A Message Across Time: Digital Preservation Using Cellular Automata

Digital Time Capsule Project

March 2025

To the Discoverer of This Capsule

Whether you are from our future or our past, this document will guide you through reading our encoded data.

1 Greetings Across Time

You've discovered something special - a message deliberately preserved to transcend time. We can't know if you're reading this centuries from now or if somehow our work has found its way to those who came before us. Either way, we've encoded information we believed worthy of preservation, using mathematical principles that should remain discoverable regardless of technological evolution.

This isn't just data storage - it's a conversation across time.

1.1 For Readers from Our Past

Reading this from before the 21st century? Some of this might sound like science fiction. We've encoded our message using a mathematical system called a "cellular automaton" - specifically Conway's Game of Life, invented in 1970.

Don't worry if you're unfamiliar with our references to "GPUs" (Graphics Processing Units). These specialized computing chips became ubiquitous in our time, allowing massive parallel calculations. *We used them because they're fast, but the mathematics work on any computing system - even one built with vacuum tubes or mechanical relays.* The principles are what matter, not the specific technology.

1.2 Basic Concepts for Past Readers

If you're reading this from what we would consider "the past" (before the 21st century), here are some brief explanations of concepts we use:

Binary Data Information in computing systems is represented using only two values: 0 and 1 (called bits). Eight bits form a byte, which can represent 256 different values (2^8). Text, images, and all other data are encoded as sequences of these bits. For example, the letter 'A' in ASCII encoding is represented as 01000001.

XOR Operation XOR (exclusive OR) is a mathematical operation that takes two bits and returns 1 if exactly one of them is 1, otherwise it returns 0. It can be expressed in this truth table:

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

XOR has a special property: $(A \oplus B) \oplus B = A$. This makes it naturally reversible and perfect for simple encryption. Our system uses this property extensively in both the encryption of the data and within our modified Game of Life implementation.

QR Codes These are two-dimensional barcodes developed in the late 20th century that can store information in a pattern of black and white squares. They were designed to be easily scanned and decoded by optical devices. They can store thousands of characters and include error correction to remain readable even if partially damaged.

1.3 For Readers from Our Future

From our vantage point in 2025, we can only imagine the computing systems of your era. Our technology probably seems as quaint to you as vacuum tubes would to us. You've likely developed processing architectures we can't even conceptualize.

The beauty of mathematical principles is their timelessness. The cellular automaton encoding we've used should be implementable on whatever computing substrate you've developed. The patterns and rules remain valid whether executed on our silicon chips or your quantum computers, biological processors, or technologies we couldn't imagine.

2 The Basics: What You Need to Know

2.1 Our Encryption Approach

Our preservation system incorporates multiple layers of security and encoding:

1. The original data is encrypted using XOR with a specific key
2. This encrypted data is then processed through our modified Game of Life
3. The results are stored with additional metadata including the encryption key itself (separately)

This layered approach ensures both security and reversibility. The encryption key also contains information about the number of iterations the Game of Life should run, which is essential for the decoding process.

2.2 Our Modified Game of Life

At the heart of our encoding system is a modified version of Conway's Game of Life—a cellular automaton invented in 1970. While the traditional Game of Life is not reversible, we've designed our implementation specifically to allow reversibility by preserving essential state information and incorporating XOR operations between states.

Each cell in our system exists in a 256×256 grid and can be either alive (1) or dead (0). Let $S_t(x, y)$ represent the state of cell at position (x, y) at time t. The number of living neighbors is denoted as $N_t(x, y)$, calculated as:

$$N_t(x, y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} S_t(x + i, y + j) - S_t(x, y)$$

Our forward rules follow the classical Game of Life, but with XOR operations between consecutive states:

Current State	**Neighbor Count**	**Next State**
1 (Alive)	$N_t(x, y) < 2$	0 (Dead)
1 (Alive)	$N_t(x, y) \in \{2, 3\}$	1 (Alive)
1 (Alive)	$N_t(x, y) > 3$	0 (Dead)
0 (Dead)	$N_t(x, y) = 3$	1 (Alive)
0 (Dead)	$N_t(x, y) \neq 3$	0 (Dead)

The output from each step is XORed with the previous state, creating a chain of dependencies between consecutive states that enables reversibility. What makes our system special is that we store both states S_t and S_{t+1}.

This pair of consecutive states, combined with the knowledge of the number of iterations run (which is specified in the encryption key), contains enough information to run the system backward and recover the original data.

