ColorFloat: Constant space token coloring

Ryan Zarick[1] Bryan Pellegrino Isaac Zhang[1] Thomas Kim Caleb Banister
LayerZero Labs Ltd.

Abstract

We present ColorFloat, a family of O(1) space complexity algorithms that solve the problem of attributing (coloring) fungible tokens to the entity that minted them (*minter*). Tagging fungible tokens with metadata is not a new problem and was first formalized in the Colored Coins protocol. In certain contexts, practical solutions to this challenge have been implemented and deployed such as NFT. We define the *fungible token coloring problem*, one specific aspect of the Colored Coins problem, to be the problem of retaining fungible characteristics of the underlying token while accurately tracking the attribution of fungible tokens to their respective minters. Fungible token coloring has a wide range of Web3 applications. One application which we highlight in this paper is the onchain yield-sharing collateral-based stablecoin.

1 Introduction

The Colored Coins protocol [1, 2] is a well-known protocol to "mark" tokens with metadata attributes. In this paper, we focus on solving one particular aspect of the Colored Coins problem that, to our knowledge, has no existing practical onchain solution. We define the *fungible token coloring problem* as the problem of attributing (coloring) a fungible token to the *minter* that minted it and tracking this color as the token is transferred to different *users'* wallets. The Colored Coins protocol provides a theoretical solution to this problem by losslessly tracking, per-wallet, a mapping of token quantities to colors. However, this hypothetical implementation would require O(N) storage *per-wallet* for a system of N minters, and O(N) computation to iterate over this storage per transaction. Thus, the Colored Coins protocol cannot solve this problem in a way that scales to many minters. In this paper, we present a class of scalable, storage-efficient algorithms to attribute fungible tokens to minters: *ColorFloat*.

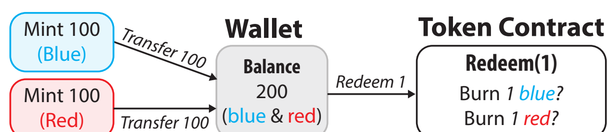

Figure 1: Existing fungible token contracts cannot fairly update attribution during redemption.

We first formally define the entities involved in the fungible token coloring problem. The *minter* generates demand for *users* to purchase fungible tokens while the per-chain *token contract* facilitates the minting, burning and/or transacting of these tokens within a given blockchain. Each minter is assigned a unique *color* (e.g., a unique numeric ID), which identifies it within the chain. In turn, any tokens attributed to this minter are associated with this color. The token contract tracks the *mint* of each minter, or the number of tokens that are attributed to that minter within a given chain. Additionally, the *vault contract* aggregates the mint across all domains (i.e., all blockchains) which we term the *circulation*. The mint of a specific color c within the set of all valid colors $\mathbf{C}$ is notated $mint_c$, and the circulation of c is notated $circulation_c$. We define the *attribution* of a color c as the proportion of $circulation_c$ to the total circulation:

$$attribution_c = \frac{circulation_c}{\sum_{x \in C} circulation_x}$$

The goal of ColorFloat is to fairly and accurately track $attribution_c$ for all colors c in the system. Within this setting, the challenges are threefold: (1) tracking the global quantity of tokens of each color, (2) propagating this information as tokens are transferred between wallets, and (3) facilitating redemption of colored tokens.

We illustrate these three challenges in the example in Figure 1. Two minters, blue and red, mint 100 tokens

[1] Inventors at LayerZero Labs Ltd.
Copyright © 2023 LayerZero Labs Ltd. All rights reserved.

each, resulting in an attribution of 100 tokens for each color. Both minters then send all 100 of their newly minted tokens to a single user's wallet, and the user redeems 1 of these 200 tokens at the token contract. The token contract must reduce the attribution of minted tokens to reflect the reduction in the global supply of tokens, but the fungibility of the tokens makes it impossible to decide which colored attribution to slash.

2 Lossy token color encoding

We postulate that a practical solution to the fungible token coloring problem should have constant storage-complexity, and thus focus primarily on exploring constant space algorithms to track token coloring. Any constant space-complexity solution to the fungible token coloring problem must be lossy, meaning information about all but a fixed number of colors will be lost during each token transfer. We propose a class of constant space-complexity fungible token coloring algorithms, each notated as ColorFloat_K where K is the number of colors that are encoded losslessly in each transaction. In this section, we begin by presenting the algorithm for ColorFloat_1 (or just "ColorFloat"), then generalize it to arbitrary K. We argue that ColorFloat_1 is the most practical point in this tradeoff space.

