**Methodology**

**3.1 Sample Selection**

Before testing could begin, suitable true random and pseudorandom generators had to be chosen. As outlined previously (**See Section 1: Introduction**) all algorithms tested represented functions from in-use languages and sites as well as industry standard packages for said languages. An appropriate digital generator needed to:

* Be in active use in either a commercial or scientific environment.
* Be able to produce at least 500 values in a single generation.
* Be able to produce a double or integer output or a non-numeric output within a chosen format (e.g., Coin Faces or Cards from a Deck).

These requirements ensured that any generators chosen for testing would represent tools currently available to both professionals in industry and the public. This also ensured that the generators in question would provide outputs suitable for individual and grouped analysis. In the event where an output would differ from the conventional integer or double values, primarily when producing a ‘random’ combination of cards from a deck, suitable generators would be required to be able to have their outputs adjusted accordingly.

In total, six pseudorandom sources matching the criteria listed above were chosen for analysis. Three industry standard languages were chosen, these being C#, Python and JavaScript, as well as reproduced algorithms of the Lehmer Generator and Von Neumann’s Middle Square Method (both written in C#). Finally, data was taken from Random.Org, a site built in 1998 by the School of Computer Science and Statistics at Trinity College in Dublin (**Random.Org, 2023**) which serves as one of the most well-known sites for random number generation on the internet. Four physical true random sources were also chosen. Dice, coins and playing cards, three sources of random chance that have been used for thousands of years were sampled for comparison to their pseudorandom counterparts. White-noise data from three different locations across Plymouth was also recorded in hopes of providing true random sequences to compare against the integer and double sequences given by the pseudorandom generators.

C# is an object-oriented, component-oriented programming language (**Microsoft, 2023**) that is primarily used for the creation of applications using the .NET framework. It is used by a variety of companies including Microsoft, Stack Overflow and Trustpilot and can be utilised to create both web and desktop applications. For this reason, it was chosen as one of the pseudorandom data sources to be tested. Three different integer sequence samples were taken from C#, two from the *rand.Next()* function and the third from the *RandomNumberGenerator.GetBytes()* function provided to C# by the Cryptography library. When using the *rand.Next()* function, the seed given to the algorithm can be chosen either by the system itself using the PC clock or by being input by the user. The *rand.Next()* function also produced two outputs for dice roll, coin flip and card shuffle simulations.

Similar to C#, Python is a high-level, general purpose programming language. It supports multiple programming paradigms, including structured, object oriented and functional programming (**Wikipedia (X), 2023**) and is used by companies such as Google, Dropbox, and Netflix as well as by data scientists alongside R for data analysis. Five sequence samples were taken from Python, three sets of integer values and two sets of double values. This is because three different functions were sampled from Python: *Randint()*, *Random()* and from the NumPy library *Numpy.Random()*. As with the *rand.Next()* function in C#, both *Random()* and *Numpy.Random()* can have their seeds selected either by the system clock or manually by the user. In addition to the integer and double sequences the Python functions were also used to produce outputs for dice, coin, and card simulations, with these simulations also being adjusted to use either the system or user input seed.

JavaScript is a programming language that is one of the core technologies of the World Wide Web, alongside HTML and CSS. As of 2023, 98.7% of websites use JavaScript (**Wikipedia (X), 2023**) so including it as a data source for this investigation was deemed essential. In addition, this would allow for comparisons to be made regarding the quality of Google’s pseudorandom generator, which also uses the *Math.Random()* function provided by JavaScript but isn’t capable of generating large batches of values at once. Only one integer sequence value was taken from JavaScript as the *Math.Random()* function provided doesn’t allow the user to manually input a seed for generation and instead only uses the PC clock to determine seed. Similarly, only one set of outputs were given for the coin and dice simulations and due to the limitations of *Math.Random()* only being able to produce integer sequences, attempts to produce a suitable card simulation were not included.

As outlined in Section 2, the Lehmer Generator was first coined by Park and Miller in 1988 as a ‘minimum standard’ for a reliable pseudorandom generator. As its variants still see use in functions today like *minstd\_rand* for C++, including the generator in this investigation was an obvious choice. The double sequence outputs given for this data source are split into four: two using a ‘version 1’ integer based and real based algorithm and two using a ‘version 2’ integer based and real based algorithm. The use of either integers or real values in these algorithms refers to the variable types used to store the seed, seed multiplier and modifier values used during calculation. As with the JavaScript functions, due to the specific type of outputs given by the Lehmer Generator it was unsuitable to be used for coin, dice, or card shuffle simulations.

