[[1]](#footnote-1)

Pseudorandom Numbers Generated via Simple Particle Simulation Evaluated for Use in Cryptographic Applications

April 10, 2018

**Sean Donachiue 24307134**

Department of Computer Science  
University of British Columbia  
Vancouver, Canada  
  
sgdonachiue@gmail.com

*Abstract –* An evaluation of cryptographic security in the use of a particle system modelling only position and velocity in 2-space and initialized with a low source of entropy to generate pseudorandom outputs without reliance on cryptographic primitives. The outputs generated were evaluated using common numerical tests for randomness. The system was conceptually evaluated against the possibility of input attacks and state compromise extension attacks in which the attacker has significant knowledge of the system. Numerical results obtained are not sufficient to claim cryptographic security in the system, however, the design appears robust to the attack types analyzed.

# INTRODUCTION

Pseudorandom number generation (PRNG) is any algorithmic process by which a random seed value obtained from an entropic source is used to generate a sequence of bits deterministically. This is in contrast with True random number generation, where the source of entropy, almost always a physical process, is used to generate each bit. However, such physical processes are often inaccessible to cryptography software due to the hardware it is carried out upon.

Random numbers are used in cryptography for several applications, including the generation of keys, salts, and nonces[1], among other things. Any bias in the set of numbers generated could expose a system in which it is used to attack. In the most trivial case, an adversary may observe bias or patterns in outputs from a system which are visible, such as nonces, and use this information to guess outputs which are not visible, such as session keys, with some probability. More sophisticated adversary models are discussed in section III.

As a result of all this, there have been significant efforts by academics as well as by groups such as the NIST and IETF to define standards and methods for cryptographically secure pseudo-random number generation (CSPRNG), as well as tests and evaluation methods for the randomness of a sequence. The system design and implementation detailed in sections IV and V relies most heavily on the concepts and recommendations made in [1], while the sources of entropy used are discussed in [2], and some of the tests for randomness utilized are detailed in [3] and were implemented in [4]. Using this body of work as primary points of reference, I designed and implemented a PRNG which allows entropy to pool in the form of a state vector *S* for a particle simulation in two-space, bounded by the range of a byte, which samples entropy gathered from the system microphone to initialize positions and velocities. This was combined with the inconsistent execution time of the CPU to update and initialize particles, further increasing the entropy available to the system. After some acceptably short amount of time has passed, the output of up to 32 bytes in length is formed by variably sampling the positions of particles in the system, keeping their velocities entirely obscured from an adversary.

The statistical randomness of byte sequences produced by the implementation was evaluated using the ENT test suite from [4], which evaluates sequences of bits for Shannon entropy, optimum compression percentage, chi-squared distribution test, arithmetic mean value, Monte Carlo Pi-value, and serial coefficient correlation. Further, the design was evaluated against the adversary models described in [1] and discussed in section III.

Note that a formal proof of the security of this design, as in [5], is well beyond the scope of this paper, and though the original goal was to achieve cryptographic security, this proved relatively far off, the test results show.

While the statistical performance of the system is not sufficient to claim cryptographic security, there are numerous ways it could be improved, which I discuss in section VII. The design itself is not easily susceptible to input attack or state compromise extension, even against an adversary with extensive knowledge of the system. It would fail easily to an adversary with perfect knowledge, but anything less is insufficient.

The performance obtained despite the lack of reliance on cryptographic primitives is encouraging, and if the uniform distribution of outputs was improved, it would be worth exploring a more rigorous proof of security in order to answer the open question posed in [1] about whether or not CSPRNGs are themselves a cryptographic primitive.

