On-Chain NFT Storage with Linear Congruential Generator

Scott Chu zc2396@nyu.edu

Linsong You ly1172@nyu.edu

Abstract

Non-fungible tokens (NFTs) have gained widespread attention as a means of demonstrating ownership of unique digital assets. However, the majority of NFT metadata is currently stored on centralized servers, which poses serious risks to the ownership and security of these tokens. In this project, we examined the technical aspects of Cryptopunks, a well-known project that utilizes on-chain NFTs. Through this analysis, we developed an enhanced on-chain solution that utilizes an invertible Linear Congruential Generator. This improvement is highly extensible, enabling users to search and customize their NFTs to a greater degree.

1. Introduction

The typical on-chain storage of NFTs is typically limited to a small amount of metadata, such as the NFT's ID and owner in ERC721, a standard implementation for NFT on Ethereum blockchain. Such protocol has limited capacity for storing additional information, making it difficult to store and retrieve more detailed information about an ERC721 NFT.

As a result, the majority of NFT metadata, including the raw data of its artwork, is typically stored on centralized servers such as Google Drive or other third-party servers. These off-chain storage solutions allow for a more streamlined handling of NFTs on the blockchain, with ownership remaining on-chain while the media itself is stored elsewhere. However, this approach does come with inherent drawbacks related to centralization, such as the potential for data loss or tampering.

The centralization of NFT off-chain storage is a concern that can be addressed through the use of InterPlanetary File System (IPFS). IPFS is a distributed storage network that operates using a peer-to-peer network of storage nodes. However, IPFS also has its own limitations, such as the lack of an effective incentive layer and the difficulty of verifying the integrity of the data it stores. To deploy NFT media, individuals may rely on centralized gateway providers to pin the data, while others NFT creators may treat IPFS storage as a secondary solution to the lack of on-chain media, thereby reintroducing the issue of a single point of failure.

Therefore, for NFTs with small dimensions, it is still more ideal to have an on-chain storage solution in order to ensure the long-term viability of a project. CryptoPunk, with dimensions of 24 by 24 pixels each, has long considered the possibility of storing the tokens directly on the Ethereum blockchain. Additionally, the attributes of CryptoPunks can also be stored on-chain alongside the images. On August 18, 2021, the creator of CryptoPunks, Larva Labs, made a community-driven advancement by deploying a smart contract, CryptoPunk Data (Ethereum: 0x16F5A35647D6F03D5D3da7b35409D65ba03aF3B2), that contains all 10,000 unique tokens.

This achievement has motivated us to investigate the technical details of this Cryptopunk smart contract, examining how a single smart contract can encompass all attributes and, more impressively, the raw

RGBA pixels for all possible Cryptopunk tokens in an cost-effective manner. In addition, we have developed and implemented an original and enhanced method that further reduces the amount of data required for on-chain storage while enabling the creation of an almost unlimited number of fully customizable CryptoPunk-style images.

2. Survey

2.1. Characteristics of CryptoPunk

The creation of the CryptoPunk Data smart contract is made possible by two important characteristics of CryptoPunk. It is necessary to discuss these characteristics as they are commonly found in the creation of NFT collections, particularly in the wake of the monumental success of the CryptoPunk project in the history of NFTs.

There are 10,000 unique CryptoPunks, each of which contains a unique combination of characters and traits. This presents an opportunity for on-chain data compression, as certain attributes, such as the base "Male punk" character and the purple cap for Punk #3782, are shared among many other CryptoPunk tokens like #3123. Notice for the same attribute, its location is unchanged among its appearances in different punks.

Hence, the character templates and all possible attributes need not to store on a punk-to-punk basis. Instead of storing all raw RGBA pixels for every punk index, the CryptoPunk Data smart contract exploits the repetition by storing only the indices that indicate which attributes are needed to reconstruct the punk, along with a database containing the pixel data for all attributes.

Besides, according to Larva Labs studio, each punk was algorithmically generated through computer code and thus no two characters are exactly alike to the purpose of digital scarcity. However, the algorithm they used is not transparent and, more importantly, we suspect that this generation process cannot be recreated using an index-based algorithm – meaning that the exact ordering, the attributes a punk has given the index, the rarity of certain traits cannot be summarized into one single formula.

