EBMP: Efficient Bitmap Encodings on Ethereum Virtual Machines

0XMOSTIMA, Penguin Logistics, Leithania
YUNSONG LIU, Carnegie Mellon University, USA
PEIYUAN LIAO, hellsegga, ?

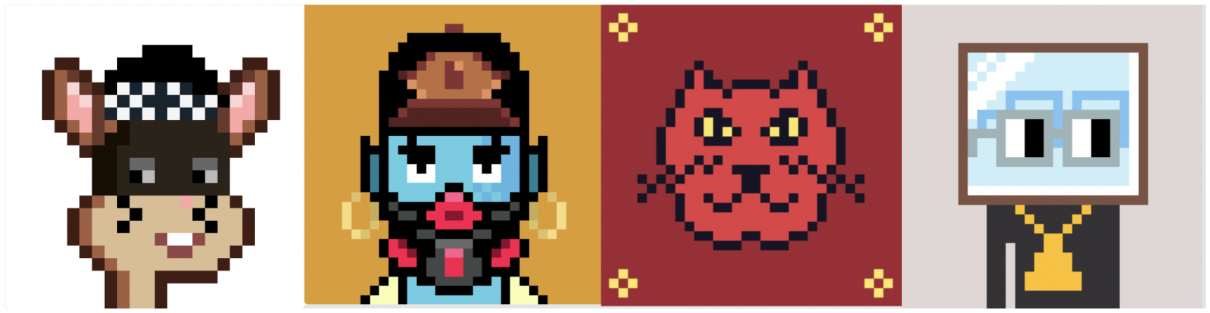
Fig. 1. Popular NFT Projects with On-chain Image: Anonymice [1], Chain Runners [2], Blitmap [3], Nouns [6]

The recent rising interests in Non-Fungible-Tokens (NFT) on cryptocurrency-backed blockchains have prompted a new series of efforts aiming to accurately store, read and render images on the Ethereum Virtual Machine (EVM, the software platform running on one of the leading blockchains, Ethereum), where smart contracts can enjoy "decentralized ownership and control" at the cost of increased computing spend. In this work, we present an efficient protocol to encode image data by directly constructing raw bytes for the device-independent bitmap (DIB) file format. The main function, implemented in Solidity, produces a shorter ERC-721-compatible tokenURI string when compared to existing methods while being more optimized in gas consumption.

CCS Concepts: • **Software and its engineering** → **Software libraries and repositories**; • **Computer systems organization** → *Peer-to-peer architectures*; • **Computing methodologies** → *Image compression.*

Additional Key Words and Phrases: still image coding, blockchain, smart contracts, non fungible tokens

ACH Reference Format:
0xmostima, Yunsong Liu, and Peiyuan Liao. 2022. EBMP: Efficient Bitmap Encodings on Ethereum Virtual Machines. In *SIGBOVIK '22: Special Interest Group on Harry Q. Bovik, April 8, 2022, Pittsburgh, PA.* ACH, 8 pages. http://sigbovik.org/

1 INTRODUCTION

Recently, there has been rising interest in crypto-currency-backed blockchains due to their potential technical and socio-economical impacts. One type of such system, pioneered by Ethereum [13], presents itself as "a decentralized but singleton compute resource," where software developers can write applications, or "smart contracts," in domain-specific languages that are then compiled to bytecode on the virtual machines running on the miners operating the blockchain. For Ethereum's case, popular programming languages for smart contracts include Solidity [11], and Vyper [12], and the virtual machine is called the Ethereum Virtual Machine (EVM). As the title suggests, this paper is mainly concerned with smart contracts on the Ethereum blockchain, which are programs running on EVM.

