Natural Computational Entropy and Random Number Generation

Ramsey Alsheikh

American Heritage Boca/Delray

Iris Thompson

Period 10

Abstract

Random number generation is the process of generating numbers that could not feasibly be predicted beforehand. The purpose of this experiment was to provide evidence that a computer's local processes could be used as a source of entropy for a random number generator (RNG). The hypothesis was that the generator using natural computational entropy would produce statistically random numbers. The local CPU registers of a machine running Ubuntu were used as a source of entropy for a newly coded random number generator. These CPU registers change their contents millions of times per second, so it was hypothesized that it would be impossible to predict their contents beforehand. A C program was coded on a system running Ubuntu Linux that used the contents of the EAX register every second to output a series of 10 bit numbers. The generator passed the Dieharder suite of statistical tests, and the hypothesis was supported. While the Dieharder tests do not "prove" randomness - as randomness is technically impossible to prove - passing it indicates it is very likely that the generator is reliable. Further research is needed to conduct more exhaustive statistical tests, and to examine other potential sources of entropy in a computer that could prove to be, effectively, more random. This experiment has provided evidence for the possibility of true random number generation that does not rely on expensive equipment. It has applications in cryptography and computer security, as random numbers are commonly used to make computer interactions more secure from hackers.

Introduction

Statement of Purpose

The purpose of this experiment would be to provide evidence for whether or not local computational processes could be used as a source of entropy for a RNG. The experimental group would be a RNG that utilizes CPU registers, and the control group would be the default C compiler RNG, widely used and accepted. The dependent variable is which of the two RNGs is being used as input for the Dieharder statistical tests, while the independent variable is the output from the tests. If the experimental RNG performs better or equal to the control RNG, it would provide evidence for local CPU registers being a secure source of entropy, which would be a convenient source of randomness for all modern computers.

Background Research

There exist two kinds of RNGs: Pseudo Random Number Generators (PRNG) and True Random Number Generators (TRNG) (Haahr, 2020). The former is the more common kind; they consist of computer algorithms that can generate number outputs that are almost untraceable back to the original inputs. The main drawback to a PRNG is that for a given input, they can produce only one output; this means that they are predetermined. If one figures out the inputs, they know the outputs. A TRNG differs from this in that instead of relying on computer algorithms, they observe phenomena external to the RNG for their “randomness”. Common TRNGs will observe the outputs of quantum mechanical processes, or measure subtle variations in the Earth’s atmosphere (Rubin, 2011). The drawback with a TRNG is that they usually require expensive equipment that is almost impossible for a casual computer user to acquire. However, one potential source of commonly available “randomness” that may offer a balance between

TRNGs and PRNGs is called natural computational entropy. Natural computational entropy refers to the inherent unpredictability of modern computing that arises from the speed of computer processors. An average computer system can perform billions of calculations per second, making the contents of the computer's memory at any given moment in time practically unpredictable (The CPU and). Since the vast majority of modern computers operate at this speed, using this entropy as the source of randomness for a RNG could be a realistic and efficient way to easily generate random numbers. If so, an average user, without access to research-grade lab equipment to observe minute physical phenomena, would be able to generate true random numbers for any purpose. This TRNG would have applications in the fields of cryptography and cybersecurity (Sidhpurwala, 2019). Random numbers are commonly used to induce an extra layer of security in online transactions, by using them as a key to an encryption cipher hiding the true contents of transmitted data (Hussain & Khalique, 2019). Making the generation of these numbers easier makes it easier to keep digital communications secure and the internet safe.

Hypothesis

If a CPU's registers are used as a source of natural computational energy for a random number generator, the generator will pass all of the dieharder statistical tests. The CPU registers are local memory caches in close proximity to the CPU, used as a quick and convenient storage location for a running program. Each program running on a computer at any given moment stores unique values in the registers depending on what is executing on the given execution cycle (often on the scale of nanoseconds) (Processes). Since even a consumer grade computer runs hundreds of different processes simultaneously, the register values change extremely rapidly. Consequently, it was hypothesized that this extreme speed would make it practically impossible to know what a computer was holding in its registers at a given moment, and thus a reliable source of entropy for an TRNG. However, if there were a repeating pattern in the register values over a given time period, the system may not be entropic enough to generate random numbers. This experiment will test this.

Method

Materials

- A computer capable of running Ubuntu 5.2.7, with administrator access
- Charging Cable
- Source code, located at https://github.com/DatOneRam/nce-rng

Procedure

1. Verify your computer has x86 architecture capable of running Ubuntu kernel version 5.2.7, and install Ubuntu.
2. Download all code from *https://github.com/DatOneRam/nce-rng*.
3. Install the dieharder suite of statistical tests with *sudo apt-get install dieharder*
4. In the folder with the files from the github repository, run *gcc -o rng.o* from the command line to compile the random number generator program
5. Run the command *./rng.o*
6. Periodically check the size of the output file, *random.txt.* When it reaches 2 gigabytes, terminate the execution of the program by pressing ctrl-c.
7. Run *dieharder -a -g 202 -f random.txt* in the same folder and let the program finish.
8. Record the output of the dieharder tests.
9. Run the command *./control.o*
10. Periodically check the size of the output file, *control.txt.* When it reaches 2 gigabytes, terminate the execution of the program by pressing ctrl-c.
11. Run *dieharder -a -g 202 -f control.txt* in the same folder and let the program finish.
12. Record the output of the dieharder tests.
13. Run *./contmap.o* and wait for it to finish executing.
14. Run *./bitmap.o* and wait for it to finish executing
15. Repeat 3 times

Results

	Trial One	Trial Two	Trial Three
diehard_birthdays	0.53579783 (PASSED)	0.99977358 (WEAK)	0.45461747 (PASSED)
diehard_operm5	0.60616033 (PASSED)	0.90072616 (PASSED)	0.46926287 (PASSED)
diehard_rank_32x32	0.50726861 (PASSED)	0.77523690 (PASSED)	0.93540002 (PASSED)
diehard_rank_6x8	0.78288838 (PASSED)	0.35146683 (PASSED)	0.85335987 (PASSED)
diehard_bitstream	0.09950032 (PASSED)	0.71848644 (PASSED)	0.23601083 (PASSED)
diehard_opso	0.92067771 (PASSED)	0.22129166 (PASSED)	0.17591068 (PASSED)
diehard_oqso	0.68864727 (PASSED)	0.52596006 (PASSED)	0.21392689 (PASSED)
diehard_dna	0.51698383 (PASSED)	0.69370959 (PASSED)	0.02755855 (PASSED)
diehard_count_1s_str	0.90214846 (PASSED)	0.05226944 (PASSED)	0.57872909 (PASSED)
diehard_count_1s_byt	0.50809128 (PASSED)	0.32218233 (PASSED)	0.45223129 (PASSED)
diehard_parking_lot	0.72516127 (PASSED)	0.43004782 (PASSED)	0.68219063 (PASSED)
diehard_2dsphere	0.11899233 (PASSED)	0.37472555 (PASSED)	0.75543998 (PASSED)
diehard_3dsphere	0.92851626 (PASSED)	0.31958302 (PASSED)	0.44378388 (PASSED)
diehard_squeeze	0.23634530 (PASSED)	0.82599456 (PASSED)	0.44378388 (PASSED)
diehard_sums	0.27394937	0.79196036	0.00801799

