Adapting Cellular Automata for Symmetric Encryption

Steven Chiacchira

Abstract

A *Cellular Automata* (CA) is a discrete-time, deterministic process which uses a set of simple rules to simulate emergent complexities in dynamic systems. These constructs, despite their simplicity in construction, are well known to yield chaotic and diffuse results, making them intriguing as an entropy source for encryption algorithms. We seek to build a proof of concept algorithm which uses CAs for this purpose.

1 Prerequisites

1.1 Cellular Automata

We shall focus on cellular automata in the discrete 2D plane (a grid of bits), as these are well studied and allow for efficient spatial diffusion of information. CAs of this type take as input a certain grid state G_i and output another grid state G_{i+1}, where each cell contains either a 0 or 1. We also call this process *evolution.* The most famous example of such a cellular automata is John Conway's Game of Life, shown in Figure 1, which defines two simple rules for obtaining the next grid state:

1. If a cell containing a "1" has 2-3 (inclusive) "1" neighbors, it stays a "1". Otherwise it becomes a "0"
2. If a cell containing a "0" has exactly three "1" neighbors, it becomes a "1". Otherwise it stays a "0"

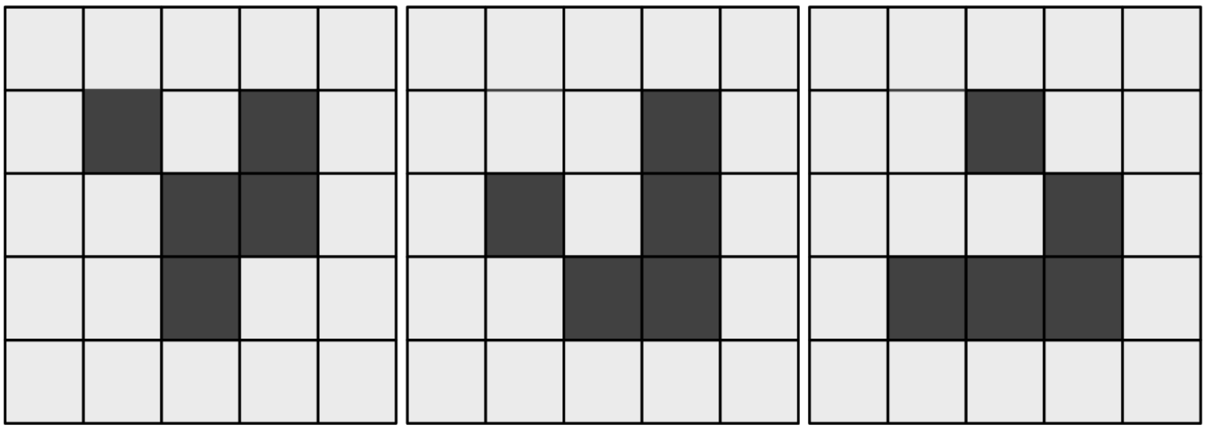

Figure 1: Three time steps from Conway's Game of Life.. This particular configuration is known as a "glider".

Where a given cell has eight adjacent neighbors (four sharing a side, four sharing only a corner), as shown in Figure 2.

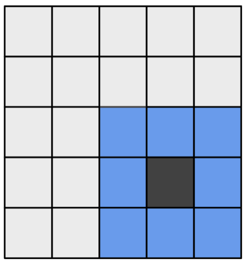

Figure 2: A gray cell with its eight adjacent neighbors highlighted in blue. This group of cells is also known as the "Moore neighborhood" of the gray cell.

These rules illustrate the power of cellular automata in creating complex, emergent behavior through state evolution.

2 Proposed Scheme

We describe in this section our proposed encryption and decryption scheme, as well as the rationale behind many of the design decisions therein. We limit ourselves to a 32-bit key and a 256 block. We choose a 256 bit block in order to ensure a given message can be packed into a square grid ($\sqrt{256} = 16$).

