JEnt v2.2.0 LFSR Conditioning Analysis

Document Version: TRD9
Date: December 9, 2025

By:
Joshua E. Hill
KeyPair Consulting Inc.
987 Osos Street
San Luis Obispo, CA 93401
United States

Yvonne Cliff
Teron Labs
Level 2, 14 Moore St
Canberra, ACT 2601
Australia

Abstract

In this paper we analyze the non-vetted LFSR-based conditioning component included in JEnt v2.2.0. We present an explicit linear model of this conditioning component, characterize its diffusion properties, and develop a heuristic analysis of its conditioned output min entropy. We then describe the interaction between this analysis and the SP 800-90B non-vetted conditioning requirements, including the observed min entropy assessment distribution for the conditioned output statistical testing that SP 800-90B requires for non-vetted conditioning components. We conclude with guidance for selecting alternate parameters that maximize either the per block output min entropy or min entropy per unit time.

Keywords

Jitter Entropy Library (JEnt), LFSR, linear model, entropy source, SP 800-90B, non-vetted conditioning, min entropy lower bound, Entropy Source Validation (ESV).

Table of Contents

1 Introduction

The CPU Jitter Random Number Generator (JEnt) library v2.2.0 ([JEnt v2.2.0], released September 2019) has been widely integrated into many cryptographic modules and it forms the basis of several entropy sources with ESV certificates (e.g., #E8, #E19, #E20, #E37, #E47, #E48, #E50, #E54, #E59, #E60, #E61, #E62, #E90, #E99, #E117, #E151, #E174, #E175, #E226, #E235). JEnt v2.2.0 was the first JEnt version that intended compliance to NIST SP 800-90B, but a substantial amount of development has occurred in the years since JEnt v2.2.0's release and there are many more recent JEnt versions that are more amenable to validation against the SP 800-90B requirements. There are several challenges when validating JEnt v2.2.0 against the requirements of SP 800-90B, [Hill 2023] but it is still useful to discuss the SP 800-90B (sometimes non-)compliance of this codebase, as it is widely integrated into settings that make replacement difficult (e.g., many Linux kernel versions include a JEnt version that is based on JEnt v2.2.0).

One change that was made to modern versions of JEnt was the inclusion of a vetted conditioning function to replace the Linear Feedback Shift Register (LFSR) used as a conditioning component in JEnt v2.2.0. This new vetted conditioner makes validation to the conditioning requirements of SP 800-90B much more straightforward. In the instance where SP 800-90B validation of an entropy source using JEnt v2.2.0 is desired, the [JEnt Design] document provides substantial validation guidance in many areas, but the included conditioning analysis is largely limited to an application of the [SP 800-90B §3.1.5.1.2] random map analysis, which is only sufficient for analysis of vetted conditioning functions[1]. As such, the vendor and test lab are left with the task of creating a technical assessment of the impact of the non-vetted LFSR-based conditioner used within the entropy source.

The levels of conditioned output min entropy documented in the published JEnt v2.2.0 ESV certificates have been inconsistent over time and are currently in a range that suggests that the documented values were constrained by the required [SP 800-90B §3.1.5.2] statistical entropy assessment of a conditioned sequential dataset[2].

In this paper, we present a theoretical basis for modeling this non-vetted LFSR-based conditioning, provide verification that this model is equivalent to the conditioning implemented within JEnt v2.2.0, and present one possible approach to using this model to find a defensible lower bound for entropy of 42.3052 bits of min entropy per conditioned 64-bit output block under the assumption that $\mathtt{osr}$ is set reasonably.

In this paper, we also provide the distribution of statistical entropy assessment ($h' \times n_{\text{out}}$) results for essentially any non-vetted conditioning function whose output is suitably pseudorandom (which applies to most non-vetted conditioning components). We finally describe some optional changes to JEnt v2.2.0 that would allow for higher conditioned output block min entropy claims (56.05 to 59.75 bits of min entropy per 64-bit block of conditioned output for roughly 95% of such validations), though this increased claim comes at the cost of decreasing the entropy output from the entropy source per unit time.

[1] Note that the included discussion in [JEnt Design §7.2.6] presumes that $\mathtt{osr} = 1$ and misinterprets the meaning of h' described in [SP 800-90B §3.1.5.2], consequently omitting any impact from the required statistical min entropy assessment of a conditioned sequential dataset described in [SP 800-90B §3.1.1]. The argument in [JEnt Design §7.2.26] is the statement "An LFSR with a primitive and irreducible polynomial is considered to not diminish the entropy", which is true of *some* LFSR-based designs, but is certainly not true with *this* conditioning component used as it is, as the LFSR data input rate is always greater than the LFSR data output rate.

[2] All current JEnt v2.2.0 ESV certs that do not further condition the LFSR output data are credited with 56.22 to 60 bits of min entropy per 64-bit block of conditioned output. Conditioned min entropy claims less than the expected statistical entropy assessment range may indicate the entropy claim was reduced as a consequence of using the LFSR-specific mathematical evidence referenced in [SP 800-90B §3.2.3, Requirement 5] and [IG D.K, Resolution 5]. [NIST SHALL ID #52] and [NIST SHALL IDs #106-#107] are marked with contradictory requirement statuses, which suggests that such mathematical evidence may not have been a requirement early in the ESV program, but the current status of these requirements is not clear.

2 LFSR Modeling

The conditioning component used within JEnt v2.2.0 is a 64-bit LFSR used in a multiplicative scrambler mode, where the raw symbol is fed into the LFSR a bit at a time, clocking the LFSR after each bit is fed in, until all 64 bits have been integrated into the LFSR. This is repeated until $64 \times$ osr "non-stuck" raw symbols are fed in, at which point the current 64-bit state of the LFSR is used as the output of the LFSR conditioning component.

The feedback polynomial for the LFSR in use is $p(z) = z^{64} + z^{61} + z^{56} + z^{31} + z^{28} + z^{23} + 1$, which is primitive. [Živković 1994] In other words, $p(z)$ is an irreducible polynomial over the ring $\mathbb{F}_2[z]$ so the quotient ring $\mathbb{F}_2[z]/\langle p(z)\rangle$ is isomorphic to the finite field $\mathbb{F}_{2^{64}}$. Further, the quotient ring element $z + \langle p(z)\rangle$ generates the full multiplicative group associated with that field, that is, the set $\{(z + \langle p(z)\rangle)^j\}_{j=1}^{2^{64}-1}$ is a group under multiplication of order $2^{64} - 1$, and is (group) isomorphic to $\mathbb{F}_{2^{64}}^{\times}$. Because the characteristic polynomial is primitive, we are guaranteed that the corresponding LFSR will have the maximal period, $2^{64} - 1$.

2.1 LFSR Conditioning as Linear Transforms

We can connect this base LFSR operation to the LFSR in multiplicative scrambler mode. The processing of an LFSR in this multiplicative scrambler mode can be expressed as a function of the current state ($\boldsymbol{s}$) and the input ($\boldsymbol{x}$); we denote this function as $\boldsymbol{f}(\boldsymbol{s}, \boldsymbol{x})$, where bolded terms represent vector quantities. Each bit of the function's output is simply an XOR of specific (feedback polynomial-determined) fixed bit places of the initial state and the input data bits. Using the commutativity of XOR and the non-effect of XORing zero bits[3], we then see that we can represent the jth bit as a sum of the effect due to the initial state and the effect of the input value, that is

$$f^{(j)}(\boldsymbol{s}, \boldsymbol{x}) = f^{(j)}(\boldsymbol{s}, \boldsymbol{0}) + f^{(j)}(\boldsymbol{0}, \boldsymbol{x}).$$

This applies for all bit places, so the vector-valued function can be similarly decomposed

$$\boldsymbol{f}(\boldsymbol{s}, \boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{s}, \boldsymbol{0}) + \boldsymbol{f}(\boldsymbol{0}, \boldsymbol{x}).$$

Though the same feedback polynomial is used for both the current state and the shifted in data, the fact that the current state is fully present initially and the data is shifted in a bit at a time makes the functions for the initial state and the input distinct.

The effect of an LFSR on each parameter is linear, that is functions $\boldsymbol{f}(\boldsymbol{s}, \boldsymbol{0})$ and $\boldsymbol{f}(\boldsymbol{0}, \boldsymbol{x})$ are linear in their respective variables, so we can express their respective actions using matrix multiplication. If we consider the values $\boldsymbol{s}$ and $\boldsymbol{x}$ as column vectors, then we can represent the function as a sum (i.e., XOR) of distinct linear operators, namely:

$$\boldsymbol{f}(\boldsymbol{s}, \boldsymbol{x}) = \boldsymbol{A}\boldsymbol{s} + \boldsymbol{B}\boldsymbol{x}, \tag{1}$$

where $\boldsymbol{A}$ and $\boldsymbol{B}$ are both 64×64 binary matrices that encode the impact of the (characteristic-polynomial-dependent) LFSR feedback on the corresponding parameter. The matrix $\boldsymbol{B}$ is the matrix that loads the input vector (raw symbol) into the internal LFSR state under the assumption that the initial state was $\boldsymbol{0}$, and the matrix $\boldsymbol{A}$ is the matrix that performs 64 LFSR operations on a specific LFSR internal state.

Both $\boldsymbol{A}$ and $\boldsymbol{B}$ are invertible, and do not commute with each other.

[3] This is a direct consequence of the field and vector space axioms for the underlying field and vector space ($\mathbb{F}_2$ and $\mathbb{F}_2^{64}$, respectively). This behavior was noted in [BL 2009, §II.D], and this development is similar.

These matrices are easy to calculate by supplying the standard basis to each parameter separately. See Appendix A for code that generates and tests these matrices under a "little endian" convention, where the least significant bit (lsb) of the LFSR state corresponds to the first value in the column vector, and using the standard basis to produce the matrix. This code also performs stochastic testing that compares the results of this matrix-based representation with those of the LFSR implementation from JEnt v2.2.0 to provide evidence that these two methods of implementing this conditioning are equivalent.

The operation represented by the matrix $\boldsymbol{A}$ is 64 iterated LFSR operations. We can also calculate the corresponding single-round version by using a subset of the conditioning code that performs only a single LFSR step; we call the matrix representation of the action of this single-step LFSR conditioning $\boldsymbol{A}_{\text{single}}$.

Each iteration of the LFSR (i.e., multiplication by $\boldsymbol{A}_{\text{single}}$) is equivalent to multiplying the field element represented by the current LFSR state by $z + \langle p(z) \rangle$. The primitive nature of the characteristic polynomial $p(z)$ shows us that if we start at any non-zero state and iteratively apply the LFSR, then we cycle through the full multiplicative group $\mathbb{F}_{2^{64}}^{\times}$ in a fixed (feedback polynomial-dependent) pattern.

