密碼工程 Quiz5
111550035 蔡昀錚

Problem 1

(a) Write a Python/C++ program to generate 1M bytes of cryptographically secure random numbers.

Ans: How to run the code:

1. Put the RNG.py file into sts-2.1.2/sts-2.1.2 (same as the assess file)

2. Open terminal and cd into sts-2.1.2/sts-2.1.2

3. Type ./assess 8388608 (here 8388608 represent 1024 * 1024 * 8 bits)

4. Type in provide command same as Quiz5_config.pdf

5. After finishing analyzing, type in "cat experiments/AlgorithmTesting/finalAnalysisReport.txt" to show the testing result.

(b) Run the NIST SP800-22 statistical test on your 1M bytes of binary cryptographically secure random numbers and analyze the test results to identify any deviations from the expected statistical properties of random numbers.

Ans: The results indicate that each test was run once, following is the brief explanation of each test:

1. **FrequencyTest**: This test checks if the number of ones and zeros in a sequence are approximately the same as would be expected for a truly random sequence. An imbalance might indicate a bias which could be exploitable.

2. **Block Frequency Test**: This test divides the sequence into blocks of equal size and then computes the frequency of ones in each block. The goal is to determine if each block's one-bit frequency is consistent with the expected half distribution for random sequences.

3. **Cumulative Sums Test**: There are two forms of this test, one for the forward direction and one for the reverse. It sums the deviations of each bit (as +1 or -1) from the expected mean (0.5), checking if the cumulative sum wanders too far from zero, which would indicate a non-random process.

4. **Runs Test**: A run is a sequence of identical bits preceded and followed by a different bit or no bit. This test examines the sequence for runs of ones and zeros to see if their occurrence is too frequent or not frequent enough which could indicate a non-random process.

5. **Longest Run of Ones in a Block Test**: This test divides the sequence into blocks and identifies the longest run of ones within each block. The distributions of the longest runs within the tested sequences are then compared to the expected distribution of a random sequence.

6. **Rank Test**: This test checks for linear dependence among fixed-length substrings of the original sequence. A high rank indicates a random sequence, whereas a low rank suggests non-randomness.

7. **Discrete Fourier Transform (Spectral) Test**: It examines the peak heights in the Discrete Fourier Transform of the sequence. An excess of peaks exceeding the 95% threshold indicates a non-random sequence.

8. **Non-overlapping Template Matching Test**: This test checks for occurrences of pre-defined templates of bits. The number of matches for the template within the sequence should conform to a particular distribution to pass as random.

9. **Overlapping Template Matching Test**: Similar to the non-overlapping version but allows for templates to overlap. It is more stringent and checks for more subtle patterns of repetition.

10. **Universal Statistical Test**: This test checks for the compressibility of the sequence, which can indicate regularities. A random sequence is expected to be less compressible.

11. **Approximate Entropy Test**: This test compares the frequency of overlapping blocks of two consecutive lengths against the expected distribution for a random sequence, looking for patterns that might repeat too often or not often enough.

12. **Random Excursions Test**: It evaluates the randomness of the number of cycles of a certain length in a random walk derived from the binary sequence. Deviations from the expected number of visits to certain states can indicate non-randomness.

13. **Random Excursions Variant Test**: This test is a variant of the Random Excursions Test and examines the total number of times that a particular state is visited in a random walk.

14. **Serial Test**: This test checks for the frequency of all possible overlapping m-bit patterns across the entire sequence. The discrepancy between the observed and expected frequency of occurrence of these patterns is used to determine the randomness.

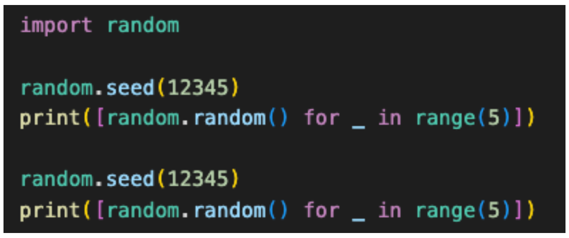
15. **Linear Complexity Test**: This test measures the length of a linear feedback shift register (LFSR) that can generate the tested sequence. A shorter than expected LFSR indicates non-randomness.

Given that the proportion of passing sequences is 1/1 for each test, and considering the test details are not fully disclosed (ex. p-values doesn’t shown), it generally indicates that each test passed based on its specific criterion. Based on the provided summary, the RNG appears to perform well across the suite of NIST SP800-22 tests, suggesting good statistical properties of randomness in the generated binary sequence. However, a deeper analysis with complete p-values and consideration of the sequence's intended use would be necessary for a comprehensive evaluation.

(c) Extra credit: Find out a non-cryptographically secure random number generator, such as random(), to demonstrate its lack of safety. Then, propose modifications to enhance its security to generate cryptographically secure random numbers that meet the highest standards of security and reliability.

Ans: The random() module in Python is a classic example of a non-cryptographically secure random number generator. The main idea of this module is the Mersenne Twister algorithm, which, while providing a high period and fast generation of pseudo-random numbers, but it can be predicted after observing a sufficient number of outputs.

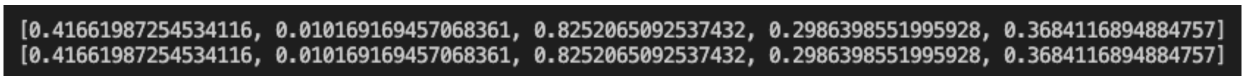
1. **Demonstration of Lack of Safety:** The predictability of random() can be demonstrated by initializing two random number generators with the same seed and showing that they produce the same output sequence. This predictability is a significant vulnerability for cryptographic applications, as it can allow attackers to predict future outputs if they can determine the current state of the generator or if they can guess the seed.



```python
import random

random.seed(12345)
print([random.random() for _ in range(5)])

random.seed(12345)
print([random.random() for _ in range(5)])
```



```
[0.41661987254534116, 0.010169169457068361, 0.8252065092537432, 0.2986398551995928, 0.3684116894884757]
[0.41661987254534116, 0.010169169457068361, 0.8252065092537432, 0.2986398551995928, 0.3684116894884757]
```

2. **Enhancing Security**: To enhance the security of random number generation in Python and meet the highest standards of security and reliability, the following modifications and considerations should be adopted:

   (1) Ensure Proper Seed Entropy: CSPRNGs rely on an initial seed value with high entropy. While `secrets` and `SystemRandom` automatically use sources of high entropy available on the system, ensuring the underlying OS and hardware provide high-quality entropy is crucial.
   (2) Regularly Refresh the RNG State: For long-running applications or those generating a large volume of random numbers, regularly refreshing the state of the RNG (by re-seeding or using mechanisms provided by the CSPRNG to incorporate additional entropy) can further enhance security.
   (3) Security Audits and Compliance: Regular security audits of the application, including the RNG component, help identify potential vulnerabilities. Adhering to industry standards and compliance requirements for cryptographic applications can guide the secure use of RNGs.
   (4) Use Well-Tested Libraries and Algorithms: Relying on well-tested cryptographic libraries and standards (e.g., NIST-approved algorithms and implementations) reduces the risk of introducing vulnerabilities through custom cryptographic implementations.

By adopting below methods, one can significantly enhance the security and reliability of random number generation in applications where the unpredictability of those numbers is crucial.