HW05: Random Number Generation Analysis

Simone Nolé
1940213

November 27, 2025

1 Introduction

The goal of this assignment is to implement and compare different Deterministic Random Bit Generators (DRBG). Specifically, I focused on constructing three Cryptographically Secure PRNGs (CS-PRNG) and comparing them each other. The algorithms chosen for this analysis are:

1. **Blum Blum Shub (BBS)**: Based on the quadratic residuosity problem.
2. **AES-CTR**: Based on symmetric cryptography (block cipher).
3. **RSA Generator**: Based on the factorization problem.

The comparison focuses on execution time, space complexity, and the statistical distribution of the generated bits.

2 Implementation Details

The algorithms were implemented in Python 3 using the standard library for math operations and the `cryptography` library for the AES implementation.

2.1 Blum Blum Shub (BBS)

This generator relies on the difficulty of finding square roots modulo a composite number $n = p \cdot q$.

- **Parameters:** I selected two large primes p and q such that $p \equiv q \equiv 3 \pmod 4$.
- **Equation:** $x_{i+1} = x_i^2 \pmod n$.
- **Output:** The least significant bit of x_{i+1}.

Since Python handles arbitrarily large integers automatically, the implementation is straightforward but expected to be computationally heavy due to the modular exponentiation at every step.

2.2 RSA Generator

Similar to BBS, but it uses the RSA encryption primitive.

- **Equation:** $z_{i+1} = z_i^e \pmod{n}$.
- **Exponent:** I used the standard public exponent $e = 65537$.

This is theoretically similar to BBS but uses a much larger exponent (65537 vs 2), which suggests it might be slower per iteration.

2.3 AES in Counter Mode ($\mathbf{CTR}_D RBG$)

This is a standard approach used in real-world applications (NIST SP 800-90A).

- **Logic:** I used AES-256. The generator encrypts a counter (or a sequence of inputs) to produce a keystream.
- **Efficiency:** Unlike the number-theoretic approaches above, AES operates on blocks of 128 bits at a time using bitwise operations (XOR, shifts).

3 Results and Analysis

Tests were performed for sequence lengths ranging from 10^3 to 10^6 bits. The results regarding time and randomness are discussed below.

3.1 Time Complexity

The execution time is the most significant differentiator between the algorithms.

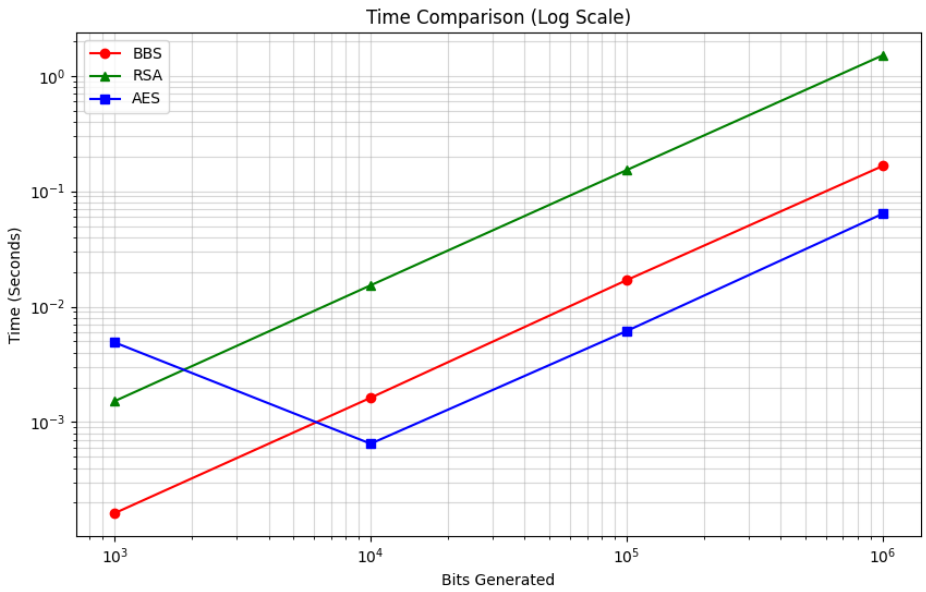

Figure 1: Time Comparison (Log Scale). BBS (Red), RSA (Green), AES (Blue).

As shown in Figure 1, the results clearly separate the algorithms into two categories:

1. **Number Theoretic (BBS, RSA):** These are significantly slower. The complexity grows linearly with the number of bits, but the constant factor is high. RSA (Green) is slower than BBS (Red) because computing $x^{65537} \pmod{n}$ is more expensive than $x^2 \pmod{n}$.

2. **Symmetric Crypto (AES):** This is orders of magnitude faster for large sequences. However, there is an interesting anomaly at the beginning of the graph (10^3 bits).

Observation on Initialization Overhead: At 10^3 bits, AES appears slower than BBS. This is likely due to the "cold start" overhead of loading the cryptographic libraries (OpenSSL backend) and setting up the Cipher object in Python. Once this fixed cost is amortized over longer sequences ($> 10^4$ bits), AES becomes extremely fast.