This is evident in the CryptoPunk Data contract. To preserve the exact ordering and the corresponding attributes, the smart contract has to store the corresponding relation between each punk index and the character type as well as the attribute(s) it might have, making up another proportion of CryptoPunks' data needed to be stored on-chain.

2.2. Compressing On-Chain Attributes

Given these two characteristics, it makes sense that the major functionalities of CryptoPunk Data contract centers around storing assets and the punk-assets correspondence. Besides, the CryptoPunk Data goes extra miles by labeling those attributes with attribute names. As shown below in the state variables of CryptoPunk Data.

```
bytes private palette;
mapping(uint8 => bytes) private assets;
mapping(uint8 => string) private assetNames;
mapping(uint64 => uint32) private composites;
```

```
    mapping(uint8 => bytes) private punks;

    address payable internal deployer;
    bool private contractSealed = false;
```

However, from the contract, it is apparent that all data needed is not in the contract code directly. Rather, the on-chain data was added through 266 transactions. By further investigating the transactions the contract deployer made, we can have a more in-depth understanding of its inner workings.

For example, all 133 possible attributes, including the type of characters, got added via the following function:

```
function addAsset(uint8 index, bytes memory encoding,
string memory name) external onlyDeployer unsealed {
    assets[index] = encoding;
    assetNames[index] = name;
}
```

By examining relating blocks, in which the deployer setup the data, we can find the one with the shortest input data:

```
  0x8972c6b4a44c972111bf18feccae7cfb26db615be400d1422c
  0763a1a2b03af5 addAsset : 18,0x6726f0,Clown Nose
```

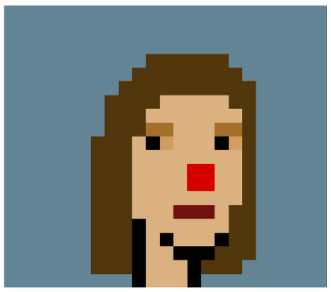
The short hex coding must represent the pixel data of attribute “Clown Nose”, which is a 2 by 2 pixel square in the punk image. With this information, we can reverse-engineer CryptoPunk’s encoding system by hypothetically creating a punk with “Clown Nose” only and pass it to the punkImage function which is used for pixel generation based on any attributes given:

```
function punkImage(uint16 index) public view returns (bytes
memory) {
    require(index >= 0 && index < 10000);
    bytes memory pixels = new bytes(2304);
    ...
            bytes storage a = assets[asset];
                uint n = a.length / 3;
                for (uint i = 0; i < n; i++) {
                    uint[4] memory v = [
                        uint(uint8(a[i * 3]) & 0xF0) >> 4,
                        uint(uint8(a[i * 3]) & 0xF),
                        uint(uint8(a[i * 3 + 2]) & 0xF0) >> 4,
                        uint(uint8(a[i * 3 + 2]) & 0xF)
                    ];
                    for (uint dx = 0; dx < 2; dx++) {
                        for (uint dy = 0; dy < 2; dy++) {
                            uint p = ((2 * v[1] + dy) * 24 + (2 *
v[0] + dx)) * 4;
                            if (v[2] & (1 << (dx * 2 + dy)) != 0) {
                                bytes4 c = composite(a[i * 3 + 1],
                                        pixels[p],
                                        pixels[p + 1],
                                        pixels[p + 2],
                                        pixels[p + 3]
                                    );
                                pixels[p] = c[0];
                                pixels[p+1] = c[1];
                                pixels[p+2] = c[2];
                                pixels[p+3] = c[3];
                            } else if (v[3] & (1 << (dx * 2 +
dy)) != 0) {
                                pixels[p] = 0;
                                pixels[p+1] = 0;
                                pixels[p+2] = 0;
                                pixels[p+3] = 0xFF;
    ...
    return pixels;
}
```

The process embedded in the punkImage function will iterate through the hex data “0x672f0” in such a way that the first byte of every three bytes represents the X, Y coordinate of the pixel block in the punk image. Given the 24 by 24 dimension, only 4 bits are needed for one dimension. Besides, the second byte of every three bytes represents an index to the state variable palette, which is set one time by transaction (0xa92ea3630a13abf3a7322043406df32744c8dc3db1107cf8e244506fba8b284c) which added 128 3-bit RGB color information. Each 3-byte pixel matrix simply store an index to the palette array.

