Cracking Chaos: Making, Using, and Breaking PRNGs

1nfocalypse

July 13, 2025

Abstract

Pseudo-Random Number Generators are often overlooked and core features of our computational experience. From research and processes irrelevant to security (i.e. Monte-Carlo simulations) to essential security functionality like secret generation, random number generation plays a significant part in our ability to utilize the modern internet. In turn, they have a unique history, threat model, and set of applications. We will discuss the history of pseudo-random number generation, the types of random number generators, where they are supposed to be utilized, and how to break them, when relevant. Additionally, we will discuss the future direction of random number generation in light of preparation for the advent of large-scale quantum computing.

1 Random Numbers and You: What is RNG and Why Should I Care?

Randomness is somewhat difficult to rigorously define. Within the real world, or any analytical system, we can intuitively understand the desired behavior as an independent sample over a uniform distribution. For example, consider rolling a fair die. Any number is equally likely, and as such, we call the output of the event "random". However, within the discrete computational field, we often rely on what's called a Random Oracle as a reference for computational randomness. To build intuition, a random oracle can be understood as a box with a gnome inside. The gnome has a pair of fair dice (or an equivalent source of true randomness), an infinitely long piece of paper, and a pen. When an input is provided to the gnome, it will roll the dice, write down the input and the results, and then provide the output. When it is provided with a novel input, the same process happens. However, when the same input is provided, the output generated for the same input previously will be outputted. This provides a foundational basis for evaluating the theoretical efficacy of a pseudo random number generator. Additionally, multiple statistical tests are often employed to determine if the output of a random number generator can be considered sufficiently uniformly random, i.e. TestU01, Dieharder, and NIST SP-800-22 in decreasing order of strictness.

Beyond definitions, we find that PRNGs find a plethora of use cases in finance via Monte-Carlo testing and modeling, AI/ML, numerical computation like the Randomized Numerical Linear Algebra library, gaming in terms of world generation and loot drops, and security in regards to secret generation, stream ciphers, and certain block cipher modes of operation. As such, knowing how they work, and relevant weaknesses, can guide implementation and use, while ignorance can lead to catastrophic exploitations in the case of an ignorant developer.

2 True Random Number Generation: What?

To first begin, we revisit the example of rolling a fair die. Since it's unknowable what the result will be, and it's equally likely to be any result, we can understand the outcome of this as random behavior. However, this differentiates from pseudo-random number generation, as with pseudo-random number generation, we are guaranteed that the same inputs lead to the same outputs. Rolling the same die twice does not guarantee the same outcome, since it is not deterministic. As such, rolling a die is considered a source of true randomness, or real entropy. This is extremely useful, as from a truly random bitstring, it takes 2^{l-1}, where l is the bitlength, attempts, on average, to brute force the correct string without knowing it. This derives from the fact that there is no means of determining how the number was generated, and consequently, there is no more efficient search in classical computation that will yield the correct value than brute force. As such, half of the space must be searched to meet the expected point at which the correct number will be found. This provides a cornerstone of modern cryptography - functionally, we attempt to design systems with equivalent security to lengths of these random bitstrings, i.e. “128-bit security”. In practice, true random numbers are generated from external sources - temperature, loading times, state, and other pieces of entropy. This gives us the security assurance, and is a necessary component of practicing secure cryptography.

The easiest example to point to is /dev/random and /dev/urandom, which are common sources for random numbers in Linux based on system noise. Historically, both systems would gather sufficient entropy and manipulate it with deterministic algorithms. This would preserve the sampled entropy without allowing outputs to be too “close” to one another, typically by hashing them, which at the time would also get around export laws. However, these generators have issues - /dev/urandom is non blocking, and will output low-entropy data if pushed too hard, and /dev/random was blocking (changed to non-blocking stream cipher output in 2020[Lut20]), meaning it would stop outputting completely if its entropy pool was insufficient. However, there are tasks where random numbers are demanded at a high frequency and a high demand for quality, such as Monte-Carlo methods, and additionally, it’s desirable for them to be deterministic and repeatable. As such, true random number generation is widely considered insufficient for actual application past sensitive, albeit basic, functions.

3 Insecure Pseudo-Random Number Generators

As a result of demand of computational simulations, a class of deterministic random number generators is born. Initially, these systems predate heavy use of cryptography online, and in fact, predate use of modern cryptography by most of the public. However, they proved to be both interesting and valuable to computer scientists, and continue to play that role today, from very simple Linear Congruential Generators being used for trivial cases, to counter-based PRNGs being used in intensive modeling in finance, medical research, and statistical evaluation.

The use case for insecure pseudo-random number generation is one where speed and quality is emphasized over security. For example, it is completely irrelevant if the numbers used for RNLA can be inverted. All that matters to the implementer is that sufficient randomness can be used to obtain a quality approximation. The same can be said for financial modeling - there is no need to enforce security requirements on pseudo-random events in the model, as inverting them gives an attacker nothing other than knowing when the next event will happen in the model (or how it will happen). However, some uninformed developers do unfortunately utilize insecure PRNGs in systems where being able to invert, or “crack”, the generator, yield exploits. Consider a scenario in which a password reset token is generated by a single server utilizing a PRNG. If an attacker were to invert the PRNG, and thus predict the next output, the attacker could then use said token to arbitrarily reset user passwords, which actually happened with CVE-2015-5267, considered 7.5 severity[CVEb]. Alternatively, consider a PRNG that only outputs a small range of numbers. While the results are approximately equidistributed, brute forcing the range can be quite easy, which the maintainers of OpenSSL learned very painfully in 2008 (CVE-2008-0166, 7.5)[CVEa].

To understand the benefits and shortcomings of these generators, we will go through some examples of insecure PRNGs, their use cases, and how to break them (if a method is currently known).

3.1 Linear Congruential Generators

Linear Congruential Generators are a class of PRNGs defined by the relation:

$$x_{n+1} = (ax_n + c) \bmod m$$

Originally published and known as the *Lehmer Generator* in 1951, it was further generalized in 1958[Tho58]. Specifically, the Lehmer generator defines $c = 0$ and $m \in \mathbb{P}$, where $\mathbb{P}$ is the set of prime numbers. The generalization allows for alternate choices to be made, of which the most common are:

$$m \in \{2^n | n \in \mathbb{N}\}, c = 0$$
$$m \in \{2^n | n \in \mathbb{N}\}, c \neq 0$$

Naturally, LCGs are exceptionally sensitive to parameter choice. Careful choice of parameters is paramount in creation of a LCG, as it becomes very easy to accidentally force a very small period or a sequence that has no appearance of randomness. Trivially, we find $a = 1, c = 1$ creates a counter mod q, which is decidedly not random.

In the original construction of the Lehmer Generator, a common choice is a Mersenne prime, or a prime of the form $2^n - 1, n \in \mathbb{N}$. This yields a period of length $m - 1$, in fact true for any choice of prime as m, when a has the property that $a^k - 1$ is only divisible by m when $k = m - 1$[WP69]. Additionally, the structure of Mersenne primes leads to some optimization opportunities. Regardless, the optimization opportunities still require an expensive division operation, and as such, are not often considered to be particularly efficient in this context. Additionally, in consideration of a typical buffer size, we do not have uniformity in output, as the values above the prime are ignored. This precludes the data from being considered uniformly random over the bitstring, and can be a distinguishing feature.

