Learning Permuted Congruential Sequences with Transformers

Tao Tao
Department of Physics, University of Maryland
`tao2021@umd.edu`

Maissam Barkeshli
Department of Physics, University of Maryland
Meta FAIR
Joint Quantum Institute, University of Maryland
`maissam@umd.edu`

Abstract

We use pseudo-random number generators (PRNGs) as a controlled benchmark to probe Transformers' ability to uncover hidden recurrence. Focusing on Permuted Congruential Generators (PCGs), which combine linear recurrences and bit-wise shift, XOR, rotation and truncation operations. We show that Transformers can successfully perform in-context prediction on unseen sequences from diverse PCG variants, in tasks that are beyond published classical attacks. Surprisingly, we find even when the output is truncated to a single bit, it can be reliably predicted by the model. We analyze embedding layers and uncover a novel clustering phenomenon: the model spontaneously groups the integer inputs into bitwise rotationally-invariant clusters, revealing how the model processes the input sequences.

1 Introduction

Recent works have investigated how Transformers learn modular arithmetic tasks, uncovering phenomena such as grokking, structured internal representations, and interpretable attention patterns (Power et al. [2022], Gromov [2023], Zhong et al. [2023], Nanda et al. [2023], Doshi et al. [2024], Charton and Kempe [2024]). We consider the task of next-element prediction in recurrence-based pseudo-random number sequences, with a focus on Permuted Congruential Generators (PCGs). These generators are of practical importance, as they serve as the default PRNG in NumPy. PCG generates outputs based on the recurrence:

$$s_i = (as_{i-1} + c) \bmod m, \qquad x_i = f(s_i), \tag{1}$$

where s_i is the hidden linear congruential state at step i, and x_i is the output. The parameters a, c, and m denote the multiplier, increment, and modulus, respectively, and are fixed for a given generator. The function f consists of a series of shifts, XORs, rotations and truncations to improve statistical quality and increase prediction difficulty. Transformers can learn linear congruential generators (LCGs) (Tao et al. [2025]), but PCGs are far tougher: they pass BigCrush at only 49-bit state ($m{=}2^{49}$) or less, whereas LCGs require 88 bits ($m{=}2^{88}$) (O'Neill [2014], L'Ecuyer and Simard [2007]).

2 Experimental Settings

Here we describe the generator variants, the datasets for training and evaluation, and the model architecture and training setups.

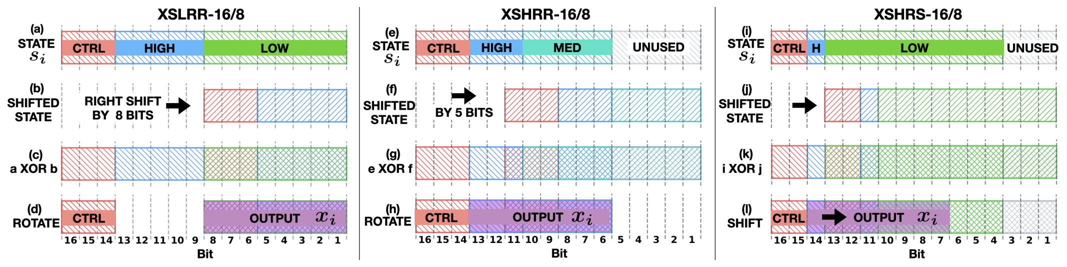


Figure 1: Depiction of PCG protocols at $m = 2^{16}$ with 8-bit output. **Left: XSLRR-16/8.** (a) State s_i. The top 3 bits are control bits. (b) s_i is right-shifted by 8 bits. (c) The shifted state is XORed with s_i. (d) The lower 8 bits are retained and rotated right by the value of the control bits to produce the output. **Middle: XSHRR-16/8.** (e) State s_i, with the top 3 bits as control; the lowest few bits are unused. (f) s_i is right-shifted by 5 bits. (g) The shifted state is XORed with s_i. (h) The upper 8 bits immediately following the control bits are retained and rotated right by the control bits to produce the output. **Right: XSHRS-16/8.** (i) State s_i, with the top 2 bits as control bits. (j) s_i is right-shifted by 3 bits. (k) The shifted state is XORed with s_i. (l) Starting from after the control bits, the output window is right-shifted by the control bits, producing the output.

