ColorTrace: Fungible token coloring and attribution

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

Abstract

We formally define the *fungible token coloring* problem of attributing (coloring) fungible tokens to originating entities (*minters*), and present, to our knowledge, the first practical onchain algorithm to solve it. Tracking attribution of colored tokens losslessly using existing approaches such as the Colored Coins protocol is computationally intractable due to the per-wallet storage requirements growing in proportion to the number of minters. Our first contribution is an elegant solution to the single-chain token coloring problem, where colored tokens are atomically burned and minted to ensure each wallet only contains tokens of a single color. Our second contribution is an extension to this single-chain token coloring algorithm to allow safe and efficient crosschain token transfers. We present *ColorTrace*, an onchain algorithm to achieve globally consistent, economically feasible, fungible token coloring.

1 Introduction

Current decentralized finance (DeFi) applications often rely on fungible tokens (e.g., stablecoins) to provide an easy-to-use, reliable, and trustworthy payment mechanism. However, there exists a key deficiency with current fungible tokens: once they are issued (*minted*), there is no practical onchain algorithm to keep track of the entity (*minter*) that minted them. This makes it impossible to (1) track the origin of each token as they are transferred and redeemed, and (2) proportionally reward minters for minting tokens. We formalize this dilemma as the *fungible token coloring* problem of "coloring" fungible tokens to associate each token to the entity that originally minted it, and present a novel algorithm to solve this problem.

[1] Inventors at LayerZero Labs Ltd.

Fungible token coloring covers the majority of the Colored Coins problem [1, 3], with the exception of singleton metadata attributes which have already been solved by NFTs [2]. Applying the Colored Coins protocol to the fungible token coloring problem would losslessly map tokens to minters in every wallet, which is impractical due to two fundamental computational constraints: onchain storage and compute cost. Even within a single blockchain, it is prohibitively expensive to losslessly maintain the per-minter attribution for every wallet balance. A system of N minters would result in $O(N)$ storage complexity *per wallet* and $O(N)$ computational complexity to iterate over both the sender and receiver wallets every time tokens are transferred. Therefore, any practical token coloring algorithm must be lossy.

Our algorithm, *ColorTrace*, is the first to solve the fungible token coloring problem in O(1) storage complexity. This is achieved by requiring all transfers to *recolor* tokens such that the receiver wallet balance and sent tokens are of the same color. In addition to solving the fungible token coloring problem in a single-chain context, we solve the additional economic and safety challenges to extend ColorTrace to support provably-safe crosschain token transfers. The safety of crosschain transactions is guaranteed by the *delta-zero* invariant, a single invariant which we use to formally prove the safety and validity of ColorTrace's crosschain recoloring methods.

Crosschain token coloring opens new opportunities in the Web3 ecosystem, and this paper focuses on one obvious and powerful example: the collateral-based stablecoin. Under the current paradigm, stablecoins are issued in exchange for collateral, and the collateral is used to purchase yield-bearing assets. Many minters (e.g., applications) generate demand for these stablecoins by providing services for users, yet the token issuer is unable to proportionally reward minters due to the lack of a mechanism to track the color of each fungible token. Token coloring enables the issuer to proportionally share yield with minters based on their quantified contribution to overall

Alice	Bob	Attribution	Wallet
mint 20	mint 10	Alice: 20	Alice: 20
		Bob: 10	Bob: 10
send 10 $\rightarrow$	receive 10	Alice: 20	Alice: 10
		Bob: 10	Bob: 20
	redeem 10	???	

Figure 1: Fair redemption requires token coloring.

demand for the stablecoin.

Without a fungible token coloring algorithm, it is impossible to track minter attribution for proportional yield sharing as we demonstrate in Figure 1. In this example, Alice and Bob mint 20 tokens and 10 tokens respectively. Alice then sells 10 tokens to Bob, where they are mixed with his existing holdings. Bob tries to redeem 10 tokens, but the fungibility of the tokens makes it impossible to decide which minter(s) to deduct attribution from.

The remainder of this paper is structured as follows: Section 2 formally defines the fungible token coloring problem, Section 3 details the required security properties of ColorTrace, Section 4 presents the algorithmic foundation of our solution, Section 5 discusses ColorTrace's countermeasures against various attack surfaces.

2 Fungible token coloring

In this section, we formalize the fungible token coloring problem and related terminology.

