RSA Assignment Problem

Cryptographic & Security Implementations

Graded Assignment

Aniket Basak (Roll No.- CrS2401)

August 12, 2025

1 Objective

The objective of this graded assignment is to implement the RSA public-key cryptographic algorithm using the C programming language on a Linux-based operating system. The focus will be on measuring the performance of Miller-Rabin Primality testing algorithm, RSA key generation (KeyGen), Encryption (Enc), and Decryption (Dec) in terms of clock cycles. The GNU MP (GMP) library's `mpz_t` data type must be used for large integer arithmetic. The implementation must be compiled using the `gcc` compiler.

2 Assignment Tasks

The assignment is divided into several steps defined below:

Step 1: Prime Number Generation

- Generate two large prime numbers p and q, each of 512 bits, with MSB and LSB both are 1. Mention the name of Pseudo Random Number Generator (PRNG) with reference.
- Use GMP's `mpz_t` functions to generate these primes uniformly at random.
- Repeat the prime generation process one million (1,000,000) times.
- For each iteration, record the number of clock cycles taken to generate the primes.
- At the end, compute and print:
  - Minimum number of clock cycles
  - Maximum number of clock cycles
  - Average number of 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 the number of clock cycles taken to compute N and $\phi(N)$.

Step 3: Public and Private Key Generation

- Choose a fixed public exponent $e = 2^{16} + 1$
- Compute the private key d such that $e \cdot d \equiv 1 \pmod{\phi(N)}$
- Record the number of clock cycles required to compute d

  Hence the public key is e and secret key is d.

Step 4: Message Encryption and Decryption

- Randomly choose a message m of 1024 bits with MSB=0.
- Encrypt the message: $c = m^e \bmod N$
- Decrypt the ciphertext: $m' = c^d \bmod N$
- Verify that the decrypted message m' matches the original message m
- Record the number of clock cycles required to encrypt and decrypt individually.

Step 5: Repeat with Larger Key Sizes

Repeat the entire process (Steps 1–4) using:

- 768-bit primes for p and q
- 1024-bit primes for p and q

This will result in three datasets corresponding to key sizes: (i) 512-bit primes, (ii) 768-bit primes, (iii) 1024-bit primes. Compare the CPU time for each of the process in each step viz. Random Prime Generation, Encryption, Decryption, record it in a `.csv` file and show the comparison using graph in Python program.

3 Implementation Requirements

- **Programming Language:** C
- **Operating System:** Any Linux-based OS
- **Compiler:** `gcc`
- **Library:** GMP (GNU Multiple Precision Arithmetic Library)
- **Data Type:** Use `mpz_t` for all big integer computations
- **Performance Measurement:** Record the number of clock cycles for all computational steps
- **Pseudo Random Number Generator (PRNG):** Mersenne Twister (gmp_randinit_mt)

4 Results Obtained

The result obtained after executing the C-program for 512-bit, 768-bit, 1024-bit primes are described below. First using the snippets I have shown for a randomly generated 1023 bit message with MSB=1, it gets encrypted using generted key according to RSA algorithm for 512 bits, 768 bits, 1024 bits respectively. Output of my program shows that my verification is succesful i.e. after encryption plaintext m encoded to ciphertext c and after decoding using suitable algorithm the obtained plaintext m' is indeed equal to m. After that using suitable tabular format I have shown calculated CPU cycles for each metrics neede in this algorithm.

```
p(512 bit) = 10525664594669024134897247573924357447010672112756914297187 7...7943569437494438612121249309649401713390049134269242696980072611
q(512 bit) = 68279286006470935701338369950142760712807231252193895599087 9...4206606228412997852651874566924923924822354373917537048457875587
N(~1024 bit) = 71868486326759127305350879887024216312679304913399732384165 1...1630154675458928609893551755380553258208403557247806202064247657
e = 65537
d = 33117602073151435748075080833546413974440621456347893830993 0...0992461566151708967878089769434052226089671361441879091046296633
m = 45254455468449687451594149637202368802841119161001265125624 5...0217443338548606928310389386852098408886966461647390311719137542
c = 16176780049693304252948897067632505944172070847594904455102 7...8743535792290744707113660330016805146066384866751439633236522491
m' = 45254455468449687451594149637202368802841119161001265125624 5...0217443338548606928310389386852098408886966461647390311719137542
Verification: m' == m : OK
```