We verified the order of the underlying LFSR by noting that $\boldsymbol{A}_{\text{single}}^{2^{64}-1} = \boldsymbol{I}$, and verifying that this is not true for any proper divisor of $2^{64} - 1$. We also verified that

$$\boldsymbol{A} = \boldsymbol{A}_{\text{single}}^{64}.$$

Because $\gcd(2^{64} - 1, 64) = 1$, we see that it is also the case that $\boldsymbol{A}$ has order $2^{64} - 1$, which we similarly verified.

We can visualize these binary matrices as in the following diagrams (where a white value is 0 and a 1 entry is depicted in color). The matrix $\boldsymbol{A}_{\text{single}}$ is of the expected form; note that the feedback polynomial can be directly inferred from the first row with this basis ordering.

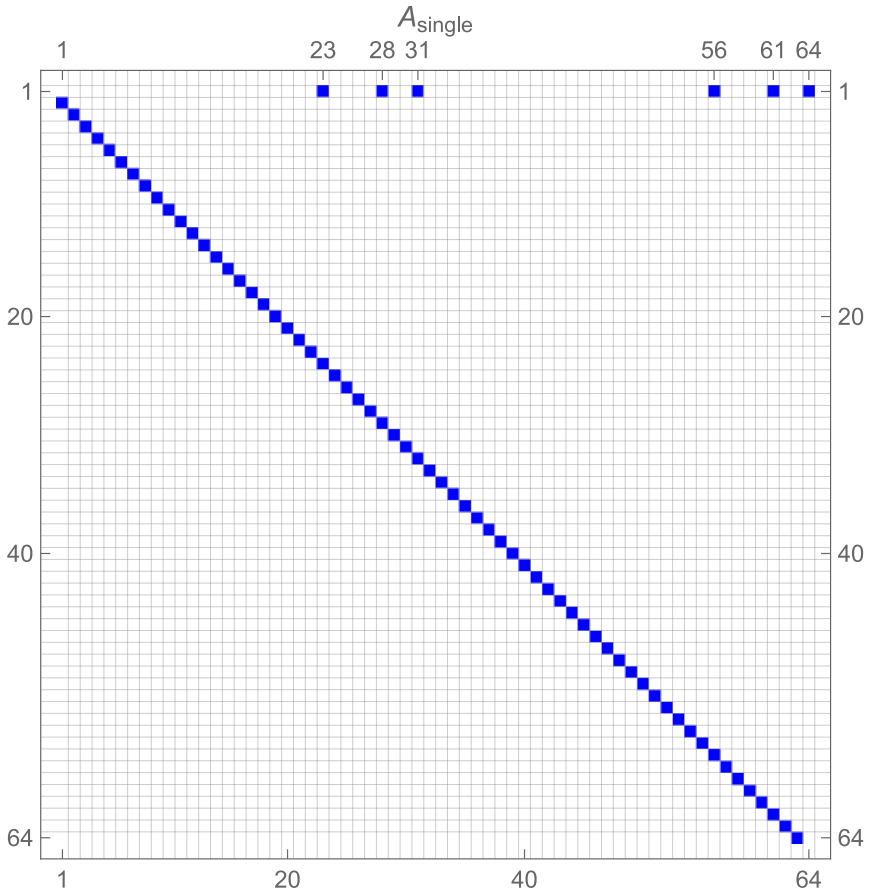

Figure 1. A_{single} in Matrix Form

For the other matrices it is not quite as easy to directly read out design information, but one can get some sense for the action of the repeated LFSR operation ($\boldsymbol{A}$) and the load-in transform ($\boldsymbol{B}$).

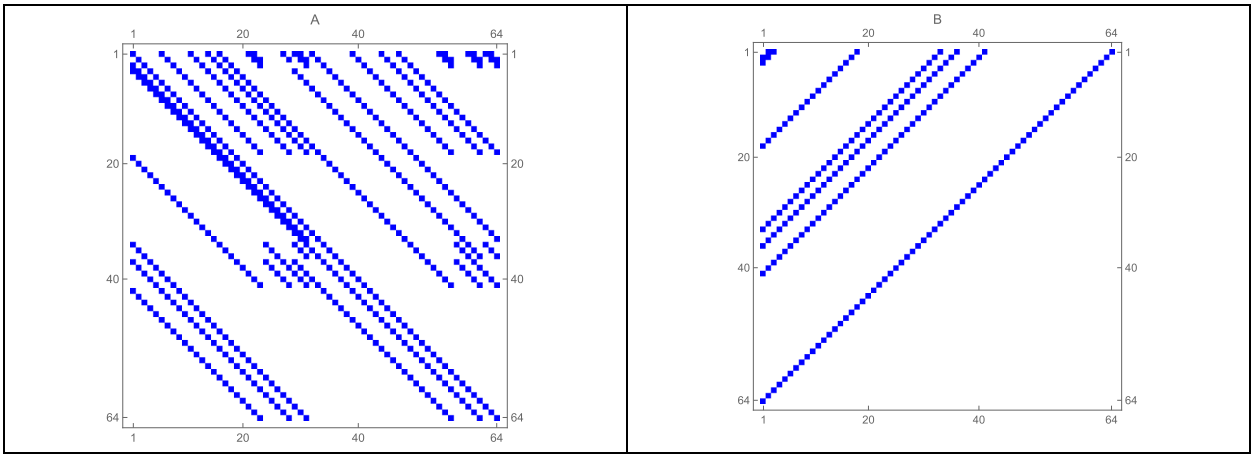

Figure 2. The Matrices A (Left) and B (Right)

2.2 The Conditioning in Closed Form

To simplify later notation, we write the per-input processing as

$$\boldsymbol{g}_j(\boldsymbol{x}) = \boldsymbol{A}^j \boldsymbol{B}\, \boldsymbol{x}. \tag{2}$$

We note that the LFSR state for the current step is wholly established by the LFSR state after the prior step completed, which is in turn dependent on the initial LFSR state and all prior raw symbols input. This gives the following recurrence relation:

$$\begin{aligned}\boldsymbol{o}_j(\boldsymbol{s}_j, \boldsymbol{x}_j) &= \boldsymbol{f}(\boldsymbol{s}_j, \boldsymbol{0}) + \boldsymbol{f}(\boldsymbol{0}, \boldsymbol{x}_j) \\ &= \boldsymbol{f}(\boldsymbol{o}_{j-1}(\boldsymbol{s}_{j-1}, \boldsymbol{x}_{j-1}), \boldsymbol{0}) + \boldsymbol{f}(\boldsymbol{0}, \boldsymbol{x}_j).\end{aligned}$$

Expanding the first few terms of this recurrence relation gives us

$$\begin{aligned}\boldsymbol{o}_1(\boldsymbol{s_1}, \boldsymbol{x_1}) &= \boldsymbol{f}(\boldsymbol{s}_1, \boldsymbol{0}) + \boldsymbol{f}(\boldsymbol{0}, \boldsymbol{x}_1) \\ &= \boldsymbol{A}\boldsymbol{s}_1 + \boldsymbol{B}\boldsymbol{x}_1 \\ &= \boldsymbol{A}\boldsymbol{s}_1 + \boldsymbol{g}_0(\boldsymbol{x}_1).\end{aligned}$$

As $\boldsymbol{s}_2 = \boldsymbol{o}_1(\boldsymbol{s}_1, \boldsymbol{x}_1)$, we have

$$\begin{aligned}\boldsymbol{o}_2(\boldsymbol{s}_2, \boldsymbol{x}_2) &= \boldsymbol{f}(\boldsymbol{o}_1(\boldsymbol{s}_1, \boldsymbol{x}_1), \boldsymbol{0}) + \boldsymbol{f}(\boldsymbol{0}, \boldsymbol{x}_2) \\ &= \boldsymbol{f}(\boldsymbol{A}\boldsymbol{s}_1 + \boldsymbol{B}\boldsymbol{x}_1, \boldsymbol{0}) + \boldsymbol{f}(\boldsymbol{0}, \boldsymbol{x}_2) \\ &= \boldsymbol{A}^2\boldsymbol{s}_1 + \boldsymbol{A}\boldsymbol{B}\boldsymbol{x}_1 + \boldsymbol{B}\boldsymbol{x}_2 \\ &= \boldsymbol{A}^2\boldsymbol{s}_1 + \boldsymbol{g}_1(\boldsymbol{x}_1) + \boldsymbol{g}_0(\boldsymbol{x}_2).\end{aligned}$$

Similarly, we have $\boldsymbol{s}_3 = \boldsymbol{o}_2(\boldsymbol{s}_2, \boldsymbol{x}_2)$, so

$$\begin{aligned}\boldsymbol{o}_3(\boldsymbol{s}_3, \boldsymbol{x}_3) &= \boldsymbol{f}(\boldsymbol{o}_2(\boldsymbol{s}_2, \boldsymbol{x}_2), \boldsymbol{0}) + \boldsymbol{f}(\boldsymbol{0}, \boldsymbol{x}_3) \\ &= \boldsymbol{f}(\boldsymbol{A}^2\boldsymbol{s}_1 + \boldsymbol{A}\boldsymbol{B}\boldsymbol{x}_1 + \boldsymbol{B}\boldsymbol{x}_2, \boldsymbol{0}) + \boldsymbol{f}(\boldsymbol{0}, \boldsymbol{x}_3) \\ &= \boldsymbol{A}^3\boldsymbol{s}_1 + \boldsymbol{A}^2\boldsymbol{B}\boldsymbol{x}_1 + \boldsymbol{A}\boldsymbol{B}\boldsymbol{x}_2 + \boldsymbol{B}\boldsymbol{x}_3 \\ &= \boldsymbol{A}^3\boldsymbol{s}_1 + \boldsymbol{g}_2(\boldsymbol{x}_1) + \boldsymbol{g}_1(\boldsymbol{x}_2) + \boldsymbol{g}_0(\boldsymbol{x}_3).\end{aligned}$$

From the evident pattern we can expand to the general case:

$$\boldsymbol{o}_j(\boldsymbol{s}_1, \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots \boldsymbol{x}_j) = \boldsymbol{A}^j\boldsymbol{s}_1 + \sum_{k=1}^{j} \boldsymbol{g}_{j-k}(\boldsymbol{x}_k) \tag{3}$$

The initial state of the LFSR is $\boldsymbol{0}$, so if we are tracking the evolution of the LFSR state from the beginning, the first summand has no impact (as $\boldsymbol{A}^j\boldsymbol{0} = \boldsymbol{0}$).

This general form suggests that we should be interested in the matrices of the form $\boldsymbol{A}^j\boldsymbol{B}$, as these completely define all processing on the inputs prior to the final XOR.