For a slightly modified construction, called a Mixed Congruential Generator, we take $m \in \mathbb{P}$ and $c \neq 0$. These generators are uncommon for a reason - the advantages to them are very limited, only really being considered for machines in which bitwise addition is meaningfully faster than bitwise multiplication. However, they suffer the same consequences of LCGs in being exceptionally sensitive to parameter choice, with additional constraints required. An investigation of them concludes that they are to be used judiciously, and are often inferior to purely multiplicative methods. [TH64]

For the consideration of $m \in \{2^n | n \in \mathbb{N}\}, c = 0$, we arrive at a very efficient LCG. However, LCGs of this form are unable to express a period of $m - 1$, instead being limited to $\frac{m}{4}$ at maximum when equipped with parameters $a = 3 + 8k$ or $a = 5 + 8k$, for $k \in \mathbb{N}$ and an odd seed. As such, they are not often used. [Gü12]

For our last iteration, we consider the case where $m \in \{2^n | n \in \mathbb{N}\}, c \neq 0$. These generators are capable of a period equal to m with certain constraints called the Hull-Dobell Theorem[Cha16], being

- $GCD(m, c) = 1$
- $a - 1$ is divisible by the prime factors of m
- $a - 1$ is divisible by 4 if m is divisible by 4

Additionally, m must be comprised of many repeated prime factors for desirable traits to manifest; this is still insufficient to obtain a high quality generator. In fact, finding a high quality generator of this form is considered extremely challenging. As such, when these generators are in use, it is often in highly specific configurations with additional constraints imposed. An easy reference is Numerical Recipes' $ranqd1$ procedure, detailing a modulus of 2^{32}, a multiplier $a = 1664525$, and an increment $c = 1013904223$[ran]. Additional constraints can relate to only outputting substrings due to low periodicity in the least significant bits.

Linear Congruential Generators are most often found in either legacy or embedded systems. They are exceptionally cheap generators, and as such, find success in environments where resources are highly constrained. They should never otherwise be considered for modern use, and naturally, should never be utilized for any application in which future outputs are sensitive.

Detecting a Linear Congruential Generator often relies on a phenomenon in which outputted points, when interpreted in a multidimensional context, tend towards hyperplanes. This phenomenon is called Marsaglia's Theorem[Mar68], and is often detected by a "spectral test". Once a Linear Congruential

Generator has been determined to be the source of random numbers, breaking them is quite easy.

Given four complete outputs, we can determine an integer multiple of m by creating a 3×3 matrix of the following form

$$\begin{bmatrix} o_1 & o_2 & 1 \\ o_2 & o_3 & 1 \\ o_3 & o_4 & 1 \end{bmatrix}$$

By taking the determinant of this matrix over the integers, we arrive at a multiple of the modulus by Marsaglia's Theorem. By performing this operation with multiple sequences of output and calculating the GCD of the determinants, we are able to deduce with overwhelming probability the modulus of the matrix. An excellent practical description can be found in a writeup by Haldir[Hal04].

Upon enumerating m, we are next tasked with determining a, c, which is quite simple. Set up a system of equations of the form:

$$o_1 a + c \equiv o_2 \bmod m$$

$$o_2 a + c \equiv o_3 \bmod m$$

In the case that $(o_2 - o_1), m$ are coprime, then a solution for a can be had as $(o_3 - o_2)(o_2 - o_1)^{-1} \bmod m$ [Bur05]. In the case that the generator is not of this form, and the result of the GCD > 1, one is able to reduce the solution field, as the designer of the generator made poor choices for the parameters. Ironically, this makes our job slightly harder, with our inversion reliant on the linear Diophantine equation $ax - ny = b$, equivalent to our form $ax \equiv b \bmod n$ $(a(o_2 - o_1) \equiv (o_3 - o_2) \bmod m)$ [Bur05]. More readily, we have the form $ax_0 - b = ny_0$, for some $y_0 \in \mathbb{Z}$. We rely on the fact our congruence $ax \equiv b \bmod n$ has solutions iff $d|b, d = GCD(a, n)$. This tells us that we have d possible (and equivalent) solutions for a in our field. Given that we are able to divide by d, we set up the equation

$$\frac{a}{d}x - \frac{n}{d}y = \frac{b}{d}$$

which can be solved via the Extended Euclidean Algorithm for a particular x_0, y_0. From this point, to find an adequate set of solutions, we set up the following equations:

$$x = x_0 + \frac{n}{d}t$$

$$y = y_0 + \frac{a}{d}t$$

Which yield the value (and its equivalent values) used for a in the generator, for $t \in \mathbb{N}$. Once a has been enumerated, c becomes trivial to match from the same equations.

In the case of output constraints, i.e. truncation, we can utilize the LLL Basis Reduction Algorithm for Lattices in an attack called Stern's algorithm. The attack is significantly more complex and substantially more verbose than others discussed within this paper, and as such, only an overview will be provided. It is offered due treatment in "On Stern's Attack Against Secret Truncated Linear Congruential Generators"[SC05] for those seeking additional clarification. The foundation of the attack relies on the linearity of the generator. It utilizes the linear relationships between the partially outputted bits to construct a lattice basis defined by the differences between outputs. Applying LLL to this lattice, it then finds a vector λ such that the linear combination of output differences, weighted by the elements of λ, sum to 0. From this vector, we are able to define a polynomial such that $P(a) \equiv 0 \bmod m$. By collecting multiple such vectors, the determinant of the lattice can be calculated, in turn yielding the modulus m. In addition, with the given polynomials, if a value is found such that $P_i(a) \equiv 0 \bmod m$, the value of a has been deduced[SC05].

3.2 Linear Feedback Shift Registers

Linear Feedback Shift Registers, otherwise known as Tausworthe generators after the inventor, are a form of stateful generator simply defined as a shift register, where the input (new) bit is a linear (or, in some cases, affine) function of its previous state. The typical linear functions are either XOR or XNOR, with the former being significantly more common. These generators typically rely on selection of a number of bits in the current state as "taps", which are values that affect the next state of the generator via the given function. By careful selection of taps, one is able to create a relatively high quality generator with an adequate period. LFSRs, naturally, are still sensitive to parameter choice, however, the qualifying metric is oftentimes simply picking taps that correspond to a primitive polynomial in $GF(2)$ [Tau64].

LFSRs are often found in one of three formats - Fibonacci LFSRs, Galois LFSRs, and XOR-Shift LFSRs[MG04a][Mar03]. We will first discuss Fibonacci LFSRs, as originally presented by Tausworthe. We define an n-length LFSR as the n-bit register it occupies. We additionally define a primitive polynomial in $GF(2), P$, to dictate our taps, called a feedback polynomial. The "on" coefficients of the polynomial, i.e. those equal to one, are considered the taps, and read from left to right (similarly to little-endian bit ordering). Given a starting seed, we advance the state. The last (rightmost) bit is outputted as the first output bit. The values of the bits at the indices of the taps are combined, either by a XOR or XNOR, and fed into the start of the register (leftmost bit). This is an advancement of the state by one bit. If the feedback polynomial is indeed primitive, a configuration like this will lead to a maximal length period equal to $2^n - 1$, where n is the degree of the feedback polynomial. Given a XOR configuration, this excludes the state in which the register is comprised of all 0's, which will never change, thus motivating the XNOR construction, which will begin immediately without a seed and only halt when the register is all 1's, thus removing the need for a direct seed. It should be noted, however, that XNOR variants are uncommon.