# Related Work

The need for number generation procedures closely approximating a uniform random distribution has given rise to a considerable number of approaches to the problem, but there are commonalities across all approaches. For instance, a universal model of a PRNG is a system which produces pseudorandom outputs from unpredictable inputs (from [1]) These unpredictable inputs are referred to as sources of entropy, which are described extensively by [2] and [5]. Once one has gathered sufficient entropy, it can be used as the seed to produce the required amount of cryptographically strong pseudo-randomness. Sources of entropy include (but are not limited to):

1. Sound and video input
2. Disk drive methods
3. Ring oscillator methods
4. Clocks
5. Device methods (serial numbers)
6. Execution timing
7. External events (e.g. key input, packet arrival times)

Some examples of CSPRNGs employing combinations of these entropy sources (listed in brackets) are the NIST approved HMAC-DRBG (which takes any buffer as entropy input from the calling function)[6] the Linux distribution of random.c (2, 4, 5, 7)[7], and the Windows CryptGenRandom function(2, 4, 5, 7)[8]. Though the inputs provided by these methods are unpredictable, they need not vary greatlyto produce outputs indistinguishable from random ones if the entropic bits are mixed or de-skewed appropriately as described in [2]. For comparison, the implementation I discussed in sections V uses 1 and 6. I also considered using traceroute timings, an example of 7, but decided these could be too consistent to be worthwhile, which the IETF warns about in [2] when discussing using packet arrival times as entropy.

A more detailed model for a strong PRNG is found in fig. 1 from [1], describing the entropy pool and the countermeasure of periodically reseeding it with additional entropy each time the generator is queried. The implementation I produced fits this model, except for the generate step, which typically involves providing the seed value as input to a secure hash function, as in HMAC-DBRG and random.c. My implementation eschews this for reasons detailed in section IV.

Numerous statistical methods for evaluating randomness have been developed, these are covered perhaps most exhaustively by the NIST in [3], as they are the primary subject of that document. The prototype analyzed here used those implemented in [4], of which three are also present in [3]; the monobit frequency test, entropy, and the chi-squared test. Additionally, it was proved in [9] that if a sequence satisfies the next-bit test (“given the first *k* bits of a random sequence, there is no polynomial-time algorithm that can predict the *(k+1)th* bit with probability of success non-negligibly better than 50%”) it will satisfy all other statistical tests for randomness. Of the tests utilized in this evaluation, the next bit test is most closely approximated by the serial correlation coefficient test, which describes the degree of correlation between bits or bytes in a sequence. The tests used are described in more detail in section VI-A.

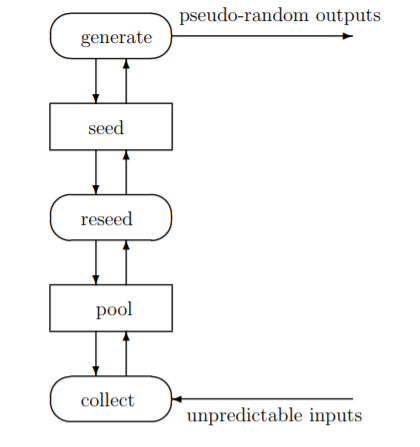


Fig 1. A model developed in [1] for describing and designing PRNG’s capable of resisting state extension cryptanalytic attacks

# Adversary Model

Following the set of attacks mounted against PRNGs in [1], I evaluate the design against the possibility of direct cryptanalytic attack, input attacks, and state compromise extension attacks (backtracking, permanent compromise, iterative guessing, meet-in-the-middle).

The design and prototype was primarily intended to defend against direct cryptanalytic attack by achieving as much statistical randomness as possible with the limited entropy sources available, maximizing defense against a “naïve” attack model which seeks to learn outputs which have a high probability of being generated such that attempts to guess would have significantly better than *1/n* chances of occurring, where *n* is the number of values which are possible to generate. In the implementation tested here, the ideal value of *n* is 2256, as the prototype implementation generates outputs up to 32 bytes long.

Input attacks are those in which an adversary is able to control or influence the inputs to the PRNG, allowing them to determine internal operations from the resulting outputs. These may be broken down further, but for reasons detailed in section VI, the system implemented is not easily susceptible to these, so I will refrain from doing so.

State compromise extension attacks are a class of attack which succeed if an adversary knows the output of the PRNG with state *St*, and they are able to guess the outputs at *St–1*(backtracking attack)or *St+1* (iterative guessing attack), and by extension, predict any output from a single compromised state. There are also meet-in-the-middle-attacks which combine backtracking and iterative guessing, as well as permanent state compromise attacks, which are those in which *St* is compromised for any value of *t*.

