Homework 05: Secure Pseudo-Random Number Generators

Nicolas Leone
Student ID: 1986354
Cybersecurity

November 22, 2025

Contents

1 Introduction

Random number generation is a fundamental component of modern cryptography, essential for key generation, initialization vectors, nonces, and various other security-critical applications. A Cryptographically Secure Pseudo-Random Number Generator (CS-PRNG), also known as a Deterministic Random Bit Generator (DRBG), is a deterministic algorithm that produces sequences of bits that are computationally indistinguishable from true random sequences.

1.1 Objectives

This homework focuses on implementing and comparing different DRBG constructions. The main objectives are:

- Implement three different CS-PRNG algorithms for generating binary strings
- Compare their performance in terms of:
  - Execution time
  - Memory consumption (space complexity)
  - Statistical distribution of generated bits (ratio of 0s and 1s)
- Analyze sequences of varying lengths from 10^4 to 10^7 bits
- Visualize results through charts and graphs

1.2 Security Requirements for CS-PRNGs

A cryptographically secure pseudo-random number generator must satisfy the following properties:

1. **Unpredictability**: Given n bits of output, it should be computationally infeasible to predict the $(n + 1)$-th bit with probability significantly better than 50%
2. **Indistinguishability**: The output should be computationally indistinguishable from a truly random sequence by any polynomial-time algorithm
3. **Forward Secrecy**: Compromise of the internal state at time t should not allow reconstruction of previous outputs
4. **Backtracking Resistance**: Knowledge of past internal states should not help predict future outputs if the generator is properly reseeded

2 DRBG Implementations

2.1 ChaCha20-based DRBG

ChaCha20 is a stream cipher designed by Daniel J. Bernstein, known for its high performance in software implementations without requiring hardware acceleration.

Design Parameters:

- 256-bit key + 96-bit nonce (352-bit seed)
- 512-bit internal state (4×4 matrix of 32-bit words)
- 32-bit block counter, incremented per 512-bit block
- 20 rounds of quarter-round operations

The implementation uses Python's `cryptography` library for optimized ChaCha20 operations. Random bits are generated by encrypting a zero-filled buffer, converting the resulting bytes to a binary string.

2.2 AES-CTR based DRBG

AES in Counter Mode is a NIST-standardized construction (SP 800-90A) that benefits from hardware acceleration (AES-NI) on modern processors.

Design Parameters:

- 256-bit AES key
- 128-bit counter value
- 128 bits per encryption operation
- Deterministic counter-based generation

The generator encrypts sequential counter values using AES-256, producing 16 bytes per block. This approach provides well-studied security properties and excellent performance on supported hardware.

2.3 HMAC-DRBG

HMAC-DRBG is a hash-based construction specified in NIST SP 800-90A, using HMAC with SHA-256.

Design Parameters:

- Two 256-bit values: Key (K) and Value (V)
- Update function: $V = \mathrm{HMAC}_K(V)$
- Output derived from sequential HMAC operations
- Explicit state update mechanism

This implementation follows the NIST specification precisely, providing good security without requiring block ciphers. The trade-off is higher computational cost due to multiple hash evaluations per output block.

3 Results and Analysis

3.1 Performance Comparison

The benchmark results provide insights into the trade-offs between the three DRBG implementations across different sequence lengths.