Galois LFSRs are similar, however, on advancement of state, non-tap values are shifted to the right one, while tap values are XOR'd with the output bit before being advanced. These generators are capable of the same outputs as Fibonacci generators, and are slightly more efficient in software-only environments.

XOR-Shift LFSRs, similarly, are capable of full-period outputs utilizing only XOR and shift operations. The theory primarily lies in the concept that, given a $1 \times n$ non-zero vector y, there exists a nonsingular matrix $T : yT, yT^2, yT^3, \ldots$ produces the nonzero permutations of y, and that $ord(T) = 2^n - 1$[Mar03]. This implies that one is able to enumerate every non-zero binary vector in the order of T, suggesting a maximal period LFSR. However, computation of matrix-vector multiplication is expensive $(O(n^2))$[BLA]. Thus, a special matrix structure must be made available in order for this to be a feasible generator. One such special form is where the matrix is all 0's, with a diagonal of 1's in a specific portion such that multiplying it by itself in tandem with an addition of the identity is equivalent to a XOR and shift operation in sequence. Experimentally, it is found that three such operations are required to fully represent the space of nonzero vectors of a given length, and the number of shifts for each iteration is expressed as a "triple". A list of said triples for common word lengths can be readily found in Marsaglia's paper[Mar03].

LFSRs are prolific in usage, particularly as counters and for digital circuit testing due to their exhaustive periods. Additionally, they see light usage in cryptography - notably in Bluetooth's E0 stream cipher, which has numerous academic breaks[Vai00] and a practical break[YLV05]. They are also oftentimes found in signal scramblers. Like LCGs, they thrive in resource-constrained environments. It is not advisable to use LFSRs for cryptographic purposes, as their linear relationship makes cryptanalysis much more approachable.

Detecting LFSRs is not so easy as detecting LCGs. A common test is the construct a matrix out of output bits of the suspected block size, and then larger than the suspected block size. If the data is from a LFSR, it is likely that the matrix will be singular in the larger case, and certain that it will be nonsingular in the former[GM85]. This test is part of the NIST SP-800-22 test, and exploits the linearity of the generator, along with the linearity test.[NIS10]. However, with LFSRs, it's oftentimes most pragmatic to attempt to detect and crack them in one go if the generator type is unknown, and attempt to match outputs afterwards to verify. The approach that will be discussed is the Berlekamp-Massey Algorithm, which finds the shortest LFSR for a given output sequence. A good rule of thumb is that if the returned length of the generator is short, you have likely encountered and cracked a LFSR. If it is large (approximately half the size of the inputted bitlength), the bitstream you are testing almost certainly did not come from a LFSR. Further testing after deriving a LFSR is required to verify the correctness of the inversion.

Berlekamp-Massey essentially functions by iterating over the given output sequence, and making educated guesses about the polynomial that governs it. For motivation, we first consider a more rudimentary algorithm predicated on iterative construction and solution of linear systems based on the available output stream. By splitting the stream into equations of a presumed length of the LFSR, we arrive at a nonsingular system by definition, which can thus be efficiently solved to enumerate the feedback polynomial coefficients. Visually, for a LFSR of length 4, this is shown as

$$\begin{bmatrix} o_4 & o_3 & o_2 & o_1 \\ o_5 & o_4 & o_3 & o_2 \\ o_6 & o_5 & o_4 & o_3 \\ o_7 & o_6 & o_5 & o_4 \end{bmatrix} \begin{bmatrix} c_4 \\ c_3 \\ c_2 \\ c_1 \end{bmatrix} = \begin{bmatrix} o_5 \\ o_6 \\ o_7 \\ o_8 \end{bmatrix}$$

With c_n being the tap coefficients and o_n being the output bits. Naturally, in practicality, we will not be able to easily determine the size of the LFSR. The naive solution is to simply guess at the size of the LFSR, and iteratively solve systems until the full sequence can be mirrored. This, however, is very slow - n iterations of a general solve, leading to a $O(n^3 log(n))$ runtime[EK13] (if binary search is used - $O(n^4)$ if brute forced). To get around this, we instead focus on finding discrepancies in the proposed coefficients on the fly, lending to a reduction to $O(n^2)$[AMV96]. This is accomplished by creating a polynomial "guess", starting at $C(x) = 1$. It will then iterate and calculate a discrepancy between its output and guess, d, by adding the product of output bits and the presumed coefficient of $C(x)$ over $GF(2)$. If this is equal to zero, there is no discrepancy, and the algorithm advances. If there is a discrepancy, then $C(x)$ is adjusted such that there isn't one given the new bit, and the number of presumed errors is updated. This will, inevitably, yield a set of coefficients that produce the same bitstream utilizing a LFSR, as we have formulated a polynomial with the coefficients required to eliminate all discrepancies in the sample of output, meaning that the polynomial likely matches the feedback polynomial of the original LFSR.

This does not guarantee that we have correctly inverted the LFSR - simply that we have created a LFSR mirroring the output thus far[Mas69]. One necessarily must provide $2n$ output bits in order to recover the exact polynomial. Functionally, this means that one will never go wrong by providing more input if attempting to invert an unknown LFSR.

3.3 Mersenne Twisters

Mersenne Twisters are a stateful PRNG also built on linear relationships. They draw from an extension of LFSRs, called Generalized Feedback Shift Registers. GFSRs follow the relation

$$x_{l+n} = x_{l+m} \oplus x_l \text{ for } l, m, n \in \mathbb{N}$$

where a given x is a bitstring of length w acting as a state word. While very fast, outputs are not particularly random in higher dimensions, and the period is significantly limited[MM92a]. Twisted GFSRs (TGFSRs) improve this by instead following the relation

$$x_{l+n} = x_{l+m} \oplus x_l A | l \in \mathbb{N}$$

Where A is a $w \times w$ matrix over $GF(2)$. For a suitable choice of parameters, this lengthens the period from what is practically obtained by GFSRs ($2^n - 1$) to $2^{nw} - 1$, outputting all w-length bitstrings except the zero vector[MM92a]. To improve computation speed, A is chosen such that it is in companion matrix form, having a band of 1's along the first superdiagonal, and the bottommost row being comprised of chosen subparameters a_w such that it is a companion to an irreducible polynomial P in $GF(2)$ and constructing a linear recurrence of order w.[MM92b]. Visually, we find a 4×4 ($w = 4$)

example as

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ a_0 & a_1 & a_2 & a_3 \end{bmatrix}$$

At this point, we have constructed a Twisted Generalized Shift Feedback Register (Rationalized), which are otherwise shortened to TGFSR(R). However, due to the choice of utilizing A in this form, outputs in higher dimensions become correlated, significantly affecting the statistical properties of output[MM92b], similar to problems encountered with LCGs. The solution to this is to temper the generator, essentially analyzing the structure of outputs and constructing a corrective linear transformation to maintain equidistribution in higher dimensions. Given a set of outputs x and a $GF(2)$ matrix P, one is able to construct a new sequence $Z : z_i = x_i P$, which in turn can be expressed as a TGFSR sequence given a similar (equivalent) rational normal form R for P. Thus, with a P capable of enforcing equidistance of output, we are able to perform a whitening transform on $x \to xP = z$, correcting the issues with higher dimensional distribution.