2.1 Encryption and Decryption

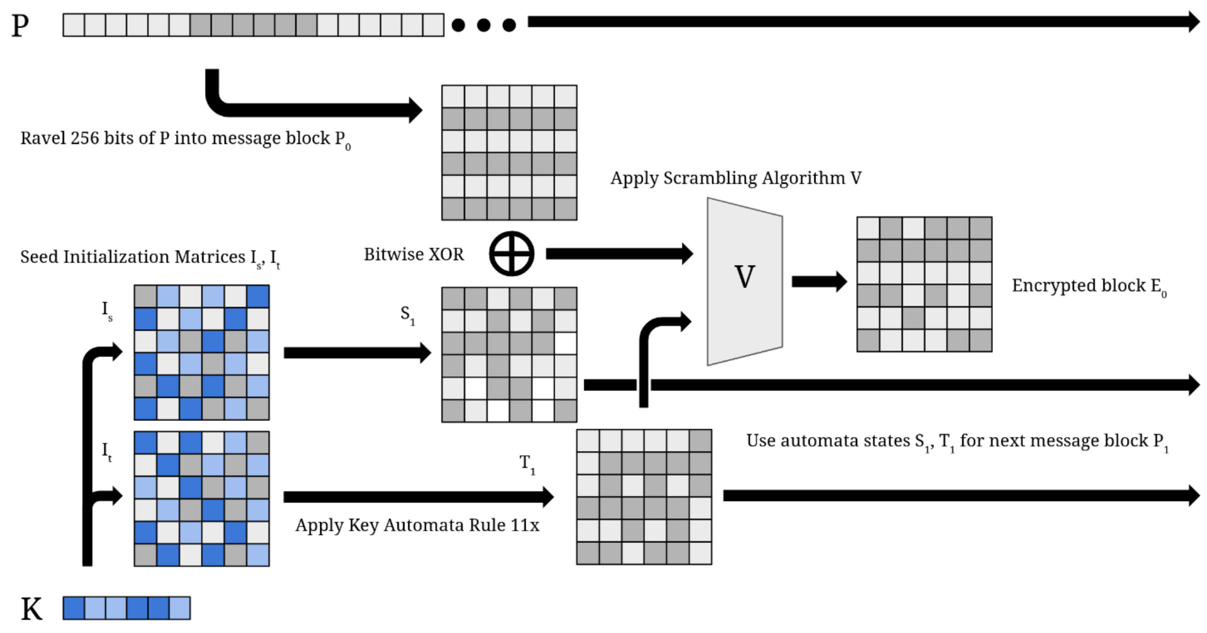
P
Ravel 256 bits of P into message block P_0
Apply Scrambling Algorithm V
Seed Initialization Matrices I_s, I_t
Bitwise XOR
V
Encrypted block E_0
I_s
S_1
I_t
T_1
Use automata states S_1, T_1 for next message block P_1
Apply Key Automata Rule 11x
K

Figure 3: Our encryption scheme.

Our overall encryption scheme is shown in Figure 3. In order to encrypt a message P, first split P into M 256-bit blocks $P_0, P_1, ...P_{M-1}$. We additionally recommend that the final block be padded with either randomly sampled noise, or with words randomly sampled from a large text corpus, as this will prevent sparse final blocks from leaking information about the final automaton states.

Following splitting the message, perform the following steps on block P_i:

1. Use the key scheduler defined in Section 2.2 to obtain block transpose key T_i and block shift key S_i.
2. XOR the plaintext block P_i with S_i.
3. Scramble the result using scrambling algorithm V defined in Section 2.2.3 with key T_i to obtain ciphertext block E_i.

$$E_i = V(P_i \oplus S_i, T_i)$$

The decryption of ciphertext block E_i can be accomplished using a reverse process:

1. Use the key scheduler defined in Section 2.2 to obtain block transpose key T_i and block shift key S_i.
2. Unscramble the result using inverse scrambling algorithm V^{-1} defined in Section 2.2.3 with key T_i to obtain ciphertext block E_i.
3. XOR the result with S_i to obtain plaintext block P_i.

$$P_i = V^{-1}(V(P_i \oplus S_i, T_i), T_i) \oplus S_i = P_i \oplus S_i \oplus S_i = P_i$$
$$\Rightarrow P_i = V^{-1}(E_i, T_i) \oplus S_i$$

2.2 Key Scheduling

We first use 32-bit key K to initialize two 256-bit block keys: transpose key T_0 and shift key S_0 using the Key Automata rule described in Section 2.2.2. In detail, K is first used to seed the two 16×16 matrices, I_t and I_s.