3.2 Randomness Quality (Bias)

To verify the quality of the generators, I calculated the ratio of 0s in the output sequences. For a good random source, this should be close to 0.5.

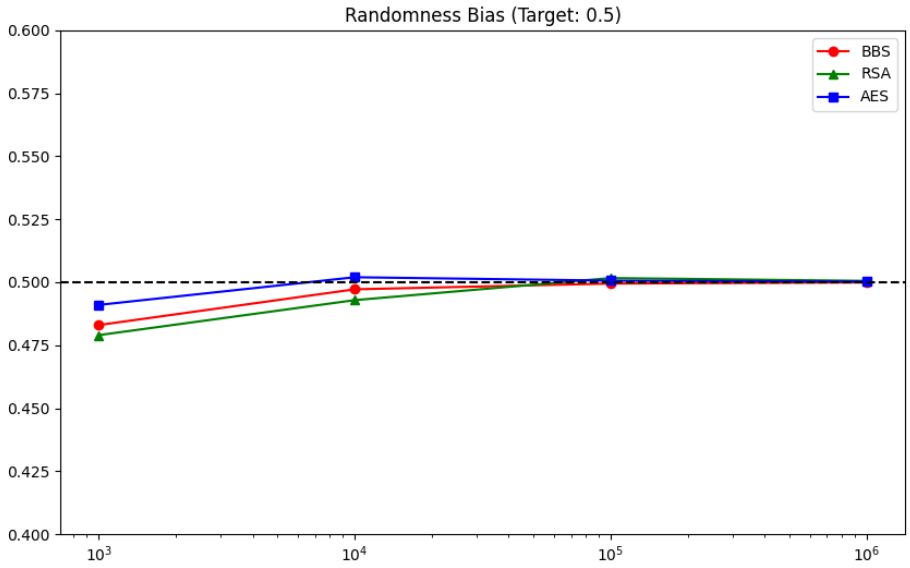

Figure 2: Ratio of Zeros generated by each algorithm.

All three algorithms produced a ratio very close to 0.5 (oscillating between 0.49 and 0.51), satisfying the basic requirement for a pseudo-random generator. No obvious bias was detected in the generated samples.

3.3 Space Complexity Analysis

The analysis of space complexity requires distinguishing between the storage of the generated sequence and the internal state of the generator.

3.3.1 Output Storage (Empirical Results)

As shown in the generated graph (`result_space.png`), the memory usage grows linearly ($O(N)$) for all three algorithms. This is expected because the assignment requires storing the full sequence of length N in memory to perform the statistical analysis. Since Python stores integers (0s and 1s) in dynamic lists, the memory footprint is identical for BBS, RSA, and AES sequences of the same length.

3.3.2 Internal State (Theoretical Analysis)

A more significant comparison concerns the memory required to maintain the generator's *state* between iterations ($O(1)$ constant space):

- **AES-CTR State:** Requires storing the 256-bit Key (32 bytes) and the 128-bit Nonce/Counter (16 bytes). The total state is extremely small (≈ 48 bytes).
- **BBS and RSA State:** These require storing the current residue x_i and the modulus n. For secure parameters, n is typically 2048 or 4096 bits. While still $O(1)$ relative to the output length, the state of Number-Theoretic generators is significantly larger (in terms of bits) than the compact state of a symmetric cipher like AES.

Conclusion on Space: While the output storage is identical, AES is more memory-efficient in terms of internal state, making it more suitable for constrained environments (e.g., embedded systems or smart cards) compared to BBS/RSA.

4 Conclusion

The experiment confirms the trade-off between performance and theoretical hardness. **BBS and RSA** offer security based on difficult number theory problems (factorization) but are too slow for high-throughput applications (like encrypting a disk). **AES-CTR** is the practical choice: it is secure (assuming AES is secure) and extremely efficient, except for a negligible initialization cost for very short sequences.