Mersenne Twisters directly follow from TGFSR(R) constructions, with a few alterations. Most obviously, they utilize a different recurrence of the form:

$$x_{k+n} = x_{k+m} \oplus (x_k^u | x_{k+1}^l)A : k, n, r, m, w \in \mathbb{N}, A_{w \times w} : a_{ij} \in GF(2)$$

where n is the degree of the recurrence, r is an integer such that $0 \leq r \leq w-1$ with a hidden definition in x_k^u, $m : 1 \leq m \leq n$, with $x_0, ..., x_{n-1}$ as seeds of length w, and $|$ is used to denote concatenation[MM98]. Additionally, u, l refer to the upper and lower portions of the specified bitstrings, with $u = w-r$ upper bits and $l = r$ lower bits. This expression is very dense, and can be more readily comprehended as the output of the PRNG is equivalent to an internal state word XOR'd with the product of A and split, then concatenated contiguous internal state words. Mersenne Twisters maintain the form of A, and maintain tempering as discussed with TGSFR(R) generators. The alteration to the recurrence relation is of particular note, allowing for equidistribution in even higher dimensions[MM98], which in turn enables the massive period length equal to $2^{19937} - 1$ for the MT19937 variant. Additionally, they utilize a very large state buffer, making use of the structure to maximize period length.

Mersenne Twisters are regarded as exceptionally high quality PRNGs. They are often employed as the primary choice of PRNG when resources are not a concern. However, they do have issues. Originally, they could require time to "warm up", i.e. output was very similar until enough state had been traversed, and similar initialization states yielded similar output streams[MM07]. Additionally, if the initial state has an excess of 0's, the first several thousand outputs will also tend towards an excess of 0's[MM05]. They are otherwise not easy to detect, and lack general distinctive traits outside of the "warm up" period present in the 1998 iteration of the generator. However, when testing for specific implementations, i.e. MT19937, tests in dimension ≥ 624 would begin to fail, whereas they would otherwise pass in lower dimensions. Similarly to LFSRs, the easiest way to detect them is to treat an output sequence as having come from a Mersenne Twister. For the 32 bit variant MT19937, inversion can be possible by applying T^{-1} to the output streams, which will yield components of the internal state. By enumerating 624 successive values and applying the inverse tempering matrix, one can enumerate the complete state of MT19937, thus inverting it by copying the enumerated state into another instance of MT19937[TP15].

3.4 Counter-based PRNGs

Counter-based PRNGs are a relatively new class of PRNG, with the most notable ones being the Random123 Suite by D.E. Shaw[JS11]. Of additional interest, these are the first PRNGs that make no claim of cryptographic security we have encountered that have have no breaks we were able to find, which is a point of future research. These PRNGs are also substantially different in the sense that they are stateless - they do not require internal storage of any real state in order to produce random values. They are also substantially different in the sense that they are based on cryptographic primitives, with one generator, ARS, even utilizing AES-NI instructions. While other counter-based PRNGs exist, we will primarily discuss the Random123 suite, as it seems to be the most prolific. The three Random123 engines are ARS, Threefry, and Philox, all built with specific environments in mind. Each was designed to be easily parallelized and produce massive periods, with a lower bound established as 2^{64} parallel streams and period lengths of at least 2^{128}[JS11].

We will begin with the ARS generator. ARS is explicitly designed for systems with AES-NI instructions, widely available on the x86_64 architecture. It simply utilizes N, typically 5 or 7, calls to the AES-128 round function with the counter buffer as the cleartext, with a simplified key schedule based on a Weyl sequence ($k : GCD(k, m) = 1; 0, k, 2k, ... \bmod m$ is equidistributed)[Mar03]. It does not require key storage. ARS is faster than any other BigCrush resistant PRNG, counter or otherwise, when considered only on AES-NI equipped CPUs[JS11]. While ARS does not claim cryptographic security, and has not been studied to our knowledge for said property, the only reported break for reduced round AES in a similar vein to ARS is a known-key attack, with no other notable practical breaks. However, the simplified key schedule may provide alternate means of cryptanalysis not otherwise relevant to attempts at breaking AES. Detection of ARS is also quite difficult, as it has showcased in its statistical testing, which would make attacking it quite challenging for an unknowing observer.

Threefry, similarly to ARS, is based off of a well-known block cipher, Bruce Schneier's Threefish[JS11]. It largely retains the structure of Threefish, utilizing a Substitution-Permutation network over a reduced round variant, and additionally dropping the tweak in the key schedule[JS11]. Threefry also retains the rotation constants from Threefish for a word width of 64, with the number of words ≥ 4. They are otherwise generated by D.E. Shaw. Threefry thrives on CPU architectures that do not support the AES-NI instructions, and also works well on GPUs. It thus is ideal for alternative architectures that require CPU use, i.e. ARM-based architectures. Once again, detection is likely non-trivial; the same principles that make it a high quality PRNG make it hard to differentiate from uniformly random noise. Additionally, there are no known practical cryptographic attacks, even against reduced round variants, although no security assurance has been made and cryptanalysis specific to Threefry has not been performed to our knowledge.

Lastly, Philox is an entirely new counter-based PRNG. It is based on weakened well-studied primitives, rather than preexisting algorithms, utilizing a very simple Substitution-Permutation network along with a very short Feistel network. It was primarily intended for usage on GPUs, and has been implemented in Nvidia's cuRAND library. It has, by far, the highest throughput, at 145.3 GB/s[JS11] on a Nvidia GPU, beating out XORWOW and MRG variants. Curiously, it was adopted into ISO C++ 26 for utilization in the Standard Template Library, and as such, may become the most commonly used variant of a counter-based PRNG. Threefry would have likely been the superior option, given the architecture-agnostic approach of the STL and its higher CPU throughput[JS11]. Additionally, having implemented Philox for GCC's libstdc++-v3, we found that the main benefit of parallelization was prevented by the interface stipulated by ISO standards. SIMD instructions may help with computation speed as standard interfaces are implemented for C++26 compliance. Philox is a new system, and no cryptanalysis has been performed on it or a related system to our knowledge. Cryptanalysis of it is an excellent topic for further research, given its inclusion in two prolific libraries.

4 Computationally Secure Pseudo-Random Number Generation/DRBGs

Computationally Secure Pseudo-Random Number Generators (CSPRNGs), or alternatively Deterministic Random Bit Generators (DRBGs) in the NIST verbiage, are specific variants of PRNGs that exist for the case where inversion is catastrophic. For example, secret generation, MFA token generation, and nonce/IV generation all rely heavily on these forms of generators. They still maintain the same requirements as PRNGs, but are additionally designed with security in mind rather than just statistical quality.