# System Design

The internal state of the system consists of n particles with position and velocity vectors in two-space bounded by the range of a byte*.* For *np* particles, there are up to *2n* bytes of entropy available, since the position values serve as the final outputs of the system. In addition to *np*, the system takes a value *tpool* as input, which is the amount of time to allow the simulation to update by integrating velocities into positions after all particles have been initialized. It is important to choose a value of *t­pool*, which is large enough that any given particle has the potential to have travelled anywhere in the simulation space. In other words, *tpool* is the amount of time required for entropy to build in the system before a pseudorandom output can be generated.

The design itself is composed of four functions, shown in the order in which they are initially called: init, update, pullBytePairs, and initOne:

**init**(n, sound\_buffer[]) {

t = clock.now()

for(i = 0; i < n; i++) {

Xpos 🡨 sound\_buffer[i\*Δt]

Ypos 🡨 sound\_buffer[i\*2\*Δt]

Xvel 🡨 sound\_buffer[i\*3\* Δt]

Yvel 🡨 sound\_buffer[i\*4\* Δt]

now = clock.now()

Δt = now – t

update(Δt)

Particle p 🡨 (Xpos, Ypos, Xvel, Yvel)

Particles[i] 🡨 p

}

}

**update**(Δt) {

for(Particle p in Particles) {

p.Xpos += (p.Xvel \* Δt) + 1

p.Ypos += (p.Yvel \* Δt) + 1

if (p.Xpos > 127)

p.Xpos -= 255

else if (p.Xpos < -128)

p.Xpos += 255

if (p.Ypos > 127)

p.Ypos -= 255

else if (p.Ypos < -128)

p.Ypos += 255

}

}

**pullBytePairs**(n, Δt, output[]) {

for ( j = 0; j < n; j+=2 ) {

Particle p 🡨 S[(Δt + j) % n]

output[j] 🡨 p.position.x

output[j+1] 🡨 p.postion.y

//The next line deletes the data in //Particle p from the state vector

delete S[(Δt + j) % n]

initOne(Δt, sound\_buffer, p.velocity)

update(Δt)

}

}

//c1…4 are arbitrary constants

**initOne**(Δt, sound\_buffer[], vel{x,y}) {

Xpos 🡨 sound\_buffer[vel.x]

Ypos 🡨 sound\_buffer[vel.y]

Xvel 🡨 sound\_buffer[c1\*Δt] + (c3 % Δt)

Yvel 🡨 sound\_buffer[c2\*Δt] + (c4 % Δt)

update(Δt)

}

These four functions roughly map to the components of the conceptual design shown in fig. 1, though not perfectly. init is involved both with the collection of entropy and the initialization of the entropy pool. This is then heavily modified by update, which maps to the reseed step. A seed is generated from the system by pullBytePairs, and this seed is used as a pseudorandom output value. Finally, initOne accounts for the reverse input to the entropy pool which we can see represented by the downward arrows in fig. 1. This is achieved by initializing new particles by indexing into the sound buffer using the velocity (an internal value) of the last particle added to the output.

Notably absent is the generate step which typically involves the application of a secure hash function or one way compression function to mix the bits of the seed value to obtain further “randomness” in the final generated value. I chose not to use one of these methods because of an open question posed in [1], which is incidentally also the source of the conceptual model I have approximately mapped my functions. The question posed was whether or not CSPRNGs can be considered as their own kind of cryptographic primitive. The authors put forward the idea that to be a true cryptographic primitive, the CSPRNG in question must not use other cryptographic primitives to achieve cryptographic security. This question served as both a major inspiration for the design as well as a convenient excuse not to implement such methods, given the time and manpower constraints of the project.

