Noah Johnson

CSC 475.01

Prof. Bryan Catron

November 18, 2021

Techniques, Applications, and Limitations of Random Number Generation

Introduction

Computing is based on logical, structured data storage and processing. Every instruction is interpreted, parsed, stored in memory, and then executed based on a set scheduling priority, and this process applies regardless of the complexity of the script being run. However, randomness may be preferred over logic in numerous disciplines for which we may use computers, among them cryptographic encryption, selection of representative samples for statistical inference, Monte Carlo simulations, and machine learning algorithms such as random forests. True randomness as I define it involves two key aspects: a long-run uniform distribution of values (every value has the same probability of being chosen), and unpredictability of the current and future outcomes (the latter condition jointly confers independence upon a truly random sequence). Naturally, the logical computing paradigm makes both aspects nearly impossible to attain in their entirety. Most widely-used random number generators rely on some pseudo-random sequence of numbers or physical phenomena bounded by natural laws, though advancements have been made in algorithms approximating randomness and quantum computing which embraces entropy more. True randomness is something not widely available, if not unattainable, in the current state of computer science, and the discussion concerning whether it is possible at all is psychologically compelling. Is there true randomness in this world at all, and if

so, can it be harnessed by machines for use in our most pressing security and simulation applications? At this point, true randomness returned from computational machines seems too utopian to be a reality without some novel breakthrough. By understanding the current state of accessible random number generation, we can evaluate the improvements behind more mathematically rigorous randomization techniques (such as quasi-random sequences, hardware random number generation, randomness extractors, and quantum random number generation) while simultaneously evaluating how random and impartial number generation can become.

Why Not Rely on Humans?

Given the logical nature of computation, one would expect humans, with their abundant creativity, to be better at choosing random sequences of numbers. However, the opposite tends to be true. In fact, the first issue with finding true randomness is unpredictability of the results, as there exists inherent human bias which occurs when we choose what we perceive as random numbers. As humans, we are prone to choose random numbers in some pattern, even when we do not intend it. For example, there are artificial intelligence algorithms which predict how a user predicts some binary event with significantly better than random chance because it picks up on the user's tendencies for choosing seemingly random outcomes. This is akin to the colloquial argument that if a person was asked to choose a random number between 1 and 10, the most common choice is 7 because it seems "more different" than any of the other options. Hence, the subjectivity of human number generation, which can hardly conceive rational or real numbers, towards interesting values is clear. This clearly begets the idea that humans cannot generate a uniform distribution of values either, because some human distribution will include only values which the human has explicitly thought about. Very few humans actively consider

the number 3905746 when asked for a random positive integer, so we see the human distribution display significant right skew which is unsuitable for uniformity. Luckily, the solution to our ineptitudes relies on the powerful problem-solving ability of modern computers. Many computer-accessible RNGs use a pseudo-random algorithm to return a value which, to an average human eye, seems to be completely random. The collective confoundment of humans only increases with inputs like thermal noise, current system time, and ambient light readings which humans cannot gauge consistently and effectively. However, if security applications used randomization from these pseudo-random algorithms, then intruders can much more easily predict the encryption and secret keys of some cipher and consequently compromise an application's sensitive data. For this reason and more, high-quality, cryptographically-secure random number generation must be understood and implemented, as the complexity of these algorithms is not going to compromise any improvements in randomization quality.

SOURCES: (https://www.clinicalkey.com/#!/content/playContent/1-s2.0-S030698770700480X?returnurl=null&referrer=null , https://web.media.mit.edu/~guysatat/MindReader/index.html , https://tashian.com/articles/a-brief-history-of-random-numbers/)