Typically, these types of generators are either created as stream cipher engines, derived from cryptographic hashes, or follow from number theory. Occasionally, hybrid approaches are also encountered. For an example of a CSPRNG, going back to /dev/random and /dev/urandom, contemporary Linux kernel behavior seeds a ChaCha20 cipher with system entropy on boot, and now gives ChaCha20 outputs when these files are polled. This is particularly of note because it opens another motivation for CSPRNGs, being maintenance of entropy. The PRNGs discussed prior have usually had known weaknesses, which at some point enable a hostile actor to enumerate their state and thus all future outputs, reducing the apparent entropy to zero as a byproduct of the determinism of the algorithms. However, CSPRNGs work instead to preserve the entropy of their seeds. Output generated naturally reduces the effective entropy - often called machine entropy - of outputs, as it necessarily informs about state. However, CSPRNGs, by definition, are designed such that this is difficult, and oftentimes practically impossible as of this time, to exploit. Additionally, if the CSPRNG is well-maintained and has additional entropy from a TRNG source periodically injected, it becomes even harder to capitalize on this.

4.1 Stream Ciphers

Stream ciphers, by definition, utilize pseudo-random number generation as their means of encipherment. They must do so in such a way that the relationships between the generated numbers are difficult to find structure in - a notable cause of failure in many linear systems, such as those discussed earlier. Furthermore, this style of cipher has become a standard in modern cryptography, with AES-GCM and ChaCha20-Poly1305 being prolific in usage for cryptographic operations because of their speed and resistance to side-channel attacks (presuming AES-NI instructions or otherwise side-channel resistant AES implementation). As such, they are often trusted and employed as CSPRNGs.

We will first discuss RC4, a stream cipher created by RSA Security. RC4 is well known for being broken - so much so, that its use is actually directly prohibited in TLS by RFC 7465[Pop15]. However, RC4 has historically been used as a PRNG, often as the arc4random API call, thus particularly motivating our study. RC4, when employed in this configuration, did not allow user seeding, and was rather directly seeded from a TRNG source from the system, then allowing calls to it. RC4's design has never been officially released, and as such, we will base our description of it on the 2002 paper "A Practical Attack on Broadcast RC4" by Itsik Mantin and Adi Shamir[IM02].

RC4 is a stream cipher consisting of a key scheduler and an output feature. The key scheduler is not relevant for our particular study, however, has been a source of other attacks. Those interested in attacks utilizing RC4's key scheduling algorithm should refer to Fluhrer, Mantin, and Shamir's paper "Weaknesses in the Key Scheduling Algorithm of RC4"[SF01]. The output function of the cipher utilizes an internal state S, consisting of a permutation of all 2^n bit words, with n typically selected as 8 in practical implementations[IM02]. The particular ordering of S is decided by the key scheduler. Given S, the output function initializes two pointer counters $i, j = 0$. i is incremented by one, and j is incremented by adding itself and the value pointed to by i, both operations occurring modulo 2^n. The values pointed to by i, j are then swapped, and output is given as the index pointed to by $S[i] + S[j] \bmod 2^n$. It is of note that every entry of S is swapped within $N = 2^n$ consecutive outputs, denoting the state will change rapidly.

The attack we will focus on for RC4 finds applications in differentiation from random, which in turn detracts from its usefulness as a PRNG in general, as well as opens it to other attacks once differentiated. Specifically, we focus on a rather atypical form of differentiation called indistinguishability in polynomial sampling, which can be understood as allowing an observer to reset and rekey a stream many times[IM02]. This test highlights a critical flaw in RC4 that was missed in testing over the entire stream - the second output word is zero with twice the probability of any other output[IM02], allowing for early differentiation in the context of multiple different streams. Practical testing has shown that by analyzing only 200 streams, RC4 is distinguishable from random over 64 percent of the time[IM02], unacceptable for any PRNG. Those interested in further exploitation of RC4, once detected, will find additional treatment in Klein's Attack[Kle06], the Royal Holloway Attack[CG15], Mantin's Bar Mitzvah Attack[Man15], and the NOMORE Attack[MV15].

We will now move to ChaCha20, a modern and proven stream cipher. ChaCha20 is derived from Salsa20, a previous stream cipher by Daniel Bernstein[Ber08]. ChaCha20 is an ARX cipher, meaning that it utilizes addition, bitwise rotation, and XOR instructions as means for creation of confusion and diffusion. This design leads to ease of implementation and naturally resists timing attacks[YY]. Additionally, ChaCha20 has yet to be broken past 7 of its 20 rounds, leading to a high degree of confidence in the cipher[JPA08]. As previously mentioned, it is currently utilized as the CSPRNG acting as /dev/(u)random.

ChaCha20 begins by constructing a 4×4 matrix, with the first four values being constants, the next 8 being a 256-bit key, and the final four being the 32-bit counter and a 96-bit nonce/IV. It then defines a quarter round[Ber08] as $QR(a, b, c, d)$:

```
a += b; d ^= a; d <<<= 16;
c += d; b ^= c; b <<<= 12;
a += b; d ^= a; d <<<= 8;
c += d; b ^= c; b <<<= 7;
```

Four quarter rounds, with array index arguments of $QR(0..3, 4..7, 8..11, 12..15)$ comprises an odd round, and four quarter rounds with arguments of $QR(0..3, [5, 6, 7, 4], [10, 11, 8, 9], [15, 12, 13, 14])$ comprise an even round. Together, this creates a double round, ten of which are utilized for the full ChaCha20 cipher. Of additional note, in high demand environments, a tradeoff of assurance and speed can be made. As previously stated, the current best cryptanalysis of ChaCha has been 7 rounds[JPA08], in turn leading to Bernstein additionally endorsing ChaCha20/12 and ChaCha20/8[Ber08], which are reduced 12 and 8 round variants of the ChaCha cipher favoring speed over assurance of security, while still offering a degree of it. It should additionally be noted that it is good practice to periodically reseed (rekey) the generator with fresh entropy in order to retain security in case of state leak at any point.

4.2 Block Ciphers in CTR Modes

In a similar vein to stream ciphers and counter-based PRNGs, we find that block ciphers operating in counter mode can serve effectively as CSPRNGs, with a particular structure of this even being standardized by NIST as CTR_DRBG. CTR mode can be understood as a mode of operation that takes a block cipher, such as AES, Kuznyechik, or ShangMi 4, and alters the behavior to be more like a stream cipher. This mode of operation, with added authentication, is very common for actual encryption, often manifesting as AES-256-GCM (Galois/Counter Mode). However, we will be skipping over authentication, as we are only considering usage of CTR mode block ciphers as CSPRNGs.

We first begin with a general discussion of CTR mode before diving specifically into CTR_DRBG. Given a counter C, a nonce/IV N, a key k, and a plaintext p, one is able to combine C, N with some form of invertible operation, i.e. XOR, which can then be passed as cleartext to the block cipher keyed with k. The output of this operation is XOR'd with p, creating stream cipher-like behavior. For our usage, however, the pure output of the cipher is the desirable stream and may be utilized as such, so long as the underlying block cipher is still regarded as secure. In fact, it can be stated that an issue with CTR mode explicitly implies the issue is actually in the underlying block cipher, and not CTR mode[NF10]. Naturally, AES is an exceedingly common choice. It is additionally worth mentioning that block ciphers (and stream ciphers for that matter) have mandatory rekeying periods, and utilization as a CSPRNG does not preclude this. In fact, failure to judiciously rekey can introduce a means of differentiating from random - the birthday bound states a collision should occur within the first $2^{\frac{n}{2}}$ block lengths, whereas a cipher in CTR mode will not collide within the first 2^n outputs.