	(PASSED)	(PASSED)	(PASSED)
diehard_runs (averaged)	0.546953715 (PASSED)	0.464491385 (PASSED)	0.55987162 (PASSED)
diehard_craps (averaged)	0.856565975 (PASSED)	0.83337907 (PASSED)	0.16627610 (PASSED)
marsaglia_tsang_gcd (averaged)	0.88059402 (PASSED)	0.944026415 (PASSED)	0.860695775 (PASSED)
sts_monobit	0.81019327 (PASSED)	0.55123354 (PASSED)	0.51039525 (PASSED)
sts_runs	0.31191600 (PASSED)	0.16517256 (PASSED)	0.92519205 (PASSED)
sts_serial (averaged)	0.49333323 (PASSED)	0.60276911 (PASSED)	0.58153574 (PASSED)
rgb_bitdist (averaged)	0.63338412 (PASSED)	0.50530684 (PASSED)	0.53438601 (PASSED)
rgb_minimum_distance (averaged)	0.85166802 (PASSED)	0.61062768 (PASSED)	0.19975601 (PASSED)
rgb_permutations (averaged)	0.51854853 (PASSED)	0.62623962 (PASSED)	0.35003433 (PASSED)
rgb_lagged_sum (averaged)	0.50591272 (PASSED)	0.54421392 (PASSED)	0.56676919 (PASSED)
rgb_kstest_test	0.20554114 (PASSED)	0.34436122 (PASSED)	0.56680642 (PASSED)
dab_bytedistrib	0.78773355 (PASSED)	0.08685945 (PASSED)	0.31520235 (PASSED)
dab_dct	0.83762451 (PASSED)	0.99558521 (PASSED)	0.50593935 (PASSED)
dab_filltree (averaged)	0.53685526 (PASSED)	0.708793715 (PASSED)	0.31968870 (PASSED)
dab_filltree2 (averaged)	0.25254166 (PASSED)	0.77794901 (PASSED)	0.89232524 (PASSED)
dab_monobit2	0.29253065	0.97567730	0.14423945

	(PASSED)	(PASSED)	(PASSED)

Table 1. The Dieharder P-Values of the Experimental RNG

	Trial One	Trial Two	Trial 3
diehard_birthdays	0.98942289 (PASSED)	0.31561856 (PASSED)	0.42299549 (PASSED)
diehard_operm5	0.78696250 (PASSED)	0.82143815 (PASSED)	0.92882544 (PASSED)
diehard_rank_32x32	0.99207211 (PASSED)	0.99735437 (WEAK)	0.24778314 (PASSED)
diehard_rank_6x8	0.78658890 (PASSED)	0.94619074 (PASSED)	0.91175131 (PASSED)
diehard_bitstream	0.31853898 (PASSED)	0.89655440 (PASSED)	0.92642471 (PASSED)
diehard_opso	0.19809957 (PASSED)	0.89655440 (PASSED)	0.75339079 (PASSED)
diehard_oqso	0.98435767 (PASSED)	0.28018339 (PASSED)	0.68659579 (PASSED)
diehard_dna	0.03642798 (PASSED)	0.34620057 (PASSED)	0.13878596 (PASSED)
diehard_count_1s_str	0.83311576 (PASSED)	0.09480269 (PASSED)	0.30056564 (PASSED)
diehard_count_1s_byt	0.05595186 (PASSED)	0.58171965 (PASSED)	0.97342159 (PASSED)
diehard_parking_lot	0.92463763 (PASSED)	0.59549008 (PASSED)	0.61550204 (PASSED)
diehard_2dsphere	0.87190818 (PASSED)	0.38308065 (PASSED)	0.2277650 (PASSED)
diehard_3dsphere	0.29940272 (PASSED)	0.38308065 (PASSED)	0.79187703 (PASSED)
diehard_squeeze	0.94857045	0.62702154	0.00413728

	(PASSED)	(PASSED)	(WEAK)
diehard_sums	0.21577444 (PASSED)	0.43886429 (PASSED)	0.33529347 (PASSED)
diehard_runs (averaged)	0.75918973 (PASSED)	0.74057467 (PASSED)	0.52508538 (PASSED)
diehard_craps (averaged)	0.68457775 (PASSED)	0.67698426 (PASSED)	0.87127953 (PASSED)
marsaglia_tsang_gcd (averaged)	0.68861461 (PASSED)	0.65658766 (PASSED)	0.32991172 (PASSED)
sts_monobit	0.54466869 (PASSED)	0.15135596 (PASSED)	0.94059854 (PASSED)
sts_runs	0.31508641 (PASSED)	0.96106185 (PASSED)	0.80988587 (PASSED)
sts_serial (averaged)	0.59578272 (PASSED)	0.48701981 (PASSED)	0.54337365 (PASSED)
rgb_bitdist (averaged)	0.67894468 (PASSED)	0.56618983 (PASSED)	0.54995465 (PASSED)
rgb_minimum_distance (averaged)	0.64596946 (PASSED)	0.29559392 (PASSED)	0.20093876 (PASSED)
rgb_permutations (averaged)	0.52226834 (PASSED)	0.63571045 (PASSED)	0.47586589 (PASSED)
rgb_lagged_sum (averaged)	0.55644548 (PASSED)	0.57194566 (PASSED)	0.50010992 (PASSED)
rgb_kstest_test	0.32532012 (PASSED)	0.22374572 (PASSED)	0.45618229 (PASSED)
dab_bytedistrib	0.24997884 (PASSED)	0.64392446 (PASSED)	0.56843172 (PASSED)
dab_dct	0.28050363 (PASSED)	0.82638846 (PASSED)	0.31857008 (PASSED)
dab_filltree (averaged)	0.63490383 (PASSED)	0.61164415 (PASSED)	0.70562447 (PASSED)
dab_filltree2	0.06748201	0.84205831	0.713142895

(averaged)	(PASSED)	(PASSED)	(PASSED)
dab_monobit2	0.63353703 (PASSED)	0.97324251 (PASSED)	0.98009045 (PASSED)