2.3 Our Encoding Method

We use the Game of Life in a novel way to encode data:

1. We convert the original data to binary (1s and 0s)
2. We encrypt this binary data using XOR with our key
3. We arrange these encrypted 1s and 0s into a 256×256 grid
4. We run the modified Game of Life on this grid for the number of iterations specified in the key
5. During each iteration, the output state is XORed with the previous state
6. We store both the final state AND the state before it
7. With both states and the knowledge of iteration count from the key, the original data can be recovered by running the Game of Life backward
8. The generated file can now be stored in a QR code on a physical film for long-term conservation

Encoding Process Overview

Original Data $\rightarrow$ XOR $\rightarrow$ Binary Grid $\rightarrow$ Game of Life $\rightarrow$ DENIS File

2.4 The DENIS File Format

We store the encoded data in a custom format called DENIS. Note that all numeric values in the header are stored in little-endian byte order (least significant byte first):

```
DENIS File Structure:
+-------------------------+
| Header (24 bytes)       |
| +-----------------------+
| | Magic "DENIS" (5B)    |
| | Version (1B)          |
| | File Type (3B)        |
| | Padding Info (2B)*    |
| | Extra Space (5-7B)    |
| | Data Size (8B)        |
+-------------------------+
+-------------------------+
| Encoded Data            |
| (variable length)       |
+-------------------------+
| Terminator (8 bytes)    |
| (All 0xFF)              |
+-------------------------+
* Only in version 2
```

2.5 The Encryption Key Format

The encryption key is a critical component of our system. It contains both the XOR encryption key and the number of iterations the Game of Life was run during encoding. Understanding this format is essential for successful decoding.

```
Key Format:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX:YYYY

Where:
X = 32 hexadecimal characters (the XOR encryption key)
: = Separator character
Y = 4 hexadecimal digits representing the iteration count
```

For example, a key might look like:

```
9ab2e7c1f5680d34129a8b7e6f5d3c21:00f4
```

This example key has:

- XOR Key: 9ab2e7c1f5680d34129a8b7e6f5d3c21
- Iteration Count: 00f4 (hexadecimal) = 244 (decimal)

The XOR key is used twice in the process:

1. Initially to encrypt the original data before running it through the Game of Life
2. Finally to decrypt the data after running the Game of Life backward

The iteration count tells the decoder exactly how many Game of Life steps to reverse. This precise count is crucial—running too few or too many iterations will result in corrupted data.

During decoding, the key should be parsed as follows:

1. Split the key at the ':' character
2. The first part is the XOR encryption key
3. The second part is the hexadecimal representation of the iteration count
4. Convert the iteration count from hexadecimal to decimal for use in the algorithm

Note that the key must be kept intact with its exact format, including the ':' separator. Any modification, even a single character, will result in incorrect decoding.

2.6 Storing a DENIS File for Long Periods of Time

As explained previously, QR codes are used to store the files (physically). The QR codes can be placed anywhere, on any item. They can also be printed (but once the colors fade out...). A good long-lasting solution is to use a silver halide microfilm on a polyester base. This solution has been proven to last at least a thousand years.

To store the data of the DENIS file, we store the bytes directly in a PNG image. Before storing the bytes, we use an error correction algorithm in case bit corruption occurs. The algorithm used is called "Hamming Codes" and here is how it works:

For a message of m bits we add, what is called, r parity bits. The number r must satisfy $2^r \geq m+r+1$. The r parity bits are placed in between data bits at positions that correspond to powers of 2 (1, 2, 4, 8, ...). For the case of a single byte error correction ($m = 8$), we can for example use $r = 4$. The encoded message will thus have a length of $m + r = 12$ bits. The positions and roles of each bit are represented in table 1

Each parity bit covers a certain set of data bits. The coverage follows the logic shown in list 2.6

- **Parity bit P1:** Covers bits whose binary position has a 1 in the least significant position (1, 3, 5, 7, 9, 11).

Position	Role
1	Parity (P1)
2	Parity (P2)
3	Data (D1)
4	Parity (P4)
5	Data (D2)
6	Data (D3)
7	Data (D4)
8	Parity (P8)
9	Data (D5)
10	Data (D6)
11	Data (D7)
12	Data (D8)

Table 1: Positions and Roles in Hamming Code Encoding (12 bits)

- **Parity bit P2:** Covers bits whose binary position has a 1 in the second position (2, 3, 6, 7, 10, 11).
- **Parity bit P4:** Covers bits whose binary position has a 1 in the third position (4, 5, 6, 7, 12).
- **Parity bit P8:** Covers bits whose binary position has a 1 in the fourth position (8, 9, 10, 11, 12).