NIST's standardized CTR_DRBG, or *Counter Deterministic Random Bit Generator*, is a specific variation of the previously described generic CTR mode CSPRNG. Specifically, it only supports 3DES (3DEA), AES-128, AES-192, and AES-256[EB15], which is completely unsurprising given the standardizing body. CTR_DRBG supports two modes of operation otherwise, one with a key derivation function, and the other without. Regardless of form, its state consists of a counter V, a key k, and a reseed counter *reseed_counter*. It initializes using an a specified initialization routine, in which k, V are initially derived from a concatenation of TRNG data, a nonce, and an additional "personalization_string" put through a hash-based derivation function. CTR_DRBG_UPDATE is then called, fully initializing the generator by iterating the counter and putting the old state through the utilized block cipher to derive a new key and counter value for use. CTR_DRBG_UPDATE is additionally called when entropy is reinjected into the generator, i.e. when additional personalized input is injected or on a complete reseed. Otherwise, the generator will behave normally as if in CTR mode, except it will return an arbitrary length of user-requested bits rather than a fixed length as the more general description above would, favoring the most significant bits in the case of a request indivisible by the block size.[EB15].

It should be noted that there is a theoretical attack on this design showcasing it is differentiable from random oracle and has a reduced security level. The attack holds that due to the counter iteration occurring between outputs when the requested number of bits is greater than the block size of the cipher, one gains an advantage in differentiation from random, and the security of the cipher is reduced to that of its block size rather than its key size[Cam06]. This does not have a practical effect in statistical measures nor on the security when AES is used as directed.[Cam06]. It is additionally worthy of stating that this form has had mixed reception - some authors reject its use[Kan07], whereas others conclude it is fine for use[VTH20]. The more modern opinion is that it is acceptable in its current form[VTH20]. An alternative with similar construction and better reception is Schneier and Ferguson's *Fortuna* CSPRNG[NF03].

4.3 Hash-Based

Hash-based PRNGs are not particularly common constructions, however, are worth mentioning due to NIST's standardization of them, and additionally as a result of Keccak's selection as SHA3. Keccak, unlike its predecessors of Merkle-Damgard construction, utilizes a sponge function, which is proven indifferentiable from random oracle[GB08]. This strongly implies that hash-based CSPRNGs may become significantly more popular in the future, as this property additionally only depends on a random permutation of data[GB08] and provides a very strong assurance of quality of randomness. As such, we will explore contemporary methods, primarily NIST's Hash_DRBG construction, to gain an understanding of existing structures in this domain as we move towards sponge-based constructions.

Hash_DRBG (which we will call HashDRBG) is another NIST DRBG, and as such maintains the same initialization function as described in CTR_DRBG[EB15]. It maintains a state of V bits, updated on each call to the DRBG, a constant C derived from the seed, and a counter *reseed_counter* to track requests and trigger a reseed. Similar to its counter-based counterpart, HashDRBG accepts an entropic input, a nonce/IV, and a personalization string on initialization. It will concatenate these values, and perform the NIST derivation function on the concatenated inputs, declaring V to be the result. It will then declare C to be the result of the same derivation function called on V with a byte of 0's prepended, and increment *reseed_counter*[EB15]. Reseeding with new entropy works similarly, concatenating a byte of value 1, V, the new entropy, and any additional input. It then derives the new seed and constant C in the same way as initialization is performed, setting *reseed_counter* equal to 1. To generate output, it must have a requested number of bits r, then deriving an iteration count as $m = \lceil \frac{r}{outlen} \rceil$. Given V, it will also prepare an internal buffer W. Then, it will iteratively hash the value of of V, incrementing it by one each time over $2^{seedlen}$ and concatenating the output with W. It will then return the requested number of W's most significant bits.[EB15]

Historically, this construction has been used with previous iterations of the SHA standard. While SHA2 is considered a secure iteration of a hash function because it has collision resistance and preimage resistance, this is insufficient to assert the quality of it as a CSPRNG/DRBG. This instead requires a proof that it behaves as a *pseudo-random family*, meaning that it is indifferentiable from random oracle when paired with a secret value[EB15], to be considered secure. While it would initially seem that collision resistance would imply the requisite diffusion characteristics intuitively, SHA2 (or any other MD construction) lacks such a proof[DJ18]. As such, the security of the scheme cannot be strongly asserted while using SHA2, despite successful heuristics with statistics suites[EB15]. However, SHA3 does have such a proof[GB08], implying that this construction, as currently understood and analyzed, can be considered safe and of high quality when utilized with SHA3/Keccak[EB15].

4.4 Number Theoretic Generators

Number Theoretic Generators are a specific class of CSPRNGs that make security assurances reliant on problems from number theory, in a similar vein to asymmetric algorithms. These forms of generators are very rare in practice, and largely only of interest in theory, often because of slower performance to more typical CSPRNGs. However, the ability to reduce to well-studied number-theoretic problems can be alluring for particularly sensitive operations, and as such, it would be remiss to not explore them.

This section will also cover DUAL_EC_DRBG, a generator proposed by NIST and later rescinded under suspicion of a backdoor being emplaced. While the precise value used to unlock said backdoor is not known, the general structure of it is, and as such, is worth exploration. Additionally, this will motivate further understanding of the consequences of insecure PRNG in a cryptographic context, along with demonstrate the importance of neutrality in standard selection and public analysis of cryptologic standards.

4.4.1 Blum-Blum-Shub

To build familiarity, we start with a relatively easy to understand number-theoretic stateful generator called Blum-Blum-Shub (BBS). Similarly to RSA, we initially derive two primes of cryptographic size p, q (i.e. via Miller-Rabin validation). Furthermore, it must hold that $p \equiv q \equiv 3 \bmod 4$ to sate the original definition[LBS86]. These primes are multiplied together to form a semiprime M, which will serve as the modulus of our relation. The relation is thus defined as:

$$x_{n+1} = x_n^2 \bmod M$$

Notably, the full output, or even a majority output, of the state transition is not given to the end user. Typically, only a parity bit is outputted[LBS86]. This drastically reduces the throughput of the generator - multiplication, especially constant operation multiplication, is expensive $O(n^2)$, and only getting a single parity bit for it makes for an extremely slow system. This is done, naturally, to prevent others from predicting the stream, as displaying the whole state, or even a majority of it, can allow for trivial reconstruction. However, since a parity bit can be considered a hardcore predicate (a one-way function $F : \{0,1\}^n \to \{0,1\}$), it does not reveal enough information to easily reconstruct the state of the generator.