Table 2. The Dieharder P-Values of the Control RNG

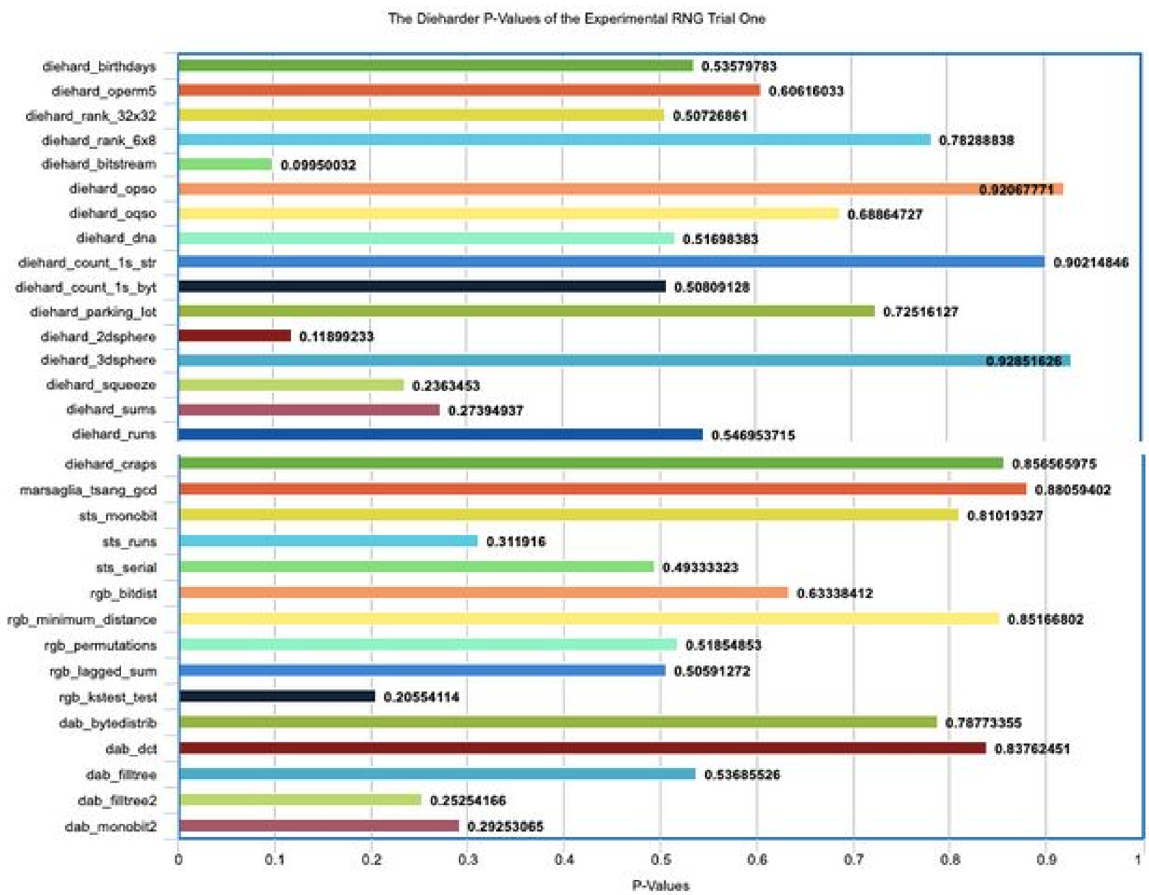

Figure 1. P-Values of the Experimental RNG Trial 1 Bar Graph

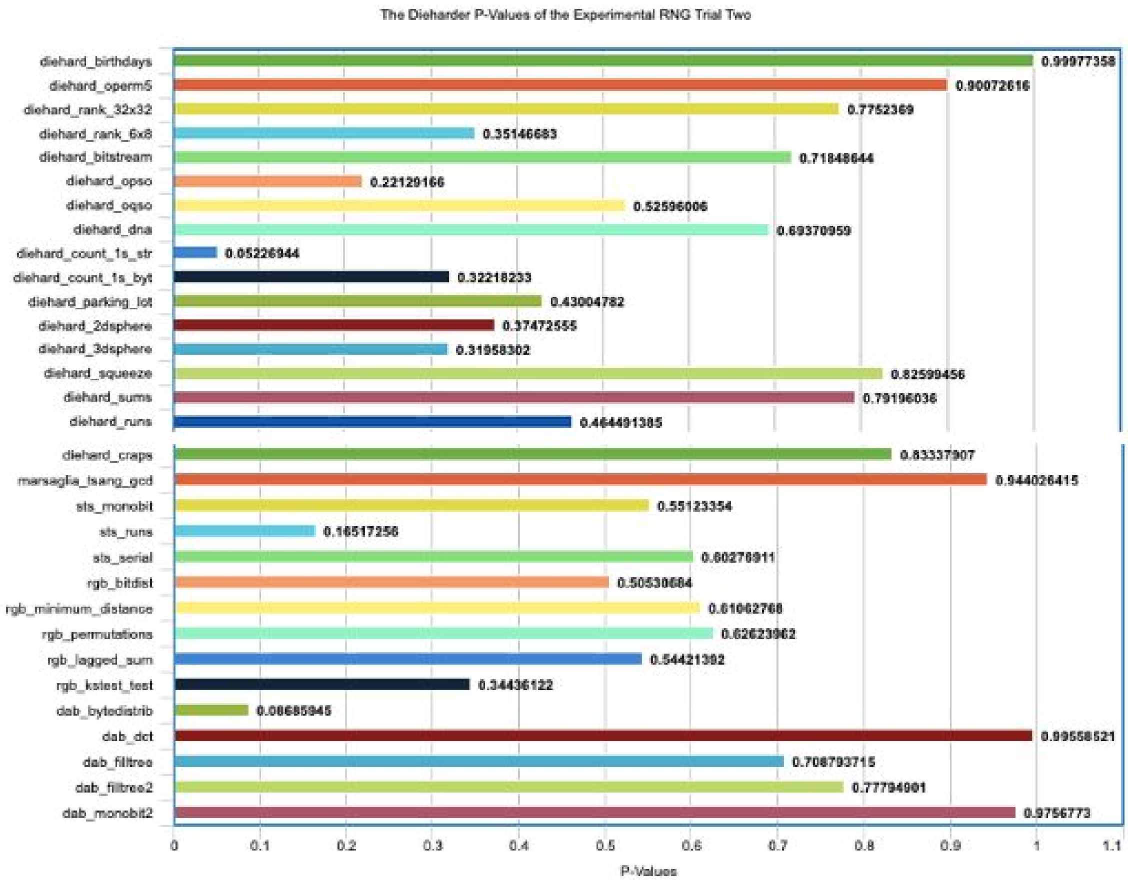

Figure 2. P-Values of the Experimental RNG Trial 2 Bar Graph

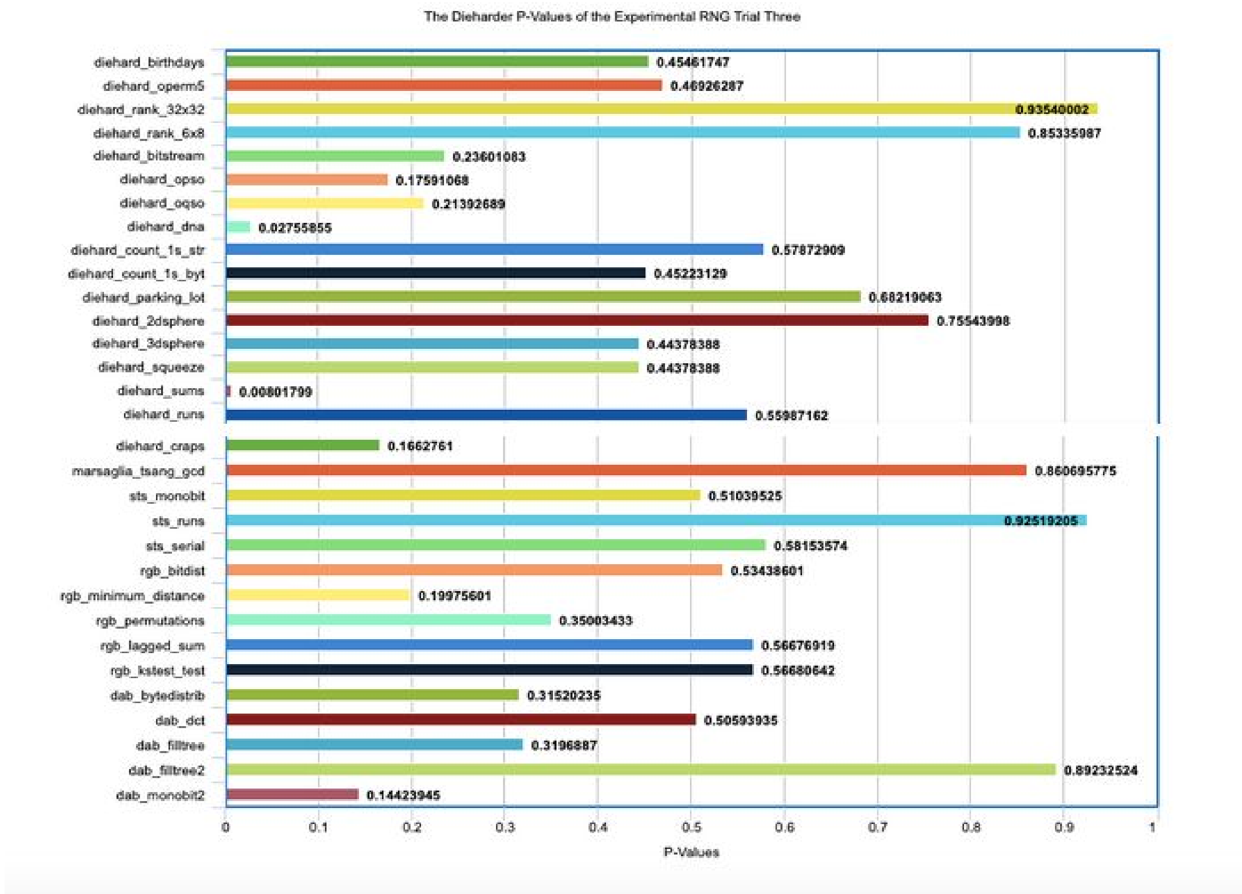

Figure 3. P-Values of the Experimental RNG Trial 3 Bar Graph

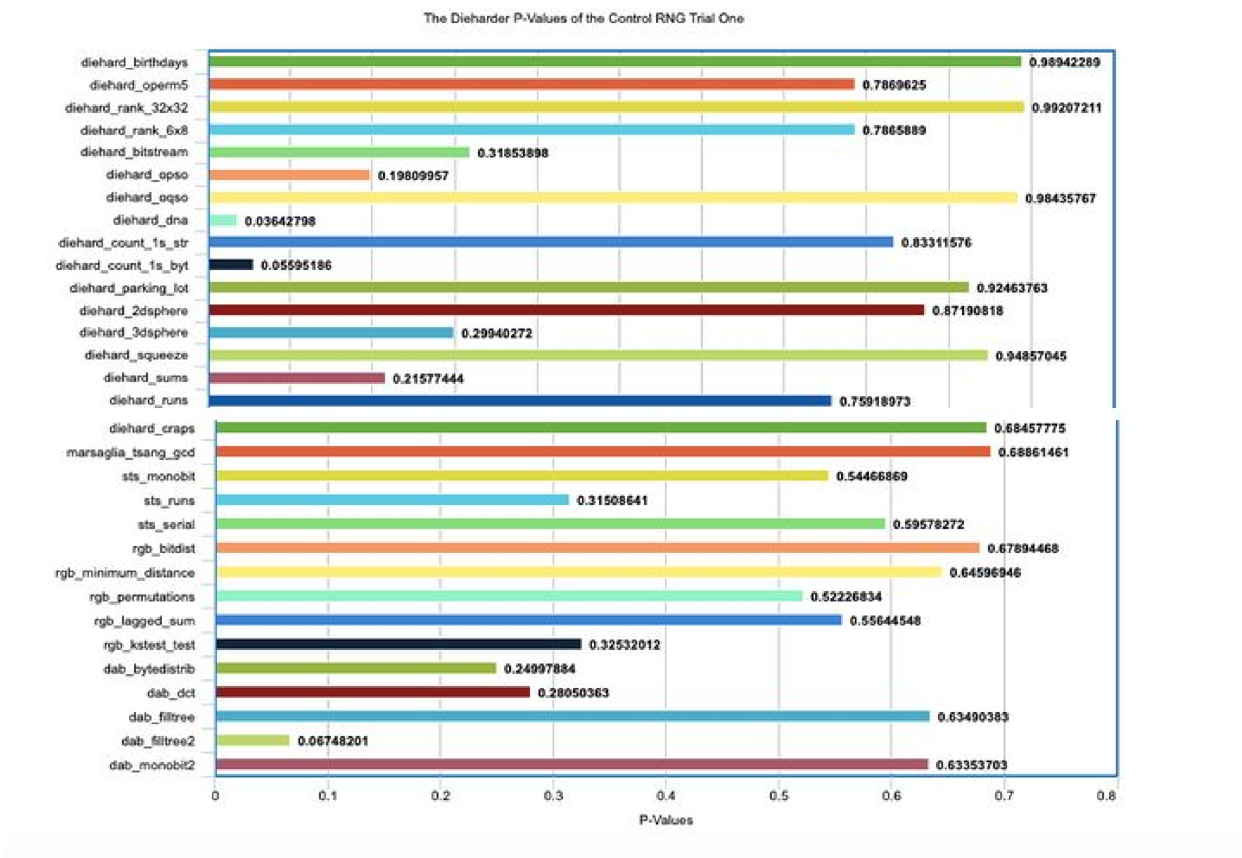

Figure 4. P-Values of the Control RNG Trial 1 Bar Graph

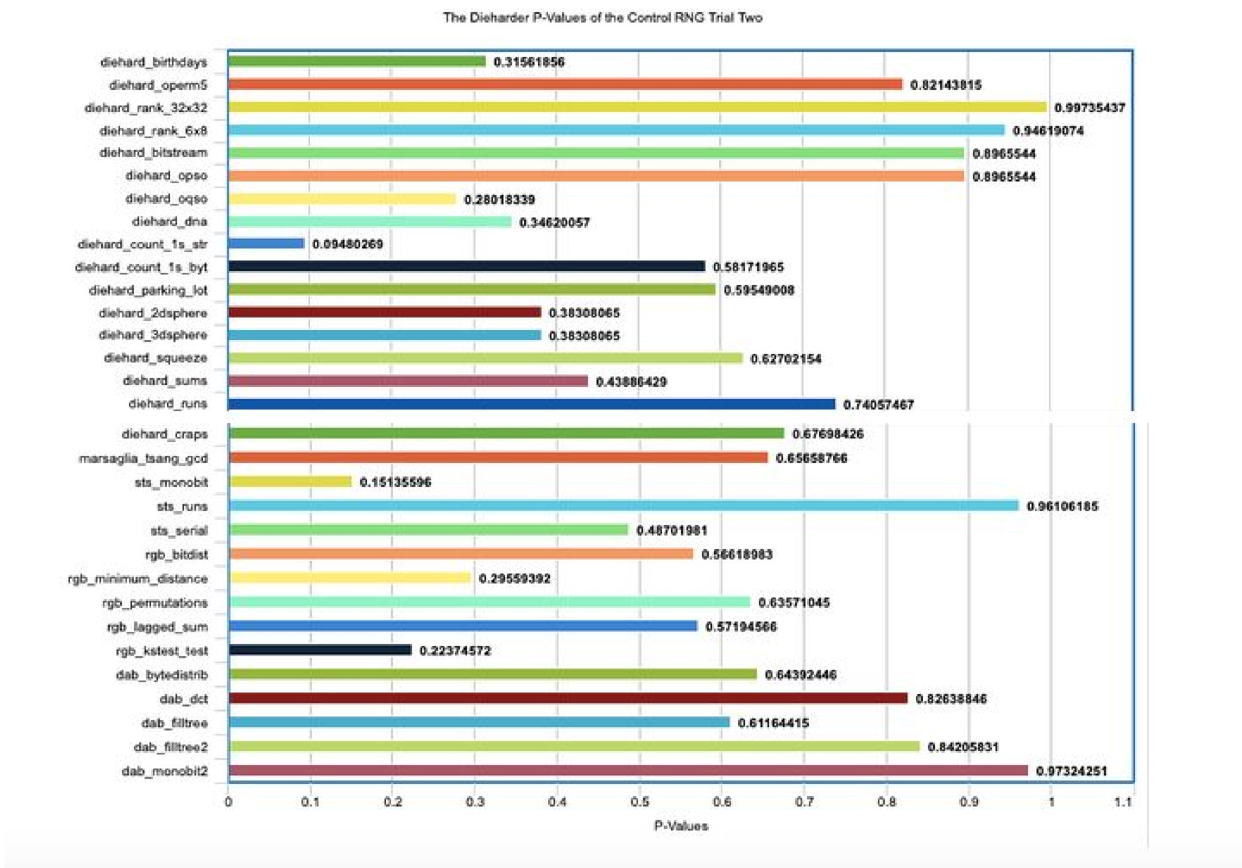

Figure 5. P-Values of the Control RNG Trial 2 Bar Graph

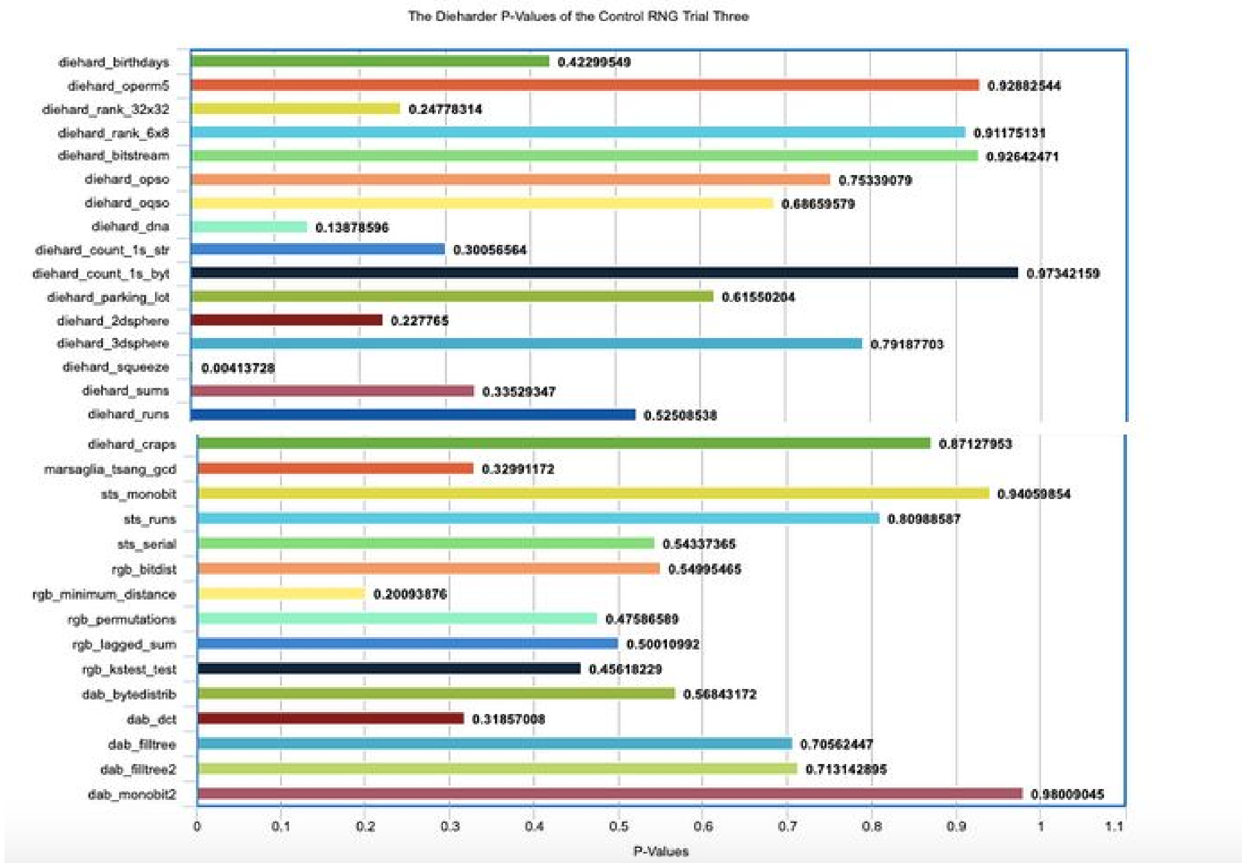