2.1 Token contract

Minting and transferring of colored fungible tokens is controlled by the *token contract*, which defines six functions: $Mint$, $Burn$, $Wrap$, $Unwrap$, $Transfer$, and $Cleanup$. $Mint$ and $Burn$ (Figure 2a) control $mint_C$ by incrementing or decrementing it respectively.

In addition to tracking $mint_C$, the token contract tracks what we term the *float* of each color $float_C$. To implement lossy color encoding, we introduce the concept of a *float token*, which can conceptually be thought of as a colored token that has been wrapped into an *uncolored* token. This is the mechanism by which we implement lossy color encoding, converting a set of colored tokens into float tokens to be represented as a single quantity. $Wrap$ and $Unwrap$ (Figures 2b and 2c respectively) control the wrapping and unwrapping of float tokens, incrementing and decrementing the float balance respectively of the specified color. $Wrap$ is always invoked on a finite quantity of colored tokens, thus limiting the value of $float_C$ to be less than or equal to $mint_C$. $Defloat$ (Figure 2d) implements a mechanism to choose colors to unwrap.

Finally, the token contract implements $Debit$ and $Credit$ to facilitate token transfers (Figures 2e and 2f respectively) between users. Transfers introduce entropy into the system in the form of float tokens, while slashing

1: **procedure** MINT(C, q)
2: $\quad mint_C \leftarrow mint_C + q$

1: **procedure** BURN(C, q)
2: $\quad mint_C \leftarrow mint_C - q$

(a) $Mint$ and $Burn$ respectively increase or decrease $mint_C$.

1: **procedure** WRAP(C, b)
2: $\quad float_C \leftarrow float_C + b$
3: $\quad$ **return** b

(b) $Wrap$ increments $float_C$ and returns the wrapped quantity.

1: **procedure** UNWRAP(C, f)
2: $\quad q \leftarrow min(float_C, f)$
3: $\quad float_C \leftarrow float_C - q$
4: $\quad$ **return** q

(c) $Unwrap$ accepts a quantity of float tokens as input and decrements the relevant color's float balance.

1: **procedure** DEFLOAT(f)
2: $\quad colors \leftarrow []$
3: $\quad$ **while** $f < 0$ **do**
4: $\quad\quad C_{rand} \leftarrow$ random color
5: $\quad\quad f \leftarrow Unwrap(C_{rand}, f)$
6: $\quad\quad colors \leftarrow colors + [C_{rand} q]$
7: $\quad$ **return** $colors$

(d) $Defloat$ unwraps float tokens to be burned, returning the array of unwrapped tokens.

1: **procedure** DEBIT(q)
2: $\quad$ **if** $b_{local} + f_{local} \geq q$ **then**
3: $\quad\quad d_f \leftarrow min(q, f_{local})$
4: $\quad\quad b_{local} \leftarrow b - (q - d_f)$
5: $\quad\quad f_{local} \leftarrow f - d_f$
6: $\quad\quad$ **return** $C_{local}(q - d_f)|d_f$
7: $\quad$ **else**
8: $\quad\quad$ Revert

(e) $Debit$ subtracts tokens from the sender's balance, exhausting float tokens before debiting colored tokens.

1: **procedure** CREDIT($Cb|f$, $policy$)
2: $\quad$ **if** $C_{local} = C$ **then**
3: $\quad\quad b_{local} \leftarrow b_{local} + b$
4: $\quad\quad f_{local} \leftarrow f_{local} + f$
5: $\quad$ **else if** $policy = \texttt{self}$ **then**
6: $\quad\quad f_{local} \leftarrow f_{local} + Wrap(Cb|f)$
7: $\quad$ **else if** $b_{local} \geq b$ **then**
8: $\quad\quad f_{local} \leftarrow Wrap(C, b) + f + f_{local}$
9: $\quad$ **else**
10: $\quad\quad f_{local} \leftarrow Wrap(C_{local}, b_{local}) + f + f_{local}$
11: $\quad\quad C_{local} \leftarrow C$
12: $\quad\quad b_{local} \leftarrow b$

(f) $Credit$ credits the receiver, wrapping tokens based on the specified policy.

Figure 2: Token contract methods.