First Attempts at Harnessed Randomness

When humans first considered randomness, they first attempted recording the outcomes of physical objects such as dice, wheels, and even turtle shells, which were poked to determine which one cracked first. In the words of late nineteenth-century statistician Francis Galton, "As an instrument for selecting at random, I have found nothing superior to dice…they tumble wildly about, and their positions at the outset afford no perceptible clue to what they will be." Galton

has a point. There's no way to document the necessary forces, velocities, and trajectories of any given dice roll and the preliminary shakes it requires, so for random numbers between 1 and 6, dice seem to be sufficiently random. The issue with relying on dice and other physical objects, however, is the lack of flexibility with its output. Numbers can be generated independently of one another, 1 through 6, but if the results of multiple dice rolls are added together to simulate the choice of higher numbers (for example, when choosing a number 1 through 12), some values (in this case, 1) are impossible or unlikely to attain, and some (6, 7, 8) are artificially too common. Hence, we desire randomization which can return a truly (or sufficiently close to it) random sequence of numbers across any conceivable range of real numbers.

Once physical computers were invented, naturally, the thoughts of many influential electrical engineers and mathematicians became fixated on what calculations can be carried out using machine power, not least of which John von Neumann in 1949. He determined a way that a machine could generate a sequence of pseudo-random numbers, ideally infinitely. The method of middle-squares, as he called it, was very simple. First, take some four-digit seed (if the number is three digits of less, add leading to the front to make four digits). Square it, make it an eight-digit number, and use the middle four digits as the next seed, repeating the process for as long as desired. However, consider that many numbers have consistent patterns which occur after multiple applications of a square; for example, when squaring numbers which end in a 1, the result also ends in 1, and this persists infinitely. If a seed were engineered which consistently resulted in 1, then the effectiveness of the randomization would be cut off by tenfold. This often occurred with the method of middle-squares since the pattern tended to repeat after some small sequence length; this time for any pseudo-random generator is called its period. Ideally, we want

the period of a PRNG to be sufficiently long that any conceivable sequence length does not repeat, or converge to some value. In von Neumann's case, the sequence eventually converges to 0 after many repetitions, and it is clear to see how this negatively affects randomness. While von Neumann himself did not keep arithmetic approximations of randomness in high regard, he realized that these methods are easiest for computers to handle and if executed correctly, they satisfy the average user's strictest demands.

SOURCES: (https://www.vice.com/en/article/wnj48y/how-to-make-a-random-number , https://tashian.com/articles/a-brief-history-of-random-numbers/)

Pseudo-Random Number Generators: Why Change if They're Already Implemented?

This compromise of effectiveness and efficiency is what the writers of the Python-native random library and the Java-native Java.util.Random class resolved when determining how best to implement randomization in their programming languages. In the decades after von Neumann, computing technology improved enough to handle much more complex mathematics in a reasonable time efficiency, and better pseudo-random sequences became the basis for randomization within the Python and Java languages. In Java's case, the relevant algorithm was first described in Donald Knuth's publication *The Art of Computer Programming*, and hence, it has achieved widespread use in other languages such as C++, Visual Basic, PHP, and SQL. The Java.util.Random library leverages a PRNG called the linear congruential generator. The fundamental idea is we generate each number following the initial state (which is just the starting seed specified to the randomizer) using a linear function of the form $x_n = ax_{n-1} + c \bmod m$ to create a sequence of "random" numbers. The goal is to choose values of a, c, and m that will accommodate a long period before the same sequence repeats again. This is primarily done by

ensuring c and m are relatively prime, and that some prime divisor of m also divides evenly into $a - 1$. In Java's specific case, m is defined to be 2^{48}, c is 11, and a is $5DEECE66D_{16}$. As the modular term m implies, the output can occupy at most 48 bits, so to get a 32-bit output, we take the upper 32 bits (bits 47 through 16, going left to right) and return them as a number. The issue, which is commonplace with many PRNGs in frequent use, is if we know the state in which the generator began, we can predict perfectly which values will be printed in following iterations. When we consider that true randomness involves some form of unpredictability of output, this hardly seems random when put under this microscope. With specified seed values, the initial state of the generator, and consequently each subsequent state, is left completely up to human biases which, while unpredictable, may not be entirely random. While the complexity has certainly eclipsed that of von Neumann's predictable algorithm, it is one of the less mathematically rigorous accepted PRNGs in computer science.