2.1 PCG Variants

PCGs come in different varieties, depending on the precise set of shifts, XORs, rotations and truncations, encapsulated in the function f in Eq. 1. When a and c are chosen according to the Hull–Dobell theorem (Hull and Dobell [1962]), the state sequence s_i in Eq. 1 achieves the maximal period m. For power-of-two moduli, however, the bits of s_i exhibit position-dependent periodicities: the k-th least significant bit cycles with period 2^k, far shorter than the full state period m (Knuth [1997]). This makes the low-order bits especially weak, revealing structural patterns in the generator. PCG permutations mitigate this weakness by redistributing high-period structure across all bit positions using operations like XOR, shifts, and rotations. We consider the following variants:

- **TLCG** (Truncated LCG): Outputs only the high bits of the state. Part of the information of the internal state is hidden by the truncation.
- **XSLRR** (XORShift Low with Random Rotation): The state is right-shifted by half the bit length of m and XORed with the original state, improving the quality of the lower half bits. This lower half is retained and rotated by an amount determined by the control bits.
- **XSHRR** (XORShift High with Random Rotation): Applies a right-shift smaller than XSLRR, then XORs with the original state. The higher bits are retained and rotated by an amount determined by control bits.
- **XSHRS** (XORShift High with Random Shift): Applies a smaller right-shift than XSLRR and XSHRR, followed by an XOR with the original state. The output window begins immediately after the control bits and is shifted right by an offset determined by those bits.

The permutations are illustrated in Figure 1. Bits are labeled from most significant (left, bit 16) to least significant (right, bit 1). Top row shows the internal state s_i, where the k-th bit in s_i has period 2^k. The lower three rows show the function f. Bits are split into high and low, with the low bits enhanced by the higher bits during the permutation; cross-hatched overlaps mark areas enhanced by XOR. The final rotation and shift in the permutation are controlled by the top bits of the state. This ensures that all bits in the output inherit the full period of the highest bit. A full description of the initial-shift calculation and pseudo-code for each generator is given in Appendix A. In practice, PCGs typically adopt a power-of-two modulus $m = 2^{\text{state size}}$, ensuring that the control bits achieve maximal period. We denote generators as *generator type-state size/output size*; for example, XSLRR-16/8 refers to an XSLRR generator with a 16-bit state and an 8-bit output.

2.2 Datasets

We consider two settings:

- **Separate**: Training and test sets each contain sequences from the output of a single generator type, with no mixing between types.
- **Combined**: Training and test sets contain sequences from all four generator types.

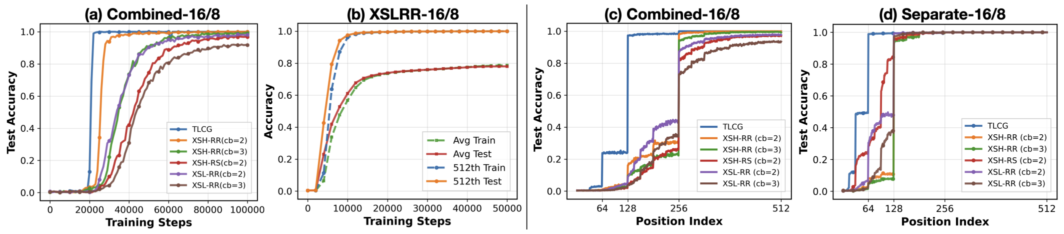


Figure 2: **(a)** Test accuracy at the 512th token during training on **combined** datasets of diverse PRNG variants. **(b)** Accuracy during training on XSLRR-16/8 dataset. "512th" refers to the model's prediction accuracy at the 512-th token. "Avg" denotes accuracy averaged across all token positions. **(c)** Final test accuracy by position index for **combined** training. **(d)** Final test accuracy when trained **separately** on each generator type, where all variants achieve near 100% accuracy with only 128 in-context elements.

