RSA Cryptographic Implementation and Performance Analysis

CRS2404

August 11, 2025

1 Problem Statement

This assignment requires implementing the RSA public-key cryptographic algorithm in C on a Linux-based system, with a focus on measuring performance in CPU clock cycles. The GNU MP (GMP) library's `mpz_t` data type is used for large integer arithmetic, and the code is compiled with `gcc`. The tasks are:

Step 1: Prime Number Generation

- Generate two 512-bit prime numbers p and q using GMP's `mpz_t` functions, uniformly at random.
- Repeat this process 1,000,000 times, recording clock cycles for each iteration.
- Compute and output the minimum, maximum, and average clock cycles.

Step 2: Compute RSA Modulus and Euler's Totient

- Compute $N = p \times q$.
- Compute $\phi(N) = (p - 1) \times (q - 1)$.
- Record clock cycles for these computations.

Step 3: Public and Private Key Generation

- Use a fixed public exponent $e = 2^{16} + 1 = 65537$.
- Compute the private key d such that $e \cdot d \equiv 1 \pmod{\phi(N)}$.
- Record clock cycles for computing d.

Step 4: Message Encryption and Decryption

- Generate a 1023-bit random message m.
- Encrypt: $c = m^e N$.

- Decrypt: $m' = c^d N$.
- Verify $m' = m$.
- Record clock cycles for encryption and decryption.

Step 5: Repeat with Larger Key Sizes

- Repeat Steps 1–4 for 768-bit and 1024-bit primes.
- Produce datasets for 512-bit, 768-bit, and 1024-bit key sizes.

2 Implementation Methodology

I implemented the RSA algorithm in C, using the GMP library for efficient large integer operations. Clock cycles were measured with the `rdtsc` instruction via inline assembly, providing high-precision timing on x86_64 processors. To optimize Step 1's prime generation (1,000,000 iterations per bit size), I used OpenMP for parallel processing.

The program was compiled with:

```
gcc -O3 -fopenmp rsa.c -o rsa -lgmp -lm
```

- `gcc`: The GNU C compiler.
- `-O3`: Enables aggressive optimizations (e.g., loop unrolling, inlining) to reduce runtime.
- `-fopenmp`: Activates OpenMP for parallelizing loops.
- `rsa.c`: The source file.
- `-o rsa`: Names the executable `rsa`.
- `-lgmp`: Links GMP for `mpz_t` operations.
- `-lm`: Links the math library for GMP dependencies.

The program was executed with:

```
nice -n 10 taskset -c 0,1 bash -c "time ./rsa"
```

- `nice -n 10`: Sets a low priority (+10) to minimize impact on tasks like browsing or LaTeX editing.
- `taskset -c 0,1`: Pins the program to two performance cores (0 and 1) on the i5-12450H, leaving two performance cores (2–3) and four efficiency cores (4–7) free.
- `bash -c "time ./rsa"`: Uses a Bash subshell to run the shell's `time` command, measuring wall-clock (`real`), user, and system time.

Parallel processing was implemented for Step 1 using OpenMP with two threads, set via:

```
omp_set_num_threads(2);
```

This distributes the 1,000,000 iterations across two threads, leveraging the i5-12450H's performance cores to reduce wall-clock time by 1.5–1.7x compared to serial execution. I chose two threads to balance speed and system responsiveness, ensuring other applications ran smoothly.

To track progress and avoid data loss, I used a batching strategy for Step 1, processing 1000 iterations per bit size (512, 768, 1024) in each batch, repeated 1000 times to reach 1,000,000 iterations. Results were saved after each batch in `results/result_<bit_size>_run_<i>.txt`, where `i` is the cumulative iteration count (1000, 2000, ..., 1000000). The batch loop is:

```
for (int batch = 1; batch <= num_batches; batch++) {
    for (int b = 0; b < num_bit_sizes; b++) {
        int bit_size = bit_sizes[b];
        // Parallel loop for 1000 iterations
        #pragma omp parallel num_threads(2) reduction(min:batch_min) ...
        // Save to results/result_<bit_size>_run_<count>.txt
    }
}
```

This allowed monitoring progress and recovering partial results if interrupted.

Prime generation used `mpz_probab_prime_p` with 5 Miller-Rabin repetitions:

```
is_prime_p = mpz_probab_prime_p(p, 5);
```

This balances speed and accuracy (error probability ¡1/1024). Steps 2–4 used GMP's `mpz_mul`, `mpz_sub_ui`, `mpz_invert`, and `mpz_powm_sec` for efficient operations.

3 Output Datasets

The results were saved in `results/` and printed to the console.

3.1 Step 1: Prime Number Generation

For 512-bit primes (1,000,000 iterations):

- Minimum clock cycles: 769,626
- Maximum clock cycles: 250,034,701
- Average clock cycles: 6,761,931.78

For 768-bit primes (1,000,000 iterations):

- Minimum clock cycles: 2,043,210
- Maximum clock cycles: 393,817,557
- Average clock cycles: 24,958,071.37

For 1024-bit primes (1,000,000 iterations):

- Minimum clock cycles: 4,332,638
- Maximum clock cycles: 1,093,673,600
- Average clock cycles: 66,574,448.80

3.2 Steps 2–4: RSA Computations

Cycle counts for Steps 2–4 are in Table 1.