The *colored token* is the underlying token that is tracked by ColorTrace, and is governed by a *token contract* deployed on every chain. *Minters* generate demand for *users* to transact in colored tokens, with each minter assigned a dedicated *color* (i.e., an ID) that uniquely identifies it across all blockchains. Every colored token is mapped to exactly one color, and this mapping is tracked by the token contract. We refer to the total number of tokens of a color *within* a given chain as the *local circulation*, and the circulation of tokens across all blockchains as the *global circulation*. User wallet balances are represented as a tuple of *color* and *quantity*, with the balance of a *user* holding $holding_{user}$ tokens of color $color_{user}$ notated as $color_{user}|holding_{user}$ (e.g., $\texttt{B}|H$ for H number of Blue tokens).

The fungible token coloring problem is the challenge of associating (coloring) each token with a specific color, tracking the color of each colored token as they move between wallets, and maintaining a consistent record of the proportion of global circulation of each color to the total global circulation of all colored tokens. Many aspects of the fungible token coloring problem correspond to the Colored Coins problem, such as the association of metadata to fungible tokens and the tracking of metadata as the underlying token is transferred between users. The fungible token coloring problem specifically deals with the tracking of color metadata, where multiple tokens can be tagged with the same color, and tokens of a given color are aggregated into a single quantity. However, the fungible token coloring problem does not cover use cases requiring singleton attributes, as such applications would better be served by existing solutions such as NFTs [2].

A globally unique *vault*, deployed on the designated *primary chain*, tracks and updates the *mint* of each color, which represents the sum of circulation for a given color across all chains. This mint can be used to calculate the contribution of a given color to the global demand across all blockchains, which can, in turn, be used by the token foundation to fairly distribute yields. After a token is minted at the vault, it is transferred to the minter's wallet on the primary chain, from where it can be transferred to other chains or wallets.

As mentioned in Section 1, it is prohibitively expensive to update mint on the vault every time a token is recolored on a chain other than the primary chain. This necessitates relaxing the synchrony requirement between *mint* and *circulation*, and handling the resulting divergence in an efficiently, provably safe manner. ColorTrace achieves this by tracking the *delta* (Δ) between the current circulation and the last synchronized mint (*localMint*), and allowing minters to *remint* any positive delta to close the gap between the *circulation* and $localMint$. As we demonstrate in this paper, each token contract must store Δ_C for every color, but does not need to explicitly track $localMint_C$ or $circulation_C$. However, we expect most implementations of our algorithm to store $localMint_C$ or $circulation_C$ for better minter and user experience (e.g., circulation observability).

Finally, the fungible token coloring problem requires the colored token to be fungible from the perspective of the *end-users*, but not necessarily the minters. ColorTrace implements full fungibility in all user-facing *and* minter-facing operations, but other solutions, such as our recently published *ColorFloat* algorithm [5], may require minters to be conscious of the color of the tokens they hold.

3 Consistency and security

In this section, we formalize the security properties of ColorTrace and present the invariants we use to reason about the safety of ColorTrace transactions. All ColorTrace operations involve one or more of three independently-consistent transaction contexts (*domains*): the *vault*, *chain*, and *packet*. The *vault* is responsible for (1) safely linking underlying assets to colored tokens and (2) tracking global per-color circulation. The *chain* (sometimes referred to as *local chain*), is any blockchain in the network that executes token operations. Finally, the *packet* is a special domain that participates in two transactions–one on the *source* chain, and another on the *destination* chain. After tokens and metadata are written into the packet on the source chain, the packet domain is made accessible to the destination chain by transmitting its contents using a crosschain messaging protocol. To avoid confusion, we refer to the packet domain as *packet*, and the crosschain messaging protocol packet as *crosschain packet*. Operations that involve multiple blockchains communicate via the packet, but crosschain packet delivery is assumed to be asynchronous.

We continue to define four properties that guarantee the safety and fairness of all ColorTrace operations, and formalize the invariants we leverage to reason about the fulfillment of these properties.

Asset-circulation equivalence ensures that no operation changes the total number of tokens in circulation without a corresponding change in the holdings of underlying assets. The vault is assumed to correctly enforce the creation and destruction of tokens in exchange for underlying assets, and as such any transaction that changes the global supply of tokens assets must originate from within the vault. Thus, we enforce asset-circulation equivalence through the *global supply invariant*, which states that any operations that increase or decrease the aggregate global supply of tokens *must* be fully contained within the vault. The global supply invariant is composable, meaning a sequence of operations will fulfill the global supply invariant if each operation individually preserves the global supply invariant.

