**Discussion**

**5.1 Data Evaluation**

After collecting and analysing the data provided by the real and pseudorandom generators sampled, the final step was to evaluate each implementation’s overall performance. As well as these individual evaluations, where possible the generators were compared to each other, since another aim of the investigation was to attempt to identify which generators performed better than others. A key point that was considered during this discussion was made in Section 4.9, in which it was stated that when reviewing all the test data gathered and attempting to compare suitable generators the datasets able to not necessarily outperform but remain constantly effective across many tests should be considered the more effective solutions.

The C# implementations were broken down into two main categories: the Rand functions and the Cryptographic function. The Rand implementations, both seeded and unseeded, performed well in the dice, coin, and card simulations which showed that the generators effectiveness remained relatively unaffected by input and output limitations. During Chi-Squared testing, all the Rand implementation datasets ranked between the desired distribution ranges, with the dice simulation data performing just slightly better than the coin simulation data. When producing shuffled deck data, the distribution of suits for both implementations was acceptable, with most groups being between two or three cards large. None of these tests presented any clear weighting or favoured outputs within the functions. During each of the dealing tests, the hands provided aligned with the expected odds of a poker deck, with the most common hands being High Card or Pairs. The distribution of values, seen for instance in the Kolmogorov-Smirnov test, with the numeric sequence Rand datasets followed the expected trend of pseudorandom data and while never being the top performers in any of the tests conducted, both the seeded and unseeded Rand implementations consistently passed. Unfortunately, the same couldn’t be said about the Cryptographic implementation which performed poorly in many of the numeric sequence tests conducted. Despite this however, it is still believed that the function can produce a valid pseudorandom sequence and the main issue presented by the tests was that they weren’t designed for the cryptographically secure sequences being produced. The purpose of such a function is to offer the user a non-reproducible collection of bits that can be used for encryption, not a mass-produced collection of values simulating randomness.

Python provided two different options for pseudorandom generation, default Rand and NumPy. Both performed well in the simulation tests, achieving sufficiently random results and a sufficiently random distribution of shuffled cards. In the poker and Texas Hold ‘Em distribution tests, both the Python and NumPy implementations provided a likely variety of hands, featuring Three and Four of a Kind. In the numeric sequence tests both implementations performed very well, with all showing an expected trend in the Kolmogorov-Smirnov test and achieving the closest P-values to the optimal 0.5 in the Serial, Gap, Runs, and Serial Correlation tests. This is counterbalanced however by a poor performance by the Rand functions in the Birthday Spacings test and low test statistics for NumPy in the Serial Correlation test. Seeding also played a more significant role in the Python implementations than in the C# implementations, often with a large difference between values depending on the seed provided to the algorithms.

Testing began poorly for JavaScript, with anomalous frequencies of dice outcomes and an unsuitable Chi-Squared statistic. However, this method improved in further tests with far more expected values in the coin simulation. The trend seen in the Kolmogorov-Smirnov test was as expected. Although the dataset featured no standout performances, and the generator lacks any ability to alter the seed used, almost all JavaScript results were above average. In tests such as the Serial Correlation and Gap, the dataset was among the top performing implementations. While the results of testing indicated that limitations on output can notably affect performance and the generator performs less effectively than Python, JavaScript can still be considered a valid pseudorandom number generator for most commercial projects, although its use in a scientific environment wouldn’t be recommended.

Considering the claim that Random.org used atmospheric noise to produce true random number sequences, expectations regarding its performance were high. This dataset, perhaps even more so than the JavaScript implementation, embodied the statement that a consistently well performing generator will be better overall than a generator with a few notably exceptional results. In all tests performed, both simulation based and numeric sequence based, the Random.org dataset remained a not top performing generator but a repeatedly well scoring one. The only exceptions to this being in the Gap and Serial Correlation tests. While these results cannot say definitively whether the data can be considered true random, they can say that the data collected serves as a highly effective pseudorandom source, comparable to JavaScript or C#.

The Lehmer Generator data was split into two main types of implementations, integer based and real based. Throughout testing it became clear that the real implementations were significantly more effective than the integer implementations. The trends shown in the Kolmogorov-Smirnov test highlight this, with the integer trends completely different to the expected and the real trends. In other numeric sequence tests like the Serial and Gap tests the integer versions were shown to consistently either fail or score substantially worse than their counterparts. The two most notable exceptions to this were the Birthday Spacings test, in which all four versions failed, and the Serial Correlation test where the Integer Version 2 implementation came the closest to the target of 0. Overall, the Real versions of the Lehmer Generator performed acceptably, with results on par with those found in JavaScript.

Going into testing, it was assumed that the Middle Square Method would perform poorly. Throughout the numeric sequence tests performed as expected the Middle Square data did fail and was easily the worst performing dataset evaluated. Apart from Birthday Spacings, it was unable to pass a single test. Even when passing, the Middle Square data was the lowest performing passed dataset sampled. Again, this was not unexpected. The Middle Square Method served as the starting point for pseudorandom generation and as such was going to perform worse than its more modern and better designed contemporaries. It was not included in this investigation because it was considered a viable alternative to other more popular generation methods, but because it was prudent to evaluate its effectiveness in comparison to what’s on the market now, to show the improvements made after almost 75 years.

The final datasets to discuss were the White Noise implementations. These were quite different from the other datasets sampled, as the data provided came directly from a source rather than through an algorithm. This is reflected in the test results, which unfortunately were often poor. Much like the Middle Square Method, the White Noise data failed most of the tests they were surveyed in, with exception of the Birthday Spacings test in which they were the highest performing implementations. The reason for this is two-fold. For one the data sampled wasn’t given the same limitations as other implementations, meaning the data had a far wider range of possible values to sample from. Secondly, many of the tests were designed with likely possibilities in mind and so when evaluating a dataset that contained completely random sequences, this true random nature was seen as too unlikely to be valid in the majority of circumstances.