In both cases, test sequences are generated from a, c values not seen during training. The **combined** setting is more challenging, as the model must simultaneously learn and distinguish multiple generation rules, effectively forming a multi-task problem across PRNG variants. The separate setting, by contrast, isolates each variant, simplifying analysis. For scaling studies on dataset size, model size, and modulus, we focus on the XSLRR variant. Both settings go beyond the assumptions of classical analytical attacks on PCGs, which typically assume knowledge of a, m, and permutation [Bouillaguet et al., 2020], and exploit the recurrence directly. For a given modulus m, we select a and c according to the Hull–Dobell Theorem to ensure maximal period.

2.3 Model and Training Setup

We train Transformers to autoregressively predict the next number in sequences generated by PRNGs. Given an input $x_0, x_1, \ldots, x_{L-1}$ of length L, the model outputs predictions $\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_L$. We use a GPT-style decoder-only Transformer (Radford et al. [2019]) with Rotary Positional Embeddings (RoPE) (Su et al. [2023]). Models use $n_{\text{layers}} = 4$ layers, $n_{\text{heads}} = 8$ attention heads, and an embedding dimension of $d_{\text{model}} = 1024$. Training details are provided in Appendix B.

3 Transformers can in-context learn PCGs

We find that Transformers achieve reliable in-context prediction across diverse PCG variants. As shown in Figure 2(a,c), a single model trained on the **combined** dataset reaches over 90% test accuracy after having seen 512 in-context elements of a test sequence, across all PCG variants. For all generators we fix the generator state to 16 bits and the output to 8 bits. For XSLRR and XSHRR we evaluate both 2- and 3-control-bit (cb) configurations, while for XSHRS the maximum feasible number of control bits is 2. Transformers can simultaneously learn multiple recurrence rules, whereas classical cracking algorithms are tailored to a single generator. When trained on **separate** generator datasets (Figure 2 b,d), models converge faster and achieve near-perfect accuracy after having seen 128 in-context elements of the test sequence. For both settings, test sets are generated from unseen a and c values, demonstrating generalization to unseen parameters. This is beyond current classical attacks, which require prior knowledge of both m and a.

Accuracy–position curves (Figure 2c,d) exhibit step-like improvements at powers of two. This is also observed in LCGs [Tao et al., 2025], where models exploit bit periodicity and show sudden accuracy gains once low-order bits complete their cycle in context. The persistence of this phenomenon in PCGs shows that, despite added permutations, significant residual bit-wise patterns appear at certain positions in the sequence that the model can exploit, although we have not studied their precise nature here. As shown in Appendix D, the model's attention patterns reflect this periodic structure.

4 What Limits Prediction Performance?

4.1 Effect of Truncations

To quantify the difficulty introduced by truncation, we study truncated LCGs where the low bits of the internal state are hidden and only the top k bits are retained as output. For $m = 2^{16}$, this yields

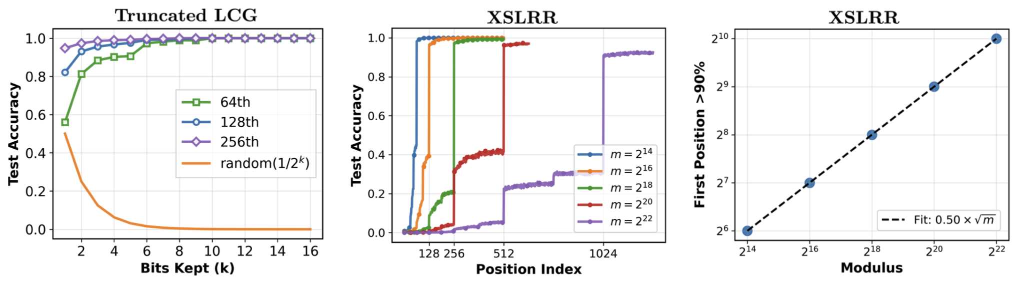


Figure 3: **Left:** Prediction accuracy at the 64th, 128th, and 256th sequence positions as a function of bits kept (k) in truncated LCGs with $m = 2^{16}$. Accuracy improves with larger k and longer context, remaining far above the random baseline $1/2^k$ even under severe truncation. **Middle:** For XSLRR, accuracy improves stepwise as more context is observed, with reliable predictions emerging once the context length reaches exactly $0.5\sqrt{m}$ elements. **Right:** Context length required to exceed 90% test accuracy scales as $\frac{1}{2}\sqrt{m}$ with modulus m.

a 2^{16-k}-to-1 mapping from states to outputs, so smaller k increases ambiguity. To examine this effect, we train separate models for each k (Figure 3, left). Despite the severe information loss, the models are surprisingly robust to truncation. Even with $k = 1$, the model attains 95% accuracy at the 256th element, far above the random-guessing baseline of $1/2^k$. At earlier positions (e.g., the 64th element), performance is lower under heavy truncation but improves quickly as k increases. These results indicate that Transformers can extract patterns even from heavily truncated outputs, with longer contexts compensating for reduced information.