Moreover, the last byte of 3-byte pixel coding includes one color mask and one black mask for the 2 by 2 pixel block. In “Clown Nose” case, color mask is set to 0xf (0b1111), indicating that there are no pixel subtracted from the 2-by-2 pixel block. Similarly, if a pixel is subtracted, a black mask can be applied to the pixel.

```
0x6726f0:
3 bytes, 1 matrix:
X: 0110 | Y: 0111 | color:00100110 | color:1111 |
black0000:
      x: 12 / y: 14  -  palette[38] color / no mask
      x: 12 / y: 15  -  palette[38] color / no mask
      x: 13 / y: 14  -  palette[38] color / no mask
      x: 13 / y: 15  -  palette[38] color / no mask
```



CryptoPunk 91

This encoding scheme is efficient as complex attributes can be further broken down into multiple 2-by-2 pixel blocks - not only square blocks, but also complex shapes with proper masking. For instance, the base character template of punk 9, “Female 4”, can be represented by the following hex string:

```
0x34000835000c360dc337000d38000c390004430008440e81440d6
0450df0460d38460f40470df0480df0490dd24a0dc34b0dc353000a
540df0550df0560dc0560f10561020570df0580df0591340590db05
a0d695b0df063000a640df0650df0660df0670df0680de169135069
0da06a0d5a6b0003730002740db4750df0760f52761080770df0780
df0790d787a000184000285000386000387000388000389000l
```

```
(from tx:
0x077a4274c477a23bdc8571c4ec9cde77fd75b3be973713428c01f
35c50ef0368)
```

2.3. Indexing Problem

Despite the effectiveness of attributes encoding, the CryptoPunk Data smart contract has to record the attributes every punk has.

However, the number of attributes a punk can have varies from 1 to 8 (including the base character archetype). 84.02% of punks have less than 7 attributes and only 1 punk has the luxury of having 8 attributes. Therefore, the encoding of their attribute’s indices would be quite wasteful in terms of precious storage space.

The wastefulness is confirmed in transaction:

```
0x067d7a7e00000001224a00000000076000000000000012d4
01400000000021a3d2d2300000067d816100000000235000000
00000657565e00000010d410000000000238331200000057
b4e0000000006834f7200000056c65000000000783707c00
0000054d4b5f000000040d2d15000000023d461d00000000
3184a000000000245470000000054d6b8400000000332000
000000033d411f000000056c4b0000000003303d35000000
003220f0000000000556000000000066300000000000033d3a
0000000001322300000000067d818400000000012826000000
00056200000000000003182b00000000033a0000000000076c
58000000000012a3d3a133e0000065c00000000005695e0000
000076c5866000000003181e150000000003223b2f00000001
452d2700000005607300000000006810000000000034100000
00000021b3d00000000066c4f000000000116000000000000
034a23000000000056b6100000000002373c0000000002453d2
d00000001173d161400000077f7c000000000143000000000
00032c0000000000011e000000000033f000000000002284
3000000000445100000000076c620000000007630000000
0000054d7d825e0000013d1e1d000000066c700000000066
e67000000000054c000000000056c81540000000013514000
0000002283c00000000067d517352000076c817e0000000
12a3d3c1400000006837d6b0000001173d0f000000317162c
000000033513000000000130460000000000230162c000000
0012a0f0000000004112d00000000033f1d0000000043f00
0000000031e201d000000033d4123000000000551000000
000054d7d717e0000033d4300000000074d690000000000550
000000000002432c00000000077f6d0000000076c7959000
000076b0000000000057d795200000000023d4a0000000007
7a00000000000022d000000000078363740000000024313000
0000000
```

```
(from tx:
0x18206900876bc517c422fe7fe42908d265e82f46028e92430ba79
66cbac0b2df)
```

Among 266 transactions to the contract (0x16F5A35647D6F03D5D3da7b35409D65ba03aF3B2), 100 of them are *addPunks*, which documented the attribute indices for all punks, from index 0 to index 9999, making up 40000 bytes of data, in which contains a large number of zeros.

