HW1

Brandon Walker

1. Introduction

This document outlines the test strategy for the dice roller application. The primary objective is to validate that the software produces fair and genuinely random results. The strategy is designed to identify potential flaws in the random number generation process, particularly issues related to seeding and statistical distribution, by simulating both common user behavior and extreme edge cases.

2. Testing Approach

My testing approach will focus on two distinct methodologies to comprehensively evaluate the application's randomness and stability.

- **Successive Call Testing:** This method involves a high volume of individual, consecutive requests to the dice roller. The goal is to detect any patterns or non-random behavior that may emerge when the software is called repeatedly in a short period. This directly addresses the risk of improper seeding, where the same sequence of numbers could be generated if the seed is not updated correctly between calls.
- **Large Call Testing:** This method involves a small number of requests, each asking for a very large number of dice rolls in a single batch. This approach is crucial for verifying the overall statistical distribution of the results and for ensuring the application can handle massive data sets without performance degradation or failure.

3. Test Cases

3.1 Successive Call Test Cases

This test case is designed to stress the application's ability to produce independent and unpredictable results with each new request.

- **Test Case 1: Rapid-Fire Multi-Dice Rolls**
    - **Description:** A script will be used to make a very large number of separate calls to the application, each requesting a roll of four six-sided dice (4d6).
    - **Expected Outcome:** I will analyze the sequence of rolls for any unnatural runs. This test is specifically designed to identify potential issues with the random number generator's integrity when producing multiple numbers in a single call. For instance, a high frequency of rolls where all four dice land on the same number would be flagged for investigation. The application should not exhibit any noticeable performance degradation.

3.2 Large Call Test Cases

This test case focuses on evaluating statistical fairness and system stability under significant load.

- **Test Case 2: Combined Large Batch Analysis**
    - **Description:** Request a few, massive batches of six-sided die rolls to push the application to its operational limits.
    - **Expected Outcome:** This single test will serve three purposes. First, the application should not crash, freeze, or return a corrupted data set, and the response time should be within an acceptable range. Second, i will verify that the **overall distribution** of the rolls is uniform. The frequency of each number from 1 to 6 should be approximately equal. A deviation of more than 1% from this will be considered a failure. Third, I will analyze the returned data for any long, **consecutive runs of the same number**, which could indicate a flaw in the random number generation algorithm's state at a large scale.

4. Metrics and Analysis

For each test case, I will analyze the data using the following methods:

- **Run Analysis:** I will count the number of consecutive identical rolls to identify patterns that deviate from pure random chance.
- **Frequency Analysis:** I will use a histogram or frequency table to compare the observed distribution of rolls to the expected uniform distribution.
- **Error Reporting:** Any application crashes, timeouts, or unexpected errors will be documented and reported immediately.

5. Conclusion

This strategy provides a robust framework for validating the core functionality of the dice roller application. By systematically testing both rapid-fire seeding issues and large-scale distribution anomalies, I can build confidence that the software provides fair, unpredictable, and reliable results.

Appendix A

DiceRoller.h

```
#pragma once
#include <string>
#include <vector>
#include <cstdint>
#include <random>

uint64_t hex_to_uint64(const std::string& hex_str);
size_t write_callback(void* contents, size_t size, size_t nmemb, void* userp);
std::string get_cloudflare_beacon();

class DiceRoller {
public:
	DiceRoller();
	std::vector<int> rollDice(int numDice);
private:
	std::mt19937 engine;
};
```