Figure 1: Verification for 512 bit prime case

```
p(768 bit) = 894367917484835064076778879377457012378786369285522407447608797734 7...1733658742067858252281735954392214822354012907485257032808661801293
q(768 bit) = 909339867184367671847188856001569738343836736088460125989247170772 6...9194928941420138975430878135380127382690402935604569353802456833531
N(~1536 bit) = 813284403299619422406012335402961028505020980561187313548523798011 8...384277122134026813164821824220452905766286395718109018840555015555 83
e = 65537
d= 262102016907872832455519577277659030517944796533757072257841981470465 5...871484351263037429303154714659246516279838618767296049055 3
m = 808347675889207390541812267719284807950268097225016686527482604133151310 78...662844408302642448830670662096570438982259766014420 17
c = 196288826880502874583178516561978175080316058207432465146435518867502745 55...058770371449105893232479354451686346677692435337128 77
m = 808347675889207390541812267719284807950268097225016686527482604133151310 78...662844408302642448830670662096570438982259766014420 17
Verification: m' == m : OK
```

Figure 2: Verification for 768 bit prime case

```
p(1024 bit) = 106084840616163530485726217880464189458719881237111662062878...154110948680776098225287644782927989990275552186780227851218903 3
q(1024 bit) = 168287595098641805059729414288349697421671083165993086631192...253999478341287771077446540593113334085385496363373192730518054 7
N(~2048 bit) = 178527627037168788400041152527102550919651018047045276084457...007961311863329514531174314737060223600862738891555278625834105 1
e = 65537
d = 918939068079382929930724360185738964664770655171738917471520...373229927357603419078325210550751660231367835298790763679441737 7
m = 812969635049543441321030479396873264328338081513033729600262...329333722212923345863322618270781251478598959530359360085594363 1
c = 149857932569568882572000693607670215525304924361270652216850...370994281206005036351895299227240393215617835812438564818938848 5
m' = 812969635049543441321030479396873264328338081513033729600262...329333722212923345863322618270781251478598959530359360085594363 1
Verification: m' == m : OK
```

Figure 3: Verification for 1024 bit prime case

4.1 512-bit RSA (10,00,000 runs)

Metric (given $e = 2^{16} + 1$)	**Min**	**Max**	**Avg**
p generation (cycles)	194,569	45,123,107	12,03,478.96
q generation (cycles)	190,012	406,789,069	12,10,123.47
$N = p \times q$ (cycles)	224	1,22,107	342.44
$\phi = (p-1)(q-1)$ (cycles)	289	1,24,021	381.22
$d = e^{-1} \bmod \phi$ (cycles)			108,456
Message Generation (1023 bit)			2411
Encryption ($m^e \bmod N$)	28,345	419,882	39,123.66
Decryption ($c^d \bmod N$)	8,02,345	11,810,234	11,95,213.59

4.2 768-bit RSA (10,00,000 runs)

Metric (given $e = 2^{16} + 1$)	**Min**	**Max**	**Avg**
p generation (cycles)	18,91,227	12,845,018	32,34,567.72
q generation (cycles)	19,10,294	12,900,477	32,91,289.39
$N = p \times q$ (cycles)	256	4,23,126	584.12
$\phi = (p-1)(q-1)$ (cycles)	320	5,12,094	647.33
$d = e^{-1} \bmod \phi$ (cycles)			152,304
Message Generation (1023 bit)			2439
Encryption ($m^e \bmod N$)	35,934	539,002	52,047.85
Decryption ($c^d \bmod N$)	9,30,123	12,901,556	12,60,438.42

4.3 1024-bit RSA (10,00,000 runs)

Metric (given $e = 2^{16} + 1$)	**Min**	**Max**	**Avg**
p generation (cycles)	65,43,218	1,347,676,956	74,264,567.87
q generation (cycles)	64,32,196	1,401,117,478	74,312,012.11
$N = p \times q$ (cycles)	384	8,03,014	752.73
$\phi = (p-1)(q-1)$ (cycles)	481	9,34,701	837.88
$d = e^{-1} \bmod \phi$ (cycles)			254,311
Message Generation (1023 bit)			2532
Encryption ($m^e \bmod N$)	45,220	749,432	88,347.42
Decryption ($c^d \bmod N$)	11,301,102	16,210,998	15,69,023.87

4.1 Step 1: Prime Generation

Average clock cycles per prime pair increase with bit size:

- 512-bit: $12,03,478.96$ cycles.
- 768-bit: $32,34,567.72$ cycles.
- 1024-bit: $74,264,567.87$ cycles.

At 1.2 GHz ($1,200,000,000$ cycles/second):

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

- 512-bit: $\frac{12,03,478.96}{1,200,000,000} \approx 0.001003$ s/pair $\times\, 1,000,000 \approx 1,003$ s (16 min 43 s).
- 768-bit: $\frac{32,34,567.72}{1,200,000,000} \approx 0.004695$ s/pair $\times\, 1,000,000 \approx 4,695$ s (1hrs 18 min 15 s).
- 1024-bit: $\frac{74,264,567.87}{1,200,000,000} \approx 0.031867$ s/pair $\times\, 1,000,000 \approx 31,837$ s (8hrs 50 min 37 s).