Hence, the *punks* state variable along takes up 40000 bytes of on-chain. Compared to *assets*, the encoded pixel data only took up 951 bytes of on-chain data based on the sum of *addAsset* length.

Nevertheless, the NFT pixel data compressing method employed by CryptoPunk Data is still considerably more space-efficient (40.951 kB vs. 823.3 kB) compared to the *punks.png* (github.com/larvalabs/cryptopunks/blob/master/punks.png), the file Larva Labs hashed into the original CryptoPunk contract if it were stored on chain.

3. Method

In light of the indexing problem mentioned above, our group researched and implemented an enhanced on-chain storage solution using an customized invertible Linear Congruential Generator.

A linear congruential generator (LCG) is a type of pseudorandom number generator (PRNG) that uses a linear equation to generate a sequence of numbers that are apparently random. Inspired by the *libraryofbabal.info*, a website that approximates Jorge Luis Borges's idea, uses the a memoryless Linear Congruential Generator. In the so-called library of Babel, every book location is represented by its wall/shelf volume numbers corresponding to a unique coordinate. Such a coordinate will be used as the seed in the Linear Congruential Generator, which deterministically produces a large unique number that will be translated into a base-29 number, which will then be translated into alphabetical letters along with three punctuations.

3.1. Linear Congruential Generator

LCG is called a "pseudorandom" number generator because the sequence of numbers it generates is not truly random but is generated according to a fixed set of rules and therefore appears random. Since we are addressing the indexing problem of CryptoPunk, we are intending to use the LCG in the same way as *libraryofbabel.info*, meaning that LCG is not used as a random number generator but an algorithm to encode certain content.

More importantly, given the storing and retrieving functionalities implemented in the CryptoPunk Data contract, information encoded by LCG must be able to be recovered exactly by a reverse algorithm unsimilar to one-way hash function.

The LCG algorithm is defined by the following equation:

$$\textbf{DECODED} = (\textbf{A} * \textbf{INPUT} + \textbf{B})\ \%\ \textbf{M}$$

Where A, B, and M are constants that define the specific algorithm.

In normally circumstances, the first input value of LCG is chosen by the user and is called the "seed" value. The seed value is used to initialize the LCG and its output will be used as the input for next random number generation.

Although LCGs seem to be relatively straightforward, efficient to compute, reversing decoded information deterministically across a large domain of possible inputs proposes serious challenges.

3.2. Hull-Dobell Theorem

By studying the previous works on related subjects, we found that the Hull-Dobell theorem is quite useful, providing valuable insights into how to choose constants in the equation.

The Hull-Dobell theorem is a result in the mathematical theory of partial differential equations. It is often used to prove the existence of solutions to certain types of partial differential equations, such as the heat equation or the wave equation.

Let f(x,y) be a continuous function defined on an open region R in the xy-plane. Suppose that there exists a function g(x,y) such that:

g(x,y) is continuous on R,

g(x,y) is differentiable with respect to x and y on R, and

f(x,y) = g(x,y) + xyh(x,y) for all (x,y) in R, where h(x,y) is a continuous function on R.

Then there exists a function u(x,y) and a constant c such that:

u(x,y) is continuous on R,

u(x,y) is differentiable with respect to x and y on R, and

f(x,y) = u(x,y) + c for all (x,y) in R.

The function u(x,y) is called the Hull-Dobell transform of f(x,y), and c is called the Hull-Dobell constant.