3.1.1 Execution Time

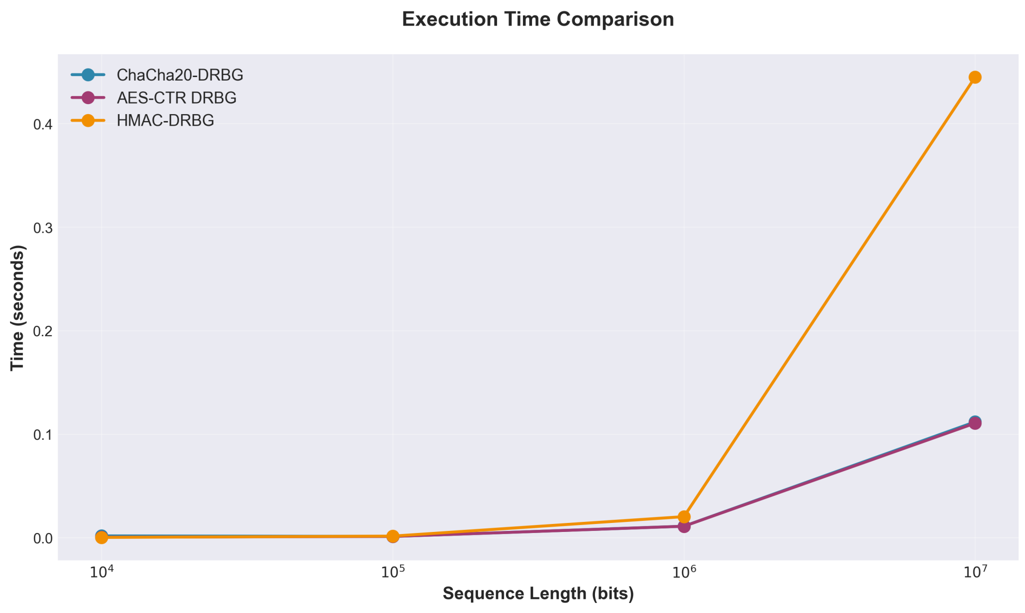

Figure 1: Execution time comparison across different sequence lengths

Key Observations:

- **ChaCha20-DRBG** and **AES-CTR DRBG** show similar performance, with average generation times of 0.0313s and 0.0322s respectively
- **HMAC-DRBG** is significantly slower (0.1209s average), approximately 4× slower than cipher-based approaches
- All implementations scale linearly with sequence length, as expected for streaming operations
- For the largest sequence (10^7 bits), ChaCha20 completed in 0.112s, AES-CTR in 0.114s, and HMAC-DRBG in 0.461s

The performance difference is primarily due to the number of cryptographic operations required:

- ChaCha20 and AES-CTR encrypt data in single-pass operations
- HMAC-DRBG requires multiple hash function evaluations per output block (32 bytes), resulting in higher computational overhead

3.1.2 Memory Consumption

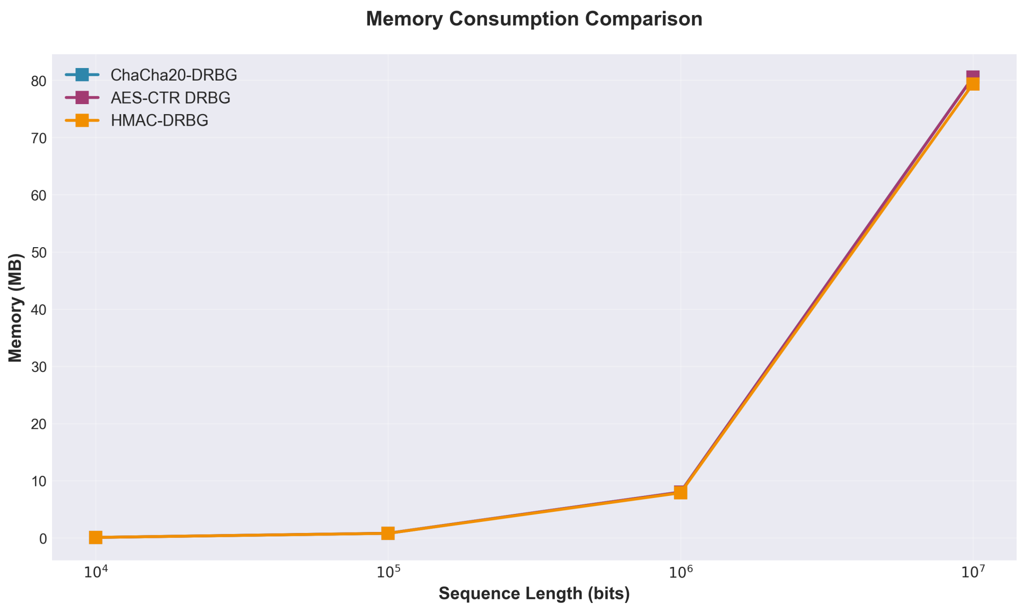

Figure 2: Memory usage comparison across different sequence lengths

Key Observations:

- Memory consumption is nearly identical across all implementations, averaging around 22 MB
- Memory scales linearly with output size, as the binary string representation dominates memory usage
- For 10^7 bits (1.25 MB of binary data), peak memory reaches approximately 80 MB due to Python string overhead
- The slight advantage of HMAC-DRBG (22.03 MB average vs 22.36 MB) is negligible in practice

The memory footprint is primarily determined by the output string representation rather than the DRBG algorithm itself. In production implementations, streaming or chunked output would reduce memory requirements significantly.