The system hardware microphone was chosen as the primary entropy source because it seems to me that gathering one second of audio with a high sample rate would be far more acceptable to a hypothetical user of any system than gathering even a fraction of a second of video. Further, ring oscillator and disk drive methods are beyond the scope of my ability to implement. Device values such as serial numbers are not particularly random, and internal operating system values such as process ID’s were not easily accessible, not to mention using these methods reduces the portability of design considerably. External events are often not particularly random, as they have a certain consistency if the user of the system follows any kind of schedule with regards to the usage of their machine in daily life. By process of elimination, only clock values and the execution timings provided by these remained. Fortunately, these methods are also trivial to implement. In [2], members of the IETF warn against using clock values, as some machines may have low resolution clocks. In order to simulate a low resolution clock, and therefore improve the portability of the design, all clock values used in the implementation are in milliseconds.

The following principles of design for secure systems are employed in the design:

Open Design – Knowledge of the inner workings of the system does not compromise the outputs unless an adversary also has extensive knowledge of the data within the sound buffer as well as the execution environment. Without access to these systems on the machine running the PRNG, knowledge of the design is not a security risk, so it should be made freely available.

Psychological Acceptability – The user inputs a number of pseudorandom bytes to generate. The PRNG generates that many pseudorandom bytes. Intuitive and acceptable.

Isolation – Outputs are isolated from the values they are generated from, by the definition of a PRNG.

Encapsulation – The PRNG functions are encapsulated in a class called cryptorand.c in the implementation.

Modularity - The PRNG could feasibly be used “as is” on any windows system, or at least this was a goal in the design. Additionally, the PRNG could be augmented with other sources of entropy, since the entropy collection is separate from cryptorand.c

Least Astonishment – Similar to psychological acceptability, the design behaves as expected for a random number generator.

# System Prototype

The design prototype was implemented in C++ 14 using Microsoft Visual Studio 2017. As is often the case with prototypes, the design did not transfer cleanly into implementation. Below, I describe departures from the design, as well as details of the implementation, including any fixed inputs.

For simplicity, the general output size of the design was eschewed in favour of a 256-bit output, corresponding to an entropy pool of 16 particles. As per the recommendations of the IETF, even 128 pseudorandom bits are enough for most cryptographic applications today, though this will change as hardware continues to improve. Since the evaluation methods operate on data streams rather than blocks, different output sizes produce identical statistical results.

The state vector was implemented as a from-scratch (a bad practice) linked list of Particle structures to facilitate easy deletion at arbitrary indices in pullBytePairs.

The waveform audio functions of [10] were used to implement a modular component separate from the PRNG which accesses the default system microphone, called “Conexant SmartAudio HD”. A 2-channel, 24-bit, 48000 hz sample rate was used, as this was the highest possible. Audio was sampled at this rate for one second. Finally, *tpool* was fixed at 700 milliseconds, as this is more than enough time for a particle to travel the entire simulation space.

Due to the fact that all possible byte values were included as potential output, C++ char\* formatting cuts outputs off early if they contain a NUL character. Outputs would also be split if they contained a newline character. These points do not affect the validity of the statistical evaluation. However, in practice this would be unacceptable, as a user will occasionally receive an output which is shorter than the one they requested.

# System Evaluation

## Evaluation Methodology

Five tests were used to evaluate the degree of randomness present in 728434 bytes generated in relative silence by the PRNG in sequences of up to 32 bytes each (Due to the NUL character issue). The tests are described below:

Shannon Entropy[11] – The measure of information density in the stream, given by (1),

 (1)

where *xi* is a sample from a given distribution *X*, and *n* is the number of values *xi* can take on. The result of (1) is the

Shannon entropy measured in bits. This is the average amount of information obtained from any sample of the distribution. This is inversely proportional to the optimum compression percentage of that stream For a uniform random distribution, the Shannon entropy is 1/*n*.

Chi-Squared Distribution Test­ [12] – A statistical hypothesis test where the null hypothesis is that the distribution being tested is truly random. The test evaluates how likely it is that the distribution being tested is indeed a uniform random distribution. This test is the most commonly used test for the randomness of a distribution because it is extremely sensitive to errors in PRNGs. P-values between 10% and 90% are considered likely to be random.