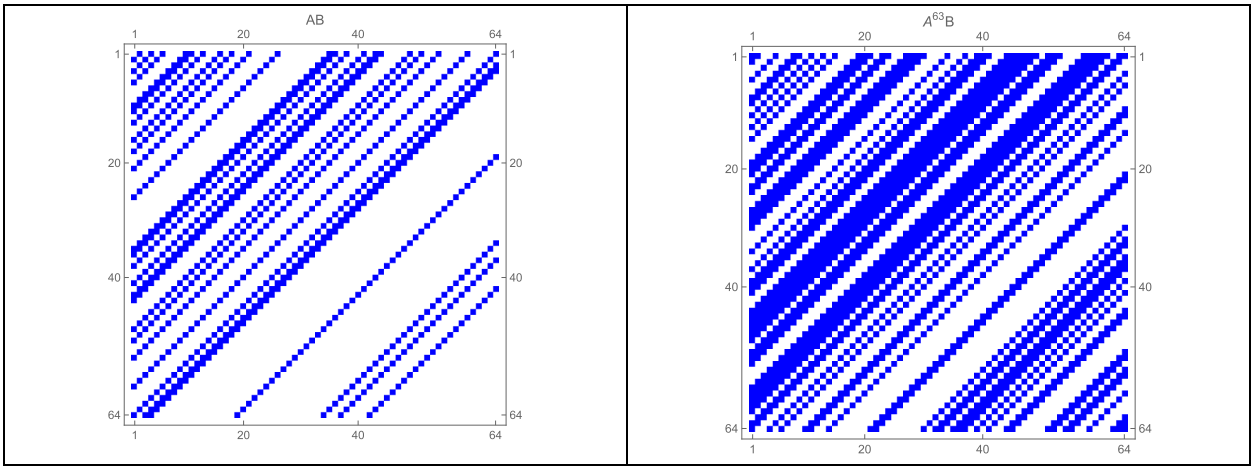

Figure 3. Some Example Matrices of the Form A^jB

Matrices of the form $\boldsymbol{A}^j\boldsymbol{B}$ are symmetric (that is $(\boldsymbol{A}^j\boldsymbol{B})^{\text{Transpose}} = \boldsymbol{A}^j\boldsymbol{B}$) for all non-negative integers j less than 12800 (thus this is true for all conditioning that occurs for `osr` values up to 200)[4].

3 The Impact of LFSR Processing on Min Entropy

3.1 Min Entropy of Each Summand

We have found that the LFSR conditioning function's output is the sum (XOR) of the iteratively LFSR-processed initial internal state and $64 \times \texttt{osr}$ LFSR processed raw symbols.

This presentation of the per-raw symbol processing highlights the importance of the choice of feedback polynomial: each output is essentially the sum (XOR) of increasingly LFSR-processed versions of the input values.

The function $\boldsymbol{g}_j(\boldsymbol{x})$ is bijective: in this processing, an input is loaded into the LFSR using the matrix $\boldsymbol{B}$, which is invertible (and can thus be reversed by multiplying by the matrix inverse $\boldsymbol{B}^{-1}$), and repeated LFSR processing is equivalent to iterative left multiplication by the matrix $\boldsymbol{A}$. The matrix $\boldsymbol{A}$ is invertible, so all these matrix multiplications can be inverted by left multiplying the result with $\boldsymbol{A}^{-1}$ the same number of times. Formally we have

$$\boldsymbol{g}_j^{-1}(\boldsymbol{y}) = \boldsymbol{B}^{-1}\boldsymbol{A}^{-j}\boldsymbol{y}.$$

A quick verification shows that $\boldsymbol{g}_j^{-1}(\boldsymbol{y})$ is the (two-sided) inverse function of $\boldsymbol{g}_j(\boldsymbol{x})$: [5]

$$\begin{aligned}\boldsymbol{g}_j^{-1}\left(\boldsymbol{g}_j(\boldsymbol{x})\right) &= \boldsymbol{B}^{-1}\boldsymbol{A}^{-j}(\boldsymbol{A}^j\boldsymbol{B}\boldsymbol{x})\\ &= \boldsymbol{B}^{-1}(\boldsymbol{A}^{-j}\boldsymbol{A}^j)\boldsymbol{B}\boldsymbol{x}\\ &= (\boldsymbol{B}^{-1}\boldsymbol{B})\boldsymbol{x}\\ &= \boldsymbol{x}\end{aligned} \qquad \begin{aligned}\boldsymbol{g}_j\left(\boldsymbol{g}_j^{-1}(\boldsymbol{y})\right) &= \boldsymbol{A}^j\boldsymbol{B}(\boldsymbol{B}^{-1}\boldsymbol{A}^{-j}\boldsymbol{y})\\ &= \boldsymbol{A}^j(\boldsymbol{B}\boldsymbol{B}^{-1})\boldsymbol{A}^{-j}\boldsymbol{y}\\ &= (\boldsymbol{A}^j\boldsymbol{A}^{-j})\boldsymbol{y}\\ &= \boldsymbol{y}\end{aligned}$$

As this per-summand input processing is bijective, the entropy present in each summand is necessarily unchanged by this portion of the conditioning. This statement should *not* be taken to mean that the overall conditioning is bijective; we will see that it is not for all parameter selections used within JEnt v2.2.0.

3.2 Statistics of the Evolution of Individual Summands

This presentation of the processing performed by the LFSR conditioning component makes it clear that, as more raw symbols are input to the conditioning component, the state resulting from each previously input raw symbol ($\boldsymbol{B}\boldsymbol{x}_i$) is additionally LFSR processed. The internal state after each input is the sum (XOR) of all these terms, so it is useful to consider the statistical properties of streams produced by each term of the sum $(\boldsymbol{A}^j\boldsymbol{B}\,\boldsymbol{x}_i)_{j=0}^{n-1}$, as this gives us some intuition of the impact of the LFSR processing as more raw symbols are integrated into the internal state.

LFSRs have a long history of use as random number generators (see [Golomb 2017, Chapter 3], [PTVF 2011], and [Knuth 1998] for general treatments), particularly in hardware applications. LFSR output has some known statistical flaws, and LFSR output streams are generally distinguishable from the output stream of a true random number generator. This manifests in a number of ways:

- LFSR outputs have a fixed algebraic relationship which can be detected through a linear complexity test.

[4] This seems likely to be true for all non-negative powers, but it is not clear how to prove this general statement.

[5] Evidencing an explicit inverse function is sufficient to show that this transform is *in some sense* a bijection, but to explicitly satisfy the requirement in [IG D.K, Resolution 3], note that the kernel of the linear operator $\boldsymbol{g}_j(\boldsymbol{x})$ is trivial so this map is injective and the dimensions of the co-domain and domain are finite and equal, which (with injectivity) tells us that the map is also surjective. As this map is both injective and surjective, it is bijective.

- When the entirety of the state is output, there is substantial serial correlation due to the per-invocation shift being equivalent to a "multiply by 2". This results in an obvious relationship between many adjacent outputs which is evident when adjacent outputs are plotted as x/y values. In this application a single conditioning step results in 64 LFSR operations, which completely cycles the internal state, so the output internal state essentially contains only those bits that would have been shifted out in a bit outputting LFSR.
- Output bits display better auto-correlation results than expected for a random bitstream.
- There can be no repeated output values until the entire multiplicative group has cycled through; this is detectable in large datasets.
- More generally, there are frequency domain anomalies in LFSR states, particularly a notable bias towards low frequencies.

Output distinguishability concerns are not directly relevant for this application, but we will later care if the data streams are distinguishable by the SP 800-90B tests. For each possible initial raw symbol, $\boldsymbol{x}_i$ (in some example raw data distribution), we iteratively produced the output $(\boldsymbol{A}^j \boldsymbol{B}\, \boldsymbol{x}_i)_{j=0}^{124999}$, resulting in 1 million bytes of output data per initial symbol. We then produced a bitwise assessment for this data sample (as done with conditioned outputs in [SP 800-90B §3.1.5.2]). Raw data distributions consisting of very few possible input values necessarily produce "bursty" assessment result histograms which prevent comparison with our reference distribution. As such, we characterize all possible initial values for a system consistent with a raw data (wider, Intel-TSC-style sub-) distribution with 256 possible values for this testing.

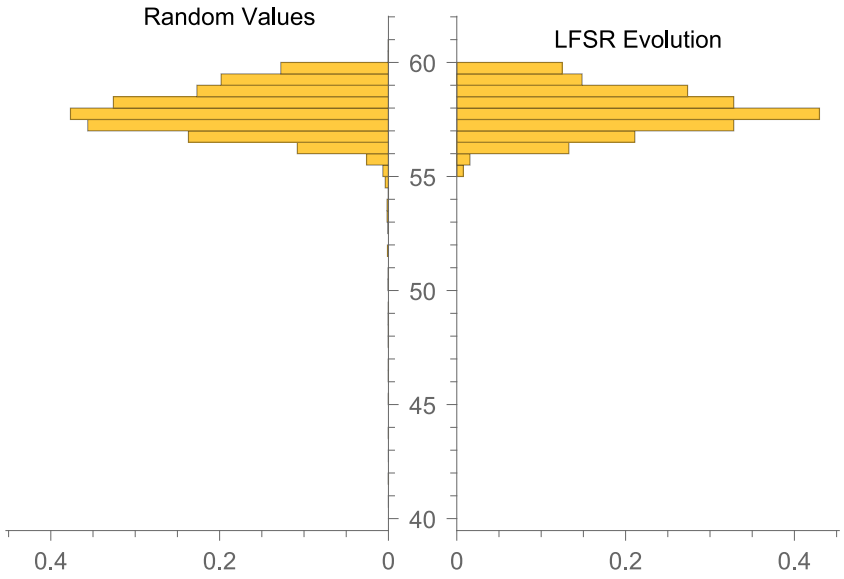

Figure 4. Comparison of Assessments for Random Data vs. LFSR Conditioned Output

The results are very similar to those coming from the testing of random data (see Section 4.2 for details). This suggests that the identified statistical LFSR issues don't significantly impact the results of this SP 800-90B assessment and support our expectation that this LFSR produces pseudorandom streams with fairly good statistical characteristics.

3.3 Heuristic Analysis of the Conditioned Output Min Entropy

If the initial state is known and a conditioned value were output after inputting only a single raw value, then all processing of that value would be bijective, there would be no entropy accumulation (so the output entropy would be the same as the input entropy), and the input could be re-constructed from the output. This is not the way that this conditioning function is used, as $64 \times$ osr raw symbols are input between each 64-bit output. As such, even though some components of this conditioning function are bijections, *the overall conditioning function is not bijective*.

To complete our understanding of this conditioning function, we need only consider the impact of the XOR of these terms.