Ethereum's (reasonably) decentralized and immutable nature gives rise to a variety of use cases, one of which is non-fungible tokens, which are often associated with concepts of digital ownership, scarcity, and the creator community. From a technical perspective, non-fungible tokens are implemented as an interface with a unified application binary interface (ABI) so that any smart contract respecting this ABI is expected to behave in a certain way (in this case, like a non-interchangeable unit of data) with respect to the "token standard" specified. For the case of Ethereum, this standard is called ERC-721 [5] in which the method TOKENURI is specified. It reads as follows:

```
/// @notice A distinct Uniform Resource Identifier (URI) for a given asset.
/// @dev Throws if `_tokenId` is not a valid NFT. URIs are defined in RFC
///  3986. The URI may point to a JSON file that conforms to the "ERC721
///  Metadata JSON Schema".
function tokenURI(uint256 _tokenId) external view returns (string);
```

With the "ERC721 Metadata JSON Schema" defined as follows:

```
{
    "title": "Asset Metadata",
    "type": "object",
    "properties": {
        "name": {
            "type": "string",
            "description": "Identifies the asset to which this NFT represents"
        },
        "description": {
            "type": "string",
            "description": "Describes the asset to which this NFT represents"
        },
        "image": {
            "type": "string",
            "description": "A URI pointing to a resource with mime type image/* representing
                the asset to which this NFT represents. Consider making any images at a width
                between 320 and 1080 pixels and aspect ratio between 1.91:1 and 4:5 inclusive.
                "
        }
    }
}
```

This implies that for an arbitrary user interacting with a smart contract on the EVM that respects the ERC-721 standard, they can expect to get a representation of the underlying data from a non-fungible-token by calling TOKENURI with the respective token ID.

The motivation of this paper came from the observation that a large portion of data represented by NFTs as of now is images due to its increasing adoption in digital art. Currently, there are two ways from which this content is delivered: one is through traditional SaaS and IaaS providers, where TOKENURI points to the address of the content stored on the actual server. This can be seen as a compromise between centralization and decentralization: while the URL address string is immutable, the integrity and authenticity of the underlying content are not guaranteed by the consensus mechanism of the blockchains. The alternative approach, which is the subject of interest for this paper, is to store data directly on blockchain and efficiently render it using EVM itself. In other words, the returned string from TOKENURI would contain a well-formed ERC-721 metadata JSON without relying on any 3rd party rendering software or hosted servers. While elevating data availability and security, this approach also incurs an orders-of-magnitude extra cost for reading and writing large amounts of data to the blockchain.

EBMP aims to alleviate this for non-fungible tokens on Ethereum: efficiently encode and render bitmaps for ERC-721-compatible smart contracts. While the majority of existing methods in the literature focus on drawing each pixel as a separate shape in HTML SVG graphics, we propose to instead directly construct raw bytes in the BMP file format [7], including the file header, device-independent bitmap (DIB) header and the image data, which is then encoded in base-64 and rendered first through the DATA:IMAGE/BMP;BASE64, mime type available for an <IMAGE> in SVG, then the DATA:IMAGE/SVG+XML;BASE64, mime type available for the IMAGE field for a ERC-721 compatible TOKENURI method. Our experiments demonstrate that EBMP produces a much shorter representation for a bitmap that is readable by common NFT platforms while achieving a considerable reduction in gas consumption compared to existing methods.

2 IMPLEMENTATION DETAILS

The full implementation of the encoding method is available below, with detailed in-line comments. EBMP accepts an arbitrary RGB image, with height and width divisible by 4, with pixels arranged in row-major order, flattened with the three colors as their 1-byte representations placed adjacent to each other in red-green-blue order:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity 0.8.11;

import {Base64} from "./Base64.sol";

contract EBMP {
    function uint32ToLittleEndian(uint32 a) internal pure returns (uint32) {
        unchecked {
            uint32 b1 = (a >> 24) & 255;
            uint32 b2 = (a >> 16) & 255;
            uint32 b3 = (a >> 8) & 255;
            uint32 b4 = a & 255;
            return uint32(b1 | (b2 << 8) | (b3 << 16) | (b4 << 24));
        }
    }