The value of those parity bits is computed by XORing all the bits the parity bit covers.

During decoding, you should recalculate parity bits and compare them with the received parity bits. If a parity check fails, it gives a binary number (syndrome) representing the position of the erroneous bit.

If the syndrome is zero, no error is detected. If a non-zero syndrome is obtained, it directly points to the bit position that needs correction. Flip the bit at the position indicated by the syndrome to correct the error.

3 Decoding: How to Recover the Original Data

3.1 The Decoding Process

To recover the original data:

1. Read the DENIS file header to understand what's encoded and extract the encryption key information
2. Parse the iteration count from the key

3. Extract the pairs of Game of Life states
4. For each pair, run the modified Game of Life backward for the specified number of iterations, applying the appropriate XOR operations with the key at each step
5. Convert the resulting grid back to binary data
6. Apply a final XOR decryption using the main key
7. Combine all chunks to reconstruct the original file

Decoding Process Overview
DENIS File → Extract States → Reverse Game of Life → De-XOR → Original Data

3.2 Reversibility: The Key Innovation

3.2.1 Forward Evolution with XOR

The forward evolution of our system follows the standard Game of Life rules but adds an XOR operation between consecutive states. This can be expressed mathematically as:

$$S'_{t+1}(x,y) = \begin{cases} 1, & \text{if } S_t(x,y) = 1 \text{ and } N_t(x,y) \in \{2,3\} \\ 1, & \text{if } S_t(x,y) = 0 \text{ and } N_t(x,y) = 3 \\ 0, & \text{otherwise} \end{cases}$$

$$S_{t+1}(x,y) = S'_{t+1}(x,y) \oplus S_{t-1}(x,y)$$

Where $\oplus$ represents the XOR operation between the new state and the previous state.

```
// Forward evolution algorithm with inter-state XOR
for each cell (x,y) in the grid:
  N = count of living neighbors
  if cell(x,y) is alive (1):
    if N < 2 or N > 3:
      next_state'(x,y) = 0  // Cell dies
    else:
      next_state'(x,y) = 1  // Cell survives
  else:  // cell(x,y) is dead (0)
    if N == 3:
      next_state'(x,y) = 1  // Cell is born
    else:
      next_state'(x,y) = 0  // Cell remains dead

  // Apply XOR with previous state
  next_state(x,y) = next_state'(x,y) XOR prev_state(x,y)
```

3.2.2 Backward Evolution with XOR

The key innovation of our system is that given two consecutive states S_t and S_{t+1}, we can calculate S_{t-1}. This involves understanding the XOR relationship between states and applying the reverse Game of Life rules.

For backward evolution, we first need to recover S'_{t+1} by applying:

$$S'_{t+1}(x, y) = S_{t+1}(x, y) \oplus S_{t-1}(x, y)$$

Then we can analyze the relationship between S_t and S'_{t+1} to infer S_{t-1}:

```
// Backward evolution algorithm with inter-state XOR
function reverseStep(S_t, S_{t+1}, S_{t-1}):
  // First, recover the intermediate state before XOR
  S'_{t+1} = new grid(256x256)
  for each cell (x,y):
    S'_{t+1}(x,y) = S_{t+1}(x,y) XOR S_{t-1}(x,y)

  // Now determine the previous state based on S_t and S'_{t
    ↪ +1}
  S_{t-2} = new grid(256x256)
  for each cell (x,y):
    current = S_t(x,y)    // Current state
    next = S'_{t+1}(x,y)  // Next state (with XOR removed)

    // Analyze the transition to infer previous state
    prev = inferPreviousState(x, y, S_t, S'_{t+1})
    S_{t-2}(x,y) = prev

  return S_{t-2}
```

The function ‘inferPreviousState‘ implements complex logic that analyzes the patterns of both S_t and S'_{t+1} to determine the most likely value of $S_{t-2}(x, y)$.