We saw in Section 3.2 that the evolution of each term of the sum (fixing the input term and iteratively applying the LFSR processing) results in pseudorandom outputs and our testing in that section is consistent with this expectation. Substantial entropy cancelation would only occur in the final sum (XOR) step when there was a very specific algebraic relationship between the inputs. This noise source is not expected to output data with any particular algebraic relationship between the outputs so substantial reduction of the entropy (beyond that due to the final XOR of many 64-bit quantities) is not expected.[6] Similarly, the prior state (which is itself an XOR of many such values) is not expected to have any particular algebraic relationship with future inputs so we do not expect substantial cancelation between the LFSR-processed initial LFSR state and LFSR-processed inputs.

The starting internal LFSR state is either identically 0 (if we are tracking the conditioning from the start) or is some already output value which cannot be credited as contributing additional entropy. As such, this term does not contribute to the min entropy estimate.

As noted in Section 2.1, these LFSR conditioning steps are bijective so the min entropy of the $\boldsymbol{A}^j\boldsymbol{B}\boldsymbol{x}_k$ summand is equal to the min entropy of $\boldsymbol{x}_k$, though the distribution of the min entropy throughout the 64 bits is changed as the LFSR processing is iterated. The min entropy in $\boldsymbol{x}_k$ resides mainly in the low-order bits of the raw delta values. Iteratively applying the LFSR operation spreads the influence of each bit of $\boldsymbol{x}_k$ across the full block; each bit of the raw symbol $\boldsymbol{x}_k$ is expected to influence approximately half of the bits in the 64-bit $\boldsymbol{A}^j\boldsymbol{B}\boldsymbol{x}_k$ expression. The LFSR processing thus uniformizes entropy distribution across the full 64-bit vector. As such, we model the entropy content of each bit of the $\boldsymbol{A}^j\boldsymbol{B}\boldsymbol{x}_k$ summand as containing 1/64$^{\text{th}}$ of the min entropy of $\boldsymbol{x}_k$.

The goal of this simplifying assumption is that, by focusing on the (post-uniformized) LFSR-processed per-bit min entropy, we can avoid a detailed analysis which both tracks each bit of each raw symbol throughout the LFSR processing and explicitly establishes the min entropy distribution within the raw symbols.

In order to support the assumption that the bits of each of the inputs are well distributed, we can examine the count of the 1s of the rows of the matrix of $\boldsymbol{A}^j\boldsymbol{B}$; this tells us how many input bits contribute to a single output bit in the expression $\boldsymbol{A}^j\boldsymbol{B}\boldsymbol{x}_k$. In Figure 5, we show the minimum, median and maximum number of input bits that contribute to a single output bit from $\boldsymbol{A}^j\boldsymbol{B}\boldsymbol{x}_k$ (where the minimum / median / maximum is across the 64 output bits) for j in the range 0 to 100. The behavior does not significantly vary for larger values of j.

[6] An active attacker who knew when specific values were shifted in could cancel out these values in perpetuity by selecting an input so that $\boldsymbol{B}\boldsymbol{x}_j$ was equal to the sum of the relevant iteratively-LFSR-scrambled summands, but that requires that an attacker can both know prior inputs to the LFSR and completely specify future inputs; this style of attacker would have already completely compromised the entropy source, so this is not an interesting attack scenario.

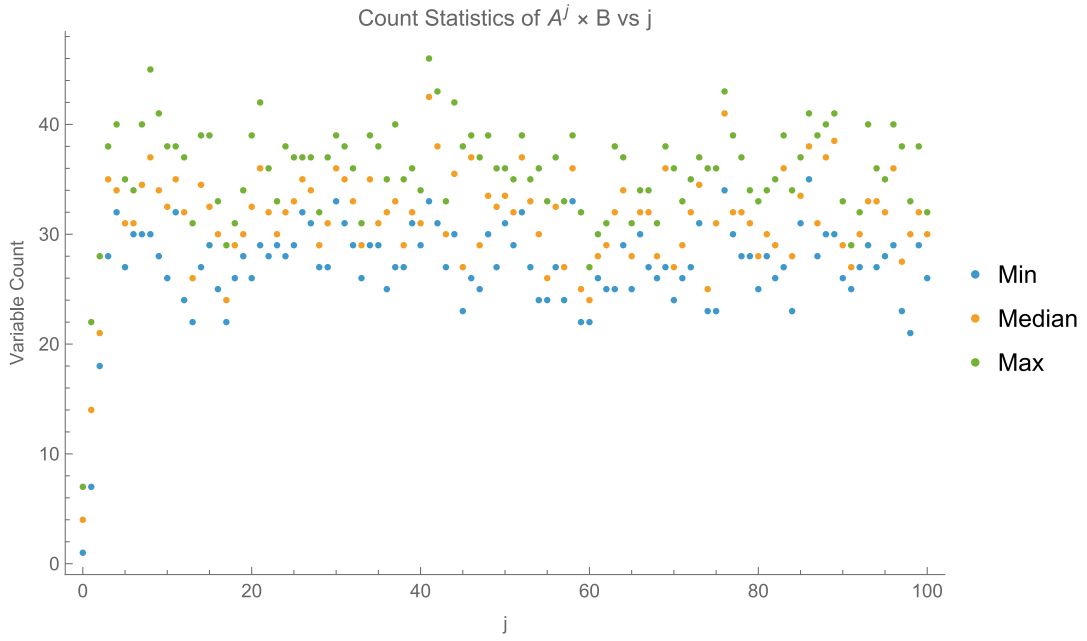

Figure 5. Variable Count Statistics

Because the matrix $\boldsymbol{A}^j\boldsymbol{B}$ is symmetric for all the depicted j values, this is the same graph as the statistics of the count of the number of 1s in each column, so we can also use this graph to understand how many output bits are influenced by each input bit.

We see that generally the last few symbols input into the conditioning function (roughly the last three, corresponding with j=0, j=1 and j=2) have notably lower representation in the output, but the raw symbols otherwise have a fairly uniform impact on all the bits of the output. The underrepresentation of the last three raw symbols is a case where the general uniformity assumption is not quite correct, but it is important to note that these inputs still have an impact. The information for the last few inputs is not well represented across all the bits of the final LFSR state, but those transforms are still invertible so the information is present (just not yet well distributed). This graph looks essentially the same for larger j values, so it is only the last few inputs that behave substantially differently.

For JEnt v2.2.0, the final 64-bit sum (XOR) that integrates the entropy contribution from each of the LFSR-processed raw inputs has exactly $64 \times \mathtt{osr}$ summands, each of the form $\boldsymbol{A}^j\boldsymbol{B}\boldsymbol{x}_k$. Each bit place of the output is established only by the corresponding bit place of each of the $\boldsymbol{A}^j\boldsymbol{B}\boldsymbol{x}_k$ summands.

Within a JEnt entropy analysis, $\mathtt{osr}$ is selected to allow the claim that $\boldsymbol{x}_k$ has at least $\frac{1}{\mathtt{osr}}$ bits of min entropy. Under our simplifying assumption, we further presume that each bit of $\boldsymbol{A}^j\boldsymbol{B}\boldsymbol{x}_k$ contains at least $\frac{1}{64\times\mathtt{osr}}$ bits of min entropy.

Given two independent bits with known min entropy m_1 and m_2, this corresponds to the most likely symbol probabilities $p_1 = 2^{-m_1}$ and $p_2 = 2^{-m_2}$. Because we are looking at the most likely symbol for each of these cases, it has probability $p_i \geq \frac{1}{2}$, so these probabilities can be written as $p_1 = \frac{1+\varepsilon_1}{2}$ and $p_2 = \frac{1+\varepsilon_2}{2}$ for some $0 \leq \varepsilon_i \leq 1$.

In each of the four possible cases (the most likely symbols for the two bits are both 0, both 1, or mixed 0 and 1), the probability of the most likely symbol of the XOR of these two bits is $\frac{1+\varepsilon_1\varepsilon_2}{2}$, so the resulting min entropy is a direct result of this probability. Similarly, if we XOR n bits with the same bias, then the probability of the most likely symbol is $\frac{1+\varepsilon^n}{2}$.

This gives us an estimate for the entropy of the LFSR output on a per-bit basis. We can then reconstruct the entropy of the full 64-bit symbol by multiplying this result by 64.

Putting this together, we find that the resulting conditioned min entropy assessment after inputting w raw symbols to the LFSR under this uniform assumption is then:

$$h_{\text{heuristic}}^{\text{uniform}}(\texttt{osr}, w) = 64\left(1 - \log_2\left(\left(2^{1-1/(64\times\texttt{osr})} - 1\right)^w + 1\right)\right). \tag{4}$$

The behavior of JEnt v2.2.0 is thus described by $h_{\text{heuristic}}^{\text{uniform}}(\texttt{osr}, 64 \times \texttt{osr})$. If we instead consider a more conservative approach where we do not account for the last three underrepresented symbols input into the conditioning function, then we get $h_{\text{heuristic}}^{\text{uniform}}(\texttt{osr}, 64 \times \texttt{osr} - 3)$.

This JEnt v2.2.0 behavior is graphically depicted in the relevant range in Figure 6.

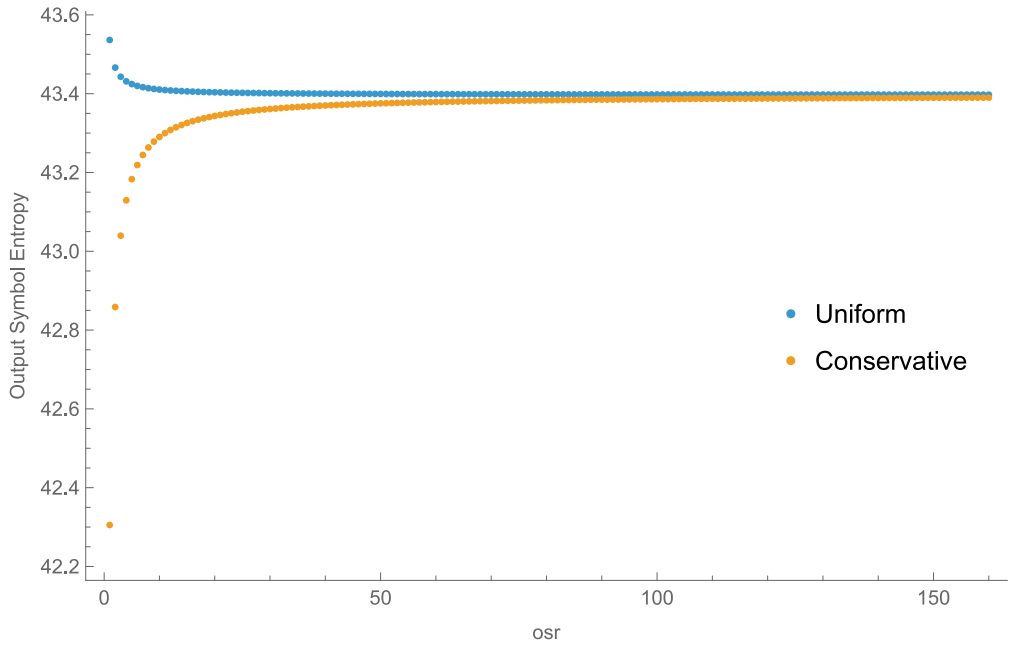

Figure 6. LFSR Output Min Entropy vs. OSR

In limit, we see that

$$\lim_{\texttt{osr}\to\infty} h_{\text{heuristic}}^{\text{uniform}}(\texttt{osr}, 64 \times \texttt{osr}) = \lim_{\texttt{osr}\to\infty} h_{\text{heuristic}}^{\text{uniform}}(\texttt{osr}, 64 \times \texttt{osr} - 3) = 64 \log_2\left(\frac{8}{5}\right) \approx 43.3966.$$

The osr=1 case of the conservative accounting approach provides the worst case in this analysis, which shows that for the JEnt v2.2.0 behavior, we can expect $h_{\text{heuristic}} \geq 42.3052$ bits of min entropy per 64-bit conditioned output block for any osr setting so long as this osr setting supports a claim of $h_{\text{submitter}} = \frac{1}{\texttt{osr}}$ bits of min entropy[7].