    function encode(
        uint8[] memory image,
        uint32 width,
        uint32 height,
        uint32 channels
    ) public pure returns (string memory) {
        bytes memory BITMAPFILEHEADER =
            abi.encodePacked(
                string("BM"),
                uint32(
                    uint32ToLittleEndian(14 + 40 + width * height * channels)
                ), // the size of the BMP file in bytes
                uint16(0), // Reserved
                uint16(0), // Reserved
                uint32(uint32ToLittleEndian(14 + 40))
                // the offset, i.e. starting address, of the byte where the bitmap
                // image data (pixel array) can be found
            ); // total 2 + 4 + 2 + 2 + 4 = 14 bytes long
        bytes memory BITMAPINFO =
            abi.encodePacked(
                uint32(0x28000000), // the size of this header, in bytes (40)
                uint32(uint32ToLittleEndian(width)), // the bitmap width in pixels (signed
                    integer)
                uint32(uint32ToLittleEndian(height)), // the bitmap height in pixels (signed
                    integer)
                uint16(0x0100), // the number of color planes (must be 1)
                uint16(0x1800), // the number of bits per pixel
                uint32(0x00000000), // the compression method being used
                uint32(uint32ToLittleEndian(width * height * channels)), // the image size
                uint32(0xc30e0000), // the horizontal resolution of the image
                uint32(0xc30e0000), // the vertical resolution of the image
                uint32(0), // the number of colors in the color palette, or 0 to default to 2n
                uint32(0) // the number of important colors used, or 0 when every color is
                    important
            ); // total 40 bytes long
        bytes memory data = new bytes(width * height * channels);
        // resharding
        for (uint256 r = 0; r < height; r++) {
            for (uint256 c = 0; c < width; c++) {
                for (uint256 color = 0; color < channels; color++) {
                    data[(r * width + c) * channels + color] = bytes1(
                        image[((height - 1 - r) * width + c) * channels + color]
                    );
                }
            }
        }
        string memory encoded =
            Base64.encode(
                abi.encodePacked(
                    BITMAPFILEHEADER,
                    BITMAPINFO,
                    data
                )
            );
        return encoded;
    }
}
```

Base64.sol here is simply a default implementation for base 64 encoding, available in appendix.

To efficiently use EBMP in an ERC-721 compatible NFT contract, one may replace the image field from a traditionally URL pointing to the image to the base64 encoded string preceded by the MIME type header, like:

```
string memory json =
    Base64.encode(bytes(string(
        abi.encodePacked(
            '{"name": "EBMP", "description": "EBMP", "image":
            "data:image/svg+xml;base64,',
            Base64.encode(bytes(EBMP(img))),
            '"}'
        )
    )));
string memory ret =
    string(abi.encodePacked("data:application/json;base64,", json));
```

Where RET would be interpretable by common NFT platforms like OpenSea [9].

3 ANALYSIS

We first present an analysis of popular on-chain image-encoding protocols to show that EBMP is the most flexible, then present gas and encoding-length analysis of EBMP (instantiated on 32x32 resolution, RGB) against one of the most permissible on-chain image encoding protocol, Pixelations [10].

Since BMP is one of the simplest image formats with a rather strict implementation requirement, Brotchain independently has developed a similar encoding method to EBMP to render bitmaps on the Ethereum blockchain [4]. However, some important differences remain:

(1) Brotchain adopts a palette-based encoding for BMP, which, although saves the number of bytes needed to store an image on-chain, limits the number of possible colors presentable in an image to the palette used.
(2) Additionally, the bytes stored on blockchain do not exhibit spatial locality in the RGB color space, making further image manipulation harder.
(3) Finally, a prototype smart contract has been developed with a greyscale version of EBMP around the same time that Brotchain is released. Users may interact with the protocol at rinkeby.0xyi.xyz.

Table 1 presents a brief and non-exhaustive overview of existing methods that aim to provide a general image encoding method on the Ethereum blockchain. Anonymice [1], and Chain Runners [2] are not included in this particular table due to the observation that the smart contract is not designed nor intended to encode images across a wide variety of domains, but instead focus solely on the very pattern on the non-fungible-token it seeks to support. Though similar is true for Nouns [6], the grouping color technique along with run-length encoding (RLE) does present an opportunity to be extended beyond encoding of the Nouns character; hence it is included in the table. We observe that EBMP offers a good balance between protocol freedom and data availability, where existing protocols either only supports a fixed number of colors or image resolution, or the data it provides is not always available (here, "Calldata" refers to the fact that the image data is only visible as input arguments to a function called on the blockchain, thus making it inaccessible for other functions running on the EVM).

Table 1. Comparison of Popular On-Chain Image Encoding Methods

Protocol	Type of data	Resolution	Number of bytes	Data availability	Color space
EBMP	Bitmap	Arbitrary (div. by 4)	$3wh$	Storage	RGB
Blitmap [3]	Bitmap	32x32	268	Storage	4 colors
Pixelations [10]	Bitmap	32x32	736	Storage	32 colors
Nouns [6]	Bitmap/Vector	32x32	RLE	Storage	Varied
Brotchain [4]	Bitmap	Arbitrary	$768 + wh$	Storage	256 colors
0xmons [14]	GIF	Arbitrary	up to 125 KB	Calldata	Arbitrary

Table 2 records the average encoding length and gas cost for random 32x32 RGB images on a local emulation of EVM across different protocols. We observe that when compared to existing methods that offer general-domain image encoding support off-the-shelf, EBMP provides a 2.8x reduction in terms of gas consumption and 1.4x reduction in encoding length even though using a far-richer color space with more bytes needed. As a separate analysis, we also include gas profiling results for Brotchain. We remark that although it produces both a shorter encoding string and a lower gas cost, EBMP is capable of representing up to 2^{24} colors in RGB format, which is $2^{16} = 65536$ times more than that of Brotchain. Additionally, since Brotchain does not exhibit spatial locality of data, resharding of raw pixel bytes is no longer needed, which will also contribute to a lower gas cost. We verify this by turning off resharding for EBMP as well and re-run the test, which indeed generated even lower gas cost and deployment cost.

Table 2. Gas and average encoding length of EBMP against Pixelations on 32x32 Resolution

Protocol	Gas	Encoding Length	Bytes	Deployment Cost	Spatial Locality	Colors
EBMP	4522204	4168	3072	549181	**Yes**	**2^{24}**
Pixelations	13390461	58575	**736**	1889790	**Yes**	2^5
Brotchain	437448	**2826**	1792	799231	No	2^8
EBMP (no resharding)	**289851**	4168	3072	**413049**	No	**2^{24}**

4 BROADER IMPACTS

It is still worthy of mentioning that although EBMP has achieved significant cost reduction against existing methods, it is still orders of magnitude larger when compared to traditional IaaS and SaaS service providers. Table 3 presents such an analysis, where we derive an upper bound for platform running EVM emulators (instead of a blockchain) by assuming a single EBMP call finishes in 2 seconds (which is true for all platforms compared).

Table 3. Analysis of EBMP against computing platform

Platform	Cost	Uptime	Data Integrity	Censorship Resistance
Ethereum [1]	\$1904.75	1 (1.07 PH/s POW) [2]	Always (POW)	If set up correctly [3]
Avalanche (EVM compat.) [4]	\$35.06	1 (validators' POS)	Always (POS)	If set up correctly
GCS (n2-highmem-8) [5]	$<$ \$0.0002912	likely $\geq$ 99.5% [6]	SLA	If they want to
AWS (t4g.2xlarge) [7]	$<$ \$0.0001494	likely $\geq$ 99.99% [8]	SLA	If they want to