Conservation of error states that any divergence between the circulation of each color and the mint recorded at the vault is accurately propagated across every operation. When combined with asset-circulation equivalence, this guarantees the net error in the system to always be zero. This safety property is the most important, and we formalize it as the *delta-zero* invariant: $\sum \Delta = 0$ for every element of the power set of all domains in the network. More informally, there should be no domain in the system that violates the delta-zero invariant, and by extension no combination of domains in the system should have a nonzero sum of deltas.

Eventual finality states that every action taken by a user or minter will eventually be reflected in the state of the relevant token contract(s) or vault. For operations that affect only a single blockchain, eventual finality is trivially fulfilled as the entire operation is committed in a single local transaction. This property has two aspects–guaranteed *delivery* and guaranteed *execution*. Operations that involve multiple blockchains rely on lossless, only-once delivery of crosschain messages for eventual finality. Furthermore, the effects of the operation *must* eventually be applied (executed) to the state of the target domain. While delivery is largely governed by the network, execution is less trivial as asynchrony can result in situations where certain actions cannot be executed immediately after the destination domain receives the corresponding packet (see Section 4.2.1).

Mint-holding preservation requires that operations never reduce a minter's mint below the number of minted tokens they hold. This property is what guarantees fungibility for minter-facing operations; without mint-holding preservation, minters must take explicit steps to hold tokens of their own color to preserve their mint. However, as we mentioned in Section 1, the colored token is not required to be fungible for all minter-facing operations, so this fairness property is not a hard requirement for all solutions to the fungible token coloring problem. Detailed further in Section 4.2.1, this property is guaranteed by the delta-influence invariant which states that operations may only change the sign of Δ_C if actual tokens are recolored or transferred, and no operation may increase the magnitude of Δ_C by more than the number of tokens of color C the initiator of the operation holds.

4 Design

To simplify the consistency model, we assume the set of minters is synchronized across all chains. This can be implemented in several different ways, such as onboarding minters at the vault before accepting any transactions.

Each token contract must store Δ_C for every color C (colored delta) along with a value Δ_θ (Section 4.1.3) that tracks Δ imbalances between different blockchains; all Δ, including Δ_θ, are initially zero as there is no difference between $localMint$ and $circulation$, nor are there any Δ imbalances between blockchains until tokens are recolored or transferred. We refer to a positive Δ_C as a *surplus*–meaning more tokens of that color are in circulation than are accounted for by the vault–and a negative Δ_C as a *deficit*. If Δ never changes, the system is trivially consistent as there is no divergence between the state of the vault and the global circulation.

We define six token contract methods–RemintSend, Recolor, TransferSend, TransferReceive, SyncSend, and SyncReceive–and three vault methods: Mint, Redeem, and RemintReceive. Furthermore, we categorize Recolor, TransferSend, TransferReceive, Mint, and Redeem as the *coloring* layer, and the remaining methods as the *synchronization* layer. The safety properties of the coloring and synchronization layers are isolated, and any synchronization layer can be interchanged with the design we present in this paper provided it implements Remint (Section 4.2.1) while fulfilling all security properties (Section 3). The remainder of this section details