$Defloat(10)$	Token	Contract
Initial state	$mint_{C_1} : 10$	$float_{C_1} : 4$
	$\cdots$	$\cdots$
	$mint_{C_2} : 10$	$float_{C_2} : 8$
$C_1 \leftarrow$ random color		
$Unwrap(C_1, 10) \rightarrow 4$	$mint_{C_1} : 10$	$float_{C_1} : 0$
	$\cdots$	
	$mint_{C_2} : 10$	$float_{C_2} : 8$
$C_2 \leftarrow$ random color		
$Unwrap(C_2, 6) \rightarrow 6$	$mint_{C_1} : 10$	$float_{C_1} : 0$
	$\cdots$	
	$mint_{C_2} : 10$	$float_{C_2} : 2$
Colors: $[(C_1, 4), (C_2, 6)]$	$mint_{C_1} : 10$	$float_{C_1} : 0$
	$\cdots$	
	$mint_{C_2} : 10$	$float_{C_2} : 2$

Figure 3: $Defloat$ unwraps the float of randomly chosen colors to fulfill the request.

balances that include float tokens reduces entropy in the system. The key idea of ColorFloat is to penalize minters with high float balances (i.e., high entropy contribution) by increasing the expected value of tokens unwrapped to their color during redemption. Minters can easily reduce their entropy by unwrapping float tokens they receive.

2.2 Transfer algorithm

We leverage the float token to decrease the space complexity of each token transfer from $O(N)$ to $O(1)$. ColorFloat encoded balances are represented by a 3-tuple of [main color identifier (C), main color balance b, float balance f], notated as $Cb|f$. Each balance losslessly stores the balance of the main color, but lossily consolidates the balances of all other colors via the float token. On every transfer, the recipient's recoloring policy decides a *single* color to track losslessly, and wraps all other colored tokens into float tokens.

Transfers are mediated by the token contract through the $Transfer$ method. For illustration purposes, we define the $Transfer$ method as the sequential invocation of the *Credit* and *Debit* subroutines in a single atomic transaction (Figures 2e and 2f). Each transfer involves two parties, a sender and receiver, with the sender transferring some quantity q of tokens to the receiver. *Debit* then subtracts q tokens from the sender's balance. This operation first draws f tokens from the float balance, then, if the float balance is exhausted, it consumes b tokens from the main color balance for a final debited balance of $(c_s b|f)$. *Credit* runs after *Debit*, receiving as arguments the debited balance $c_s b|f$ and a policy for resolving color conflicts: `self` or `float-minimized`. For brevity, we specify only these two policies in this paper, but any policy desired can be implemented by modifying the implementation of *Credit*. If the main color of the debited balance matches the main color of the receiver, the balances are merged as-is (shown in Figure 2f lines 3–4). If the main colors differ and the conflict resolution policy is set to `self`, *Credit* wraps the entire debited balance to float before adding it to the receiver's balance (shown in line 6). If the conflict resolution policy is set to `float-minimized`, the color with the larger balance is chosen to replace the receiver's color, with the smaller balance wrapped into float as illustrated in lines 7–12.

Crediting and debiting will never wrap more tokens of a color than the total mint of that color, which enforces the invariant that $\forall C, float_C \leq mint_C$.

When a token balance $(Cb|f)$ is burned, the main color can be burned by calling $Burn(C, b)$. However, the float tokens have no color and thus cannot be burned with the *Burn* function. $Defloat$ (Figure 2d) facilitates burning of float tokens by unwrapping them into random colors. The key observation that makes this scheme fair is that a high float balance for a given color implies a high rate of attrition for that token. Therefore, minters should be incentivized to minimize this introduction of entropy by creating demand for users to configure that minter's color as their main color on the token contract. As such, random selection of colors to unwrap float tokens into results in a proportional relationship between a minter's float balance and the expected value of tokens chosen by $Defloat$.

Figure 2d lines 3–6 show this process when unwrapping f float tokens. $Defloat$ randomly selects a color, unwraps up to f tokens into that color, and repeats until all f tokens have been unwrapped. After all f float tokens are unwrapped into colored tokens, the main balance and the newly unwrapped tokens are burned. The random color selection on line 4 can be implemented onchain by using a pseudorandom hash of some user and/or transaction-specific metadata (e.g., using the hash of the user's public key and transaction parameters), which we illustrate in Figure 3. After calculating which colors should be burned in which quantities, the tokens can be burned in the same transaction. Note that $Unwrap$ will never unwrap more tokens of a color C than $float_C$, as shown in Figure 2c line 2. An example of ColorFloat is shown in Figure 4, with three users exchanging and redeeming two colors of tokens (blue and pink).