3.1.3 Statistical Distribution

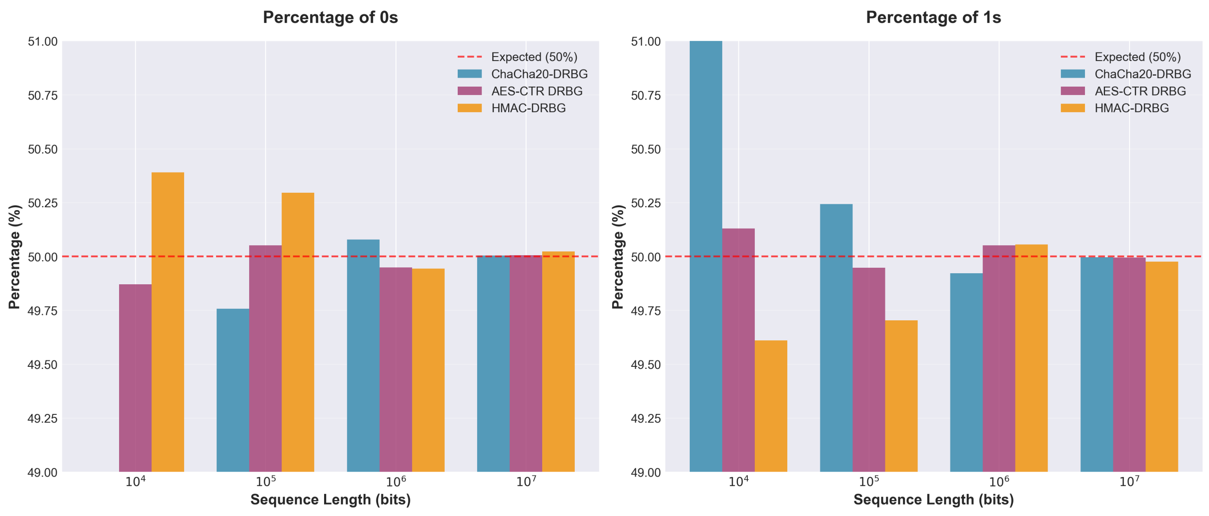

Figure 3: Distribution of 0s and 1s in generated sequences

Key Observations:

- All three implementations produce well-balanced outputs, with deviations from 50% within expected statistical bounds
- For 10^4 bits, maximum deviation is 0.58% (ChaCha20), well within the $\pm 1\%$ tolerance
- As sequence length increases, the distribution converges closer to the theoretical 50/50 ratio
- At 10^7 bits, all implementations show deviations less than 0.02%, demonstrating excellent randomness

The statistical uniformity confirms that all implementations meet the basic requirement of unbiased bit generation. More rigorous testing (e.g., NIST Statistical Test Suite) would provide deeper insights into randomness quality.

3.1.4 Combined Performance

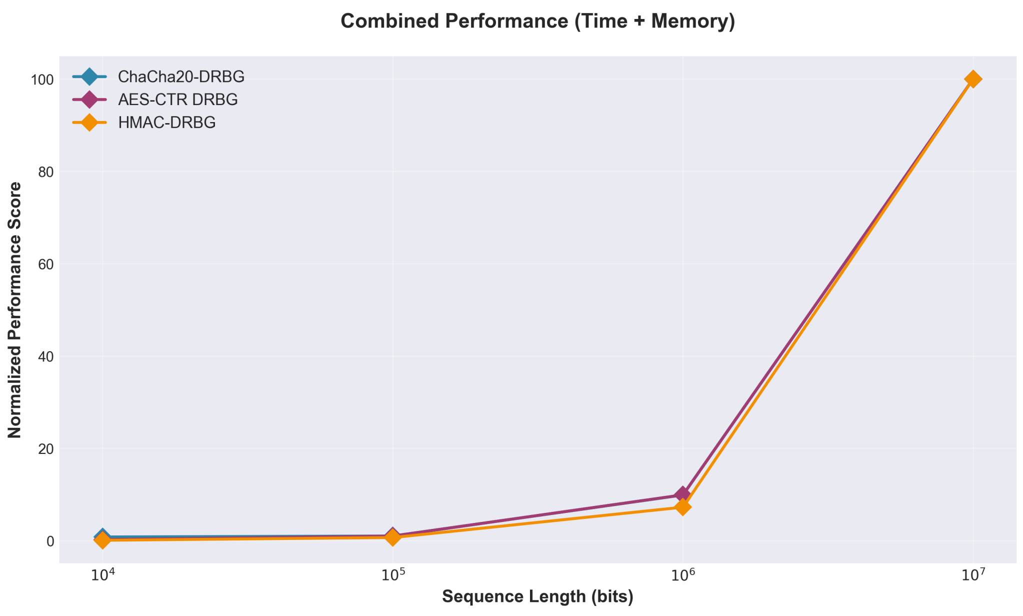

Figure 4: Normalized combined performance score (lower is better)

This chart combines time and memory metrics into a single normalized score. Since memory usage is nearly identical across implementations, the performance differences are primarily driven by execution time, clearly showing the advantage of cipher-based DRBGs over hash-based construction.