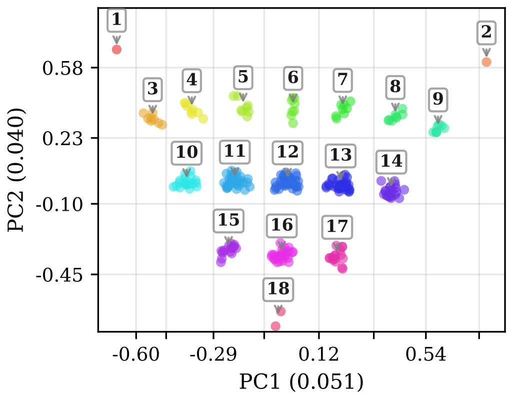
4.2 Effect of Generator Modulus:

Practical PCGs, such as the XSLRR-128/64 generator used as NumPy's default generator, operate at a scale far beyond the 16-bit state settings. To bridge this gap, we analyze how the modulus m affects prediction performance. We evaluate moduli ranging from $m = 2^{14}$ to $m = 2^{22}$ and observe a clear scaling law: the number of sequence elements required to reach at least 90% test accuracy grows as $\frac{1}{2}\sqrt{m}$. This relationship is shown in Figure 3 (middle, right) and indicates that context length becomes the primary bottleneck as the modulus increases. Compared to LCGs, where the requirement grows as $m^{0.25}$ (Tao et al. [2025]), PCGs demand substantially longer contexts, reflecting the information obscuration introduced by truncation and permutations. At large moduli, we use pretrained initialization combined with curriculum training.

5 Geometric and Algorithmic Structure in Model Representations

5.1 Token Embeddings

To understand how Transformers model PCG patterns, we analyze the token embedding layer of a model trained on XSLRR-16/8. We apply principal component analysis (PCA) to the embedding matrix and visualize the first two components in Figure 4. Representing tokens in binary form reveals that the learned embeddings encode a rotation-invariant structure, reflecting the symmetries of the generator. To formalize the structure, we use *zero-run notation* $Z(a_1, a_2, \ldots, a_k)$, where each a_i denotes the length of a contiguous run of `0`s between `1`s. The zero-run patterns and representative binary tokens for each cluster are shown in Figure 4(Right), with the complete listing provided in Table 1. We find that the first principal component (PC1) perfectly correlates the total number of zero bits N_0 in a token, while the second (PC2) perfectly correlates with the number of zero runs. Vertical bands in Figure 4 correspond to constant N_0 (e.g., clusters 6, 12, 16, and 18 all have $N_0 = 4$), while horizontal groupings reflect constant run counts (e.g., clusters 10–14 all contain two zero runs). This reveals that the model captures algorithmic structure in the token space: it organizes embeddings according to rotation-invariant features such as zero count and zero-run decomposition, implicitly mirroring the generator's permutation behavior.



Cluster	Zero-run pattern	Example
1	All 1s	11111111
2	All 0s	00000000
3	Z(1)	01111111
4	Z(2)	00111111
5	Z(3)	00011111
6	Z(4)	00001111
7	Z(5)	00000111
8	Z(6)	00000011
9	Z(7)	00000001
10	Z(1,1)	01011111
11	Z(2,1)	00101111
12	Z(3,1), Z(2,2)	00010111
13	Z(4,1), Z(3,2)	00001011
14	Z(5,1), Z(4,2), Z(3,3)	00000101
15	Z(1,1,1)	01010111
16	Z(2,1,1)	00101011
17	Z(3,1,1), Z(2,2,1)	00010101
18	Z(1,1,1,1)	01010101

Figure 4: PCA of token–embedding matrix for XSLRR-16/8 (left) and cluster summary (right). Tokens group by rotation-invariant zero-run structures; full table with all tokens provided in appendix.