This theorem can be used to determine under what conditions does the repeating period of a LCG with module M attain the theoretic maximum length, which is also M. This not allows us to reverse any integer values over the value domain of *uint32* in solidity, but also provides an effective way to reverse a decoded information. We have implemented a smart contract that acts as a reversible LCG:

```
contract LCG {
    address owner;

    uint32 public constant M =
type(uint32).max >> 1;

    uint32 public immutable A;
    uint32 public immutable B;
    uint32 public immutable Inv;

    constructor(uint32 _m, uint32 _i) {
        require((_m - 1) % 4 == 0,
"Multiplier-1 must be a mutiple of 4.");
        require(checkCoprime(_i, M + 1),
"Increment & Modulus must be coprime.");

        owner = msg.sender;
        A = _m;
        B = _i;
        unchecked { (Inv,) = extEuclidean(A, M +
1); }
    }

    function produce(uint32 _i) public view
returns (uint32) {
        unchecked { return (A * _i + B) & M; }
    }

    function inverse(uint32 _o) public view
returns (uint32) {
        unchecked { return (Inv * (_o - B) &
M); }
    }

    function extEuclidean(uint32 _a, uint32 _b)
internal pure returns (uint32, uint32) {
        if (_b == 0) { return (1, 0); }

        uint32 s;
        uint32 t;

        unchecked {
            uint32 q = _a / _b;
            uint32 r = _a - _b * q;
            (s, t) = extEuclidean(_b, r);
            s = s - q * t;
        }
        return (t, s);
    }

    function checkCoprime(uint32 _a, uint32 _b)
internal pure returns (bool) {
        uint32 gcd = _a > _b ? _a : _b;
        while (gcd != 0) {
            uint32 r = _a % gcd;
            _a = gcd;
            gcd = r;
        }
        return (_a == 1);
    }
}
```

3.3. Encoding Punk Index

Using this LCG, we are effectively encoding 32-bit information into another 32-bit information with any overlap between generated numbers. Every value input value corresponds to a unique output value which can be easily inverted back into the input value. In fact, our LCG is more space-efficient compared to the LCG used in *libraryofbabal.info* (github.com/librarianofbabel/libraryofbabel.info -algo).

With such encoding capability, we have implemented an enhanced CryptoPunk Data smart contract that can further save on-chain storage by encoding each punk's attributes using the index itself.

We built upon the CryptoPunk Data contract and setup the exact *palette* and *assets* as the CryptoPunk, but using the indexing scheme we proposed instead.

As indicated below, every 32-bit value is encoded as

background: **011** | archetype: **10111** | hair:**10100** | beard:**1011** |

eyes: **0110** | cheeks: **011** | mouth:**010** | other:**1011**

which we go through the LCG process and produces a unique index that acts as punk index.

When retrieving punk image by reversing the LCG process and get the packed index for each attributes the punk has.

4. Front-end Implementation

4.1. Code Repository

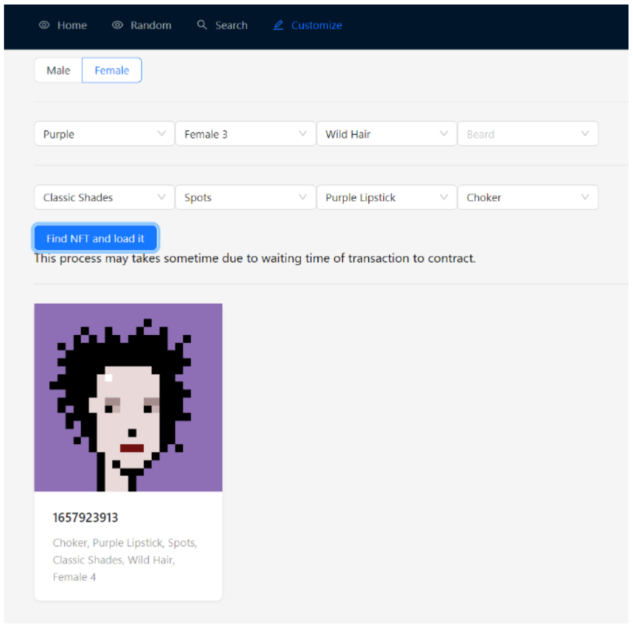
https://github.com/suker0723/LCG_NFT_2022

4.2. Frontend Ether.js

We utilize Ether.js and Hardhat to implement this project. The Hardhat Network supports an instance to deploy the contract locally and allow using MetaMask to connect. We interact with contracts on Hardhat to send assets and get NFT images.

The Ether.js let us communicate with the Ethereum Blockchain and its ecosystem. After we set up the metamask, we can call the api provided by the Ether.js to get contract methods.

The frontend part is built on React Native. We also import the ant-design library to fulfill some components' functions.



Purple | Female 3 | Wild Hair | Beard

Classic Shades | Spots | Purple Lipstick | Choker

This process may takes sometime due to waiting time of transaction to contract.

1657923913

Choker, Purple Lipstick, Spots, Classic Shades, Wild Hair, Female 4