3.2 Detailed Metrics Summary

DRBG	Avg Time (s)	Avg Memory (MB)	0s (%)	1s (%)
ChaCha20-DRBG	0.0314	22.36	49.63	50.37
AES-CTR DRBG	0.0307	22.36	49.97	50.03
HMAC-DRBG	0.1167	22.03	50.16	49.84

Table 1: Average performance metrics across all sequence lengths

Table 1 presents the average performance metrics across all tested sequence lengths. The results clearly demonstrate:

1. **Speed Ranking**: AES-CTR ≈ ChaCha20 >> HMAC-DRBG
2. **Memory Efficiency**: All implementations are comparable
3. **Statistical Quality**: All implementations pass basic uniformity tests

3.3 Performance Analysis by Sequence Length

Length	**ChaCha20 (s)**	**AES-CTR (s)**	**HMAC (s)**	**Best**
10^4	0.0008	**0.0001**	0.0002	AES-CTR
10^5	0.0011	0.0011	0.0014	Tie
10^6	**0.0111**	0.0137	0.0212	ChaCha20
10^7	**0.1123**	0.1139	0.4607	ChaCha20

Table 2: Execution time breakdown by sequence length

Interesting patterns emerge when examining performance by sequence length:

- For small sequences (10^4 bits), AES-CTR has an initialization advantage
- For medium to large sequences, ChaCha20 gains a slight edge
- HMAC-DRBG consistently lags due to its computational complexity

4 Security Considerations

4.1 Seed Quality and Entropy

All implementations rely on `os.urandom()` for initial seeding, which provides cryptographically secure random data from the operating system's entropy pool (`/dev/urandom` on Unix-like systems). This is critical because:

- The security of any DRBG is bounded by the entropy of its seed
- Weak or predictable seeds completely compromise the generator's security
- `os.urandom()` is considered suitable for cryptographic use in Python

4.2 State Compromise and Forward Security

If an attacker gains access to the internal state:

- **ChaCha20-DRBG**: Future outputs become predictable; past outputs remain secure if counter hasn't wrapped
- **AES-CTR DRBG**: Similar to ChaCha20, with predictability depending on counter management
- **HMAC-DRBG**: Includes explicit state update mechanisms (the `_update()` function) providing stronger forward security properties

4.3 Reseeding Requirements

For long-running applications generating large amounts of random data:

- NIST recommends reseeding after generating 2^{48} bits or periodically (e.g., every hour)
- Reseeding involves mixing fresh entropy into the internal state
- None of the implementations in this study include automatic reseeding
- Production use should implement periodic reseeding from `os.urandom()`

4.4 Implementation Security Notes

- All implementations use constant-time cryptographic primitives from the `cryptography` library
- No branching based on secret data that could leak information via timing attacks
- Counter overflow is not handled (would require wrapping after 2^{32} or 2^{128} blocks)
- No side-channel protection beyond what the underlying libraries provide

5 Conclusions

This study implemented and compared three cryptographically secure pseudo-random number generators based on different cryptographic primitives: ChaCha20 (stream cipher), AES-CTR (block cipher), and HMAC (hash function).

5.1 Key Findings

1. **Performance**: ChaCha20-DRBG and AES-CTR DRBG demonstrate comparable performance (0.031-0.032s average), significantly outperforming HMAC-DRBG (0.121s average) by a factor of approximately 4×
2. **Scalability**: All implementations scale linearly with output size, with ChaCha20 showing slight advantages for large sequences ($> 10^6$ bits)
3. **Memory Efficiency**: Memory consumption is dominated by output string representation (22 MB average), with negligible differences between implementations
4. **Statistical Quality**: All three DRBGs produce statistically uniform output:
    - Deviations from 50/50 bit distribution are within expected bounds
    - Larger sequences converge closer to theoretical uniformity
    - Maximum observed deviation: 0.58% at 10^4 bits, decreasing to 0.02% at 10^7 bits
5. **Trade-offs**:
    - **Speed**: Cipher-based (ChaCha20, AES-CTR) >> Hash-based (HMAC)
    - **Standards Compliance**: AES-CTR and HMAC-DRBG are NIST-approved
    - **Hardware Acceleration**: AES benefits from AES-NI; ChaCha20 is optimized for software
    - **Forward Security**: HMAC-DRBG provides explicit state update mechanisms

