303.6 us
Max exp time    : 17666.3 us
```

Figure 3: Results second part

3 Cost of Elliptic Curve Operations

3.1 Implementation

The Elliptic curves that we are going to focus on are:

- prime256v1 - 128-bit sec, prime field, general use (NIST P-256)
- secp256k1 - 128-bit sec, prime field, used in Bitcoin/Ethereum
- secp384r1 - 192-bit sec, prime field, stronger TLS/government use
- secp521r1 - 256-bit sec, prime field, high-security niche use
- sect233r1 - 112-bit sec, binary field, used in embedded/hardware

Complete code is available in the repository, below is core function:

```python
REPEAT = 10000
TO_MICRO = 1000000

def benchmark(curve):
    group = ECGroup(curve)

    tuples = [(group.random(G), group.random(G), group.random(G), group.random(ZR)) for _ in range(REPEAT)]
    a, b, x, y = zip(*tuples)
    add_times = []
    mul_times = []

    for i in range(REPEAT):
        start = time.process_time()
        _ = a[i] * b[i]
        add_times.append(time.process_time() - start)

    for i in range(REPEAT):
        start = time.process_time()
        _ = x[i] ** y[i]
        mul_times.append(time.process_time() - start)

    return {
        "curve" : curve,
        "add_min": min(add_times),
        "add_max": max(add_times),
        "add_avg": sum(add_times) / len(add_times),
        "mul_min": min(mul_times),
        "mul_max": max(mul_times),
        "mul_avg": sum(mul_times) / len(mul_times),
    }
```

Figure 4: Elliptic curve benchmark function

3.2 Results

The test was done for 10000 operations for each elliptic curve, and the results are below.

```
Starting benchmark of operations over elliptic curves 10000 operations per test
-----------------------------------------------------------------------------------------------
Benchmarking 'prime256v1'...
Benchmarking 'secp256k1'...
Benchmarking 'secp384r1'...
Benchmarking 'secp521r1'...
Benchmarking 'sect233r1'...

=====================================================================================
Benchmarking cost of point addition
=====================================================================================
Curve Name      | Min CPU (µs)    | Max CPU (µs)    | Average CPU (µs)
-------------------------------------------------------------------------------------
prime256v1      | 1.02            | 6.77            | 1.16
secp256k1       | 1.02            | 6.97            | 1.20
secp384r1       | 1.37            | 5.93            | 1.59
secp521r1       | 1.50            | 8.78            | 1.88
sect233r1       | 6.14            | 17.66           | 6.90
-------------------------------------------------------------------------------------

=====================================================================================
Benchmarking cost of scalar multiplication
=====================================================================================
Curve Name      | Min CPU (µs)    | Max CPU (µs)    | Average CPU (µs)
-------------------------------------------------------------------------------------
prime256v1      | 32.45           | 353.54          | 33.02
secp256k1       | 265.77          | 452.45          | 270.04
secp384r1       | 277.53          | 409.10          | 280.73
secp521r1       | 221.06          | 348.43          | 225.42
sect233r1       | 177.49          | 344.03          | 183.00
```

Figure 5: Results of benchmarking elliptic curves

4 Cost of Elgamal Scheme

4.1 Implementation

As with elliptic curve operation benchmarks, the same curves are used for benchmarking ElGamal encryption and decryption, and only core function is provided here with rest in GitHub repository:

```
def random_message(curve_group) -> bytes:
    return b'\x00'+ os.urandom(curve_group.bitsize() - 1)


def benchmark(curve):
    group = ECGroup(curve)
    elgamal = ElGamal(group)
    public, private = elgamal.keygen()
    encryption_times = []
    decryption_times = []

    for i in range(REPEAT):
        message = random_message(group)
        start = time.process_time()
        cipher = elgamal.encrypt(public, message)
        encryption_times.append(time.process_time() - start)
        start = time.process_time()
        decrypted = elgamal.decrypt(public, private, cipher)
        decryption_times.append(time.process_time() - start)

    return {
        "curve" : curve,
        "enc_min": min(encryption_times),
        "enc_max": max(encryption_times),
        "enc_avg": sum(encryption_times) / len(encryption_times),
        "dec_min": min(decryption_times),
        "dec_max": max(decryption_times),
        "dec_avg": sum(decryption_times) / len(decryption_times),
    }
```

Figure 6: Function for benchmarking ElGamal in Charm library

4.2 Results

The test was done for 10000 operations for each elliptic curve, and the results are below.

```
Starting benchmark of elgamal encryption and decryption scheme with 10000 operations per test
---------------------------------------------------------------------------------------------
Benchmarking 'prime256v1'...
Benchmarking 'secp256k1'...
Benchmarking 'secp384r1'...
Benchmarking 'secp521r1'...
Benchmarking 'sect233r1'...

=====================================================================================
Encryption cost benchmark
-------------------------------------------------------------------------------------
Curve Name      | Min CPU (µs)    | Max CPU (µs)    | Average CPU (µs)
-------------------------------------------------------------------------------------
prime256v1      | 83.09           | 246.55          | 96.67
secp256k1       | 546.92          | 1372.99         | 577.77
secp384r1       | 611.46          | 1291.45         | 651.50
secp521r1       | 522.21          | 1272.32         | 595.10
sect233r1       | 390.88          | 964.67          | 431.47
-------------------------------------------------------------------------------------

=====================================================================================
Decryption cost benchmark
-------------------------------------------------------------------------------------
Curve Name      | Min CPU (µs)    | Max CPU (µs)    | Average CPU (µs)
prime256v1      | 37.23           | 62.48           | 37.87
secp256k1       | 276.40          | 592.18          | 285.74
secp384r1       | 304.27          | 552.97          | 311.08
secp521r1       | 238.98          | 472.68          | 245.41
sect233r1       | 185.20          | 374.97          | 192.86
-------------------------------------------------------------------------------------
```

Figure 7: Results of benchamarking ElGamal in Charm library

References

5 References

[1] *Documentation of GNU Multiple Precision Library*; available online:
https://gmplib.org/manual/

[2] *Documentation of NTL: A Library for doing Number Theory*; available online:
https://libntl.org/

[3] *Charm framework documentation*; available online:
https://jhuisi.github.io/charm/

[4] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*, Second Edition. Chapman and Hall/CRC, 2015.