If we instead fix osr to some positive integer value and take the limit in w, we then find that

$$\lim_{w\to\infty} h_{\text{heuristic}}^{\text{uniform}}(\texttt{osr}, w) = 64,$$

so, given enough data, $h_{\text{heuristic}}$ can be made as close to 64 as desired.

[7] The result here is a truncation of the calculated value of 42.30525921443575, where truncation is used to provide a conservative estimate. Note that rounding this value to 6 significant digits yields 42.3053, which we sometimes use in this paper.

4 SP 800-90B Non-Vetted Conditioning Analysis

An assessment approach for non-vetted conditioning components is outlined in [SP 800-90B §3.1.5.2]; this is slightly updated by [IG D.K, Resolution 5], which further requires that the claimed h_{out} cannot be greater than the entropy output level justified with mathematical evidence, which we call $h_{\text{heuristic}}$. Together, we have:

$$h_{\text{out}} = \min(\text{Output\_Entropy}(n_{\text{in}}, n_{\text{out}}, nw, h_{\text{in}}), 0.999 \times n_{\text{out}}, h' \times n_{\text{out}}, h_{\text{heuristic}}).$$

The Output_Entropy(·) function described in [SP 800-90B §3.1.5.1.2] is a method for modeling entropy accumulation within a random map. For non-vetted conditioning (like the LFSR used here), this entropy may be further reduced based on the result of a bitstring-oriented statistical test of the output of the conditioning function (h', See [SP 800-90B §3.1.5.2] for details). As a practical matter, the $0.999 \times n_{\text{out}}$ is essentially never the minimum of these expressions, as we necessarily have $h' < 0.999$ for any reasonable size of conditioned sequential dataset.

4.1 The Output_Entropy(·) Function

The parameters used within the SP 800-90B Output_Entropy(·) function are summarized in Table 1.

Table 1. SP 800-90B Output_Entropy(·) Parameter Summary

Parameter	Value
Symbol Width (n)	64
n_{out}	64
nw ($\leq n_{\text{in}}$)	64
$n_{\text{in}}(w \times n)$	$w \times 64$
H	$1/\texttt{osr}$
$h_{\text{in}} = w \times H$	$\geq w/\texttt{osr}$

The only parameter that may require explanation is the narrowest width, *nw*. The value of this parameter is evident when viewed in terms of the matrix representation of the LFSR action: the internal state is a 64-bit vector, and all input data is integrated into this result, so *nw* is clearly not larger than 64 bits.

In order to show that the narrowest width is not less than 64, we note that if we select bit ℓ in the jth state

$$\boldsymbol{o}_j(\boldsymbol{s}_1, \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots \boldsymbol{x}_j) = \boldsymbol{A}^j \boldsymbol{s}_1 + \sum_{k=1}^{j} \boldsymbol{A}^{j-k} \boldsymbol{B} \boldsymbol{x}_k$$

we can determine exactly which bits in each of the j inputs (as k runs 1 to j) influence this bit by looking at the ℓth row of the matrix used to transform the kth input:

$$\boldsymbol{A}^{j-k} \boldsymbol{B}.$$

The matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ are invertible, so $\boldsymbol{A}^{j-k}\boldsymbol{B}$ is also invertible. An invertible matrix cannot have a row with all 0s (as this would cause the determinant to be 0) so this ℓth row cannot be identically 0 for any selection of k or j. This internal state is the output value, so every bit of this state is influenced by input and influences the output. As such, the narrowest width cannot be less than 64, and it cannot be greater than 64, so it is exactly equal to 64.

The numerical results of the Output_Entropy(·) function are clearly dependent on the values w (the number of symbols fed into the conditioner between outputs) and osr, but the general relationships are consistent across these parameter values. Figure 7 shows the relationship between the LFSR entropy accumulation described in this analysis and the Output_Entropy(·) function output (both for osr=1) as the number of symbols input (w) varies.

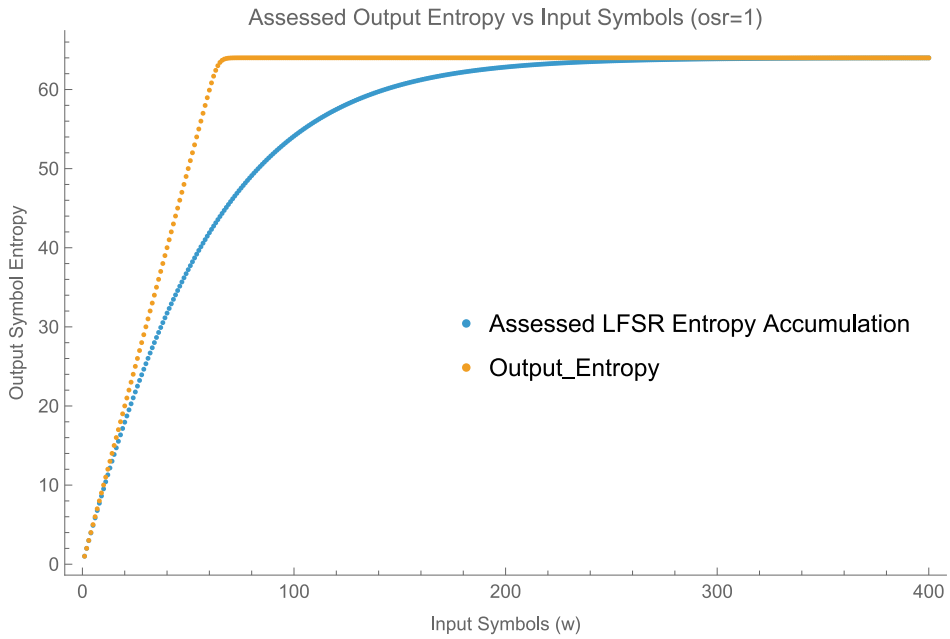

Figure 7. Output Min Entropy vs. Raw Symbols Input for osr=1

It is clear from this graph that under this analysis the LFSR accumulates entropy more slowly than modeled by the Output_Entropy(·) function.[8] This continues to be true for larger values of osr. As such, the minimum value that establishes h_{out} is not the Output_Entropy(·) result.

4.2 The Distribution of $h' \times n_{\text{out}}$ For Random Data

As seen in Section 3.2, this LFSR has reasonable statistical properties (even absent entropy being fed into it). The output of the XOR of many such pseudorandom strings will itself be pseudorandom, so the LFSR-conditioned output of JEnt is expected to be indistinguishable from random data using the SP 800-90B test suite. Similarly, there is a broad class of non-vetted conditioning components that also produce pseudorandom outputs irrespective of the quality of the data supplied to them. Most non-vetted conditioning components evaluated within the ESV process are of this class.

As part of the non-vetted conditioning procedure described in [SP 800-90B §3.1.5.2], the tester is required to statistically evaluate a conditioned sequential dataset output as a bitstring. The NIST ESV testing infrastructure presently only allows samples of 1 million bytes to be submitted for testing.[9] For the conditioned output test, this means that the sample is treated as a single bitstring of length 8 million bits.

When the NIST entropy assessment tool performs a bitstring assessment on a sample of 8 million bits of pseudorandom data, the statistical testing results are expected to conform to a specific min entropy estimate distribution. This min entropy estimate distribution conveys an effective "best case" for this step of the SP 800-90B assessment procedure, as conditioning components that produce detectably flawed (i.e., evidently non-pseudorandom) output will generally assess lower than those in this min entropy estimate distribution.

[8] A more detailed analysis where the min entropy distribution within the raw input symbol is established and the contribution of each input bit is separately credited may yield a higher $h_{\text{heuristic}}$, but there is no expectation that the Output_Entropy(·) function would be a reasonable approximation for this more detailed approach, as LFSRs are not random maps due to the linear relationships between the inputs and outputs.

[9] Testing larger pseudorandom data samples would tend to narrow the assessment distribution and shift this entire distribution higher.

To get a better sense of how the SP 800-90B bitstring min entropy estimator results are distributed for such data sets we generated 1 million data sets, each with 1 million random bytes[10] (8 million bits) and tested each using the SP 800-90B bitstring tests, just as is done with samples of non-vetted conditioned output within the ESV process. The result distribution for these random data SP 800-90B bitstring tests is presented in Figure 8.