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

1: procedure REDEEM($C, q$)
2:     $mint_C \leftarrow mint_C - q$
```

Figure 2: *Mint* and *Redeem* respectively increase or decrease the mint of color C ($mint_C$) at the vault.

the implementation and safety properties of each method.

4.1 Coloring layer

The coloring layer is the layer that governs the coloring and recoloring of colored tokens, and encompasses the user-facing methods of ColorTrace. To provide the best user experience, the coloring layer is designed to use minimal, deterministic per-transaction gas at the expense of creating divergence that must be reconciled by the synchronization layer.

4.1.1 Mint and redeem

Minting issues tokens based on some value input, for example one US dollar (USD) in exchange for one colored token. Redeeming is the opposite of minting, burning tokens at the vault to represent the release or expiry of the underlying value (e.g., transfer 1 USD to a bank account). Both minting and redeeming can only occur in a local context on the primary chain, thus achieving instant finality and by extension eventual finality. Minting and redemption directly change *mint* at the vault, and correspondingly increases the balance of the minter's wallet on the primary chain–neither of these operations affect Δ, and thus trivially satisfy the delta-zero and delta-influence invariants. Finally, as both minting and redeeming occur in the vault, asset-circulation equivalence is assumed to be fulfilled by the vault implementation.

4.1.2 Recolor

To achieve O(1) storage complexity transfers, ColorTrace enforces that all transfers be conducted in a single color. However, this potentially introduces some *divergence* (increase in magnitude of Δ) when tracking circulation across transfers. If the sender and receiver of a token transfer have different colored balances, one of the balances must be recolored to match the other. While we assume in the scenario, for simplicity, that the sent quantity is recolored to the receiver's color, any policy can be implemented for choosing which balance to recolor, and the specific policy used is orthogonal to ColorTrace. When $C_S|Q$ is recolored to $C_R|Q$, Δ_{C_S} is decreased by Q and Δ_{C_R} is increased by Q. Because no other deltas change other than the sender and receiver colors on the local chain, the net change in the sum of all Δ is zero, preserving the delta-zero invariant and by extension fulfilling conservation of error. The quantity of tokens does not change and the recoloring operation is atomic, fulfilling asset-circulation equivalence and eventual finality respectively.

The initiator of the recolor operation is limited to recoloring up to the quantity of tokens they hold, implying that mint-holding is preserved as tokens held by a minter cannot be recolored unless the minter themselves recolor it. Put more formally, if the minter is holding $localMint$ tokens and a different recoloring initiator is holding K tokens of the same color, by definition the circulation *must* be at least $localMint + K$. The caller can only induce a change in circulation by up to magnitude K, bounding the final resulting circulation such that $circulation \geq localMint = holding_{minter}$.

4.1.3 Transfer

In this section, we present the token transfer algorithm in the context of a crosschain transfer although the safety properties are identical for crosschain and single-chain transfers. The crosschain transfer operation is composed of separate *send* and *receive* transactions, whereas a single-chain transfer would simply combine these two operations into the same local transaction. Transfers between blockchains involve messaging asynchrony between the debit from the sender's balance and the credit to the receiver's balance, significantly complicating the consistency (safety) model.

TransferSend is the source transaction of the crosschain transfer, and moves q tokens of color C from the source chain into the packet. Crosschain transfers must be delta-zero on both the source and destination chains. When transferring tokens of a color C, the delta-zero invariant is trivial to guarantee when $\Delta_C \leq 0$ and difficult when $\Delta_C > 0$. The reason for this is not immediately apparent, but consider the case where $\Delta_C \leq 0$; the tokens can be transferred directly without changing Δ_C, as conceptually, $circulation_C$ and $localMint_C$ are reduced by the same amount. The fact that the user is holding K tokens guarantees that $circulation_C \geq K$, and because $circulation_C = localMint_C + \Delta_C \leq localMint_C$, the user can safely infer that $K \leq circulation_C \leq localMint_C$.

However, crosschain transfers when $\Delta_\text{C} > 0$ are less straightforward, as the user cannot guarantee the local availability of minted tokens to back the entire transferred balance. The key insight that makes crosschain transfers efficient in ColorTrace is our use of Δ_θ (delta theta), a bookkeeping mechanism to track the flow of (fungible) surpluses between chains. When a surplus flows out from a chain, it is matched by an increase in

```
procedure TransferSend($C, q$)
    if $q > \Delta_C > 0$ then
        Revert
    $holding_{C_{user}} \leftarrow holding_{C_{user}} - q$
    if $\Delta_C > 0$ then
        $surplus \leftarrow min(q, \Delta_C)$
    else
        $surplus \leftarrow 0$
    $\Delta_C \leftarrow \Delta_C - surplus$
    $\Delta_\theta \leftarrow \Delta_\theta + surplus$
    $holding_{C_{pkt}} \leftarrow holding_{C_{pkt}} + q$
    $\Delta_{C_{pkt}} \leftarrow surplus$
    $\Delta_{\theta_{pkt}} \leftarrow -1 \times surplus$
```