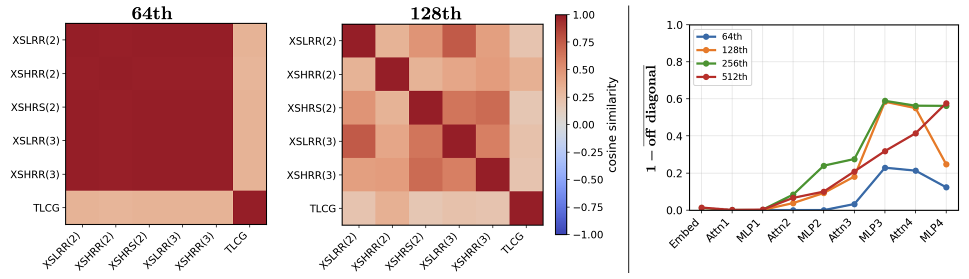


Figure 5: Cosine similarity of representations across different PRNG variants for a 4-layer Transformer trained on the **combined** dataset. Numbers in parentheses indicate the number of control bits. **Left**: At the 64th token position, third-layer MLP outputs already separate truncated LCGs from PCG variants, though PCG types remain highly overlapping. **Middle**: At the 128th token position, the same MLP outputs cleanly separate all PCG variants. Variants with the same permutation type but different control-bit counts are more similar to each other than to other types. **Right**: Generator separation across the network for selected token positions (64th, 128th, 256th, 512th) defined as $1-$ mean off-diagonal cosine similarity. Higher values indicate stronger generator separation.

5.2 Generator Separation

When trained on **combined** datasets, the model develops a permutation-agnostic grouping of tokens(Figure 6). This raises the question of how the model is able to predict different PRNG variants at test time. Despite receiving no explicit supervision about generator identity, the model's internal representations spontaneously distinguish PRNG variants. In a 4-layer model, this structure emerges most clearly in the MLP output of the third Transformer block: by the 64th token position, the model already distinguishes truncated LCGs from PCG variants, and by the 128th token, it cleanly differentiates between all PCG variants (see Figure 5, left and middle). To quantify this effect across layers, we plot the average off-diagonal cosine dissimilarity between generators at each position (Figure 5 right). Separation is weakest in the embeddings and first layer, rising sharply through the middle MLP and attention layers, suggesting that model first forms a shared representation of the underlying recurrence and then, in deeper layers, refines generator-specific distinctions.

References

Charles Bouillaguet, Florette Martinez, and Julia Sauvage. Practical seed-recovery for the pcg pseudo-random number generator. *IACR Transactions on Symmetric Cryptology*, 2020(3):175–196, Sep. 2020. doi: 10.13154/tosc.v2020.i3.175-196. URL `https://tosc.iacr.org/index.php/ToSC/article/view/8700`.

François Charton and Julia Kempe. Emergent properties with repeated examples, 2024. URL `https://arxiv.org/abs/2410.07041`.

Darshil Doshi, Aritra Das, Tianyu He, and Andrey Gromov. To grok or not to grok: Disentangling generalization and memorization on corrupted algorithmic datasets, 2024. URL `https://arxiv.org/abs/2310.13061`.

Andrey Gromov. Grokking modular arithmetic, 2023. URL `https://arxiv.org/abs/2301.02679`.

T. E. Hull and A. R. Dobell. Random number generators. *SIAM Review*, 4(3):230–254, 1962. doi: 10.1137/1004061. URL `https://doi.org/10.1137/1004061`.

Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997. ISBN 0201896842.

Pierre L'Ecuyer and Richard Simard. Testu01: A c library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, 33(4), August 2007. ISSN 0098-3500. doi: 10.1145/1268776.1268777. URL `https://doi.org/10.1145/1268776.1268777`.

Neel Nanda, Lawrence Chan, Tom Lieberum, Jess Smith, and Jacob Steinhardt. Progress measures for grokking via mechanistic interpretability, 2023. URL `https://arxiv.org/abs/2301.05217`.

Melissa E. O'Neill. Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation. Technical Report HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, September 2014.