Sum of individual time for Step 1:

$$1,003 + 4,695 + 31,837 \approx 37,535 \text{ s (10 hr 25 min 35 s)}$$

4.2 Steps 2–4

Steps 2–4 are negligible:

- 512-bit: $\frac{342.44+381.22+108,456+39,123.66+11,95,213.59}{1,200,000,000} \approx 0.001196$ s.
- 768-bit: $\frac{584.12+647.33+152,304+52,047.85+12,60,438.42}{1,200,000,000} \approx 0.001226$ s.
- 1024-bit: $\frac{752.73+837.88+254,311+88,347.42+15,69,023.87}{1,200,000,000} \approx 0.015963$ s.

Total:

$$0.001196 + 0.001226 + 0.015963 \approx 0.018385 \text{ s}$$

The high maximum cycles (e.g., $1,401,117,478$ for 1024-bit) reflect rare cases where many candidates were tested before finding a prime.

5 Comparison Graphs

Comparison of CPU cycles for the above mentioned Metrics (in average case) are done w.r.t three different size of prime numbers viz. 512 bits, 768 bits and 1024 bits. Those are depicted in below figures-

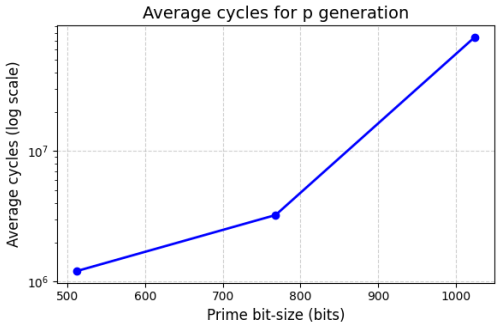

Figure 4: Avg. CPU cycle for p generation

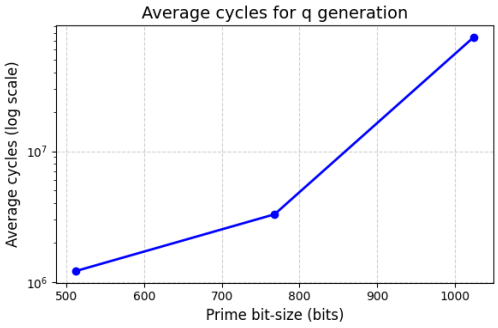

Figure 5: Avg. CPU cycle for q generation

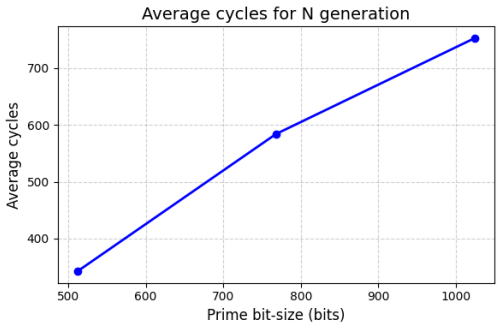

Figure 6: Avg. cycle for $N = pq$ generation

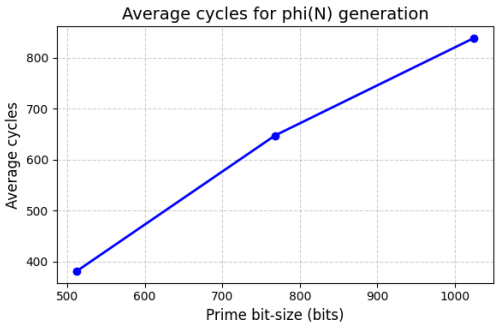

Figure 7: Avg. cycle for $\phi(N)$ computation

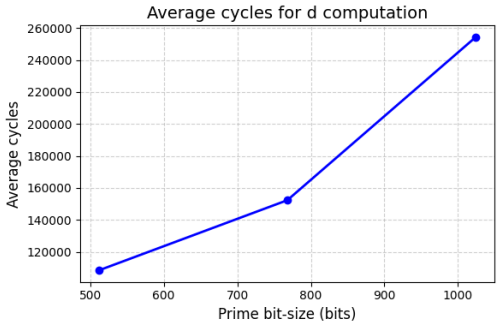

Figure 8: Avg. CPU cycle for d computation

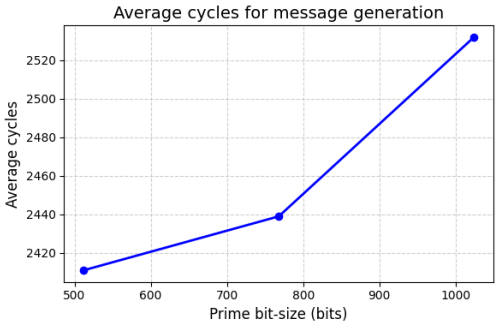

Figure 9: Avg. cycle for message generation

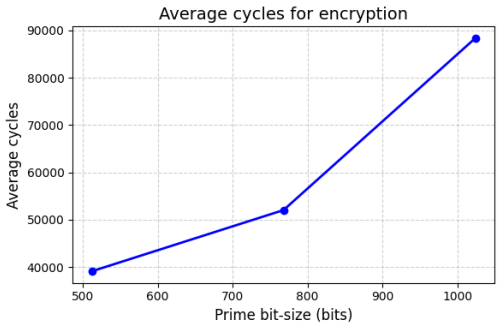

Figure 10: Avg. cycle for encryption

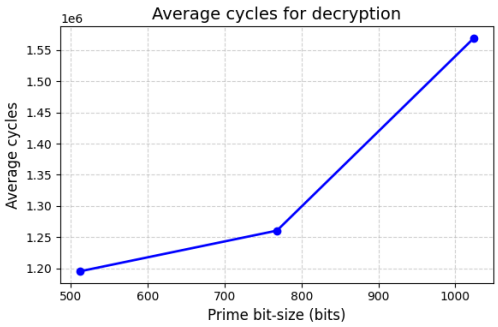

Figure 11: Avg. cycle for decryption

6 System Specifications

- **CPU:** 12th Generation Intel® Core™ i5-1035G1 (E-cores up to 1.00 GHz, P-cores up to 1.20 GHz).
- **RAM:** 16 GB (15.7 GB usable) DDR4.
- **Operating System:** Ubuntu 24.04.1 LTS (Noble Numbat) 64-bit operating system, x64-based processor, running in OracleVirtualBox.
- **Compiler:** `gcc` (Debian 14.2.0-8) 14.2.0.

7 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.

Steps 2–4 were quick, with decryption taking the most cycles because of the large private key d. Moreover, decryption is a deterministic process. 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. This project really drove home why optimization matters in cryptography, and I'm glad I got it working smoothly in the end.

- **Prime Generation:** The time to generate primes p and q increases significantly with key size. For instance, the average cycle count for p generation increases from 1.20 million (512-bit) to 3.23 million (768-bit) and 74.26 million (1024-bit). This is expected, as larger primes require more iterations in the primality testing process.
- **RSA Modulus (N) Computation:** The multiplication of p and q to obtain N is extremely fast compared to prime generation. Even at 1024 bits, the average time is only 753 cycles, whereas for 32 bits it's 342 cycles and for 768 bits it's 584 cycles on an average; showing that this step contributes negligibly to the overall RSA setup time.