DiceRoller.cpp

```
#include <iostream>
#include <random>
#include <string>
#include <vector>
#include <chrono>
#include "DiceRoller.h"

// Header for libcurl, handles HTTP requests
#include <curl/curl.h>

// Function to convert hex string to 64-bit unsigned integer
uint64_t hex_to_uint64(const std::string& hex_str) {
	uint64_t value = 0;
	std::size_t size = hex_str.length();
	for (std::size_t i = 0; i < size; ++i) {
		char c = hex_str[i];
		value <<= 4; // Shift for next hex digit
		if (c >= '0' && c <= '9') {
			value |= (c - '0');
		} else if (c >= 'a' && c <= 'f') {
```

```
                value |= (c - 'a' + 10);
            } else if (c >= 'A' && c <= 'F') {
                value |= (c - 'A' + 10);
            }
        }
        return value;
}

// Write callback for cURL to capture the response data
size_t write_callback(void* contents, size_t size, size_t nmemb, void* userp) {
        reinterpret_cast<std::string*>(userp)->append(reinterpret_cast<char*>(contents),
        size * nmemb);
        return size * nmemb;
}

// Function to fetch the beacon value from Cloudflare's API
std::string get_cloudflare_beacon() {
        CURL* curl;
        CURLcode res;
        std::string read_buffer;

        curl_global_init(CURL_GLOBAL_DEFAULT);
        curl = curl_easy_init();
        if (curl) {
            curl_easy_setopt(curl, CURLOPT_URL,
            "https://drand.cloudflare.com/public/latest");
            curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_callback);
            curl_easy_setopt(curl, CURLOPT_WRITEDATA, &read_buffer);
            curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L);

            res = curl_easy_perform(curl);
            if (res != CURLE_OK) {
                fprintf(stderr, "curl_easy_perform() failed: %s\n",
                curl_easy_strerror(res));
                read_buffer = ""; // Indicate failure
            }
            curl_easy_cleanup(curl);
        }
        curl_global_cleanup();

        // The response is JSON, so we need to parse it to get the 'randomness' value.
        // For simplicity, we'll find the substring manually. A JSON library is better for
        robustness.
        size_t start = read_buffer.find("\"randomness\":\"");
```

```
        if (start != std::string::npos) {
                start += 14; // Length of `"randomness":"`
                size_t end = read_buffer.find("\"", start);
                if (end != std::string::npos) {
                        return read_buffer.substr(start, end - start);
                }
        }
        return ""; // Return empty string on parsing failure
}

DiceRoller::DiceRoller() {
        // Get UNIX timestamp in milliseconds
        uint64_t time_seed = std::chrono::duration_cast<std::chrono::milliseconds>(
        std::chrono::system_clock::now().time_since_epoch()).count();

        // Fetch beacon randomness
        std::string beacon_hex = get_cloudflare_beacon();
        uint64_t beacon_seed = 0;
        if (!beacon_hex.empty()) {
                beacon_seed = hex_to_uint64(beacon_hex);
        }

        // Combine seeds using std::seed_seq
        std::seed_seq seed_seq{
        static_cast<unsigned int>(time_seed & 0xFFFFFFFF),
        static_cast<unsigned int>(time_seed >> 32),
        static_cast<unsigned int>(beacon_seed & 0xFFFFFFFF),
        static_cast<unsigned int>(beacon_seed >> 32)
        };
        engine = std::mt19937(seed_seq);
}

std::vector<int> DiceRoller::rollDice(int numDice) {
        std::vector<int> results(numDice);
        std::uniform_int_distribution<int> dist(1, 6); // Six-sided dice
        for (int i = 0; i < numDice; ++i) {
                results[i] = dist(engine);
        }
        return results;
}
```

main.cpp

```
#include <iostream>
#include <vector>
#include "DiceRoller.h"

int main() {
	int rollCount[6] = {0};  // Initialize all counts to 0
	DiceRoller roller;

	while(true){
		int numDice;
		std::cout << "Enter number of dice to roll (0 to quit): ";
		std::cin >> numDice;

		if (numDice == 0) break;

		std::vector<int> results = roller.rollDice(numDice);

		std::cout << "Results: ";
		for (int value : results) {
			std::cout << value << " ";
			if (value >= 1 && value <= 6) {
				rollCount[value - 1]++;
			}
		}
		std::cout << std::endl;
	}

	std::cout << "\nNumber of times each number was rolled:\n";
	for (int i = 0; i < 6; ++i) {
		std::cout << (i + 1) << ": " << rollCount[i] << std::endl;
	}

	return 0;
}
```