Alethea Power, Yuri Burda, Harri Edwards, Igor Babuschkin, and Vedant Misra. Grokking: Generalization beyond overfitting on small algorithmic datasets, 2022. URL `https://arxiv.org/abs/2201.02177`.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI*, 2019. URL `https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf`. Accessed: 2024-11-15.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023. URL `https://arxiv.org/abs/2104.09864`.

Tao Tao, Darshil Doshi, Dayal Singh Kalra, Tianyu He, and Maissam Barkeshli. (how) can transformers predict pseudo-random numbers?, 2025. URL `https://arxiv.org/abs/2502.10390`.

Ziqian Zhong, Ziming Liu, Max Tegmark, and Jacob Andreas. The clock and the pizza: Two stories in mechanistic explanation of neural networks, 2023. URL `https://arxiv.org/abs/2306.17844`.

A Permuted Congruential Generators

In this section we review the background of LCGs, truncated LCGs and PCGs.

A.1 Hull–Dobell Theorem.

For an LCG

$$s_i = as_{i-1} + c \pmod{m}, \tag{2}$$

the sequence $\{s_i\}$ has full period m if and only if the following three conditions hold:

1. c and m are relatively prime,
2. $a - 1$ is divisible by all prime factors of m,
3. if m is divisible by 4, then $a - 1$ is also divisible by 4.

A.2 Period of Low-Order Bits in LCGs

Consider the LCG

$$x_{t+1} = (ax_t + c) \bmod m, \tag{3}$$

with $m = 2^K$, c coprime to m, and $a - 1$ divisible by 4, so that $\{x_t\}$ has full period m by the Hull–Dobell theorem. Let

$$z_{t,k} = x_t \bmod 2^k \tag{4}$$

denote the lowest k bits of x_t. Then

$$z_{t+1,k} = (az_{t,k} + c) \bmod 2^k, \tag{5}$$

so $\{z_{t,k}\}$ itself is an LCG with modulus 2^k. Since c is coprime to 2^k and $a - 1$ is divisible by 4, this reduced generator achieves full period 2^k. Thus, the k-th lowest bit of an LCG with power-of-two modulus cycles with period exactly 2^k, much shorter than the full state period m.

A.3 PRNG Variants.

We consider three widely used PCG permutations, each defined for a $2n$-bit state with cb control bits and an n-bit output. The internal state evolves as:

$$s_i = as_{i-1} + c \pmod{m}, \quad m = 2^{2n}. \tag{6}$$

- **XSLRR (Xorshift Low, Random Rotation).** First apply a right shift of n bits and XOR with the original state, folding the high and low halves together. The low n bits of the result are then retained and rotated right by the control value to produce the output. Formally:

$$\text{control bits value: } v = s_i \gg (2n - cb), \tag{7}$$
$$\text{state XOR shifted state: } s_i' = s_i \oplus (s_i \gg n), \tag{8}$$
$$n\text{-bit output: } x_i = \text{rot}_v(s_i' \bmod 2^n)\,, \tag{9}$$

  where $s_i \gg n$ denotes right shift s_i by n bits and rot_v denotes an n-bit right rotation by v.

- **XSH-RR (Xorshift High, Random Rotation).** First apply a right shift by $\lfloor (n + cb)/2 \rfloor$ bits and XOR the result with the original state. The n bits immediately following the control bits are then retained and rotated right by the control value to produce the output. Formally:

$$\text{control bits value: } v = s_i \gg (2n - cb), \tag{10}$$
$$\text{state XOR shifted state: } s_i' = s_i \oplus (s_i \gg \lfloor (n + cb)/2 \rfloor), \tag{11}$$
$$n\text{-bit output: } x_i = \text{rot}_v((s_i' \gg (n - cb)) \bmod 2^n) \tag{12}$$

- **XSH-RS (Xorshift High, Random Shift).** First apply a right shift by $(n - cb - 2^{cb} + 1)$ bits and XOR the result with the original state. The n-bit output window begins immediately after the control bits, but its starting position is shifted further right by the control value v, selecting a different n-bit segment of the state. Formally:

$$\text{control bits value: } v = s_i \gg (2n - cb), \tag{13}$$
$$\text{state XOR shifted state: } s_i' = s_i \oplus (s_i \gg (n - cb - 2^{cb} + 1)), \tag{14}$$
$$n\text{-bit output: } x_i = (s_i' \gg (n + v)) \bmod 2^n. \tag{15}$$