Figure 6. P-Values of the Control RNG Trial 3 Bar Graph

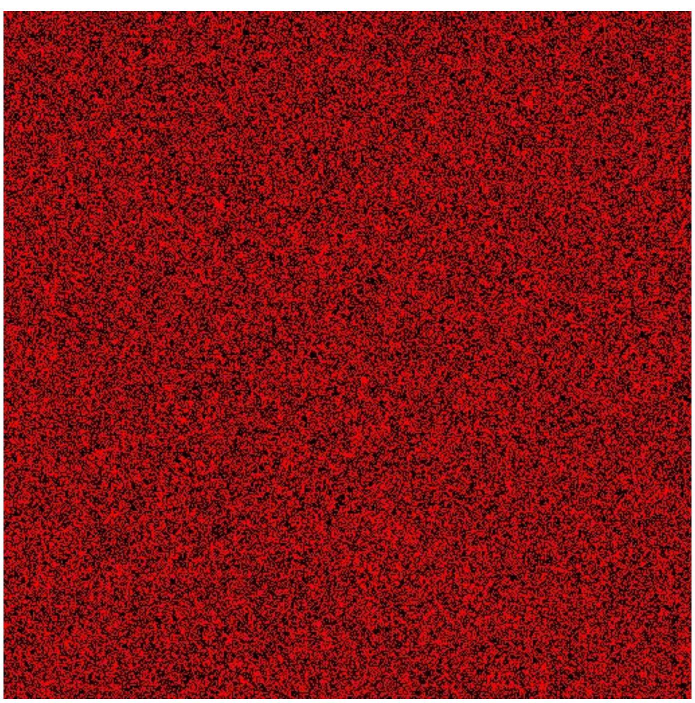

Figure 7. Bitmap of the Output of the Experimental RNG Trial One

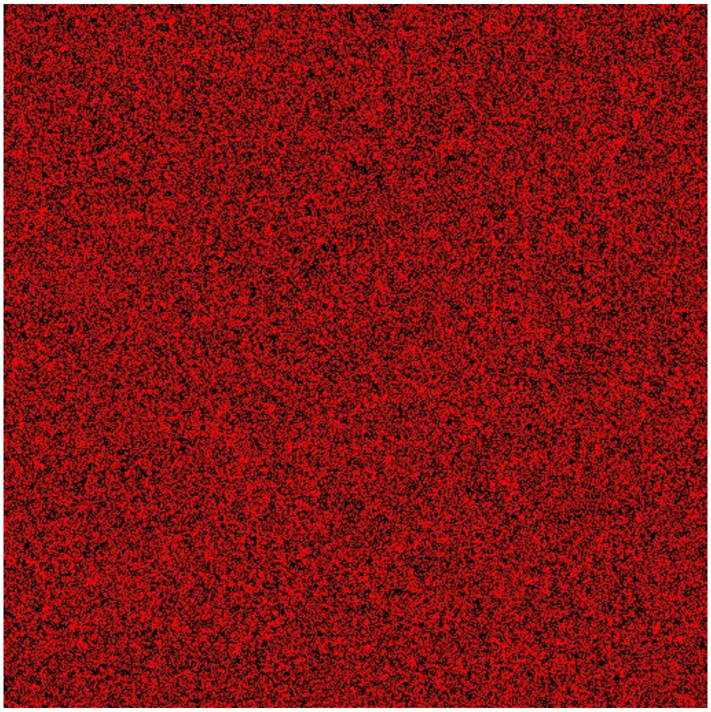

Figure 8. Bitmap of the Output of the Experimental RNG Trial Two

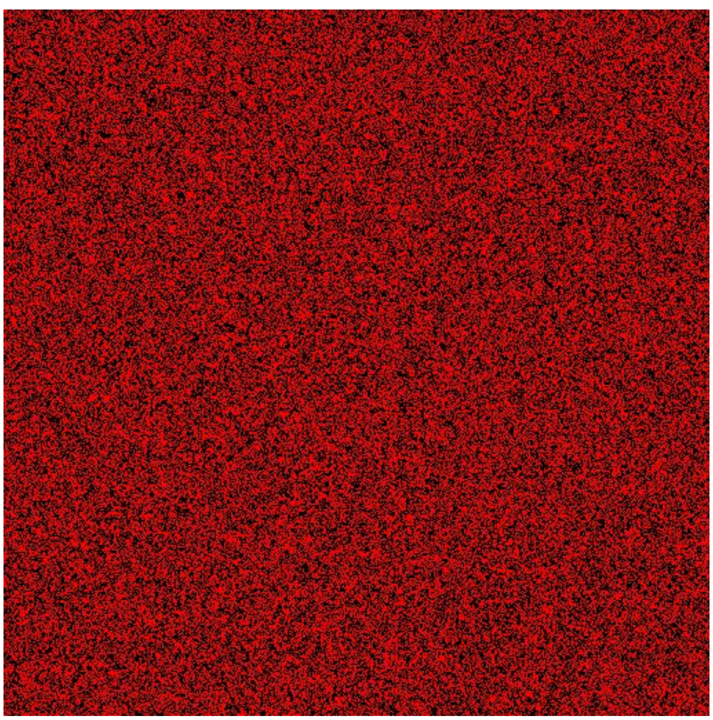

Figure 9. Bitmap of the Output of the Experimental RNG Trial Three

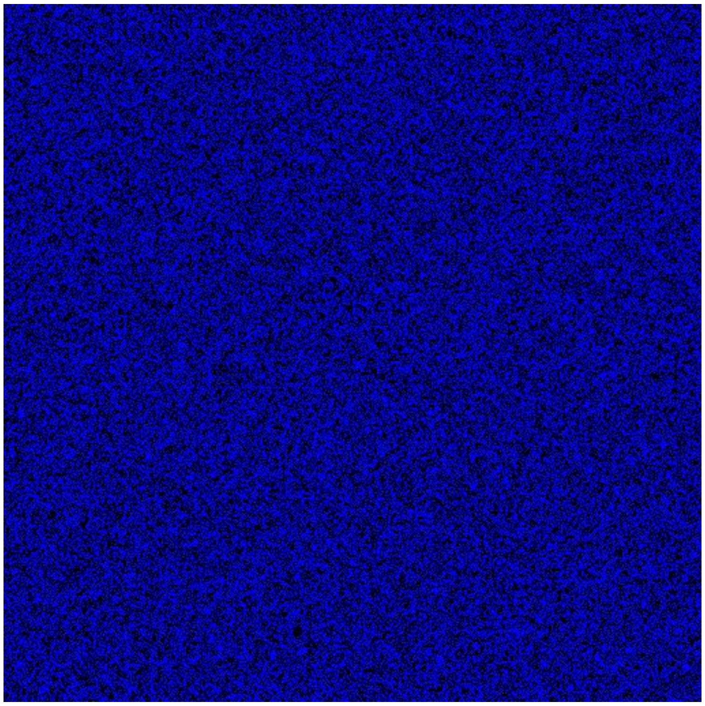

Figure 10. Bitmap of the Output of the Control RNG Trial One

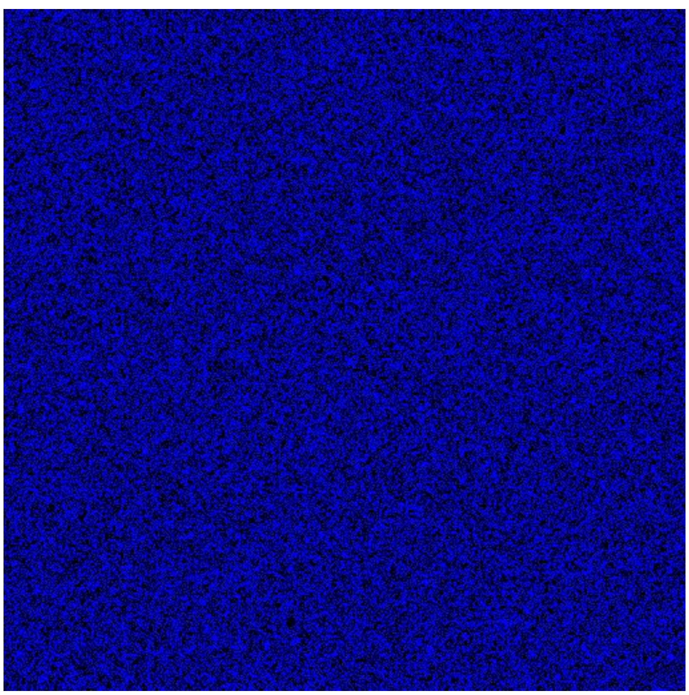

Figure 11. Bitmap of the Output of the Control RNG Trial Two