2.2.1 Block Initialization Matrices

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	2	3	4	5	6	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Table 1: The correspondance between base 32 digits (top row) and values (bottom row).

$$\begin{bmatrix}
. & A & \# & 3 & . & 2 & \# & Z & . & Y & \# & X & . & W & \# & V \\
7 & . & B & . & 4 & . & P & \# & O & . & N & . & M & \# & L & . \\
\# & 6 & \# & C & \# & 5 & \# & Q & \# & 3 & . & 2 & \# & Z & . & U \\
E & . & 5 & \# & D & . & 6 & . & R & \# & 4 & \# & 7 & . & K & \# \\
\# & D & . & 4 & \# & E & . & 7 & . & S & \# & 5 & . & Y & . & T \\
F & . & C & \# & 3 & . & F & . & A & \# & T & \# & 6 & \# & J & \# \\
\# & Q & \# & B & . & 2 & . & G & \# & B & . & U & \# & X & . & S \\
G & ,\# & P & . & A & . & Z & \# & H & . & C & \# & V & . & I & \# \\
. & R & \# & O & . & 7 & \# & Y & . & I & \# & D & . & W & \# & R \\
H & . & E & \# & N & . & 6 & \# & X & . & J & . & E & \# & H & . \\
\# & S & . & D & \# & M & . & 5 & \# & W & . & K & \# & F & . & Q \\
I & \# & F & . & C & \# & L & . & 4 & \# & V & \# & L & . & G & . \\
. & T & . & A & . & B & \# & K & . & 3 & \# & U & . & M & . & P \\
J & \# & G & \# & H & \# & I & \# & J & \# & 2 & \# & T & \# & N & \# \\
. & U & \# & V & . & W & . & X & . & Y & . & Z & \# & S & . & O \\
K & \# & L & . & M & \# & N & \# & O & \# & P & . & Q & \# & R & .
\end{bmatrix}
\begin{bmatrix}
P & \# & O & \# & N & \# & M & \# & L & \# & K & \# & J & \# & I & \# \\
\# & L & \# & K & . & J & \# & I & . & H & . & G & \# & F & . & H \\
Q & . & D & \# & C & \# & B & \# & A & \# & 7 & \# & 6 & \# & E & \# \\
\# & M & . & X & \# & W & . & V & . & U & . & T & . & 5 & \# & G \\
R & . & E & . & H & \# & G & . & F & \# & E & . & S & \# & D & . \\
\# & N & \# & Y & . & T & \# & S & . & R & . & D & \# & 4 & . & F \\
S & . & F & . & I & \# & 3 & \# & 2 & . & Q & \# & R & \# & C & . \\
\# & O & . & Z & \# & U & . & 7 & \# & Z & \# & C & . & 3 & \#E & \\
T & \# & G & \# & J & . & 4 & . & 6 & \# & P & . & Q & . & B & \# \\
\# & P & \# & 2 & . & V & \# & 5 & . & Y & \# & B & . & 2 & . & D \\
U & . & H & \# & K & . & W & . & X & \# & O & \# & P & . & A & . \\
\# & Q & . & 3 & \# & L & . & M & . & N & . & A & \# & Z & . & C \\
V & . & I & . & 4 & \# & 5 & . & 6 & \# & 7 & . & O & \# & 7 & . \\
\# & R & . & J & . & K & \# & L & . & M & . & N & . & Y & \# & B \\
W & . & S & \# & T & . & U & \# & V & \# & W & . & X & . & 6 & \# \\
\# & X & . & Y & . & Z & . & 2 & \# & 3 & . & 4 & . & 5 & . & A
\end{bmatrix}$$

Figure 4: I_t and I_s respectively. Base 32 digits represent the index of the bit used in key K for seeding. A table of base 32 digits is given in Table 1. Entries marked with a `#` are always `1`, while entries marked with a `.` are always `0`.