4 Practical Implementation Guide

4.1 Equipment Needed

To decode our data, you'll need:

- A computing device capable of running simple programs
- If available, a GPU (Graphics Processing Unit) or similar parallel processor
- The encryption key, which is provided separately from the DENIS file
- Basic understanding of binary data representation, XOR operations, and little-endian byte ordering

4.2 Step-by-Step Guide for Implementation

If you need to implement the decoder yourself:

1. Parse the DENIS file format following the header structure (remember all values use little-endian byte order)
2. Obtain the external encryption key and parse the iteration count from it (see Section 2.5)
3. Extract the Game of Life state pairs
4. Implement a modified Game of Life simulator with XOR operations and backward evolution capability
5. Process each state pair by running the specified number of backward iterations
6. Apply the final XOR decryption to the resulting grid
7. Convert the resulting grid back to binary data

Here's a simplified implementation of the backward evolution with XOR:

```
function reverseGameOfLife(S_t, S_{t+1}, iterations):
  // S_t and S_{t+1} are consecutive Game of Life states
      ↪ (256x256 grids)
  // iterations is the number of steps to reverse (found in
      ↪ the key)

  // When starting, we only have S_t and S_{t+1}
  // We need to "bootstrap" S_{t-1} for the first backward
      ↪ step
  S_{t-1} = inferInitialPreviousState(S_t, S_{t+1})

  for step = 1 to iterations:
    // Create grid to hold the calculated previous state
    S_{t-2} = new grid(256x256)

    // First, recover the intermediate state before XOR was
        ↪ applied
    S'_{t+1} = new grid(256x256)
    for each cell (x,y):
      S'_{t+1}(x,y) = S_{t+1}(x,y) XOR S_{t-1}(x,y)

    // For each cell, analyze the pattern of S_t and S'_{t
        ↪ +1}
    // to determine what S_{t-2} must have been
    for each cell (x,y):
      // Current and next state
      current = S_t(x,y)
      next = S'_{t+1}(x,y)

      // Get the neighborhood pattern in the current state
      N_t = getNeighborhood(S_t, x, y)

      // Get the neighborhood pattern in the next state
      N_{t+1} = getNeighborhood(S'_{t+1}, x, y)

      // Based on the transition from S_t to S'_{t+1} and
          ↪ the
      // neighborhood patterns, infer what S_{t-2}(x,y) must
          ↪  have been
      S_{t-2}(x,y) = inferPreviousState(current, next, N_t,
          ↪ N_{t+1})

    // Move one step backward in time
    S_{t+1} = S_t
    S_t = S_{t-1}
    S_{t-1} = S_{t-2}

  // After the backward iterations, S_t contains the
      ↪ encrypted original data
  // Apply the final XOR decryption with the key
  result = new grid(256x256)
  for each cell (x,y):
    keyBit = getBitFromKey(key, x, y)  // Get appropriate
        ↪ bit from the key
    result(x,y) = S_t(x,y) XOR keyBit

  return result
```

5 For Future Historians and Archivists

5.1 Why We Chose This Method

We selected a combination of XOR encryption and Conway's Game of Life for this time capsule because:

- XOR is mathematically simple yet provides effective obfuscation
- The Game of Life is based on simple mathematical rules that should remain discoverable in any era
- Our modified implementation has the property of being reversible
- It creates visually distinct patterns that aid in verification
- The combined approach provides both security and redundancy

5.2 Importance of the Encryption Key

The encryption key is not just for security—it serves specific purposes in our system:

- It determines the initial XOR encryption of the data before it enters the Game of Life process
- It specifies the number of Game of Life iterations that were run (embedded within the key)
- It is required for the final decryption step after the Game of Life has been reversed
- Without the correct key, even with the correct algorithm, decoding will fail

The key is deliberately kept separate from the DENIS file itself and should be available as a separate document, written on the margins of this document, or displayed somewhere within the environment where you discovered this time capsule.

When you find the key, you'll need to extract the iteration count information from it to perform the correct number of reverse steps. See Section 2.5 for the detailed format and parsing instructions.

5.3 Technical Context of Our Era

As of 2025, when this capsule was created:

- We commonly used 64-bit computing architectures
- GPUs (Graphics Processing Units) were specialized processors capable of highly parallel computation
- OpenGL was a common graphics programming interface
- Our storage capacities were typically measured in gigabytes and terabytes
- XOR encryption was considered a basic technique, often used as a building block in more complex cryptographic systems

This information may help you understand the technical choices we made.

6 Final Notes

6.1 Verifying Successful Decoding

You'll know you've successfully decoded our data when:

- The size matches what's specified in the header

- For text files, the content is human-readable
- For images, the data forms a recognizable picture
- If checksums are included in the metadata, they match the decoded content

6.2 A Message of Goodwill

Whether you are from our past learning about technology that didn't yet exist in your time, or from our future where our methods may seem primitive, we hope this time capsule serves as a bridge across time. The mathematical principles of cellular automata and simple cryptography show how complex behaviors can emerge from simple rules—a universal truth that transcends our specific technological era.

May this modest effort to preserve information give you insight into our time, our knowledge, and our hopes for communication across generations.