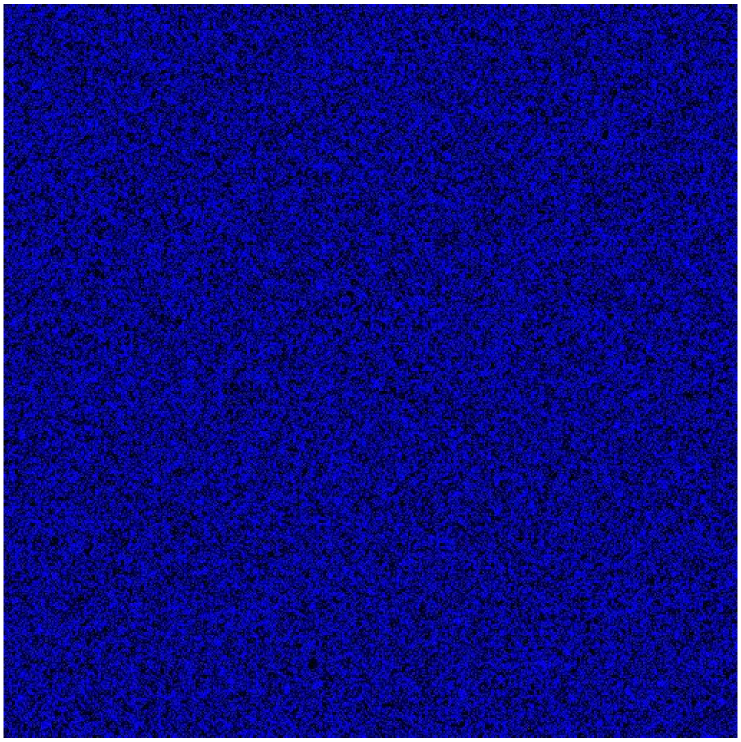

Figure 12. Bitmap of the Output of the Control RNG Trial Three

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//#include <openssl/sha.h>
#include <time.h>

void ramseysleep(int milliseconds)
{
        clock_t start_time = clock();
        while (clock() < start_time + milliseconds);
}

int main()
{
        FILE *fp;
        fp = fopen("example3.input", "a");
        while (1 == 1)
        {
                        ramseysleep(1000);
                        register int *var asm ("eax");
                        srand((int)var);
                        int sum = rand();
                        fprintf(fp, "%10d\n", sum);
                        //printf("%10d\n", sum);
        }

        return 0;
}
```

Figure 13. Code of the Experimental RNG

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

void ramseysleep(int milliseconds)
{
	clock_t start_time = clock();
	while (clock() < start_time + milliseconds);
}

int main()
{
	FILE *fp;
	fp = fopen("control3.input", "a");
	int sum = 1;
	while (1 == 1)
	{
			ramseysleep(1000);
			sum = rand();
			fprintf(fp, "%10d\n", sum);
	}

	return 0;
}
```