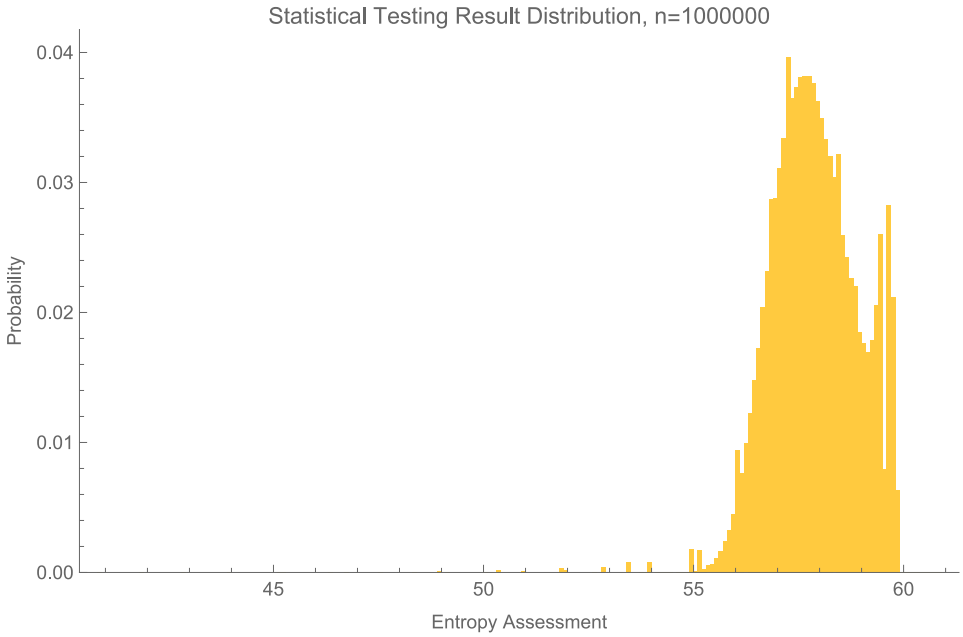

Figure 8. Distribution of the Bitstring Assessments of 8 Million Pseudorandom Bit Samples

We summarize the high-level statistical properties of this distribution in Table 2.

Table 2. Assessment Statistics

Percentile (%)	Value
≈ 0 (Min)	40.8494
50 (Median)	57.8352
99	59.7484
≈ 100 (Max)	60.8971

We can also determine a sequence of nested two-sided bounds that are expected to contain the result for this testing with some specific probability.

Table 3. Expected Result Bounds

Probability the Result is in the Bound (%)	Result Interval
95	[56.0599, 59.7484]
99	[54.9680, 59.8852]
99.9	[50.3425, 59.8852]

We can also examine one-sided results to get some impression of how unlikely some specific result is.

[10] Random generation was done within the [Theseus] package, using the XOR of the outputs of the well-regarded xoshiro256** PRNG and the RdRand instruction.

Table 4. Probability that Result is Greater Than or Equal to the Reference

Reference Value	p-Value (%)
59.6	5.6
59.7	2.8
59.75	0.71
59.9	0.025
60.4	0.0091
60.8	0.00050

This data shows us that for this data set size we have a low probability (< 1%) of getting a statistical test result greater than or equal to 59.75, and a very low probability (<0.025%) of getting a statistical test result greater than or equal to 59.9.

4.3 Alternate Selections for the w Parameter

JEnt v2.2.0 normally outputs a block of conditioned output after $w = 64 \times osr$ raw symbols have been sent to the conditioning component, resulting in the $h_{\text{heuristic}}$ value provided in Section 3.3. If this behavior were to be altered, how should w be set?

Section 3.3 shows that $h_{\text{heuristic}}$ can get asymptotically close to the maximal claim of 64 bits of min entropy per output by sending more data to the conditioner between outputs. Theoretically, sending more data into the conditioner always results in a higher $h_{\text{heuristic}}$ value, which is in some sense "better". When does it make sense to stop?

Two distinct (and conflicting) goals both seem reasonable. The first is making the largest possible claim for h_{out}, and the second is getting as much entropy as possible from the entropy source in some fixed amount of time.

4.3.1 Largest Possible h_{out} Assessment Goal

If we want to make the largest possible claim for h_{out}, then we need to control the chance that the $h_{\text{heuristic}}$ estimate is less than $h' \times n_{\text{out}}$ and thus reduces h_{out}. We have seen that for this conditioner the value that ultimately establishes h_{out} is either $h' \times n_{\text{out}}$ or $h_{\text{heuristic}}$. The LFSR output is expected to have excellent statistical properties given any reasonable amount of input, so the value we find for $h' \times n_{\text{out}}$ is essentially stochastic in nature and expected to be distributed as seen in Figure 8. We can use this expected result distribution to better understand how large the $h_{\text{heuristic}}$ claim should be to avoid the case where $h_{\text{heuristic}}$ reduces the final h_{out} assessment.

If we wanted to accumulate entropy until there was at least a 50% chance that the $h_{\text{heuristic}}$ value was higher than the statistical test result, then we would accumulate entropy to at least the median of the $h' \times n_{\text{out}}$ min entropy estimates. Similarly, if we wanted there to be a high (99%) chance that the $h_{\text{heuristic}}$ value was higher than the statistical test result, then we would accumulate entropy to at least the 99$^{\text{th}}$ percentile of the $h' \times n_{\text{out}}$ min entropy estimates. Finally, if we wanted there to be an essentially 100% chance that the $h_{\text{heuristic}}$ value was higher than the statistical test result, then we would accumulate entropy to at least the maximum observed $h' \times n_{\text{out}}$ min entropy estimate.

We can determine how many symbols should be input into the LFSR conditioner so that $h_{\text{heuristic}}$ is at least these reference entropy levels under this assessment approach. This information is summarized in Figure 9.

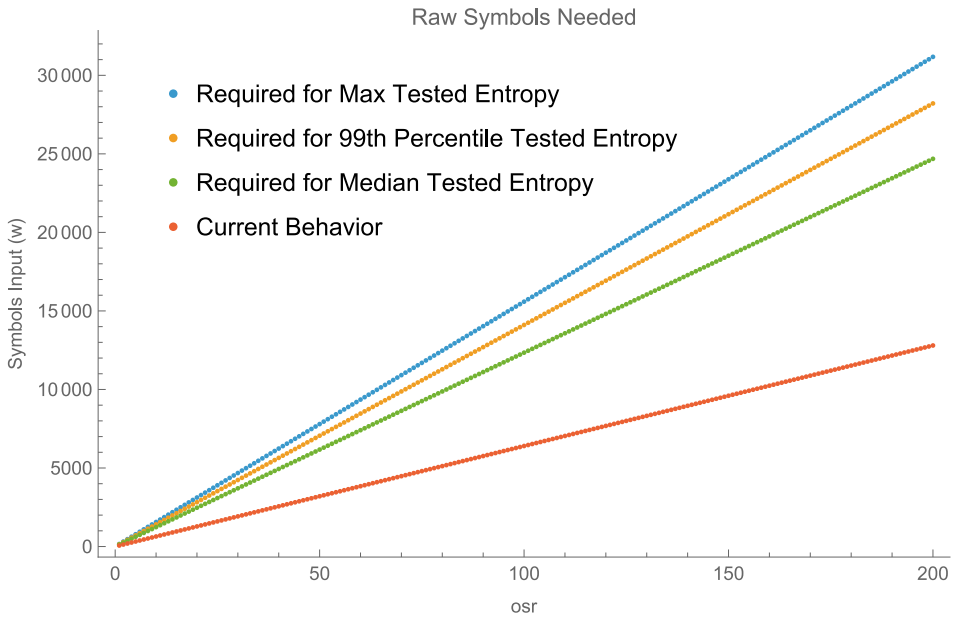

Figure 9. Number of Raw Symbols Required to Prevent Likely Assessment Reduction

The depicted relationship is essentially linear, so we can simply scale w by the scaling factor ($\mathcal{S}$) as needed to meet our goal. These scaling factors are presented in Table 5.

4.3.2 Largest Entropy Per Unit Time Goal

Each raw symbol takes approximately the same time to produce, so if we are interested in getting as much entropy as possible from the system in some fixed time, then we need to maximize the impact of the min entropy present in each raw symbol on the conditioned output assessment. To gauge the impact of each additional symbol on $h_{\text{heuristic}}$, we examine the marginal increase in $h_{\text{heuristic}}$ per additional raw symbol input. More explicitly, if we denote the value of $h_{\text{heuristic}}$ after inputting j raw symbols into the LFSR conditioner as $h^{(j)}_{\text{heuristic}}$, then the marginal increase in $h_{\text{heuristic}}$ for the jth symbol is

$$\Delta_j = h^{(j)}_{\text{heuristic}} - h^{(j-1)}_{\text{heuristic}},$$

that is the change in $h_{\text{heuristic}}$ after inputting the jth symbol. A graph showing the marginal increase in $h_{\text{heuristic}}$ for $\texttt{osr} = 1$ is depicted in Figure 10; higher values of `osr` yield less steep declines, but the general trend is the same for larger settings of `osr`.

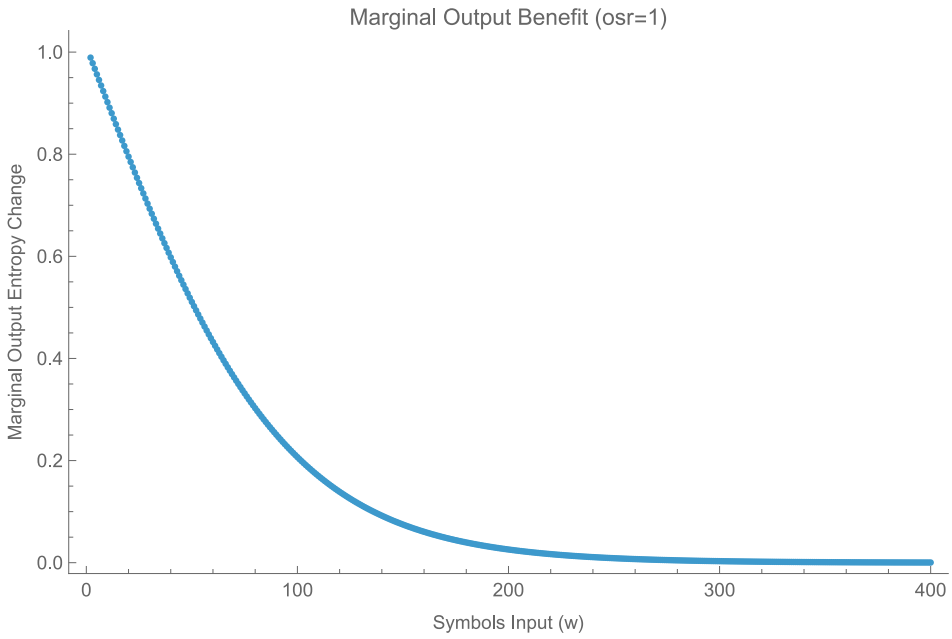

Figure 10. Marginal $h_{\text{heuristic}}$ Increase Per Raw Symbol Input (osr=1)

This implies that we get the highest output of entropy per unit time by keeping w as low as possible.