Step	512-bit	768-bit	1024-bit
Step 2: N and $\phi(N)$ (cycles)	5,618	1,485	2,359
Step 3: Compute d (cycles)	15,961	4,786	7,032
Step 4: Encryption (cycles)	63,600	100,927	173,280
Step 4: Decryption (cycles)	727,766	2,000,828	4,748,965
Verification	Successful	Successful	Successful

Table 1: Clock cycles for Steps 2–4 across key sizes

All verifications confirmed $m' = m$, validating the implementation.

4 Detailed Analysis

The results show RSA's computational scaling with key size, with Step 1 dominating due to its 1,000,000 iterations.

4.1 Step 1: Prime Generation

Average clock cycles per prime pair increase with bit size:

- 512-bit: 6,761,931.78 cycles.
- 768-bit: 24,958,071.37 cycles.
- 1024-bit: 66,574,448.80 cycles.

At 4.4 GHz (4,400,000,000 cycles/second):

$$Time(s) = \frac{AverageCyclesperPair \times 1,000,000}{Frequency}$$

- 512-bit: $6,761,931.78 \div 4,400,000,000 \approx 0.001537\, s/pair \times 1,000,000 \approx 1,537\, s(25min37sec)$.
- 768-bit: $24,958,071.37 \div 4,400,000,000 \approx 0.005677\, s/pair \times 1,000,000 \approx 5,677\, s(94min37sec)$.
- 1024-bit: $66,574,448.80 \div 4,400,000,000 \approx 0.015130\, s/pair \times 1,000,000 \approx 15,130\, s(252min10sec)$.

Total serial time for Step 1: $1,537 + 5,677 + 15,130 \approx 22,344\, s(6hr12min24sec)$. With 2 threads and 35% overhead:

$$TotalStep1time \approx \frac{22,344}{2} \times 1.35 \approx 15,083\, s(4hr11min23sec).$$

4.2 Steps 2–4

Steps 2–4 are negligible:

- 512-bit: $(5,618 + 15,961 + 63,600 + 727,766) \div 4,400,000,000 \approx 0.000188\,s$.
- 768-bit: $(1,485 + 4,786 + 100,927 + 2,000,828) \div 4,400,000,000 \approx 0.000472\,s$.
- 1024-bit: $(2,359 + 7,032 + 173,280 + 4,748,965) \div 4,400,000,000 \approx 0.001110\,s$.
- Total: $0.000188 + 0.000472 + 0.001110 \approx 0.00177\,s$.

Total estimated execution time: 15,083 s (4 hr 11 min 23 sec), dominated by Step 1.

The high maximum cycles (e.g., 1,093,673,600 for 1024-bit) reflect rare cases where many candidates were tested before finding a prime.

5 System Specifications

- **CPU**: 12th Generation Intel® Core™ i5-12450H (E-cores up to 3.30 GHz, P-cores up to 4.40 GHz).
- **RAM**: 16 GB SO-DIMM DDR4 3200. **Operating System**: Kali Rolling (2024.4) x64, running in VirtualBox.
- **Compiler**: gcc (Debian 14.2.0-8) 14.2.0.

6 Observations

This assignment was a real eye-opener about RSA's computational demands. Step 1's prime generation was the biggest hurdle, needing 1,000,000 iterations for each key size. I tackled this by batching iterations into 1000 groups of 1000 iterations per bit size (512, 768, 1024), saving results after each batch in files like `result_512_run_1000.txt`. This let me keep tabs on progress and ensured I wouldn't lose everything if the program crashed or was interrupted. It was especially useful for debugging and understanding how far along the program was.

Using OpenMP with two threads was a game-changer for Step 1. By splitting the work across two performance cores, I cut the runtime significantly compared to running it all on one core. The `taskset -c 0,1` command kept the program on cores 0 and 1, leaving the rest of my CPU free for other tasks like browsing or editing LaTeX, and `nice -n 10` made sure it didn't hog resources. The estimated runtime of 4 hours, 11 minutes, 23 seconds, based on cycle counts, seems long but makes sense in VirtualBox, where overhead can slow things down.

Steps 2–4 were quick, with decryption taking the most cycles because of the large private key d. The successful verifications gave me confidence in the math, and GMP's functions made the big integer stuff surprisingly painless. The cycle counts show how RSA's cost skyrockets with larger keys—1024-bit primes took about 10x more cycles than 512-bit. This project really drove home why optimization matters in cryptography, and I'm glad I got it working smoothly in the end.

7 External Sources

- GNU MP (GMP) library documentation: For `mpz_t` functions and prime generation (`https://gmplib.org/`).
- Stack Overflow: Guidance on `rdtsc` and OpenMP usage (various threads on inline assembly and parallel loops).
- OpenMP documentation: For thread management and parallel directives (`https://www.openmp.org/`).
- Grok by xAI: Used to understand sample code for parallel processing with OpenMP, GMP, and `mpz_t` operations.
- Linux man pages: For `nice`, `taskset`, and `time` usage.