Figure 14. Code of the Control RNG

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <openssl/sha.h>
#include "libbmp.h"
#include "libbmp.c"
#include <unistd.h>
#include <time.h>

void ramseysleep(int milliseconds)
{
	clock_t start_time = clock();
	while (clock() < start_time + milliseconds);
}

int main()
{
	int size = 512;
	bmp_img img;
	bmp_img_init_df(&img, size, size);
	for (int y = 0; y < size; y++)
	{
		for (int x = 0; x < size; x++)
		{
			ramseysleep(1000);
			register int *var asm ("eax");
			srand((int)var);
			int sum = rand();
			if (sum % 2 == 0)
				bmp_pixel_init(&img.img_pixels[y][x], 255,0,0);
			else
				bmp_pixel_init(&img.img_pixels[y][x], 0,0,0);
		}
	}
	bmp_img_write(&img, "experimental.bmp");
	bmp_img_free(&img);

	return 0;
}
```

Figure 15. Code to Generate the Experimental Bitmap

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <openssl/sha.h>
#include "libbmp.h"
#include "libbmp.c"
#include <time.h>

void sleep(int milliseconds)
{
	clock_t start_time = clock();
	while (clock() < start_time + milliseconds);
}

int main()
{
	int size = 512;
	bmp_img img;
	bmp_img_init_df(&img, size, size);
	for (int y = 0; y < size; y++)
	{
		for (int x = 0; x < size; x++)
		{
			sleep(1000);
			int var = rand();
			if ((int)var % 2 == 0)
				bmp_pixel_init(&img.img_pixels[y][x], 0,0,255);
			else
				bmp_pixel_init(&img.img_pixels[y][x], 0,0,0);
		}
	}
	bmp_img_write(&img, "control.bmp");
	bmp_img_free(&img);

	return 0;
}
```

Figure 16. Code to Generate the Control Bitmap

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |           filename             |rands/second|
        mt19937|                   example.input|  1.60e+08  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.53579783|  PASSED
      diehard_operm5|   0|   1000000|     100|0.60616033|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.50726861|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.78288838|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.09950032|  PASSED
        diehard_opso|   0|   2097152|     100|0.92067771|  PASSED
        diehard_oqso|   0|   2097152|     100|0.68864727|  PASSED
         diehard_dna|   0|   2097152|     100|0.51698383|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.90214846|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.50809128|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.72516127|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.11899233|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.92851626|  PASSED
     diehard_squeeze|   0|    100000|     100|0.23634530|  PASSED
        diehard_sums|   0|       100|     100|0.27394937|  PASSED
        diehard_runs|   0|    100000|     100|0.65816193|  PASSED
        diehard_runs|   0|    100000|     100|0.43574550|  PASSED
       diehard_craps|   0|    200000|     100|0.76343322|  PASSED
       diehard_craps|   0|    200000|     100|0.94969873|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.94086284|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.82032520|  PASSED
         sts_monobit|   1|    100000|     100|0.81019327|  PASSED
            sts_runs|   2|    100000|     100|0.31191600|  PASSED
          sts_serial|   1|    100000|     100|0.12188324|  PASSED
          sts_serial|   2|    100000|     100|0.16580772|  PASSED
          sts_serial|   3|    100000|     100|0.89016545|  PASSED
          sts_serial|   3|    100000|     100|0.35202098|  PASSED
          sts_serial|   4|    100000|     100|0.60384022|  PASSED
          sts_serial|   4|    100000|     100|0.94236811|  PASSED
          sts_serial|   5|    100000|     100|0.49001427|  PASSED
          sts_serial|   5|    100000|     100|0.48045422|  PASSED
          sts_serial|   6|    100000|     100|0.15982320|  PASSED
          sts_serial|   6|    100000|     100|0.17426548|  PASSED
          sts_serial|   7|    100000|     100|0.40485683|  PASSED
          sts_serial|   7|    100000|     100|0.29913325|  PASSED
          sts_serial|   8|    100000|     100|0.81238598|  PASSED
          sts_serial|   8|    100000|     100|0.68813402|  PASSED
          sts_serial|   9|    100000|     100|0.07826869|  PASSED
          sts_serial|   9|    100000|     100|0.13165877|  PASSED
          sts_serial|  10|    100000|     100|0.38195558|  PASSED
          sts_serial|  10|    100000|     100|0.99682719|   WEAK
          sts_serial|  11|    100000|     100|0.52924307|  PASSED
          sts_serial|  11|    100000|     100|0.83991174|  PASSED
          sts_serial|  12|    100000|     100|0.16147790|  PASSED
          sts_serial|  12|    100000|     100|0.87650686|  PASSED
          sts_serial|  13|    100000|     100|0.90867776|  PASSED
          sts_serial|  13|    100000|     100|0.84490133|  PASSED
```

Figure 17. Sample Dieharder Output from Experimental Data