A more rigorous alternative to the LCG leads into the discussion of Python's analogous PRNG called the Mersenne Twister. The complexity is higher than both von Neumann's algorithm and the primitive LCG. The general idea involves a recurrence relation of 624 32-bit integers being generated over a sequence of bitwise operators, such as XOR, AND, and shift left/right. From the starting seed, we set the generator's initial state, then the 624 values are generated by performing an XOR with the last 2 bits of the current 32-bit integer with the first 2 bits of the same number. Next, multiply that result by some 32-bit coefficient ($6C078965_{16}$), add to this the current integer index (1, 2, … 624), and use the final answer as the input for the next number. Next, perform a right shift of 11 bits on the current results, shift it left 7 bits then AND it with $9D2C5680_{16}$, perform an XOR with this number and the same value shifted left another

15 bits and ANDed with $EFC60000_{16}$, and finally perform an XOR with this value and the same value shifted right 18 bits. Surely, the average user would not take a second look at these formulas and take the "randomness" at face value, but anything algorithmically derived like this cannot be sufficiently random for all purposes. If a seed value results in the same sequence of numbers each time, it is hardly unpredictable and can be reverse-engineered if we know the generator's initial state. In fact, the main takeaway from this last point is that the Mersenne twister, in all its mathematical rigor with demonstrated randomness properties, one could attack what was intended to be a secure cryptographic key every time we attempt to make a connection to TCP. A similar cryptographic argument applies for the Java LCG, and it tracks how it would be even worse for encoding randomly generated ciphers due to the greater simplicity of its underlying algorithms. For these purposes, most widely-used PRNGs generally cannot produce cryptographically secure values and require the use of other utilities to establish sufficient security of connections.

SOURCES: (https://docs.oracle.com/javase/8/docs/api/java/util/Random.html , https://research.nccgroup.com/wp-content/uploads/2020/07/randomness.pdf , https://www.win.tue.nl/~marko/2WB05/lecture5.pdf , http://www.math.sci.hiroshima-u.ac.jp/m-mat/MT/ARTICLES/mt.pdf , http://www.columbia.edu/~ks20/4106-18-Fall/Simulation-LCG.pdf , https://docs.python.org/3/library/random.html , http://voodooguru23.blogspot.com/2019/01/the-mersenne-twister.html)

Real-World Examples of PRNG Issues

Before discussing the theory governing cryptographically secure random number generation and how they represent an improvement over PRNGs, bad random number generation

through pseudo-random algorithms has caused catastrophic issues in industry. In one example, the encrypted web chatting service Cryptocat unintentionally made their random bit generation algorithm easier to crack by leaving an off-by-one error in their implementation. The algorithm chooses some random byte (8 bits), which has 256 possible outcomes (2^8), to turn into a digit from 0 to 9. Since there are 10 possible digits, Cryptocat thought they should throw away the last 6 outcomes and only have 250 applicable bytes. However, their code allows any value from 0 to 250, *inclusive*, to be added, which involves an extra value which will return '0' by the encryption. Hence, it became easier to hijack a TCP connection by guessing the random digit was a '0.' Another example revolves around the random number generator for security keys in Debian OpenSSL originally contained what the compiler considered redundant code. However, not only did numerous keys become suddenly very vulnerable, but there is no evidence there has been a permanent fix from the developers such that these keys are no longer vulnerable or at least out of use. Some of these insecure security keys may still be being used, meaning any services which still use these keys are still vulnerable, even after 13 years. Finally, early Netscape encrypted messages suffered a similar fate. Because of regulations against strong cryptography overseas, they only possessed 40-bit precision for their secret keys. The seeds used for these secret keys depended on the current time of day (in seconds and microseconds), the process ID (pid), and the parent process ID (ppid). Any computer on a Netscape browser could use the ps command to determine which processes and parent processes were running at a given time, so two variables are known by any potential attacker. There exist tools such as tcpdump which can estimate the time at which a process was completed, usually down to second accuracy. This leaves only microseconds, and these million outcomes can be iterated through via brute force to find the secret key in less than 30 seconds. Netscape soon responded with more sources

of entropy to determine their randomly-generated secret keys, but it shows just how vulnerable a reputable browser becomes to outside attackers with only haphazard pseudo-random number generation. Thus, the cryptographic and cybersecurity communities require PRNGs which either cannot reveal sensitive information in any way or presents such computational complexity that it is impossible for hackers to gain information.