Arithmetic Mean Value – The mean value of a uniform random distribution is the middle value in the distribution. This is not a very useful test of randomness on it’s own, as a distribution containing nothing but the middle value would hit the mark perfectly while being obviously non-random, however, the information it tells us about the skew of the distribution is still useful for fine tuning a PRNG. For a sequence of random bytes, the mean value is 127.5.

Monte Carlo Estimation of Pi – The implementation of the method used and described in [4] (which cites [13]) interprets successive 6-byte sequences as 24-bit (x.y) coordinates in a square. A circle is inscribed in this square and the number of points falling within the circle are counted and divided by the total number of points, which approximates the ratio of the area of the circle to the area of the square, in turn allowing us to calculate an accurate approximation of pi given the randomness of the input. Therefore, when using output from a PRNG as input to the method, we can evaluate the randomness of that output by how closely it approximates pi.

Serial Correlation Coefficient – A measurement of dependence between adjacent values in a sequence which can vary from -1 to 1, with random sequences approaching a coefficient of 0. Knuth describes Equations (2) and (3) in [13] for determining a “good” value for the coefficient *C* when testing for randomness in *n* quantities.

 (2)

 (3)

For a random input to the Serial Correlation Coefficient Equation described by Knuth (but omitted here) [[2]](#footnote-2) the value of *C* should fall in the range of Equation (3) 95% of the time.

Refer to section III for descriptions of adversary models the design is evaluated against in part B.

## Results of the evaluation

The set of tests described in part A were applied to a 728,434-byte file produced by the PRNG which was initialized once, updated itself for 700ms, then the output file was generated by calling pullBytePairs in a loop of 30,000 iterations. The results can be found in table 1.

Direct Cryptanalytic Attack: Given the performance of the PRNG on the numerical tests for randomness, it is plausible that a direct cryptanalytic attack on outputs produced with a single initialization of the PRNG could succeed. That is, it is plausible that an attacker could distinguish between outputs from a single initialization run and a set of truly random values using similar tests as those employed here. Despite this, knowing the values are not truly random does not imply that the attacker could reliably guess outputs.

|  |  |
| --- | --- |
| Shannon Entropy | 7.467205 bites per byte |
| Optimum Compression | 6% reduction in file size |
| Chi Squared Distribution | 0.01% |
| Arithmetic Mean Value | 117.2595 |
| Monte Carlo Estimation of Pi | 3.050945183 (error 2.89%) |
| Serial Correlation Coefficient | 0.008292 |

Input Attack: An attacker could easily determine outputs from the system if they had full knowledge of the design in addition to knowledge of data in the sound buffer and the value of each timestep, though this seems extremely improbable. Suppose instead that the adversary knows only the system design and the value of each timestep, but not the data in the sound buffer, the variable selection of particles to the output (i.e. A particle can exist for anywhere from 1 to the end of time) prevents the attacker from ascertaining values in the sound buffer from outputs they see, since any byte pair in the output could have been moving through the space for any amount of time since the program was initialized.

State Compromise Extension Attack: Suppose an adversary has read this paper, and therefore has full knowledge of the

operations carried out by the PRNG, as well as the data in the sound buffer, but timestep values remain hidden. Assume (rather generously[[3]](#footnote-3)) that *Δt* is a sample from a uniform random distribution with a range of 1ms to 26ms. Since the sound buffer is filled only once per initialization, particle velocities may only take on as many values as *Δt*. For any state *S­t* given by the positions of 16 particles in the entropy pool at time *t,* then the next state *St+Δt* can take on 800 possible values. Computing these 800 state branches takes a paltry 51,200 operations. However, each branch has another 800 branches after the next timestep. Thus, to perform an informed brute force search from one output to the next, the adversary has 80016 branches to compute, which can be performed in ~1.44x1051 operations. For a 3Ghz CPU running a single thread, this would take ~1.52x1034 years. For reference, the birth of the universe is closer to the present than our adversary would be to brute forcing from one output to the next under these conditions.