```
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.98942289|  PASSED
      diehard_operm5|   0|   1000000|     100|0.78696250|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.99207211|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.78658890|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.31853898|  PASSED
        diehard_opso|   0|   2097152|     100|0.19809957|  PASSED
        diehard_oqso|   0|   2097152|     100|0.98435767|  PASSED
         diehard_dna|   0|   2097152|     100|0.03642798|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.83311576|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.05595186|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.92463763|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.87190818|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.29940272|  PASSED
     diehard_squeeze|   0|    100000|     100|0.94857045|  PASSED
        diehard_sums|   0|       100|     100|0.21577444|  PASSED
        diehard_runs|   0|    100000|     100|0.76117957|  PASSED
        diehard_runs|   0|    100000|     100|0.75719989|  PASSED
       diehard_craps|   0|    200000|     100|0.79736718|  PASSED
       diehard_craps|   0|    200000|     100|0.57178832|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.41705550|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.96017371|  PASSED
         sts_monobit|   1|    100000|     100|0.54466869|  PASSED
            sts_runs|   2|    100000|     100|0.31508641|  PASSED
          sts_serial|   1|    100000|     100|0.92946444|  PASSED
          sts_serial|   2|    100000|     100|0.66330023|  PASSED
          sts_serial|   3|    100000|     100|0.95818657|  PASSED
          sts_serial|   3|    100000|     100|0.98462386|  PASSED
          sts_serial|   4|    100000|     100|0.42797576|  PASSED
          sts_serial|   4|    100000|     100|0.92908147|  PASSED
          sts_serial|   5|    100000|     100|0.87479662|  PASSED
          sts_serial|   5|    100000|     100|0.54242918|  PASSED
          sts_serial|   6|    100000|     100|0.40321081|  PASSED
          sts_serial|   6|    100000|     100|0.14789883|  PASSED
          sts_serial|   7|    100000|     100|0.74082778|  PASSED
          sts_serial|   7|    100000|     100|0.93603216|  PASSED
          sts_serial|   8|    100000|     100|0.99128477|  PASSED
          sts_serial|   8|    100000|     100|0.33766603|  PASSED
          sts_serial|   9|    100000|     100|0.11567696|  PASSED
          sts_serial|   9|    100000|     100|0.54562829|  PASSED
          sts_serial|  10|    100000|     100|0.88073498|  PASSED
          sts_serial|  10|    100000|     100|0.95330654|  PASSED
          sts_serial|  11|    100000|     100|0.57721968|  PASSED
          sts_serial|  11|    100000|     100|0.38891208|  PASSED
          sts_serial|  12|    100000|     100|0.29728817|  PASSED
          sts_serial|  12|    100000|     100|0.20623876|  PASSED
          sts_serial|  13|    100000|     100|0.30039970|  PASSED
          sts_serial|  13|    100000|     100|0.91855609|  PASSED
```

Figure 18. Sample Dieharder Output from Control Data

Figure 19. Computer Used to Run Both Control and Experimental Groups

Calculations

The tabulated data is taken directly from the output of the dieharder tests; there are no intermediate calculations. The bash command used to generate the output was “*dieharder -f [input filename] -a >> [output filename]*”. The angle brackets redirected the output to a text file. The p-value from this file was recorded, which is the probability that a random number would have produced the result it did (How do I). Effectively, this means that if the p-value is very close to 0 or very close to 1, the data performed poorly on that test. The parentheses in the tables saying PASSED or WEAK are also from dieharder itself, relating if the given data performed well on the specific statistical test. Because of the nature of random numbers, it is inevitable that, run enough times, an RNG will generate some numbers that seem unlikely to be random. That is why three trials were performed - if the datum had failed repeatedly on specific tests, it would be indicative that the RNG was not suitable on that measure (How do I). The other outputs produced by dieharder were test-specific, and not relevant to a holistic discussion of the effectiveness of the tested RNG. The bitmaps shown are colored black if the output number is odd, and red or blue if the output is even. In a random spread of data, there should be an approximately equal spread of odd and even numbers.

Discussion

Conclusion

In this experiment, both the built-in C programming language RNG and a natural computational entropy RNG were tested. Both RNGs passed the vast majority of the dieharder statistical tests. In total, the control RNG returned WEAK on seven tests. In total, the experimental RNG returned WEAK on exactly seven tests as well. Neither of the RNGs outright failed any statistical test. By all measures, the two RNGs performed extremely similarly with the dieharder suite. Furthermore, an analysis of the bitmaps of the two RNGs outputs reveals no discernible pattern to the eye. Taken together, this experiment has provided evidence that natural computational entropy can be used as a source of randomness for RNGs on par with modern random number generation technology. Thus, the hypothesis was supported.

Applications

This experiment is of use to scientists through demonstrating the effectiveness of natural computational entropy as a source of randomness and highlighting its merit. This experiment suggests that operating systems developers may want to consider using computational processes in the computer as a basis for a system's random number generation. Doing so would add another layer of randomness at no additional cost, since entropic facets of computing, like the contents of registers, are basic, essential components of modern computers. Furthermore, this experiment suggests that computer cryptographers and cybersecurity professionals can use natural computational entropy in their practice, which is an urgent field that affects the global economy (Cyberattacks now cost, 2019). It can be used to encrypt online communication so that it is difficult for a malicious actor to predicate the key to interpret the cipher code. This practice of using random numbers in cybersecurity is well established, and natural computational entropy can contribute to that by eliminating the need for an expensive lab setup to observe minute phenomena. This would make the internet much easier to secure for both professionals and amateur computer scientists who do not have access to research-grade equipment.

Limitations

Randomness can never technically be "proved" (Gordon, 2014). Knowing enough information about any system allows you to predict everything within that system. For example, if one flips a coin, it is technically possible to predict which side will face up if you know the contour of your thumb, the coin, the speed of the wind, and the shape of the floor. However, it is nigh impossible to know everything about some systems, and so they can be considered practically random. Additionally, further testing with different statistical tests on several different computer architectures would help to confirm the validity of these results.

Error Analysis

While the computer was kept in the same place for all number generation, experiencing the same workload, it is possible that the numbers were influenced by their environment. For example, the programs open in the background could have theoretically influenced the register values, though this is unlikely. Furthermore, the environment the computer was in underwent normal variations in temperature and noise. As always, there is the possibility of human error. This experiment yielded vast amounts of data for every trial, and in the process of recording them the researcher may have misread a number or made a typo.

Future Analysis

It would be beneficial to this experiment's results to have the code peer reviewed by experienced programmers, to verify the RNGs used were logical and contained no errors. It would be worth investigating the performance of the RNGs on different operating systems, as the operating system is intimately involved with managing the register values. Additionally, since registers are not the only source of natural computational entropy, there may exist a more entropic computational process that would be better suited for random number generation. Testing this process would also be of use to the scientific community. Finally, testing random number generators on different kinds of hardware would be worthwhile as well.

References

Cyberattacks now cost companies $200,000 on average, putting many out of business. (2019, October 13.). https://www.cnbc.com/2019/10/13/cyberattacks-cost-small-companies-200k-putting-many-out-of-business.html

Gordon, A. (2014, June 14). *Does Randomness Actually Exist?* Pacific Standard. Retrieved November 22, 2020, from https://psmag.com/social-justice/free-will-enigma-machine-code-randomness-actually-exist-89285

Haahr, M. (2020). *Introduction to Randomness and Random Numbers*. https://www.random.org/randomness/

How do I interpret the results from dieharder for great justice [Online forum post]. (n.d.). Stack Overflow. https://stackoverflow.com/questions/29366721/how-do-i-interpret-the-results-from-dieharder-for-great-justice

Hussain, I., & Khalique, A. (2019, June). *Random Number Generators and their Applications: A Review*. https://www.researchgate.net/publication/335920952_Random_Number_Generators_and_their_Applications_A_Review

International Rules for Pre-College Science Research: Guidelines for Science and Engineering Fairs 2020–2021. (2021). In *International Rules for Pre-College Science Research: Guidelines for Science and Engineering Fairs 2020–2021*. Retrieved November 22, 2020, from https://sspcdn.blob.core.windows.net/files/Documents/SEP/ISEF/2021/Rules/Book.pdf

Palm Beach Regional Science & Engineering Fair 2020 -21 Handbook for Parents, Students, and Teachers. (n.d.). In *Palm Beach Regional Science & Engineering Fair 2020 -21 Handbook for Parents, Students, and Teachers*. Retrieved November 22, 2020, from https://docs.google.com/document/d/165S_Yj5MttssMM0ytzIBrXRHIXvieE__nc17EeZbikI/edit

Processes [Lecture notes]. (n.d.). University of Illinois at Chicago. Retrieved November 20, 2020, from https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/3_Processes.html

Rubin, J. M. (2011, November 1). *Can a computer generate a truly random number?* Retrieved November 15, 2020, from https://engineering.mit.edu/engage/ask-an-engineer/can-a-computer-generate-a-truly-random-number/

Sidhpurwala, H. (2019, June 5). *Understanding random number generators, and their limitations, in Linux*. Retrieved November 15, 2020, from https://www.redhat.com/en/blog/understanding-random-number-generators-and-their-limitations-linux

Society for Science and the Public (2016-17). International Science and Engineering Fair 2016-17: International Rules & Guidelines. Washington, DC: Society for Science and the Public.

The CPU and the fetch-execute cycle. (n.d.). BBC. Retrieved November 15, 2020, from https://www.bbc.co.uk/bitesize/guides/zws8d2p/revision/2

2020-2021 RULES SUPPLEMENT To the International Science and Engineering Fair Rules. (n.d.). In *2019-2020 RULES SUPPLEMENT To the International Science and Engineering Fair Rules*. Retrieved November 22, 2020, from https://ssefflorida.com/wp-content/uploads/2016/10/2020_21-SSEF-Rules-Supplement8.19.2020-Autosaved.pdf