Shown in Figure 4 are initialization matrices I_t and I_s. The design of these matrices was intentional, and based on a few assumptions.

1. It is desirable that the initial state of each CA contains roughly the same number of `1` and `0` cells. In order to achieve this, 64 cells are initialized with a constant (independent of K) `1` value, and 64 cells are initialized with a constant `0` value. This applies a form of Laplacian smoothing to our initial automaton states, so we are biased to obtain a state with the desired uniform property.
2. It is desirable that our key values interact with the constant values immediately, as otherwise the constant values will yield invariant sections of the early-stages of the evolved automaton. Thus, the constant and seeded values are arranged in a checkerboard pattern.
3. It is desirable that no key can yield a bilaterally symmetric initial state, as CAs with symmetric initial states will remain symmetric, providing patterns in the generated noise which an adversary can take advantage of. The chosen constant values were taken from a pattern which is not bilaterally symmetric. This guarantees no K can yield an initially symmetric CA, but it does not preclude a symmetric state from arising during evolution.
4. It is desirable that the key be distributed evenly through the initially seeded matrix, but not in a symmetric manner.
5. It is desirable that the two matrices not be symmetric to each other.

Following seeding, the Key Automata rule is applied 11 times to the seeded matrices I_t and I_S to obtain T_0 and S_0 respectively. In order to obtain the next pair of block keys T_1 and S_1, the Key

Automata rule is applied an additional 11 times to T_0 and S_0. This can be repeated until enough blocks are obtained to encrypt the message.

The decision to use 11 applications of the Key Automata rule is explained in Section 2.2.2.

2.2.2 Key Automata Rule

As was done with our initialization matrices, we define a number of maxims we can follow to maximize the efficacy of our automaton as an entropy source. Specifically:

1. Few attractors and stable states exist. Such states would make easy guesses for decrypting patches of a message, weakening our cipher.
2. Low repetition frequency. If a CA state is repeated frequently, it is possible that some transpose key T_i or shift key S_i will be repeated for message blocks, allowing for a differential analysis.
3. A roughly equivalent number of `0` and `1` cells in each evolved state. This limits the capability of probabilistic attacks.
4. Most states have a precursor state, as this maximizes the effective key space of each block key.
5. The rule is "symmetrical", meaning no particular set of cells has some special property. This both limits the occurrence of patterns in the automata, and also reduces its reliance on a small subset of the key.
6. The rule is simple enough to process quickly, allowing for fast encryption and decryption (see Section 2.1).

From these maxims, we propose the following Moore-neighborhood Class III Cellular Automata rule based on empirical observation.

1. If a cell is a `1` and has 2-4 (inclusive) neighboring `1` cells it stays a `1`; otherwise it becomes a `0`.
2. If a cell is a `0` and has 2-6 (inclusive) neighboring `1` cells it becomes a `1`; otherwise it becomes a `0`.

Where we additionally allow border cells to neighbor opposing border cells to promote faster diffusion of information. This property additionally makes all of the cells symmetric to each other, as without it border cell behavior would deviate from inner cell behavior. Although we do not prove any of our maxims about this CA rule due to the undecidable nature of related problems, we empirically observe that some are approximately satisfied. Numbers obtained from these experiments are given in Appendix B.

The decision to evolve a CA 11 times before each use as described in Section 2.2.1 was motivated by two factors: first, it is clear by our rule that any given cell can only affect its eight neighboring cells. Due to this, for each cell to affect each other cell in generation, it is necessary that at least 8 iterations of our CA rule are applied (recall that the space our cells occupies loops). Additionally, a CA will always repeat after some number of generations G. Choosing a prime number reduces the likelihood that the CA's repetitions will align with the states used in encryption.