Code (made in Python)

```
import time
import sys
import os
import matplotlib.pyplot as plt

# Prova a importare la libreria crittografica
try:
    from cryptography.hazmat.primitives.ciphers import Cipher, algorithms,
modes
    from cryptography.hazmat.backends import default_backend
except ImportError:
    print("ERRORE CRITICO: La libreria 'cryptography' non è installata.")
    print("Esegui questo comando nel terminale: pip install cryptography")
    sys.exit(1)


class BBS_Generator:
    def __init__(self, length_bits):
        self.length = length_bits
        self.p = 30000000091 #prime to q such that is congruent to 3 mod 4
        self.q = 40000000003 #prime to p such that is congruent to 3 mod 4
        self.n = self.p * self.q
        self.s = 1234567 #such that GCD(s, n) = 1
        self.state = pow(self.s, 2, self.n)

    def generate(self):
        bits = []
        start_time = time.perf_counter()
        current_x = self.state

        for _ in range(self.length):
            current_x = pow(current_x, 2, self.n)
            bits.append(current_x % 2)

        end_time = time.perf_counter()
        return bits, end_time - start_time


class RSA_Generator:
    def __init__(self, length_bits):
        self.length = length_bits
```

```
        self.p = 30000000091 #prime
        self.q = 40000000003 #prime
        self.n = self.p * self.q
        self.e = 65537 #such that GCD(e, (p-1)•(q-1)) = 1
        self.s = 1234567 #seed
        self.state = self.s

    def generate(self):
        bits = []
        start_time = time.perf_counter()
        current_z = self.state

        for _ in range(self.length):
            current_z = pow(current_z, self.e, self.n) #current_z^e (mod n)
            bits.append(current_z % 2)

        end_time = time.perf_counter()
        return bits, end_time - start_time


class AES_CTR_Generator:
    def __init__(self, length_bits):
        self.length = length_bits
        self.key = os.urandom(32)
        self.nonce = os.urandom(16)

    def generate(self):
        num_bytes = (self.length + 7) // 8
        start_time = time.perf_counter()

        cipher = Cipher(algorithms.AES(self.key), modes.CTR(self.nonce),
backend=default_backend())
        encryptor = cipher.encryptor()
        keystream = encryptor.update(b'\x00' * num_bytes) +
encryptor.finalize()

        bits = []
        for byte in keystream:
            for i in range(8):
                if len(bits) < self.length:
                    bits.append((byte >> i) & 1)

        end_time = time.perf_counter()
        return bits, end_time - start_time
```

```python
def run_analysis():
    lengths = [1000, 10000, 100000, 1000000]

    results = {
        'BBS': {'times': [], 'zeros': [], 'space': []},
        'RSA': {'times': [], 'zeros': [], 'space': []},
        'AES': {'times': [], 'zeros': [], 'space': []}
    }

    print(f"{'Algo':<5} | {'Bits':<10} | {'Time (s)':<12} | {'0s Ratio':
<10}")
    print("-" * 50)

    for l in lengths:
        # BBS
        bbs = BBS_Generator(l)
        bits, t = bbs.generate()
        results['BBS']['times'].append(t)
        results['BBS']['zeros'].append(bits.count(0))
        results['BBS']['space'].append(sys.getsizeof(bits))
        print(f"BBS   | {l:<10} | {t:.8f}   | {bits.count(0)/l:.4f}")

        # RSA
        rsa = RSA_Generator(l)
        bits, t = rsa.generate()
        results['RSA']['times'].append(t)
        results['RSA']['zeros'].append(bits.count(0))
        results['RSA']['space'].append(sys.getsizeof(bits))
        print(f"RSA   | {l:<10} | {t:.8f}   | {bits.count(0)/l:.4f}")

        # AES
        aes = AES_CTR_Generator(l)
        bits, t = aes.generate()

        if t < 1e-9: t = 1e-9
        results['AES']['times'].append(t)
        results['AES']['zeros'].append(bits.count(0))
        results['AES']['space'].append(sys.getsizeof(bits))
        print(f"AES   | {l:<10} | {t:.8f}   | {bits.count(0)/l:.4f}")
        print("-" * 50)

    return lengths, results

def plot_graphs(lengths, results):
    # 1. TIME
    plt.figure(figsize=(10, 6))
```

```
    plt.plot(lengths, results['BBS']['times'], 'o-', color='red',
label='BBS')
    plt.plot(lengths, results['RSA']['times'], '^-', color='green',
label='RSA')
    plt.plot(lengths, results['AES']['times'], 's-', color='blue',
label='AES')
    plt.xlabel('Bits Generated')
    plt.ylabel('Time (Seconds)')
    plt.title('Time Comparison (Log Scale)')
    plt.xscale('log')
    plt.yscale('log')
    plt.grid(True, which="both", ls="-", alpha=0.5)
    plt.legend()
    try:
        plt.savefig('time_plot.png')
        print("Grafico salvato: time_plot.png")
        plt.show()
    except Exception as e:
        print(f"Impossibile mostrare il grafico: {e}")

    # 2. RANDOMNESS (Bias)
    plt.figure(figsize=(10, 6))
    r_bbs = [z/l for z, l in zip(results['BBS']['zeros'], lengths)]
    r_rsa = [z/l for z, l in zip(results['RSA']['zeros'], lengths)]
    r_aes = [z/l for z, l in zip(results['AES']['zeros'], lengths)]

    plt.plot(lengths, r_bbs, 'o-', color='red', label='BBS')
    plt.plot(lengths, r_rsa, '^-', color='green', label='RSA')
    plt.plot(lengths, r_aes, 's-', color='blue', label='AES')
    plt.axhline(0.5, color='black', linestyle='--')
    plt.ylim(0.40, 0.60)
    plt.xscale('log')
    plt.title('Randomness Bias (Target: 0.5)')
    plt.legend()
    plt.savefig('randomness_plot.png')
    plt.show()

if __name__ == "__main__":
    lengths, data = run_analysis()
    plot_graphs(lengths, data)
```