In contrast to the Lehmer Generator, Von Neumann’s 1949 Middle Square Method is considered to be a highly flawed method for many practical purposes (**Wikipedia (X), 2023**). While other pseudorandom algorithms were included in this investigation due to their continued use in commercial or scientific fields, the Middle Square Method was included due to the ease in recreating it and as an example of a generator proven to be flawed in producing random numbers. One integer sequence was outputted from the Middle Square algorithm used and much like the Lehmer Generator, due to the specific output of the algorithm, which also doubles as the seed for its next iteration, dice, coin, and card shuffle simulations could not be performed.

Random.org, created by Dr Mads Haahr in 1998, aims to offer true random numbers to anyone on the Internet (**Random.org, 2023**). This is done using an algorithm based on atmospheric noise, a principle that was used as the inspiration for the white noise data collected for this investigation. The claim, that through the use of atmospheric noise this digital generator was capable of producing true random instead of pseudorandom outputs, made the inclusion of Random.org data invaluable as not only did the generator meet the criteria specified above but if it could be proved through empirical testing that this generator is capable of producing true random outputs then it would serve as a benchmark when comparing the other pseudorandom generators featured. Three outputs were taken from Random.org: an integer sequence, a dice roll simulation, and a coin flip simulation. Due to the limitations in the format of possible outputs, a card shuffle simulation wasn’t considered to be viably possible with this generator.

While the data provided by Random.org could be used for comparison between true random and pseudorandom numeric sequences, the best tools for aiding in the evaluation of the ability of pseudorandom generators to replicate true random sequences were physical generators. Much like many of the pseudorandom generators used in this investigation, physical generators have been used in both commercial and scientific environments and are able to produce outputs in a variety of formats. Dice, coins and playing cards, although more commonly seen as components in games rather than number generators for study were chosen for this investigation due to the large number of commercial tools available that are designed to replicate them in a digital environment. All the programming language functions, as well as the data given by Random.org, evaluated in this investigation were capable of simulating one or more of these physical generators. Dice rolls were simulated to test a generator’s effectiveness when the range of possible results were limited from 0-100 to 1-6. Coin flip simulations expanded upon this idea by limiting possible results to a binary 0 or 1 while card shuffle simulations aimed to evaluate a generator’s effectiveness when both the number of iterations was reduced (from 500 to 52) and the range of possible results was reduced.

The use of white noise for the purpose of random number generation has been explored across several papers (**See Section 2: Literature Review**). In these examples, the most common method involved identifying the colour code of a sequence of pixels, then correlating the colour ids to bit values in a true random output. For this investigation a similar method was used; however, in place of video data, audio data was used instead. This audio data was then converted to a waveform and the audio levels were used to form a true random sequence of integers. Three sets of audio data were collected for this investigation, with each being recorded at a different location around the Plymouth area. By collecting from three distinct locations, three noticeably different waveforms could be created, as the white noise from all these areas are from different sources. The first location sampled was a busy roundabout, the second was a public park, and the third was next to the sea.

**3.2 C# Implementation**

The C# pseudorandom generators were created in a .NET framework project. Because the development environment uses a different method for generating a base seed in a .NET framework file compared to a .NET core file, it was important that a framework project was used. This was because .NET framework files, much like files in other languages, use the system clock for generating a ‘random’ base seed and being able to use the same method ensured consistency between languages. Due to the object-oriented nature of C#, individual functions were used to house each of the generators. These functions would then be called by the program upon being run. In total nine functions were created. To be able to analyse generator outputs not only by the C# implementations but also by all pseudorandom and true random generators external file storage was a necessity. While there are many file types capable of storing the data given including notepad, CSV, or excel it was decided that JavaScript Object Notation (JSON) files would be used. JSON is standard in many areas of the computing industry and can be read and written to by all the languages sampled in this investigation. These files are also designed to be used for storage of data structures such as arrays or lists which were also used throughout the implementation of the pseudorandom generators.

A screenshot of a computer program

Description automatically generated

*Figure 1. A screenshot of the Main body of the program and the Implementation 1 call*

Before any functions are called, a set of lists are created to hold the various outputs generated. When the function is called, as seen with *RandImplementation1* in Figure 1, the relevant output list is passed by reference into the function, allowing it to be modified outside of the program’s Main body. *ReturnValues*, *ReturnCardValues*, and *ReturnDiceValues* are all lists of types double, string and int that store outputs from the generators. Once a function has been called, a for loop is used to print the contents of the output list to the user before the output list is converted to a JSON object and exported as a JSON file.

A computer screen with green text

Description automatically generated

*Figure 2. A screenshot of the RandImplementation1 function*

*RandImplementation1* uses the *rand.Next()* function and a system generated seed to produce 500 values between 0 and 100. When first called, the *Values* list which is a reference to the *ReturnValues* list is emptied. Then a new Random variable is created which contains the *.Next()* method. By leaving rand without an integer value during declaration, the system will automatically produce a seed value based on the system clock of the PC. A for loop is then set to iterate 500 times and during each iteration a random integer is added to the *Values* list.