2.2.3 Scrambling Algorithm

We have now defined a method for generating pseudorandom noise of the same size as our message block. However, it is possible that contiguous regions of our message will be uniformly transformed by a simple XOR, meaning partial reconstruction may be possible. To mitigate this, we also introduce a scrambling algorithm V to ensure the message bits are well dispersed before XORing.

V takes two inputs, a 16x16 message block P_i and a 16x16 block transposition key T_i. The process of V can be described in three steps:

1. Starting with bit $3j \bmod 4$ of row j of T_i, concatenate every fourth bit in the row to obtain a 4 bit unsigned integer r_j.
2. Iterate over the rows in P_i in ascending order, swapping the jth row of the message with the r_jth row of the message.

3. Repeat this process over the columns of the resulting scramble, starting with the $3j + 3 \bmod 4$th bit of each column and concatenating every fourth bit to yield c_j, then swapping column j with column c_j.

In order to reverse the scramble given ciphertext block E_i, calculate the column c_j values and iterate over the columns of the encrypted block in descending order, swapping each column j with column c_j. Then repeat with the rows to obtain P_i.

These specific bit indices were chosen with the goal of assuring no bit would be used twice in generating a column or row index.

3 Strengths of Approach

3.1 Modularity

Although it is not directly related to security, we note that our method is modular, as the proposed initialization matrices I_t and I_s, Key Automata Rule, and scrambling algorithms could all be replaced by other similar algorithms. Thus, if any component of the system is found to be insecure it can be simply replaced. Any other initial matrices or CA rules to be used should still satisfy the maxims given in Section 2.2.1 and Section 2.2.2 respectively.

4 Reference Implementation

A reference implementation of the defined encryption algorithm is available on GitHub[1] to be compiled with the Rust compiler. We choose rust for our implementation to facilitate speed of execution, which is elaborated upon in Appendix A. Our reference implementation merely implements the encryption scheme and does not pad any messages with random noise as recommended in Section 2.1.

5 Threats to Validity

5.1 Speed of Encryption

Despite our implementation exceeding our expectations with regards to its speed (see Appendix A), it does not hold up to contemporary, hardware based implementations of common cryptographic protocols. Most off the shelf hardware is unable to simulate cellular automata without high-level software implementations. This means our algorithm is slower than current SOA methods of encryption. However, CA, can be implemented in hardware, meaning this is not an inherent barrier in the methodology.

5.2 Naivety of Implementation

We admit that we have little domain knowledge in either cellular automata or the field of encryption, with most of our experience being built over a two week period. Additionally, although previous work exists in using cellular automata for encryption, we intentionally avoided it in designing our cipher and evaluation methods. This was done because we believe the greatest contribution of our cipher to be its novelty, and wanted to avoid being influenced by earlier designs.

5.3 Potential Attacks

5.3.1 Probabilistic Attacks

Our experimental results in Appendix B reveal potential avenues for probabilistic attacks of our cipher. Specific flaws of our automata rule which could be taken advantage of include:

[1]https://github.com/Ezuharad/talos-cipher

1. A mean cell value of .506 across multiple automaton states, which is 0.006 higher than the ideal of 0.5. This bias lends strength to probabilistic attacks on long ciphertexts.
2. The mean cell value within any specific automaton state fluctuates quite heavily, with a standard deviation of 18.2. This allows attackers to credibly assume that a given automaton state will contain a majority of either `1` or `0` cells.
3. The Key Automata Rule is especially sensitive to its initial state, and we observe that keys similar under the hamming distance have a tendency to collapse to the same automaton state quickly. This flaw is quite severe, as it drastically shrinks the effective key space of our cipher, making it weaker to brute force strategies. Potential remedies for this flaw are given in Section 6.1 and Section 6.2.

We also note that further exploration of the joint probabilties of cell neighborhoods (that is, how much information a cell gives us about its neighbors) may yield further avenues for attacks.