Paperspace C6 [9]	$<$ \$0.00008889	likely $\geq$ 99.99% [10]	SLA	If they want to
my M1 MacBook Pro [11]	$<$ \$0.0001268	me	also me	N/A

[1] at the time of writing, Ethereum's price is \$2832.87

[2] POW stands for Proof-of-Work, and POS stands for Poof-of-Stake, which are different forms of achieving consensus on blockchains. Ethereum and Avalanche guarantee full uptime and data integrity given that consensus is achieved, which is theoretically breakable under a series of scenarios that are outside the scope of this paper.

[3] If the set-up to run EBMP on blockchains is incorrect, then it may not be censorship resistant.

[4] at the time of writing, Avalanche's price is \$80.44

[5] on-demand hourly price for this instance at the time of writing is \$0.52405

[6] https://cloud.google.com/compute/sla

[7] on-demand hourly price for this instance at the time of writing is \$0.26880

[8] https://aws.amazon.com/legal/service-level-agreements/

[9] on-demand hourly price for this instance at the time of writing is \$0.16

[10] https://www.paperspace.com/security

[11] assuming a laptop lasts for a year, and the cost of purchasing a M1 MacBook Pro is \$1999.00.

5 CONCLUSION

EBMP is a gas-efficient way to encode bitmaps on Ethereum Virtual Machines, which has implications in non-fungible-token implementations running on popular blockchains. Future improvements could be made on different resharding methods and extending the protocol to allow encoding of images with length or width not divisible by 4.

ACKNOWLEDGMENTS

We thank Diana from A-SOUL for the emotional support. You should definitely check out some of their wonderfully-made music videos:
https://www.bilibili.com/video/BV1vQ4y1Z7C2
https://www.bilibili.com/video/BV1FX4y1g7u8/

REFERENCES

[1] Anonymice. 2021. *Anonymice.* https://opensea.io/collection/anonymice
[2] Mega City. 2021. *Chain Runners.* https://opensea.io/collection/chain-runners-nft
[3] dhof. 2021. *Blitmap.* https://opensea.io/collection/blitmap
[4] divergence. 2021. *Brotchain.* https://opensea.io/collection/brotchain
[5] W. Entriken, D. Shirley, J. Evans, and N. Sachs. 2018. EIP-721: Non-Fungible Token Standard. *Ethereum Improvement Proposals* no. 721 (Jan. 2018). https://eips.ethereum.org/EIPS/eip-721
[6] Nouns Foundation. 2021. *Nouns.* https://opensea.io/collection/nouns
[7] Ron Gery. 1992. DIBs and Their Use. https://docs.microsoft.com/en-us/previous-versions/ms969901(v=msdn.10)?redirectedfrom=MSDN
[8] Georgios Konstantopoulos. 2021. *foundry.* https://github.com/gakonst/foundry
[9] OpenSea. 2022. OpenSea, the largest NFT Marketplace. https://opensea.io/
[10] Pixelations.xyz. 2021. *Pixelations.* https://opensea.io/collection/pixelations-xyz
[11] Solidity Team. 2022. *Solidity.* Ethereum Foundation. https://docs.soliditylang.org/en/v0.8.13/
[12] Vyper Team. 2022. *Vyper.* https://vyper.readthedocs.io/en/stable/
[13] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.
[14] xmon.eth. 2021. *0xmons.* https://opensea.io/collection/0xmons-xyz

A SOFTWARE ARTIFACTS

The software artifacts to reproduce the experiments in the paper, including an additional copy of the EBMP protocol, can be found here: https://github.com/0xmostima/EBMP. It is implemented in Solidity, and Foundry [8].

B SAMPLE BASE64 IMPLEMENTATION IN SOLIDITY

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

/// [MIT License]
/// @title Base64
/// @notice Provides a function for encoding some bytes in base64
/// @author Brecht Devos <brecht@loopring.org>
library Base64 {
    bytes internal constant TABLE =
        "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";

    /// @notice Encodes some bytes to the base64 representation
    function encode(bytes memory data) internal pure returns (string memory) {
        uint256 len = data.length;
        if (len == 0) return "";
        // multiply by 4/3 rounded up
        uint256 encodedLen = 4 * ((len + 2) / 3);
        // Add some extra buffer at the end
        bytes memory result = new bytes(encodedLen + 32);
        bytes memory table = TABLE;
        assembly {
            let tablePtr := add(table, 1)
            let resultPtr := add(result, 32)
            for {
                let i := 0
            } lt(i, len) {

            } {
                i := add(i, 3)
                let input := and(mload(add(data, i)), 0xffffff)
                let out := mload(add(tablePtr, and(shr(18, input), 0x3F)))
                out := shl(8, out)
                out := add(
                    out,
                    and(mload(add(tablePtr, and(shr(12, input), 0x3F))), 0xFF)
                )
                out := shl(8, out)
                out := add(
                    out,
                    and(mload(add(tablePtr, and(shr(6, input), 0x3F))), 0xFF)
                )
                out := shl(8, out)
                out := add(
                    out,
                    and(mload(add(tablePtr, and(input, 0x3F))), 0xFF)
                )
                out := shl(224, out)
                mstore(resultPtr, out)
                resultPtr := add(resultPtr, 4)
            }
            switch mod(len, 3)
                case 1 {
                    mstore(sub(resultPtr, 2), shl(240, 0x3d3d))
                }
                case 2 {
                    mstore(sub(resultPtr, 1), shl(248, 0x3d))
                }
            mstore(result, encodedLen)
        }
        return string(result);
    }
}
```