SOURCES: (https://www.cs.tufts.edu/comp/116/archive/fall2013/ali.pdf , https://people.eecs.berkeley.edu/~daw/papers/ddj-netscape.html , https://www.researchgate.net/publication/255569058_Predictable_PRNG_In_The_Vulnerable_Debian_OpenSSL_Package)

Cryptographically Secure PRNGs: Worth Universal Implementation?

Cryptographically secure PRNGs are the only algorithm-based RNGs which can reliably generate secret keys, nonces (numbers used only once in a cryptographic correspondence), and salts (additional one-way inputs used to secure passwords) which cannot be reverse-engineered to front an attack on a user. Two specific tests differentiate it from a normal PRNG. First, if there exists no polynomial-efficiency algorithm which can reliably predict the next bit of some random sequence of bits with marginally greater than 50% accuracy, we say the algorithm which produced the string passes the next-bit test. Upon passing the next-bit test, it has been proven that the algorithm satisfies all probabilistic tests for randomness necessary to conclude the algorithm is effective. Second, if an attacker were to ascertain the current state and output value of the RNG, we say the algorithm withstands state compromise extensions if it is impossible to determine previously generated values and with enough entropy dictating the generator, it should be impossible to predict future outcomes also. The LCG and Mersenne twister algorithms fail

both of these tests spectacularly, as do many insecure PRNGs. Some CSPRNGs are built based on block ciphers, which encrypt entire bit strings at once with one key. A block cipher is said to be in counter mode if the values which are encrypted are successive values of some function (in this case, we choose a simple increment function, adding 1 each successive round through the encryption). By choosing a single key to encrypt each number, this confirms the period is 2^n, which for a standard 128-bit cipher, represents a sufficiently long period to feel comfortable about that aspect of randomness. As long as the key and original bit string are kept secret, this method is reasonably secure. Another type of cipher, a stream cipher, in counter mode can function as a CSPRNG if a known pseudorandom bit string is returned from a different CSPRNG. This original string must be kept secret, as it is XORed with the plaintext to be encrypted, and encrypting the counter as described above with the block cipher results in another pseudorandom string usually with a longer period, and thus more sufficient in most cases.

Not all CSPRNG involve cryptography, however. Some mathematical algorithms involve such exorbitant time complexity that a breach is functionally impossible. The Blum Blum Shub algorithm mirrors a LCG; however, the main difference is that the next number is given by $x_n = x_{n-1}^2 \bmod M$. This time, M represents the product of two large primes, p and q, and the seed x_0 must be relatively prime with M (meaning p and q cannot be factors of x_0). Further, the algorithm works optimally when p and q are congruent with 3 mod 4 (that is, the remainder of $p/4$ and $q/4$ should both be 3) and the greatest common divisor of $(p - 3)/4$ and $(q - 3)/4$ are appreciably small. It works chiefly because if the plaintext is randomly selected, knowledge of the key and the factors of M are required to guess even one bit of the encrypted message due to the unpredictability of the number generation stream, given the particular

properties of p and q. In essence, then, the computational complexity of finding factors which return a given number x_n is too great. Even though this technique is proven to be completely secure from outside attacks, its greatest asset is its greatest drawback in practice. The computational complexity of finding random numbers in this manner is so high that it should only be considered when extreme encryption is required. The Blum-Micali algorithm results in a similar conclusion using the discrete logarithm problem (essentially, the absence of an algorithm to determine the k such that $\log_b a = k$, where a and b are any real number) to justify computational complexity sufficient enough to ensure no attacker can find the key or plaintext bit string. Similarly, too, this algorithm should only be used in cases of extreme security need.