We also consider truncated LCGs, where the output is formed by retaining only the top k bits of the internal state s_i, hiding the lower-order bits. This preserves the recurrence structure and full period of the LCG while exposing only partial information about the state. Formally:

$$x_i = s_i \gg (2n - k) \bmod 2^k, \tag{16}$$

B Training Details

Combined Dataset (Figure 2a,c). For each generator type, we select n_a=n_c=1024 training multipliers a and increments c using the Hull–Dobell theorem. One sequence per (a, c) pair is generated with its initial state x_0 sampled randomly using NumPy's RNG. All sequences from all generator types are merged into a single dataset and reshuffled at the start of each epoch to randomize the sampling order.

We train a Transformer with depth $n_{\text{layers}} = 4$, $n_{\text{heads}} = 8$ attention heads, and d_{model}=1024 for 100k steps with batch size 512 (about 8 epochs). The learning rate is 0.0001 with weight decay 1.0, using 5000 warm up steps (linear) followed by cosine decay. Training is performed on two NVIDIA A100 GPUs and takes roughly 8 hours.

Separate Datasets (Figure 2b,d). We follow the same procedure for selecting a and c but train a separate model on each generator type individually. Each model is trained for 50k steps with batch size 512 (roughly 4 hours on two A100 GPUs). We perform a grid search over learning rate and weight decay for each generator to ensure fair comparison across configurations.

Truncated LCG (Figure 3 left). We evaluate truncated LCGs with $m = 2^{16}$, varying the number of retained bits k from 1 to 16, which determines the effective output range. Each integer is tokenized into two base-256 digits, except for special cases where a smaller base performed better: for k=7 we use base-128 with one digit, and for k=9 we use base-64 because training with base-256 consistently yielded worse performance. This tokenization dramatically reduces the vocabulary size (e.g., k=16 would otherwise require 65,536 symbols) and empirically improves convergence. Because each number is split into two tokens, the context length doubles, and a prediction is counted correct only when both digits are predicted correctly. For each configuration, we train for 50k steps with batch size 512, taking roughly four hours on two NVIDIA H100 GPUs for two-digit experiments and about two hours for one-digit experiments.

C Token Embeddings

C.1 XSLRR token clusters

Table 1 presents the detailed token clusters for XSLRR-16/8, listing each cluster's canonical binary pattern, its rotationally equivalent variants, and the corresponding tokens assigned to each group.

C.2 Combined token

Figure 6 shows the token embeddings of a model trained on combined datasets, projected onto the first two principal components. Unlike models trained on XSLRR, this model develops a more general, permutation-agnostic grouping of tokens. The embeddings form two broad bands corresponding to even and odd tokens. Along PC2, tokens are roughly ordered from top to bottom by increasing numbers of 1-bits.

D Attention Pattern

To analyze how attention spans evolve across the network, Figure 7 shows the distribution of token distances for the top-8 most-attended keys per query, averaged over all positions and heads in each layer (Distances of $q - k = 0$ are excluded because self-attention peaks there and would dominate the scale). In the first layer, attention is dominated by long-range periodic connections, with strong peaks