5.3.2 Scrambling Algorithm

We note that although our scrambling algorithm does allow contiguous patches of bits to be spread across the message block, it also preserves the sum of the rows and columns of the block following transposition (that is, the sets of row and column sums do not change following scrambling). This may allow an attacker to reconstruct the original message if they are able to make a reasonable guess as to the shift automata state used in the XOR. This problem could be mitigated by using another scrambling algorithm, perhaps one which shifts rows or columns in addition to just swapping them.

6 Potential Improvements and Future Work

6.1 Seeding Matrices Through Time

As observed in Appendix B, the Key Automata Rule is very sensitive to its initial state, and multiple keys collapse to the same initial state quite often. We term this problem state collision, after a similar phenomenon observed in designing hash functions. Because each CA state is determined only by its previous state, this very strongly decreases our key space. One potential solution for this problem is to seed the the initialization matrices I_s, I_t not only across space, but also across time, perhaps by overwriting selected cells with key bits at predetermined time steps. We see in our results that once an automaton has escaped its initial state it rarely overlaps with another's. Thus, by extending the use of our key across multiple time steps, we provide more opportunities for deviation.

6.2 Larger Automata Sizes

The use of a larger automaton cellspace could prevent state collisions, as more possible states could be evolved from an initial, seeded state. We believe this to be an inferior option compared to that in Section 6.1, as we observe few state collisions past the first few initial states. Thus, this approach would more substantially increase computing costs and likely yield similar results.

7 Conclusion

We propose a novel symmetric encryption algorithm based on cellular automata and provide justification for our approach. Additionally, we implement and evaluate our algorithm, showing its strengths and weaknesses as a general purpose cipher.

A Benchmarks

Two implementations of the Key Automata rule were created. The first, written in rust, is the faster of the two and was used in our final benchmarks. A second implementation was written as a PyTorch neural network with a desire to leverage the GPU for fast encryption, but was ultimately much slower than the rust version. We believe this disparity to arise from two factors:

1. PyTorch is intended to be used for neural network training, not cellular automata simulation. The framework was chosen merely due to the implementer's familiarity. Using CUDA for a GPU implementation would likely yield significantly improved results.
2. The simulated CA is only 16x16 cells, and is read from every 11 iterations in order to encrypt data. This is a suboptimal scenario for employing the GPU, as we need to lock the simulation and read data from the GPU every time we want to encrypt or decrypt a block. It's therefore possible that CPU SIMD registers simply outperform CUDA cores at this scale.

Despite the rust implementation's significant edge over its PyTorch-based counterpart, it is still a naive one due to the implementer's lack of familiarity with the language[2] and problem domain. Thus, many of the speed gains come from optimizations made by LLVM during compilation, and it's likely the hardware could be used more effectively.

We use a Dell XPS 13, equipped with a 5 GHz Intel Core i7 Comet Lake Processor, for all experiments. A 4.1 MiB file (a txt file containing the King Jame's Bible) was used as our plaintext for encryption. Over three trials the time to encrypt the entire file came to an average of 52.44 seconds, and the time to decrypt the file came to a similar average of 53.75 seconds. Thus, the average encryption speed is about 655,000 bits/s and the average decryption speed about 639,000 bits/s on this hardware.

B Empirical Results of Key Automata Rule

Because the Key Automata Rule is critical in our algorithm's security, we seek to determine its strength as a random number generator. We use the rust code given in our repository for our experiments.

BA Quantitative Results

BAA Mean Cell Values

We first evaluate how many bits are, on average, a `1` or a `0` given an automaton state. This is germane to our shift matrix, as we want to XOR a message block P_i with a generated shift key S_i which is, on average, half `0`s and half `1`s. Seeding matrix I_s three times with a random 32 bit number and iterating 32,000 generations, we see an average cell value of .506 with a standard deviation of 18.2. Although we are happy with the mean of .506, the standard deviation of 18.2 is quite high. The standard deviation of the n-observation binomial distribution over the Bernoulli distribution with mean μ is given by the formula $\sqrt{n\mu(1-\mu)}$. Applying this to our scenario with 255 observations, we see an ideal standard deviation would be 8, which is less than half of 18.2. This weakens our cipher, as attackers can assume that most of the bits in any given automaton state are either a `0` or a `1`.