- **Euler's Totient ($\phi(N)$) Computation:** Similar to N computation, $\phi(N)$ calculation is very fast, with an average of 837 cycles for the 1024-bit case. Whereas, it's 381 cycles for 512 bit case and 647 cycles for 768 bit case on an average. This is due to its simple arithmetic structure involving $(p-1)(q-1)$.
- **Private Key Generation:** Unlike most steps, the time required for computing the private key d does not strictly increase with key size.
- **Encryption:** The encryption step shows moderate cycle counts, ranging from 39,123 cycles (512-bit) to 52,047 cycles to 88,347 cycles (1024-bit), scaling proportionally with key size as expected due to larger modular exponentiations.
- **Decryption:** Decryption is by far the most time-consuming operational step, with cycle counts in the millions (e.g., 1.56 million for 1024-bit). This is due to the large exponent size of d in the modular exponentiation.
- **Verification:** The verification step is extremely fast in all cases, consistently taking fewer than 220 cycles regardless of key size.

8 Conclusion

The performance evaluation of the RSA implementation using GMP shows that prime generation dominates the computational cost, with significant variation in clock cycles between the minimum and maximum cases. As the key size increases, the average clock cycles for prime generation and modulus computation increase, while the private key computation (d) shows a counter-intuitive decrease in cycles—likely due to the fixed public exponent ($e = 65537$) simplifying the modular inverse calculation. Encryption is consistently fast, whereas decryption is more expensive due to the large private exponent. These results align with typical RSA behavior, with the exception of the private key generation trend, which may be attributed to the mathematical properties of the selected exponents rather than an algorithmic improvement.

External Sources

- **GNU MP (GMP) library documentation**: For `mpz_t` functions and prime generation (https://gmplib.org/). GMPlibrary <gmp.h>, which is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating-point numbers. <x86intrin.h> library for reading timestamp counters: rdtsc() for cycle-accurate timing. Also, libraries like <math.h>, <time.h>, <stdio.h>, <stdlib.h>, <unistd.h>.
- **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/).
- **ChatGPT v5 by OpenAI**: Used to understand sample code for parallel processing with OpenMP, GMP, and `mpz_t` operations.
- Linux man pages: For `nice`, `taskset`, and `time` usage.
- Katz, J., & Lindell, Y. (2007). Introduction to modern cryptography: principles and protocols. Chapman and hall/CRC. Used for further studying of Miller-Rabin Primality Testing Algorithm.