ColorFloat$_K$ is a generalization of ColorFloat$_1$ that encodes balances as a (2k+1)-tuple encoded as $C_1 b_1 | C_2 b_2 | \ldots | C_k b_k | f$. In ColorFloat$_K$, debit must enable users to specify which colors to debit after the float is exhausted, and credit must implement a more expressive policy for resolving color conflicts. For example, debit can accept a ranked list of colors to preferentially subtract from, and credit could losslessly encode the

Action	Alice	Bob	Carol	Mint	Float
Alice mints 80 blue tokens	B80|0			B : 80	
Alice transfers 40 tokens to Bob	B40|0	B40|0		B : 80	
Carol mints 80 pink tokens	B40|0	B40|0	P80|0	B : 80 P : 80	
Bob transfers 10 tokens to Carol (B10 converted to float)	B40|0	B30|0	P80|10	B : 80 P : 80	B : 10
Carol transfers 30 tokens to Alice (P20|10 debited, B0|30 credited)	B40|30	B30|0	P60|0	B : 80 P : 80	B : 10 P : 20
Alice unwraps 20 float into blue (10 blue and 10 pink randomly chosen)	B60|10	B30|0	P60|0	B : 80 P : 80	B : 0 P : 10
Bob burns 20 tokens (all blue)	B60|10	B10|0	P60|0	B : 60 P : 80	B : 0 P : 10
Alice burns 10 tokens (10 float automatically unwrapped to pink)	B50|0	B10|0	P60|0	B : 60 P : 70	B : 0 P : 0

Figure 4: Three parties minting, exchanging, unwrapping, and burning tokens. Defloating is marked in red.

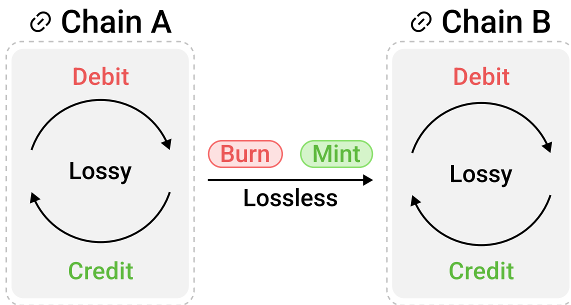

Figure 5: Transfers within a chain are lossy, but transfers between chains are lossless.

K largest balances and wrap the remaining colors into float tokens. The generalization of ColorFloat$_1$ to ColorFloat$_K$ depends largely on domain-specific implementation details (recoloring policy) so we omit a formal discussion of ColorFloat$_K$ for brevity.

Crosschain token transfers must be lossless, as cost constraints prevent float balances from being synchronized across multiple chains. This, in turn, makes it impossible to fairly burn tokens that have been transferred to a different chain. We suggest that crosschain token transfers losslessly encode a fixed number of colored tokens, which can be dynamically updated to reflect a set of minters who have the highest circulation across both chains in the transaction. When tokens of a given color are transferred from a source chain to a destination chain. However, there is a reduction in the mint on the source chain with a corresponding increase in the mint on the destination chain; however, the *circulation* of the token does not change. Conceptually, crosschain transfers are very simple: debit and burn a non-float balance on source, then mint and credit the corresponding balance on destination (illustrated in Figure 5). To convert a balance with a nonzero float to a non-float balance, the user can *Unwrap* any float tokens they hold to their wallet's main color. Crosschain transfers can be facilitated by atomically moving tokens from the source chain to the destination chain with instant guaranteed finality using a messaging protocol such as LayerZero [3].

3 Conclusion

We presented the ColorFloat family of algorithms, solving the fungible token coloring problem by implementing lossy– but fair– attribution of tokens to minters. The constant space complexity of ColorFloat enables the practical onchain application of these algorithms. Multiple different-colored tokens are lossily aggregated into a single quantity by wrapping them into *float* tokens, introducing entropy as less-desirable tokens are wrapped into float tokens. These token colors that introduce entropy into the system are fairly penalized when tokens are burned, with colors that were wrapped in larger quantities more likely to have large portions of their mint slashed during token burning. We believe this algorithm can be used in a variety of blockchain applications by allowing multiple minters to gain fair attribution while contributing value to the same token ecosystem.

References

[1] Assia, Y., Buterin, V., LiorhakiLior, M., Rosenfeld, M., and Lev, R. Colored coins whitepaper. https://www.etoro.com/wp-content/uploads/2022/03/Colored-Coins-white-paper-Digital-Assets.pdf.

[2] Rosenfeld, M. Overview of colored coins. https://bitcoil.co.il/BitcoinX.pdf, 2012.

[3] Zarick, R., Pellegrino, B., and Banister, C. Layerzero: Trustless omnichain interoperability protocol. https://layerzero.network/pdf/LayerZero_Whitepaper_Release.pdf.