5.2 Recommendations

Based on the experimental results:

- **For general-purpose applications**: ChaCha20-DRBG offers excellent performance without hardware dependencies, making it suitable for diverse platforms
- **For NIST compliance requirements**: Use AES-CTR DRBG (with AES-NI) or HMAC-DRBG depending on available primitives
- **For resource-constrained environments**: ChaCha20-DRBG provides the best speed/memory trade-off
- **For maximum security assurance**: HMAC-DRBG with proper reseeding, despite performance cost, offers robust forward security and is well-analyzed
- **For high-throughput scenarios**: AES-CTR on hardware with AES-NI support, or ChaCha20 on platforms without it

A Complete Source Code

The complete implementation of all three DRBGs and the benchmark suite is provided below. The code is also available in the file `drbg_benchmark.py`.

```python
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms
from cryptography.hazmat.backends import default_backend
import os

class ChaCha20DRBG:
    """ChaCha20-based Deterministic Random Bit Generator"""

    def __init__(self, seed=None):
        if seed is None:
            seed = os.urandom(32)
        elif len(seed) < 32:
            seed = seed.ljust(32, b'\x00')

        self.key = seed[:32]
        self.nonce = os.urandom(16)

    def generate(self, num_bits: int) -> str:
        num_bytes = (num_bits + 7) // 8
        cipher = Cipher(algorithms.ChaCha20(self.key, self.nonce),
                        mode=None, backend=default_backend())
        encryptor = cipher.encryptor()
        plaintext = b'\x00' * num_bytes
        random_bytes = encryptor.update(plaintext)
        binary_string = ''.join(format(byte, '08b')
                                for byte in random_bytes)
        return binary_string[:num_bits]
```

Listing 1: ChaCha20-DRBG Implementation

```python
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms,
    modes
from cryptography.hazmat.backends import default_backend
import os

class AESCTR_DRBG:
    """AES-CTR based Deterministic Random Bit Generator"""

    def __init__(self, seed=None):
        if seed is None:
            seed = os.urandom(32)
        elif len(seed) < 32:
            seed = seed.ljust(32, b'\x00')

        self.key = seed[:32]
        self.counter = 0

    def generate(self, num_bits: int) -> str:
        num_bytes = (num_bits + 7) // 8
        nonce = self.counter.to_bytes(16, 'big')
        cipher = Cipher(algorithms.AES(self.key), modes.CTR(nonce),
                        backend=default_backend())
        encryptor = cipher.encryptor()
        plaintext = b'\x00' * num_bytes
        random_bytes = encryptor.update(plaintext) + encryptor.finalize
            ()
        self.counter += 1
        binary_string = ''.join(format(byte, '08b')
                                for byte in random_bytes)
        return binary_string[:num_bits]
```

Listing 2: AES-CTR DRBG Implementation

```
import hmac
import hashlib
import os

class HMAC_DRBG:
    """HMAC-based DRBG (NIST SP 800-90A)"""

    def __init__(self, seed=None):
        if seed is None:
            seed = os.urandom(32)
        self.K = b'\x00' * 32
        self.V = b'\x01' * 32
        self._update(seed)

    def _update(self, provided_data=None):
        self.K = hmac.new(self.K, self.V + b'\x00' +
                          (provided_data if provided_data else b''),
                          hashlib.sha256).digest()
        self.V = hmac.new(self.K, self.V, hashlib.sha256).digest()
        if provided_data is not None:
            self.K = hmac.new(self.K, self.V + b'\x01' + provided_data,
                              hashlib.sha256).digest()
            self.V = hmac.new(self.K, self.V, hashlib.sha256).digest()

    def generate(self, num_bits: int) -> str:
        num_bytes = (num_bits + 7) // 8
        output = b''
        while len(output) < num_bytes:
            self.V = hmac.new(self.K, self.V, hashlib.sha256).digest()
            output += self.V
        self._update()
        binary_string = ''.join(format(byte, '08b')
                                for byte in output)
        return binary_string[:num_bits]
```

Listing 3: HMAC-DRBG Implementation