4.3.3 Selection of w Parameter Result Summary

To characterize overall efficiency, we are interested in getting some notion of how much benefit we get by increasing w. A reasonable raw efficiency (expressed as average assessed conditioned min entropy per raw symbol input) is

$$E = \frac{\overline{h_{\text{out}}}}{w} = \frac{\overline{h_{\text{out}}}}{\lceil \mathcal{S} \times 64 \rceil \times \texttt{osr}}.$$

This has a dependency on the selection of `osr`, so we instead normalize the efficiency with respect to the original behavior, namely

$$E_{\text{original}} = \frac{42.3053}{64 \times \texttt{osr}}.$$

The normalized efficiencies are then

$$\begin{aligned} E_{\text{normalized}} &= \frac{E}{E_{\text{original}}} \\ &= \frac{h_{\text{out}}}{\lceil \mathcal{S} \times 64 \rceil \times \texttt{osr}} \times \frac{64 \times \texttt{osr}}{42.3053} \\ &= \frac{h_{\text{out}}}{42.3053} \times \frac{64}{\lceil \mathcal{S} \times 64 \rceil}. \end{aligned}$$

This tells us how efficient each of these options is as compared to the efficiency of the original implementation. Smaller normalized efficiency values denote less efficiency. The modeled selections for w are summarized in Table 5.

Table 5. Strategies for the Selection of w

Approach	Chance of $h_{\text{heuristic}}$ Reducing h_{out}	Scaling Factor ($\mathcal{S}$)	$w = \lceil \mathcal{S} \times 64 \rceil \times osr$	Average h_{out}	Normalized Efficiency ($E_{\text{normalized}}$)
Original Behavior	$\approx 100\%$	1	$w = 64 \times \texttt{osr}$	42.3053	1
$h_{\text{heuristic}}$ is likely above $h' \times n_{\text{out}}$	$\leq 50\%$	1.96875	$w = 126 \times \texttt{osr}$	57.4610	0.689904
$h_{\text{heuristic}}$ is very likely above $h' \times n_{\text{out}}$	$\leq 1\%$	2.25000	$w = 144 \times \texttt{osr}$	57.8775	0.608041
$h_{\text{heuristic}}$ is always above $h' \times n_{\text{out}}$	$\approx 0\%$	2.48438	$w = 159 \times \texttt{osr}$	57.8780	0.550683

The "Average h_{out}" value in this table is established by the interaction between $h_{\text{heuristic}}$ for the original behavior, a lower bound for $h_{\text{heuristic}}$ for the described choice of w (under the conservative accounting of Section 3.3), and the $h' \times n_{\text{out}}$ distribution. The value $h' \times n_{\text{out}}$ is stochastic in nature and is expected to vary between validations. (See Figure 8 for the expected distribution of $h' \times n_{\text{out}}$.)

This analysis shows that inputting more than the largest noted number of raw symbols between LFSR outputs is not expected to provide any SP 800-90B assessment benefit. We also see that the original behavior is the most efficient of the analyzed selections for w.

5 Conclusion

We presented a theoretical basis for modeling JEnt v2.2.0's LFSR non-vetted conditioning component. This model is demonstrated to be equivalent to the LFSR-based conditioning component present in JEnt v2.2.0. Various characteristics of this LFSR conditioner were verified using this model. We then used this model to develop a heuristic analysis of the conditioned output min entropy, finding a defensible lower bound for an entropy claim of 42.3052 bits of min entropy per conditioned 64-bit output block (under the assumption that `osr` is set reasonably).

We analyzed the distribution of the SP 800-90B bitstring assessments of sample sets of 8 million bits which applies to any non-vetted conditioning function that produces pseudorandom output.

We also described a change to JEnt v2.2.0 that would allow for higher conditioned output block min entropy claims, though this increased claim comes at the cost of decreasing the entropy output from the entropy source per unit time.

The proposed analysis provides a sound basis for the entropy claims stated herein and is a significant step forward in providing a solid justification for the level of entropy output by JEnt v2.2.0.

5.1 Future Work

Future work in this area could include a more detailed analysis where the min entropy distribution within the raw symbol is established and the contribution of each input bit is separately credited.

6 References

- [BL 2009] Marco Bucci and Raimondo Luzzi. *Fully Digital Random Bit Generators for Cryptographic Applications*. IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 55, no. 3, pp. 861-875, April 2008. https://doi.org/10.1109/TCSI.2008.916446.
- [IG] CMVP. *Implementation Guidance for FIPS 140-3 and the Cryptographic Module Validation Program*. NIST, September 2, 2025. https://csrc.nist.gov/csrc/media/Projects/cryptographic-module-validation-program/documents/fips%20140-3/FIPS%20140-3%20IG.pdf.
- [Golomb 2017] Solomon W. Golomb. *Shift Register Sequences*. World Scientific, 2017.
- [Hill 2023] Joshua E Hill. *What to Expect When You're Expecting: (Part II: … to Assess the CPU Jitter Random Number Generator v2.2.0 Against SP 800-90B)*. Presentation Version 20230711. CMUF Entropy Working Group. https://www.untruth.org/~josh/sp80090b/JEnt%20v2.2.0%20Validation%20Notes%2020230711-2.pdf.
- [Theseus] Joshua E. Hill. *Theseus*. Version 20250626. https://github.com/KeyPair-Consulting/Theseus/archive/refs/tags/20260626.tar.gz.
- [Knuth 1998] Donald E. Knuth. *The Art of Computer Programming*. Vol. 2, 3rd ed. Addison-Wesley, 1998. pp 29-32.
- [JEnt Design] Stephan Müller. *CPU Time Jitter Based Non-Physical True Random Number Generator*. October 24, 2022. https://www.chronox.de/jent/CPU-Jitter-NPTRNG-v2.2.0.pdf.
- [JEnt v2.2.0] Stephan Müller. *Jitterentropy Library v2.2.0*. September 2019. https://www.chronox.de/jent/releases/historic/jitterentropy-library-2.2.0.tar.xz.
- [NIST SHALL] NIST CAVP. *90B-Shall-Statements*. NIST, August 9, 2021. https://csrc.nist.gov/CSRC/media/Projects/cryptographic-module-validation-program/documents/esv/90B%20Shall%20Statements.xlsx.
- [PTVF 2011] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Third Edition. Cambridge University Press, 2011. pp 380-386.
- [SP 800-90B] Meltem Sönmez Turan, Elaine Barker, John Kelsey, Kerry A. McKay, Mary L. Baish and Mike Boyle. *NIST Special Publication 800-90B, Recommendation for the Entropy Sources Used for Random Bit Generation*. NIST, January 2018. https://doi.org/10.6028/NIST.SP.800-90B.
- [Živković 1994] Miodrag Živković. *A Table of Primitive Binary Polynomials*. Mathematics of Computation. Vol. 62 no. 205 (1994), 385-386. https://poincare.matf.bg.ac.rs/~ezivkovm/publications/primpol1.pdf.

Appendix A LFSR Conditioning Matrix Generation and Verification

A.1 Introduction

The code included in this appendix can be used to generate the matrices $\boldsymbol{A}$, $\boldsymbol{A}_{\text{single}}$, and $\boldsymbol{B}$. This code also performs stochastic testing to provide evidence that the described implementation of the LFSR processing as linear operations is equivalent to the version implemented within JEnt v2.2.0.

There is also a program to model all the raw noise symbols in a range and then produce 1 million bytes of conditioned output per analyzed raw noise symbol by repeatedly LFSR processing the existing state.