SOURCES:

(https://archive.org/details/Introduction_to_Modern_Cryptography/page/69/mode/2up, https://www.schneier.com/wp-content/uploads/2017/10/paper-prngs.pdf, https://www.uobabylon.edu.iq/eprints/paper_1_17913_649.pdf , https://shub.ccny.cuny.edu/articles/1986-A_simple_unpredictable_pseudo-random_number_generator.pdf , https://www.cs.ru.ac.za/research/g02c2954/Final%20Writeup.htm#_Toc119059685)

Tradeoff for Quasi-Random Sequences

The chief issue of CSPRNGs is the complexity required to ensure cryptographic security from the base PRNG. Further, as we have discussed about PRNGs, the uniformity of their outputs are sometimes put into question, namely in the Cryptocat example, in von Neumann's middle-squares method, and in the LCG algorithm if poor choices are chosen for a and/or m. However, a substitute for randomly chosen numbers are low-discrepancy sequences, also known

as quasi-random sequences. Both create uniform distributions, albeit in distinct ways. With any PRNG, CSPRNG, or "random" algorithm, there is an equal chance of the next number landing in an interval of some size as a distinct interval of the same size. With a low-discrepancy sequence, the universal distance between all the numbers (i.e., the discrepancy) is minimized such that it seems as though the next value is dependent on the previous values. Examples include the Sobol, Halton, Neiderreiter, and Faure sequences. Unfortunately, none of these are not exactly random since, like PRNGs, they are not wholly unpredictable. When considering the previous few numbers, one can reasonably pinpoint the general interval where the next value will fall, so if an attacker found previous bit strings, future numbers can be predicted just as easily, if not more easily than with PRNGs. The incentive of this technique comes in grid optimization or Monte-Carlo simulation contexts. When the dimensionality of the grid is unknown, quasi-random sequences allow for halting at convergence points, where further optimization of the current dimensions is ineffective. By prioritizing uniformity over unpredictability, and in effect possessing only one of the two desirable quantities of truly randomness numbers, quasi-random sequences provide faster convergence than the same methods using pseudorandom numbers. Further, the quasi-Monte Carlo method for numerical integration relies on stochastic calculus techniques which are commonly used in quantitative finance to predict volatile markets. They present an advantage over the normal Monte Carlo method which relies on pseudorandom numbers in moderate dimensionality, which makes it a good benchmark for numerical integration as used in risk assessment. These applications do not provide an answer to the central question of how random RNG can become because by definition, these quasi-random numbers are deterministic values masquerading as random values, even more so than pseudo-random sequences.

SOURCES:

(https://www.nag.com/industryarticles/introduction_to_quasi_random_numbers.pdf, http://dsec.pku.edu.cn/~tieli/notes/numer_anal/MCQMC_Caflisch.pdf , https://www.math.ucla.edu/~caflisch/Pubs/Pubs1990-1994/Morokoffsisc1994.pdf , https://mathworld.wolfram.com/QuasirandomSequence.html)

The Randomness of the Physical World

From one extreme of our definition of randomness (prioritizing uniformity relative to the previous values) to another (prioritizing unpredictability), the discussion pivots towards an entirely distinct class of random number generators. These physical RNGs do not leverage a mathematical algorithm, but instead read and sense some physical phenomenon and use the seemingly infinite states these objects have under the laws of physics to return a random number based on this input. For example, the HotBits website hosted by Fourmilab in Switzerland processes user requests for random numbers by returning the time between one particle's radioactive decay time with that of some other particle in the sample, as measured by a Geiger-Muller tube hooked up to a high-powered computer. With a very long (on a nuclear scale, that is) half-life of 30.17 years, Cesium-137, the isotope HotBits uses in its analysis, typically waits long enough between individual particle decays so that the computer can perceive the difference in time of the decays. Relying on the innate chaos and entropy of quantum mechanics and radioactivity is about as random as one can ascertain. Further, the Java package randomX provides access to a repository of HotBits radioactive decay data which is accessible on local machines, so the accessibility issue of HotBits only taking requests for internal evaluation is partially alleviated as well.