## Discussion of the evaluation results

Though the PRNG output does not pass any of the tests for randomness, in every case except the chi squared distribution it performs relatively well. We can see from the arithmetic mean value that the outputs have a slight left skew, which I suspect is a result of the values indexed from the sound buffer biasing the output. The error obtained in estimating the value of pi with the Monte Carlo method is also fairly low, but indicative of a non-random distribution/ The range for a serial correlation coefficient signifying no correlation in the data was calculated as -0.002342 < *C* < 0.002345 using Equations (2) and (3). Though the file evaluated does not pass this test, the coefficient obtained is in the same order of magnitude as a “good” value. For comparison, the C++ file which implements the methods described in section IV has a coefficient of 0.419610. Overall, these statistical results are promising, and I believe that with appropriate tuning, results indicative of a random distribution could be obtained with this prototype.

Surprisingly, the design is not especially vulnerable to state compromise extension attacks, despite the fact that it neither utilizes catastrophic reseeding nor a one-way compression function to gather fresh entropy or alter the entire state vector periodically. Even if the velocities are known in addition to the positions for a given state, the variable timestep and selection indices for choosing which particle to remove, as well as removing only one particle per timestep result in robust state transitions.

# Discussion

Undoubtedly, the PRNG designed and implemented here has numerous flaws[[4]](#footnote-4), such as the lack of uniformity in the distribution of outputs, the reliance on the Windows API, and perhaps most importantly, the heavy dependence on inconsistency of execution time for distribution of byte values in the outputs as well as the order they are selected in. It is possible that the system would fail to start on a different Windows machine than the one it was implemented on, simply because the microphone can’t be accessed, or does not allow for recording with the sample rate I used. Finally, output strings being terminated by the NUL character before they are the required length is a serious bug which would need to be fixed (likely by excluding the character) before the system could be deployed.

Despite that, the system has numerous advantages as well. Taking clock values in milliseconds instead of the highest resolution available means the implementation is not restricted by clock hardware, and the entropy utilized from the clock is comparable to some of the systems analyzed in [1]. The system was designed without the use of any cryptographic primitives to explore an open question posed in [1], namely, are CSPRNGs a cryptographic primitive unto themselves? Due to lackluster performance on the numerical tests of randomness, the system can’t be called cryptographically secure, so the question remains unanswered. However, the lack of reliance on a hash function or cipher meant that the system holds up well against hypothetical state compromise extension attacks, since there is no single point of failure, and no key to keep secret. For comparison, all three of the NIST approved PRNGs depend upon cryptographic primitives for the security of their outputs.

Beyond the lack of cryptographic primitives, the design proposed here is fundamentally different from that of the PRNGs analyzed in [1], as well as HMAC-DBRG (analyzed in [6]), random.c, described in [7], and the Windows CryptGenRandom function, described in [8]. The difference lies in the fact that these generators are specified to run their update loops in the background and be queried for output by the user at an arbitrary time, then outputs are fed back into the generator in some manner to generate the next output in one step. This allows for potential vulnerability to state compromise extension attacks because the state space is not sufficiently large enough to be computationally infeasible to search.

I speculate that the uniformity of outputs produced by my design could be significantly improved by implementing any one of the following, alone or in combination: Periodically replacing the data in the sound buffer, using higher resolution clock values, masking outputs by XORing each byte with low-order digits of the clock, using a different prime number modulus on each byte of output.

# Conclusion

The degree of randomness achieved by the system using a comparatively small amount of entropy to other systems is not insignificant for a first pass at a design free of dependence on cryptographic primitives. Additionally, the entropy sources used are not easy to hijack, influence, or predict, resulting in strong resistance to input attacks. Finally, the potential size of the state space between outputs is far larger than that of other generators resulting in state compromise extension attacks without perfect knowledge being computationally infeasible.

I believe the results obtained and the conceptual analysis of attacks warrant further work on improving this approach. For now, the prototype may be viewed as a proof of concept of the potential for PRNGs that do not make use of cryptographic primitives.