A computer screen with green and white text

Description automatically generated

*Figure 3. A screenshot of the RandImplementation2 function*

*RandImplementation2* is nearly identical to the former implementation. The only difference between both functions is in the declaration of *rand*. When deciding on a seed to provide as an alternative to the system clock, the value 30102000 was chosen. This was because the more traditional algorithms such as the Lehmer Generator or the Middle Square Method used six to ten character seeds for optimal calculations. As only the length of the seed determined effectiveness and not the number itself, the author’s birthday (30/10/2000) was considered valid.

A computer screen shot of a program

Description automatically generated

*Figure 4. A screenshot of the RandImplementation3 function*

Unlike the previous rand functions, *RandImplementation3* utilises the C# Cryptography library. Included in this library is the *RandomNumberGenerator* class which provides a cryptographically secure set of bytes. In addition, where previous functions required the ReturnValues list to be provided, RandImplementation3 requires the user to provide the number of iterations desired which is stored in the *size* integer variable. When called, the function will create the *values* list to store the generator output, then by calling the *.GetBytes()* function the generator will produce a *size* length collection of random bytes. Once generation is complete, the values list is returned to the program Main body where the bytes are stored inside the emptied *ReturnValues* list and then exported as a JSON file.

A screenshot of a computer program

Description automatically generated

*Figure 5. A screenshot of the RandCoinSimulation1 function*

*RandCoinSimulation1* used the same *.Next()* function as seen in *RandImplementation1* however by adjusting the desired threshold from 100 to 2, the possible outputs became only 0 (heads) or 1 (tails). As seen previously, upon being called *RandCoinSimulation1* emptied the *Values* list and used a for loop to iterate 500 times. Due to the use of the system clock for seed creation, this function and implementation 1 cannot be completely reproduced, unless a user knows exactly when the code was run and how the clock data can be formatted into a usable seed.

A computer screen with green and white text

Description automatically generated

*Figure 6. A screenshot of the RandCoinSimulation2 function*

As with implementation 2, *RandCoinSimulation2* operates the same as previously, however instead uses the predetermined 30102000 seed. Although an essentially binary output is given for both coin simulations, the ReturnValues double list is used for data storage as the outputs given, unlike in other simulations such as card shuffles, do not require specialised storage. Only two coin simulations were created for the C# implementation. This was due to the output given by the cryptographic generator not being compatible with the specific output limitations of the simulations. While able to produce bytes, the cryptographic method shown in *RandImplementation3* could not be altered to only generate values between 1 and 6.

A computer screen shot of a program

Description automatically generated

*Figure 7. A screenshot of the RandCardSim1 function*

The decision to use JSON files proved valuable again when designing the card shuffle simulations. As well as needing to output to JSON, the unshuffled ‘deck’ used by the card simulation functions was able to be read into the program as a JSON file. The Newtonsoft.JSON library for C# allowed for the creation of the StreamReader object *r*, which took the input deck data and stored it in a manipulatable string list. After this, a for loop was iterated through that would randomly select a *ChosenCard* from the deck and then add that card to the ‘shuffled’ deck. Once the new card had been added to the shuffled deck, it was removed from the input deck via its list position to prevent it from being selected again.

A screen shot of a computer program

Description automatically generated

*Figure 8. A screenshot of the RandCardSim2 function*

Figure 8 shows the *RandCardSim2* function, which much like RandCardSim1 used an input deck JSON file to produce a ‘shuffled’ collection of 52 cards. As with the previous method, the .*Next()* function allowed for pseudorandom card selection, however the 30102000 user given seed was used in place of the system clock determined seed. Only two card simulations were created in the C# implementation because, much like with the coin simulations, the cryptographic method provided was unable to produce outputs suitable for card shuffling. In addition, both card simulations used the same deck input file.

A computer screen shot of a program code

Description automatically generated

*Figure 9. A screenshot of the RandDiceRoll1 function*

A computer screen shot of a code

Description automatically generated

*Figure 10. A screenshot of the RandDiceRoll2 function*

The dice roll simulations were developed with the same method as the coin simulations shown above. A *rand* variable was created using the Random class, first in *RandDiceRoll1* with the system clock seed and then again in *RandDiceRoll2* with the user given seed. A for loop allowed for 500 consecutive iterations, with each iteration producing a value between 1 and 7 (not including 7). The maximum bound available for the generator had to be increased due to a particular quirk of the function, which when given the bounds 1-6 would only generate values up to 5.