```
procedure TransferReceive($pkt$)
    $holding_{C_{pkt}} \leftarrow holding_{C_{pkt}} - q$
    $\Delta_{C_{pkt}} \leftarrow surplus$
    $\Delta_{\theta_{pkt}} \leftarrow -1 \times surplus$
    $holding_{C_{dst}} \leftarrow holding_{C_{dst}} + q$
    $\Delta_{C_{dst}} \leftarrow \Delta_{C_{dst}} + \Delta_{C_{pkt}}$
    $\Delta_{\theta_{dst}} \leftarrow \Delta_{\theta_{dst}} + \Delta_{\theta_{pkt}}$
```

Figure 3: TransferSend moves q tokens from color C into the packet, and TransferReceive correspondingly moves it from the packet into the destination token contract.

Δ_θ, and when a surplus flows into a chain it is matched by a decrease in Δ_θ. This Δ_θ facilitates constant space crosschain transfers while maintaining the delta-zero invariant on the local chain ($\sum_{C \neq \theta} \Delta_C + \Delta_\theta = 0$). In the remainder of this paper, we refer to non-Δ_θ deltas as *colored*, in contrast to Δ_θ which can be thought of as an *uncolored* delta.

Δ_θ can continue to grow in magnitude if left alone, and a large negative Δ_θ on a chain implies many local colored surpluses cannot be reminted due to insufficient local colored deficits. We reconcile these imbalances through Δ_θ synchronization (Section 4.2.2), an operation that combines a positive Δ_θ with an equal amount of negative colored Δ (deficit) and transfers it to a different chain with a negative Δ_θ. Importantly, Δ_θ makes it simple to guarantee source chain convergence upon Δ_θ synchronization by blocking synchronizations that would increase the magnitude of Δ_θ on the source chain.

There are exactly two cases when invoking TransferSend, either $\Delta_C \leq 0$ or $\Delta_C > 0 \land q \leq \Delta_C$; all other cases are explicitly prohibited by Figure 3 TransferSend line 2. In the first case, tokens are sent directly without changing Δ_C or Δ_θ. In the second case, the surplus is sent from the source chain to the destination chain with a change in Δ_θ to compensate for the surplus outflow.

TransferSend never violates asset-circulation equivalence as it atomically removes q tokens from the source chain and adds them to the packet. Conservation of error and mint-holding are met as Δ_θ and/or Δ_C change by some quantity that is bounded by the number of transferred tokens. Importantly, both the source chain and packet meet the delta-zero invariant. Finality is instant as there exists no condition that would prevent the transfer of held tokens from the source chain into the packet.

TransferReceive is the destination transaction of the crosschain transfer, moving q tokens of color C from the packet into the destination chain along with any Δ_θ and Δ_C necessary to preserve the delta-zero invariant. Asset-circulation is met as tokens are simply moved from the packet into the destination chain and these tokens can be assumed to have safely originated from the source chain by composability of the global supply invariant. The delta-zero nature of the packet guarantees the delta-zero invariant upon delivery to the destination chain; this ensures unconditional applicability of the packet to the destination chain (eventual finality). The deltas on the destination chain only change by the corresponding deltas stored in the packet, transitively fulfilling the mint-holding and error-conservation properties.

4.2 Synchronization layer

We continue by describing the design of the synchronization layer. In this paper, we focus on presenting one specific configuration of the synchronization layer that allows crosschain reminting and omnidirectional synchronization. More concretely, RemintSend can be called on a secondary chain that is different from the primary chain where RemintReceive is invoked. Likewise, synchronization can send deltas between any two blockchains. This design was chosen for its optimality within the economic constraints of currently existing blockchains, but it is not the only viable design for the synchronization layer. It is possible to limit source and destination semantics for reminting and Δ_θ synchronization to redefine the algorithm with different attack surfaces and user experience.

4.2.1 Remint

Reminting is the mechanism by which minters (M) capture the value of additional gained circulation ($\Delta_M > 0$) in their mint, simultaneously penalizing other "victim" minters (V) who have lost circulation ($\Delta_V < 0$). To remint, a portion (q) of the positive Δ_M (surplus) for color M is matched with a victim color V that has a sufficiently negative Δ_V (deficit) such that $\Delta_M - q \geq 0 \land \Delta_V + q \leq 0$. The surplus is added to the mint of the corresponding color, and the deficit is accordingly redeemed in the same crosschain transaction with instant guaranteed finality.

Action	$wallet_{src}$	$\Delta_{\theta_{src}}$	Δ_{src}	$wallet_{pkt}$	$\Delta_{\theta_{pkt}}$	Δ_{pkt}	$wallet_{dst}$	$\Delta_{\theta_{dst}}$	Δ_{dst}
Initial	$\mathtt{B}\vert 100$		$\Delta_{\mathtt{B}}0$						
Recolor $\mathtt{B}\vert 100$	$\mathtt{R}\vert 100$		$\Delta_{\mathtt{R}}100$						
to $\mathtt{R}\vert 100$			$\Delta_{\mathtt{B}}$-100						
TransferSend	$\mathtt{R}\vert 0$	100	$\Delta_{\mathtt{R}}0$	$\mathtt{R}\vert 100$	-100	$\Delta_{\mathtt{R}}100$			
($\mathtt{R}\vert 100$)			$\Delta_{\mathtt{B}}$-100						
TransferReceive	$\mathtt{R}\vert 0$	100	$\Delta_{\mathtt{R}}0$				$\mathtt{R}\vert 100$	-100	$\Delta_{\mathtt{R}}100$
($\mathtt{R}\vert 100$)			$\Delta_{\mathtt{B}}$-100						
SyncSend	$\mathtt{R}\vert 0$	0	$\Delta_{\mathtt{R}}0$		100	$\Delta_{\mathtt{B}}$-100	$\mathtt{R}\vert 100$	-100	$\Delta_{\mathtt{R}}100$
($\Delta_\theta = 100$)			$\Delta_{\mathtt{B}}0$						
SyncReceive()	$\mathtt{R}\vert 0$	0	$\Delta_{\mathtt{R}}0$				$\mathtt{R}\vert 100$	0	$\Delta_{\mathtt{R}}100$
			$\Delta_{\mathtt{B}}0$						$\Delta_{\mathtt{B}}$-100
RemintSend	$\mathtt{R}\vert 0$	0	$\Delta_{\mathtt{R}}0$			$\Delta_{\mathtt{R}}100$	$\mathtt{R}\vert 100$	0	$\Delta_{\mathtt{R}}0$
($\Delta_{\mathtt{R}}100$, [$\Delta_{\mathtt{B}}$-100])			$\Delta_{\mathtt{B}}0$			$\Delta_{\mathtt{B}}$-100			$\Delta_{\mathtt{B}}0$

Figure 4: ColorTrace uses Δ_θ to facilitate constant space, delta-zero transfers. Nonzero Δ_θ must eventually be balanced (via syncSend and SyncReceive) to enable all minters to fully capture their remint potential.