The number theory problem at play here is the famous Quadratic Residuosity Problem of Gauss, stated as follows:

Let N be a product of two distinct odd primes. Exactly half the elements of Z_N^* have Jacobi symbol +1, the other half have Jacobi symbol -1. Denote the former by $Z_N^*(+1)$ and the latter by $Z_N^*(-1)$. None of the elements of $Z_N^*(-1)$ and exactly half the elements of $Z_N^*(+1)$ are quadratic residues. The quadratic residuosity problem with parameters N and x consists in deciding, for x in $Z_N^*(+1)$, whether or not x is a quadratic residue.[LBS86]

This can be more readily understood as follows. Given a multiplicative modulo N, half of the elements will not have a root r such that $x \equiv r^2 \bmod N$ (quadratic non-residue). The other half may have such an element, and deciding if it does is computationally hard. The assertion we have thus far is that guessing the parity of $\sqrt{x}$ is hard - we must equate this to the quadratic residuosity problem. This reduction is achievable by the fact that, given $p \equiv q \equiv 3 \bmod 4$, any given quadratic residue will have 4 square roots, exactly one of which is also a quadratic residue[MG04b]. We identify this as the unique root of the given residue. Thus, in order to determine the parity of the unique root of the given residue, one must determine which root is also a residue, which is the quadratic residuosity problem.

The previous explanation is informal and follows intuition for pedagogical reasons, however, a formal proof is available in “About Random Bits” by Geisler, Krøigård, and Danielsen[MG04b]. Furthermore, this problem can be shown to be as hard as factoring once made equivalent[LBS86].

4.4.2 DUAL_EC_DRBG

DUAL_EC_DRBG (ECRNG) is a proposed stateful generator built off of elliptic curve arithmetic. While originally offering no security reduction, a third party reduction was performed by Daniel Brown and Kristian Gjøsteen[DRLB07], positing hardness based on three problems:

- Elliptic Curve Decisional Diffie-Hellman Problem: Let $\mathcal{E}$ be an elliptic curve defined over a finite field of prime characteristic $\mathbb{F}_p$. Let $G \in \mathcal{E}(\mathbb{F}_p)$ be a generator for an additive cyclic (sub)group $\mathcal{E}_G$ of prime order q over the points of $\mathcal{E}$. Given $G, s_1G, s_2G, Q : s_1, s_2 \in \mathbb{Z}_q$ and s_1, s_2 chosen uniformly at random, decide if $Q = s_1s_2G$ or if Q is a random valid point in $\mathcal{E}_G$. This problem is well-studied and assumed hard[Bon98].
- X-Logarithm Problem: Given an elliptic curve point, determine whether it's discrete logarithm is congruent to the x coordinate of an elliptic curve point. This problem was new for this generator, and reliance on it is questionable, however, it is assumed hard[DRLB07].
- Truncated Point Problem: Given a bitstring of a specific length n, determine whether it is obtained by truncating the x coordinate of a random elliptic curve point. This problem is more favorably viewed than X-Logarithm, however, is still not considered hard when insufficient bits are truncated (as was the case with ECRNG)[DRLB07].

We will now describe the algorithm. ECRNG begins with a given field characteristic p, an equation of the form $y^2 = x^3 - 3x + b$ defining an elliptic curve $\mathcal{E}$, and two defined base points $P, Q \in \mathcal{E}(F_p)$. We are equipped with several functions as follows:

- $X(x, y) \to x$: Extracts the x-coordinate of a given elliptic curve point in affine coordinates.
- $t(x) : \{0,1\}^k \to \{0,1\}^{k-n}$: Truncates n MSB of data, with the default being 16[EB12].
- $g_P(x) = X(xP)$: Performs point doubling of P x times (scalar multiplication).
- $g_Q(x) = t(X(xQ))$: Performs point doubling of Q x times, extracts the x coordinate, and truncates it.

The generator is then very simple. Select a seed $s \in F_p$, and pass it to g_P to establish the first state. The outputted values are calls to g_Q passing the current state, and the new state is generated by calling g_P on itself.

The notable problems with ECRNG do not follow from its construction, necessarily (although there was concern over variations from random[BS]). They do, however, follow from the specified constants asserted by NIST. NIST specifies several things, described as follows:

- Operations must take place on a NIST curve (P-256, P-384, P-521)[EB12]
- Truncation should be 16 or 17 bits[EB12]
- The defined points P, Q must be used to meet certifying standards, but are not explained[EB12]

Notably, the NIST curves are all of cofactor 1[DJB25], denoting that there must necessarily exist some scalar $s : sQ = P$. Of additional concern, there is no justification given for how these constants are derived[EB12]. We now consider a single output of the generator, t. If A is the point with x coordinate t, then there also must exist some scalar $r : A = rQ$. We can now construct the main portion of the backdoor[DS07]:

- $sA = srQ$ $(A = rQ)$
- $srQ = rP$ $(sQ = P)$
- $rP =$ state

With a single output, we are able to reconstruct the internal state of the generator, nullifying any concept of computational security. Naturally, this example assumes that no truncation occurs. However, given that only 16 bits were to be truncated, reconstruction of the affine x coordinate is very quick to compute by guessing and seeing if such a valid point with that coordinate lies on the curve. Many were conflicted if the backdoor was intentional, until it was revealed that it was intentionally created as part of a long-running operation to create a backdoor in public cryptography[NP]. By being able to recreate state in a CSPRNG, one in turn is almost certainly able to enumerate the secret used in forms of Diffie-Hellman key agreement, or any secret shared by a KEM. Furthermore, if one is able to do this, then they are able to deduce the shared secret used for symmetric encryption, completely breaking the process of setting up a secure channel.

Naturally, ECRNG was withdrawn from standardization after this was revealed and has not, in any form, been standardized again.

5 Future?

As we move to the post-quantum future, many are apprehensive about the cryptography governing their communication privacy. Fortunately, many sufficient systems already exist. Popular symmetric ciphers, such as AES and ChaCha20, have shown no real indication of being broken by a quantum computer. As such, they can be safely used as CSPRNGs in the configurations we have discussed. Additionally, hash-based CSPRNGs show immense promise with the standardization of SHA3, with sponge constructions proven indifferentiable even in a quantum context[GAT]. Potential areas of future interest are in post-quantum number theoretic generators, as research in this area currently seems to be sparse.

References

[AMV96] P. van Oorschot A. Menezes and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[Ber08] Daniel Bernstein. Chacha, a variant of salsa20. Technical report, The University of Illinois at Chicago, 2008.

[BLA] *Basic Linear Algebra Subprograms.*

[Bon98] Dan Boneh. The decisional diffie-hellman problem. In *Algorithmic Number Theory*, 1998.

[BS] Andrey Sidorenko Barry Schoenmakers. Cryptanalysis of the dual elliptic curve pseudorandom generator. Retrieved from IACR ePrint.

[Bur05] David Burton. *Elementary Number Theory*, volume 6. McGraw-Hill, 2005.

[Cam06] Matthew J Campagna. Security bounds for the nist codebook-based deterministic random bit generator. Technical report, Pitney Bowes Inc., 2006.

[CG15] Thyla van der Merwe Christina Garman, Kenneth Paterson. Attacks only get better: Password recovery attacks against rc4 in tls. In *USENIX*, 2015.

[Cha16] Chi Wei Cliburn Chan. Computational statistics in python. Technical report, Duke University, 2016.

[CVEa] OpenSSL Debian PRNG outputs predictable numbers.

[CVEb] moodlelib.php Mersenne Twister for password recovery tokens.

[DJ18] Ueli Maurer Daniel Jost. Security definitions for hash functions: Combining uce and indifferentiability. In *Security and Cryptography for Networks*, 2018.