Acknowledgment

S.D. Author thanks Professor Bob Gill for his lenience with regards to the tardiness of this report, as well as for providing the template on which it is based. Thanks also to the teaching assistants, Junbin Zhang and Jiahui Tang for reading and evaluating this paper late, as well doing their best to decipher S.D.’s terrible handwriting on quizzes, and for stifling their laughter at this Author’s wildly wrong answers on the same.

References

1. J. Kelsey, B. Schneier, D. Wagner, C. Hall. “Cryptanalytic Attacks on Pseudorandom Number Generators” *Fast Software Encryption, Fifth International Workshop Proceedings* (March 1998), Springer-Verlag, 1998, pp. 168-188.

Available: <https://www.schneier.com/academic/paperfiles/paper-prngs.pdf>

1. D. Eastlake 3rd, J. Schiller, and S. Crocker. "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005.

Available: <https://www.rfc-editor.org/info/rfc4086>

1. Rukhin et al. “A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications.” *NIST SP 800-22 revision 1a*, (April 2010).

Available: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf>

1. J. Walker. “ENT A Pseudorandom Number Sequence Test Program” (January 2008)

Available: <http://www.fourmilab.ch/random/>

1. Turan et al. “Recommendation for the Entropy Sources Used for Random Bit Generation.” *NIST SP 800-90B*, (January 2018).

Available: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90B.pdf>

1. K.Q. Ye. “*The Notorious PRG: Formal Verification of the HMAC-DBRG Pseudorandom Number Generator.*” B.A. Thesis, Princeton University, United States, 2016.

Available: <https://www.cs.cmu.edu/~kqy/resources/thesis.pdf>

1. L. Torvalds, Random.c Open Source Contributors. “linux/drivers/char/random.c”

Available: <https://github.com/torvalds/linux/blob/e192be9d9a30555aae2ca1dc3aad37cba484cd4a/drivers/char/random.c>

1. “CryptGenRandom” Windows CryptoAPI Documentation. Microsoft.

Available: <https://msdn.microsoft.com/en-us/library/windows/desktop/aa379942(v=vs.85).aspx>

1. A. Yao. “Theory and Application of Trapdoor Functions” In *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science,* (1982)

Available: <http://www.di.ens.fr/users/phan/secuproofs/yao82.pdf>

1. “Waveform Audio Reference.” Internet: <https://msdn.microsoft.com/en-us/library/windows/desktop/dd743833(v=vs.85).aspx>, [April 1, 2018]
2. C. E. Shannon. “A Mathematical Theory of Information.” (1948. July–October). *Bell System Technical Journal.* [PDF] 27 (3), 379–423.

Available: <http://math.harvard.edu/~ctm/home/text/others/shannon/entropy/entropy.pdf> [April 11, 2018]

1. B. Y. Ryabko, V.S. Stognienko, Y. I. Shokin. “A new test for randomness and its application to some cryptographic problems” *Journal of Statistical Planning and Inference*. [PDF]. 123, 365-376

Available: <http://boris.ryabko.net/jspi.pdf> [April 11, 2018]

1. D. E. Knuth. (1968). *The Art of Computer Programming*. (2nd Edition). [PDF] Vol 2.
2. Available: <http://plouffe.fr/simon/math/The%20Art%20of%20Computer%20Programming%20vol%202.pdf> [April 11, 2018]
3. B. Schneier. “Did NSA Put a Secret Backdoor in New Encryption Standard?” Internet: <https://www.wired.com/2007/11/securitymatters-1115/>, June 21, 2014 [April 12, 2018]

1. [↑](#footnote-ref-1)
2. A formula which looks absolutely heinous to type out in Word. [↑](#footnote-ref-2)
3. *Δ*t does range between those values, but it is likely a normal distribution rather than a random one. However, even a range of 10 results in a computationally infeasible state tree.. [↑](#footnote-ref-3)
4. It may be flawed, but at least there’s no kleptographic backdoor. Unlike the NSA, I don’t care to intentionally weaken a supposedly secure system as investigated in [15]. [↑](#footnote-ref-4)