Atmospheric noise can also function as an input to a physical RNG, used by the common RNG website RANDOM.ORG. Atmospheric noise seems like an oxymoron, as in most locations around the world, the atmosphere makes no sound. However, the radio signals picked up by RANDOM.ORG and others leverage the 40 lightning strikes which occur globally every second. The randomness is not as inherent as radioactive decay and quantum chemistry, as we observed with HotBits. However, exact weather phenomena are as unpredictable as the natural world can accomplish, and this includes the bursts of electrical energy resulting from charged air currents we refer to as lightning. There are multiple receivers across the globe which each read a 8-bit signal at a frequency of 8KHz. The first seven bits are discarded, and skew correction is performed to ensure a relatively equal distribution of 0's and 1's. The skew correction depends on the transitions between bit pairs. If there are two distinct bits, keep only one bit in the pair and discard the other; and if there are two of the same bit, read neither and discard the entire pair. Of course, this results in a loss of 75% of the input data on average, but this process leaves the bit stream modified in such a way that they now possess a high degree of entropy. The National Institute of Standards and Technology (NIST) has a comprehensive list of 15 tests which, if they are all passed, statistical randomness is satisfied, and the algorithm is sufficiently close to true randomness to produce some of the best possible results. These are many of the tests which are bypassed by an algorithm satisfying the next-bit test from the discussion on CSPRNGs. Hence, it can be argued that if the next-bit test was proven to be true on a dataset of random numbers from RANDOM.ORG, the same conclusion would be drawn. Intuitively, we see this is true because the next bit pattern corresponding to the atmospheric sound wave is unknown to any algorithm or human knowledge, so we have credence towards the statistical validity of the RANDOM.ORG physical random number generator.

As diverse as the sources of entropy are in these algorithms, we cannot confirm true randomness. The administrator at RANDOM.ORG says it himself: "Oddly enough, it is theoretically impossible to prove that a random number generator is really random. Rather, you [analyze] an increasing amount of numbers produced by a given generator, and depending on the results, your confidence in the generator increases (or decreases, as the case may be)." In essence, then, a physical random number generator can be based on processes which are defined to be truly random, but not itself truly random due to variation in pre-processing the bit string to extract entropy from the physical signals. These methods are often unintuitive and RANDOM.ORG presents theirs as almost magic, like it rearranges the remaining bits in such a way it is perfectly random. However, there exists a logic to them which requires clarification.

SOURCES: (https://www.fourmilab.ch/hotbits/ , https://www.fourmilab.ch/hotbits/how3.html , https://www.random.org/randomness/, https://www.random.org/faq/#Q2.1, https://www.random.org/analysis/, https://www.random.org/analysis/Analysis2005.pdf, https://www.nature.com/articles/s41598-020-74351-y)

Randomness Extractors

SOURCES: (https://www.cs.utexas.edu/users/diz/pubs/erf.pdf , https://par.nsf.gov/servlets/purl/10136214 , https://thenewstack.io/random-number-generation-breakthrough-boost-data-encryption/ , https://people.seas.harvard.edu/~salil/pseudorandomness/extractors.pdf)

Quantum Randomness: The Final Solution?

SOURCES: (https://www.quantamagazine.org/how-to-turn-a-quantum-computer-into-the-ultimate-randomness-generator-20190619/, https://www.nature.com/articles/s41598-021-95388-7 , https://daily.jstor.org/the-quantum-random-number-generator/)

Is True Randomness Possible?

SOURCES: (https://towardsdatascience.com/when-science-and-philosophy-meet-randomness-determinism-and-chaos-abdb825c3114, https://medium.com/intuitionmachine/there-is-no-randomness-only-chaos-and-complexity-c92f6dccd7ab , https://www.random.org/faq/#Q2.1)