BAB State Collisions

We next evaluate the probability of state collisions during automaton evolution. Seeding the block initialization matrix S_i with the first 3419 32-bit keys, we find that 1581 (nearly half!) evolve to some state which another had already achieved after 32000 generations, with the all of these collisions occuring within 2-3 time steps. This implies that although our proposed Key Automata Rule is

[2]This was their first multi-file rust project, implemented in about one week.

extremely sensitive to its inputs and can quickly deviate from other state trajectories, there is a high probability that the rule will yield collisions near the very beginning of the simulation. This effectively shrinks our keyspace by a significant factor, making brute force attacks more plausible.

The probability of state collisions goes down significantly when using a random key, with only 6 collisions occurring in 1845[3] keys. Even still, 2 of these 6 collisions occurred on the third time step of the automaton simulation, again highlighting the difficulty in escaping the initial state. We suspect that keys within a certain hamming distance of each other have a high probability of encountering this type of early collision, and propose remedies in Section 6.1 and Section 6.2.

BAC Scrambling Algorithm Indices

Finally, we evaluate the frequency of each column or row index obtained in our scrambling algorithm V (see Section 2.2.3). Due to the symmetry of our cellspace, we consider specific row and column indices to be effectively identical for a frequency analysis. Averaging over 100 keys, each used to encrypt 100 blocks of plainetext, we find a relatively flat distribution in Figure 6, albeit with some notable biases.

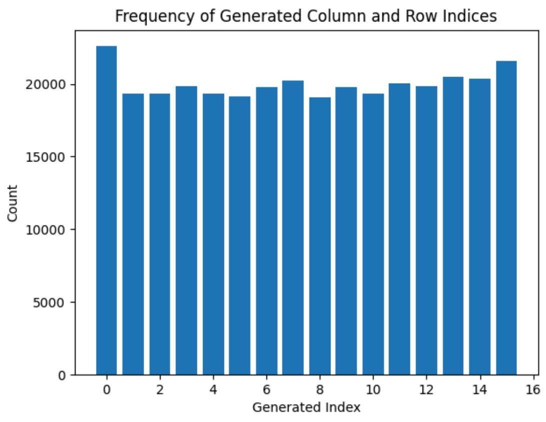

Figure 6: The relative frequencies of each row/ column index, averaged across 100 keys each used for 100 blocks of plaintext.

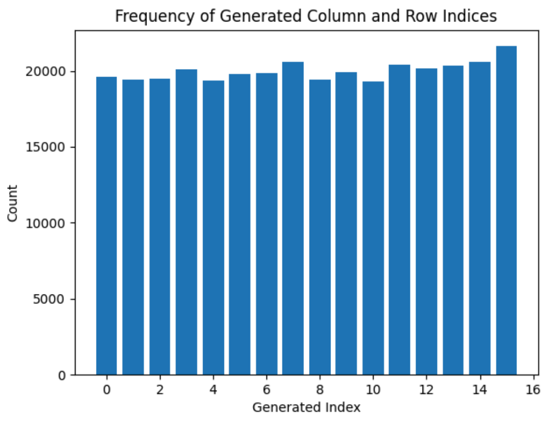

Figure 7: The relative frequencies of each row/ column index, averaged 100000 states arising from 1 key.

We additionally plot in Figure 7 the index distribution from a single seed used on 10000 blocks of plaintext, noting a similar (albeit slightly less biased) uniform shape.

Further testing on the distribution of the joint probabilities of the indices given a specific automaton state should be attempted, as we believe more significant biases may emerge.

[3]The disparity in the number of keys tested arises from RAM limitations. Because we must store all earlier seen states in a hashset, the memory cost of this test increases with time. We ran out of memory more quickly with random keys, as many of them were simulated for the full 32000 generations due to the relatively sparser collisions.