Cluster	Pattern	Rotational Equivalent	Tokens
1	Z() all bits are 1	11111111	255
2	Z(*) all bits are 0	00000000	0
3	Z(1) only 1 bit is 0	01111111	127, 191, 223, 239, 247, 251, 253, 254
4	Z(2) 2 consecutive 0	00111111	63, 126, 159, 207, 231, 243, 249, 252
5	Z(3)	00011111	31, 62, 124, 143, 199, 227, 241, 248
6	Z(4)	00001111	15, 30, 60, 120, 135, 195, 225, 240
7	Z(5)	00000111	7, 14, 28, 56, 112, 131, 193, 224
8	Z(6)	00000011	3, 6, 12, 24, 48, 96, 129, 192
9	Z(7)	00000001	1, 2, 4, 8, 16, 32, 64, 128
10	Z(1,1) 2 separated 0	01011111	95, 125, 175, 190, 215, 235, 245, 250
		01101111	111, 123, 183, 189, 219, 222, 237, 246
		01110111	119, 187, 221, 238
11	Z(2,1) 2 consecutive 0 and 1 separated 0	00101111	47, 94, 121, 151, 188, 203, 229, 242
		00110111	55, 110, 115, 155, 185, 205, 220, 230
		00111011	59, 103, 118, 157, 179, 206, 217, 236
		00111101	61, 79, 122, 158, 167, 211, 233, 244
12	Z(3,1) or Z(2,2)	00010111	23, 46, 92, 113, 139, 184, 197, 226
		00011011	27, 54, 99, 108, 141, 177, 198, 216
		00011101	29, 58, 71, 116, 142, 163, 209, 232
		00100111	39, 57, 78, 114, 147, 156, 201, 228
		00110011	51, 102, 153, 204
13	Z(4,1) or Z(3,2)	00001011	11, 22, 44, 88, 97, 133, 176, 194
		00001101	13, 26, 52, 67, 104, 134, 161, 208
		00010011	19, 38, 49, 76, 98, 137, 152, 196
		00011001	25, 35, 50, 70, 100, 140, 145, 200
14	Z(5,1) or Z(4,2) or Z(3,3)	00000101	5, 10, 20, 40, 65, 80, 130, 160
		00001001	9, 18, 33, 36, 66, 72, 132, 144
		00010001	17, 34, 68, 136
15	Z(1,1,1)	01010111	87, 93, 117, 171, 174, 186, 213, 234
		01011011	91, 107, 109, 173, 181, 182, 214, 218
16	Z(2,1,1)	00101011	43, 86, 89, 101, 149, 172, 178, 202
		00101101	45, 75, 90, 105, 150, 165, 180, 210
		00110101	53, 77, 83, 106, 154, 166, 169, 212
17	Z(3,1,1) or Z(2,2,1)	00010101	21, 42, 69, 81, 84, 138, 162, 168
		00100101	37, 41, 73, 74, 82, 146, 148, 164
18	Z(1,1,1,1)	01010101	85, 170

Table 1: XSLRR-16/8 Model Token Clusters with Rotation-Based Structures

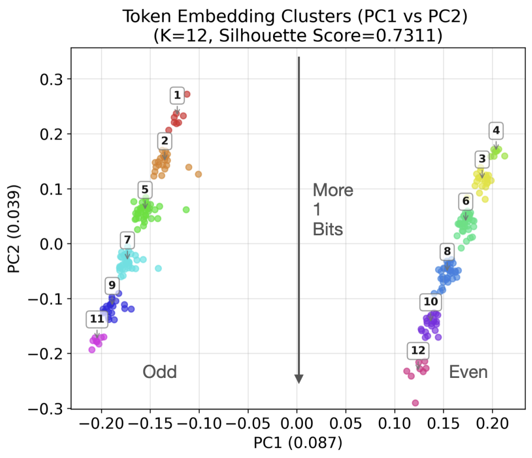
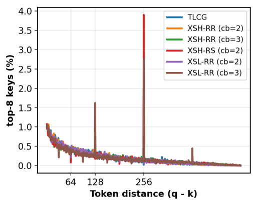
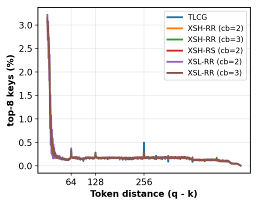


Figure 6: When trained on combined datasets, the model develops a more general grouping of tokens. The visualization shows token embeddings projected onto the first two principal components.

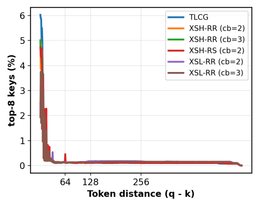
at powers of two (64, 128, 256), revealing that the model has discovered the underlying bit-periodicity of the generators. By the later layers, attention shifts toward shorter token distances, indicating that prediction increasingly relies on local context once the global recurrence has been inferred.



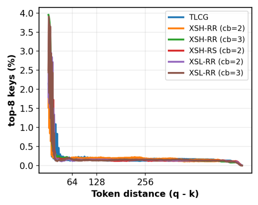
(a) Layer 1



(b) Layer 2



(c) Layer 3



(d) Layer 4

Figure 7: Token-distance distribution of the top-8 attended keys at each Transformer layer for a model trained on the combined dataset.