A.2 Matrix Calculation and Testing

```
#include <stdio.h>
#include <stdbool.h>
#include <inttypes.h>
#include <time.h>
#include <stdlib.h>
#include <assert.h>

#define TESTROUNDS 1000

//Some conventions:
//Column vectors are stored as uint64_t values
//The standard basis is e_1 = 0x0000000000000001, e_2 = 0x0000000000000002, e_3 = 0x0000000000000004, etc.

// This is the full LFSR process where the input value is shifted into the internal state.
// It is derived from JEnt v2.2.0 jent_lfsr_time() function.
// The changes here are to isolate the LFSR processing from the other JEnt functionality
// and to harmonize the variable naming with the LFSR conditioning analysis.
uint64_t lfsr_process(uint64_t s, uint64_t x) {

        //Shifting in the input (x) low order to high order
        for (uint64_t i = 1; 64 >= i; i++) {
                //This operation does a (64-i) left bit shift;
                //this puts a shifted copy of x into tmp which
                //makes bit place (i-1) of x the msb of tmp.
                //(The loop left-shifts 63 then 62 then 61 ... and finally 0 bits)
                uint64_t tmp = x << (64 - i);
                //shift the 64-bit value tmp right by 63 bits,
                //retaining only the msb of tmp
                //(which is the (i-1)th bit of x)
                //and positioning it in the lsb of tmp.
                tmp = tmp >> 63;

                //Now tmp contains only the (i-1)th bit of x in its lsb.

                /*
                * Fibonacci LFSR with polynomial of
                *  z^64 + z^61 + z^56 + z^31 + z^28 + z^23 + 1
                * which is primitive according to
                *   http://poincare.matf.bg.ac.rs/~ezivkovm/publications/primpol1.pdf
                * (the shift values are the polynomial powers minus one
                * due to counting bits from 0 to 63). As the current
                * position is always the lsb, the polynomial only needs
                * to shift data in from the left without wrap.
                */
                tmp ^= ((s >> 63UL) & 1UL);
                tmp ^= ((s >> 60UL) & 1UL);
                tmp ^= ((s >> 55UL) & 1UL);
                tmp ^= ((s >> 30UL) & 1UL);
                tmp ^= ((s >> 27UL) & 1UL);
                tmp ^= ((s >> 22UL) & 1UL);
                s <<= 1UL;
                s ^= tmp;
        }
```

```
        return s;
}

// The base LFSR always performs 64 LFSR operations (because it is shifting in a 64-bit raw value).
// This function performs a single LFSR operation so that we can calculate A_single.
uint64_t lfsr_single_process(uint64_t s) {

        uint64_t tmp = 0;

        /*
        * Fibonacci LFSR with polynomial of
        *  z^64 + z^61 + z^56 + z^31 + z^28 + z^23 + 1
        *  which is primitive according to
        *   http://poincare.matf.bg.ac.rs/~ezivkovm/publications/primpol1.pdf
        * (the shift values are the polynomial powers minus one
        * due to counting bits from 0 to 63). As the current
        * position is always the lsb, the polynomial only needs
        * to shift data in from the left without wrap.
        */
        tmp ^= ((s >> 63UL) & 1UL);
        tmp ^= ((s >> 60UL) & 1UL);
        tmp ^= ((s >> 55UL) & 1UL);
        tmp ^= ((s >> 30UL) & 1UL);
        tmp ^= ((s >> 27UL) & 1UL);
        tmp ^= ((s >> 22UL) & 1UL);
        s <<= 1UL;
        s ^= tmp;

        return s;
}

//This is a quick function that demonstrates that we can take the full LFSR process (in two variables)
//and decompose it into two separate LFSR operations (where each is essentially in one variable).
bool testLFSRDecomp(uint64_t s, uint64_t x) {
        return lfsr_process(s, x) == (lfsr_process(s, 0) ^ lfsr_process(0, x));
}

//Assemble a simple 64-bit RNG for stochastic testing.
uint64_t rand64(void) {
        uint64_t a, b, c;
        /*RAND_MAX is 0x7F FF FF FF*/
        a = ((uint64_t)rand()) & 0xFFFFFFL;
        b = ((uint64_t)rand()) & 0xFFFFFFL;
        c = ((uint64_t)rand()) & 0xFFFFL;
        return (a << 40) | (b << 16) | c;
}

//Some basic matrix math functions.
//We're trying for clarity over speed here.
void printMatrix(uint8_t M[][64]) {
        printf("{");
        for(int i=0; i<64; i++) { //Row
                printf("{");
                for(int j=0; j<64; j++) { //Column
                        printf("%u", M[i][j]);
                        if(j<63) {
                                printf(", ");
                        } else {
                                printf("}");
                        }
                }
                if(i<63) {
                        printf(",\n");
                } else {
                        printf("};\n");
                }
        }
}

// Return the result of the matrix multiplication M v as a column matrix encoded as a uint64_t
// We can calculate each term of M v = y as y _i = Sum_{k=1}^64 M_{i,k} v_k
// Here, we are operating in a binary ring, so addition is XOR
```

```
// Remember that C indexes start at 0, whereas the standard way of writing matrix column/row
// indices is to start at 1.
uint64_t applyMatrix(uint8_t M[][64], uint64_t v) {
        uint64_t output = 0;

        for(uint8_t i=0; i<64; i++) {
                uint64_t curres = 0;
                //Calculate the result for row i+1 of the result
                //Recall that C is 0 indexed, whereas matrix indexes are 1-indexed.
                //(e.g., the i=0 case is for row 1 of the matrix)
                for(uint8_t k=0; k<64; k++) {
                        uint8_t vk = ((v&(1UL<<k))==0UL)?0:1;
                        curres = curres ^ (M[i][k] * vk);
                }
                assert((curres & 0xFFFFFFFFFFFFFFFE)==0UL);
                //We now have calculated y_i; it resides in the lsb of curres.
                output = output ^ (curres << i);
        }
        return output;
}

//This is a quick function that demonstrates that we can implement the full LFSR process
//using matrix multiplication using the matrices that we calculated.
bool testLFSRMatrix(uint64_t s, uint64_t x, uint8_t A[][64], uint8_t B[][64]) {
        return lfsr_process(s, x) ==  (applyMatrix(A, s) ^ applyMatrix(B, x));
}

int main() {
        bool decompTestPass = true;
        bool matrixTestPass = true;

        uint8_t Asingle[64][64];
        uint8_t A[64][64];
        uint8_t B[64][64];

        //Calculate the linear maps by applying the functions to the standard basis.
        printf("Linear maps:\n");
        //Asingle
        for(uint64_t j=0; j<64; j++) {
                //Get the result for column j+1;
                //Recall that C is 0 indexed, whereas matrix indexes are 1-indexed.
                //(e.g., the j=0 case is for column 1 of the matrix)
                uint64_t result = lfsr_single_process(1UL<<j);
                for(uint64_t i=0; i<64; i++) Asingle[i][j] = ((result&(1UL<<i))==0UL)?0:1;
        }

        //A
        for(uint64_t j=0; j<64; j++) {
                uint64_t result = lfsr_process(1UL<<j, 0UL);
                for(uint64_t i=0; i<64; i++) A[i][j] = ((result&(1UL<<i))==0UL)?0:1;
        }

        //B
        for(uint64_t j=0; j<64; j++) {
                uint64_t result = lfsr_process(0UL, 1UL<<j);
                for(uint64_t i=0; i<64; i++) B[i][j] = ((result&(1UL<<i))==0UL)?0:1;
        }

        //Now print the calculated matrices
        printf("Asingle = ");
        printMatrix(Asingle);
        printf("A = ");
        printMatrix(A);
        printf("B = ");
        printMatrix(B);

        //Perform some stochastic testing to show that these transforms seem to be working.
        srand(time(NULL));

        printf("Random LFSR Decomposition Testing: ");
        for(uint64_t j=0; j < TESTROUNDS; j++) {
                uint64_t curI = rand64();
```

```
                uint64_t curX = rand64();
                if(!testLFSRDecomp(curI, curX))
                        decompTestPass = false;
        }
        if(decompTestPass) {
                printf("Pass\n");
        } else {
                printf("Fail\n");
        }

        printf("Random Matrix LFSR Testing: ");
        for(uint64_t j=0; j < TESTROUNDS; j++) {
                uint64_t curI = rand64();
                uint64_t curX = rand64();
                if(!testLFSRMatrix(curI, curX, A, B))
                        matrixTestPass = false;
        }
        if(matrixTestPass) {
                printf("Pass\n");
        } else {
                printf("Fail\n");
        }

        return 0;
}
```

A.3 LFSR Driven Evolution of Single Inputs

```
#include <stdio.h>
#include <stdbool.h>
#include <inttypes.h>
#include <time.h>
#include <stdlib.h>
#include <assert.h>
#include <errno.h>

/* This program enumerates a range of expected raw values, and runs the LFSR forward for each,
 * saving the outputs to support statistical testing of them.
 */

#define OUTPUTBLOCKS UINT64_C(125000) // Each block is 8 bytes, so this represents 1 million bytes

//Narrow results (ARMish)
/*
#define TESTBITWIDTH UINT64_C(3)
#define TESTOFFSET UINT64_C(120)
*/

//Wide results (x86-with-TSCish)
#define TESTBITWIDTH UINT64_C(8)
#define TESTOFFSET UINT64_C(8645)

//Some conventions:
//Column vectors are stored as uint64_t values
//The standard basis is e_1 = 0x0000000000000001, e_2 = 0x0000000000000002, e_3 = 0x0000000000000004, etc.

// This is the full LFSR process where the input value is shifted into the internal state.
// It is derived from JEnt v2.2.0 jent_lfsr_time() function.
// The changes here are to isolate the LFSR processing from the other JEnt functionality
// and to harmonize the variable naming with the LFSR conditioning analysis.
uint64_t lfsr_process(uint64_t s, uint64_t x) {

        //Shifting in the input (x) low order to high order
        for (uint64_t i = 1; 64 >= i; i++) {
                //This operation does a (64-i) left bit shift;
                //this puts a shifted copy of x into tmp which
                //makes bit place (i-1) of x the msb of tmp.
                //(The loop left-shifts 63 then 62 then 61 ... and finally 0 bits)
                uint64_t tmp = x << (64 - i);
                //shift the 64-bit value tmp right by 63 bits,
                //retaining only the msb of tmp
```

```
                //(which is the (i-1)th bit of x)
                //and positioning it in the lsb of tmp.
                tmp = tmp >> 63;

                //Now tmp contains only the (i-1)th bit of x in its lsb.

                /*
                * Fibonacci LFSR with polynomial of
                *  z^64 + z^61 + z^56 + z^31 + z^28 + z^23 + 1
                * which is primitive according to
                *   http://poincare.matf.bg.ac.rs/~ezivkovm/publications/primpol1.pdf
                * (the shift values are the polynomial powers minus one
                * due to counting bits from 0 to 63). As the current
                * position is always the lsb, the polynomial only needs
                * to shift data in from the left without wrap.
                */
                tmp ^= ((s >> 63UL) & 1UL);
                tmp ^= ((s >> 60UL) & 1UL);
                tmp ^= ((s >> 55UL) & 1UL);
                tmp ^= ((s >> 30UL) & 1UL);
                tmp ^= ((s >> 27UL) & 1UL);
                tmp ^= ((s >> 22UL) & 1UL);
                s <<= 1UL;
                s ^= tmp;
        }

        return s;
}

//Some basic matrix math functions.
//We're trying for clarity over speed here.
// Return the result of the matrix multiplication M v as a column matrix encoded as a uint64_t
// We can calculate each term of M v = y as y _i = Sum_{k=1}^64 M_{i,k} v_k
// Here, we are operating in a binary ring, so addition is XOR
// Remember that C indexes start at 0, whereas the standard way of writing matrix column/row
// indices is to start at 1.
uint64_t applyMatrix(uint8_t M[][64], uint64_t v) {
        uint64_t output = 0;

        for(uint8_t i=0; i<64; i++) {
                uint64_t curres = 0;
                //Calculate the result for row i+1 of the result
                //Recall that C is 0 indexed, whereas matrix indexes are 1-indexed.
                //(e.g., the i=0 case is for row 1 of the matrix)
                for(uint8_t k=0; k<64; k++) {
                        uint8_t vk = ((v&(1UL<<k))==0UL)?0:1;
                        curres = curres ^ (M[i][k] * vk);
                }
                assert((curres & 0xFFFFFFFFFFFFFFFE)==0UL);
                //We now have calculated y_i; it resides in the lsb of curres.
                output = output ^ (curres << i);
        }
        return output;
}

int main() {
        uint8_t A[64][64];
        uint8_t B[64][64];

        //Calculate the linear maps by applying the functions to the standard basis.
        //A
        for(uint64_t j=0; j<64; j++) {
                uint64_t result = lfsr_process(1UL<<j, 0UL);
                for(uint64_t i=0; i<64; i++) A[i][j] = ((result&(1UL<<i))==0UL)?0:1;
        }

        //B
        for(uint64_t j=0; j<64; j++) {
                uint64_t result = lfsr_process(0UL, 1UL<<j);
                for(uint64_t i=0; i<64; i++) B[i][j] = ((result&(1UL<<i))==0UL)?0:1;
        }
```

```
	for (uint64_t x=TESTOFFSET; x < TESTOFFSET+(UINT64_C(1)<<TESTBITWIDTH); x++) {
		char curfilename[256];
		FILE *fp;
		uint64_t curvalue=applyMatrix(B, x);

		printf("Testing initial value %" PRIu64 "\n", x);

		snprintf(curfilename, sizeof(curfilename), "out-%" PRIu64 ".bin", x);

		if((fp=fopen(curfilename, "wb"))==NULL) {
			perror("Can't open file.");
			exit(-1);
		}

		//Write out the initial value B x (a.k.a. A^0 B x)
		if(fwrite(&curvalue, sizeof(curvalue), 1, fp)!=1) {
			perror("Can't write to file.");
			exit(-1);
		}

		for(uint64_t j = 1; j < OUTPUTBLOCKS; j++) {
			curvalue=applyMatrix(A,curvalue);
			//write out A^j B x
			if(fwrite(&curvalue, sizeof(curvalue), 1, fp)!=1) {
				perror("Can't write to file.");
				exit(-1);
			}
		}
		fclose(fp);
	}

	return 0;
}
```