[DJB25] Tanja Lange Daniel J. Bernstein. Safecurves: choosing safe curves for elliptic-curve cryptography. Technical report, https://safecurves.cr.yp.to/, Accessed 2 June 2025.

[DRLB07] Kristian Gjøsteen Daniel R. L. Brown. A security analysis of the nist sp800-90 elliptic curve random number generator. In *Advances in Cryptology: CRYPTO 2007*, 2007.

[DS07] Niels Ferguson Dan Shumow. On the possibility of a back door in the nist sp800-90 dual ec prng. Technical report, Microsoft, 2007.

[EB12] John Kelsey Elaine Barker. Nist sp-800-90a: Recommendation for random number generation using deterministic random bit generators. Technical report, National Institute for Standards and Technology, 2012.

[EB15] John Kelsey Elaine Barker. Nist sp-800-90a rev 1: Recommendation for random number generation using deterministic random bit generators. Technical report, National Institute for Standards and Technology, 2015.

[EK13] George Yuhasz Erich Kaltofen. On the matrix berlekamp-massey algorithm. *ACM Transactions*, 2013.

[GAT] Christian Majenz Gorjan Alagic, Joseph Carolan and Saliha Tokat. The sponge is quantum indifferentiable. Pre-publication. Revised 13 May 2025. Retrieved 2 June 2025.

[GB08] Michaël Peeters Gilles Van Assche Guido Bertoni, Joan Daemen. On the indifferentiability of the sponge construction. In *EUROCRYPT '08*, 2008.

[GM85] Liang-Huei Tsay George Marsaglia. Matrices and the structure of random number sequences. *Linear Algebra and its Applications Vol. 67*, 1985.

[Gü12] Dr. Mesut Güneş. Random-number generation. Technical report, Freie Universität Berlin, 2012.

[Hal04] Haldir. How to crack a linear congruential generator. *Haldir*, 2004.

[IM02] Adi Shamir Itsik Mantin. A practical attack on broadcast rc4. In *Fast Software Encryption*, 2002.

[JPA08] Shahram Khazaei Willi Meier Christian Rechberger Jean-Philippe Aumasson, Simon Fischer. New features of latin dances: Analysis of salsa, chacha, and rumba. In *Fast Software Encryption*, 2008.

[JS11] Ron Dror David E. Shaw John Salmon, Mark Moraes. Parallel random numbers: As easy as 1, 2, 3. Technical report, D.E. Shaw Research, 2011.

[Kan07] Wilson Kan. Analysis of underlying assumptions in nist drbgs. Technical report, Pitney Bowes Inc., 2007.

[Kle06] Andreas Klein. Attacks on the rc4 stream cipher. *Designs, Codes, and Cryptography*, 2006.

[LBS86] M. Blum L. Blum and M. Shub. A simple unpredictable pseudo-random number generator. *SIAM Journal on Computing*, 1986.

[Lut20] Andy Lutomirski. [patch v3] rework random blocking. Technical report, Linux Kernel, 2020.

[Man15] Itsik Mantin. Attacking ssl when using rc4. In *Hacker Intelligence Initiative*, 2015.

[Mar68] George Marsaglia. Random numbers fall mainly in the planes. *Proceedings of the National Academy of Sciences*, 1968.

[Mar03] George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 2003.

[Mas69] James L. Massey. Shift-register synthesis and bch decoding. *IEEE Transactions on Information Theory*, 1969.

[MG04a] Andrew Klapper Mark Goresky. Fibonacci and galois representations of feedback with carry shift registers. Technical report, Institute for Advanced Study, University of Kentucky, 2004.

[MG04b] Andreas Danielsen Martin Geisler, Mikkel Krøigård. About random bits. Technical report, Aarhus University, 2004.

[MM92a] Yoshiharu Kurita Makoto Matsumoto. Twisted gfsr generators. *ACM Transactions on Modeling and Computer Simulations, Vol. 2*, 1992.

[MM92b] Yoshiharu Kurita Makoto Matsumoto. Twisted gfsr generators ii. *ACM Transactions on Modeling and Computer Simulations, Vol. 2*, 1992.

[MM98] Takuji Nishimura Makoto Matsumoto. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation, Volume 8*, 1998.

[MM05] Mariko Hagita Mutsuo Saito Makoto Matsumoto, Takuji Nishimura. Cryptographic mersenne twister and fubuki stream/block cipher. *IACR*, 2005.

[MM07] Takuji Nishimura Makoto Matsumoto. Mersenne twister with improved initialization. Technical report, Hiroshima University, 2007.

[MV15] Frank Piessens Mark Vanhoef. All your biases belong to us: Breaking rc4 in wpa-tkip and tls. In *USENIX*, 2015.

[NF03] Bruce Schneier Niels Ferguson. *Practical Cryptography.* John Wiley and Sons, 2003.

[NF10] Tadayoshi Kohno Niels Ferguson, Bruce Schneier. *Cryptography Engineering: Design Principles and Practical Applications.* Wiley Publishing, Inc., 2010.

[NIS10] NIST. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, San Vo, 2010.

[NP] Scott Shane Nicole Perlroth, Jeff Larson. N.s.a. able to foil basic safeguards of privacy on web. https://www.nytimes.com/2013/09/06/us/nsa-foils-much-internet-encryption.html. Accessed: 2 June, 2025.

[Pop15] A. Popov. Prohibiting rc4 cipher suites. Technical report, Internet Engineering Task Force, 2015.

[ran] https://oeis.org/A384451. Retrieved on 2 June 2025.

[SC05] Igor E. Shparlinsk Scott Contini. On stern's attack against secret truncated linear congruential generators. *Springer Information Security and Privacy*, 2005.

[SF01] Adi Shamir Scott Fluhrer, Itsik Mantin. Weaknesses in the key scheduling algorithm of rc4. In *Selected Areas of Cryptography*, 2001.

[Tau64] Robert Tausworthe. Random numbers generated by linear recurrence modulo two. *American Mathematical Society*, 1964.

[TH64] A.R. Dobell T.E. Hull. Mixed congruential random number generators for binary machines. *ACM*, 1964.

[Tho58] W.E. Thompson. A modified congruence method of generating pseudo-random numbers. *The Computer Journal, Vol 1, Issue 2*, 1958.

[TP15] Alex Balducci Marcin Wielgoszewski Thomas Ptacek, Sean Devlin. Cryptopals 23. Technical report, CryptoPals, 2015.

[Vai00] Juha T. Vainio. Bluetooth security. Technical report, Helsinki University of Technology, 2000.

[VTH20] Yaobin Shen Viet Tung Hoang. Security analysis of nist ctr_drbg. In *Advances in Cryptology - CRYPT0 2020'*, 2020.

[WP69] T.P. Bogyo W.H. Payne, J.R. Rabung. Coding the lehmer pseudo-random number generator. *Communications of the ACM Vol. 12 Num. 2*, 1969.

[YLV05] Willi Meier Yi Lu and Serge Vaudenay. The conditional correlation attack: A practical attack on bluetooth encryption. In *CRYPTO 2005*, 2005.

[YY] Elisabeth Oswald Yan Yan. Examining the practical side channel resilience of arx-boxes (extended). Retrieved from the International Association for Cryptologic Research ePrint.