```
1: procedure RemintSend($M, V, q$)
2:     if $q > 0 \wedge \Delta_{\mathtt{M}} - q \geq 0 \wedge \Delta_{\mathtt{V}} + q \leq 0$ then
3:         $\Delta_{\mathtt{M}} \leftarrow \Delta_{\mathtt{M}} - q$
4:         $\Delta_{\mathtt{M}_{pkt}} \leftarrow q$
5:         $\Delta_{\mathtt{V}} \leftarrow \Delta_{\mathtt{V}} + q$
6:         $\Delta_{\mathtt{V}_{pkt}} \leftarrow -q$
```

```
1: procedure RemintReceive($pkt$)
2:     if $mint_V + \Delta_{\mathtt{V}_{pkt}} \geq 0$ then
3:         $mint_M \leftarrow mint_M + \Delta_{\mathtt{M}_{pkt}}$
4:         $mint_V \leftarrow mint_V + \Delta_{\mathtt{V}_{pkt}}$
```

Figure 5: RemintSend sends a request to remint a surplus of q for color M (minter) using an equivalent deficit from color V (victim) as collateral. RemintReceive applies this request to the vault.

Multiple remints can be composed as the remint operation itself is commutative, allowing many remint operations to be encoded into the same packet for efficiency. A negative Δ_θ cannot be used as the counterparty to a surplus, so it is possible for a surplus to be un-remintable if $\Delta_\theta < 0$, which we address using Δ_θ-synchronization (Section 4.2.2).

Preserving the mint-holding guarantee requires RemintSend to never induce Δ_M to become negative nor Δ_V to become positive. Consider the case where the minter's holding is equal to $localMint_V + \Delta_V = circulation_V$ and $\Delta_V + q > 0$. A remint request that disregards this condition would reduce $localMint_V$ by $q + \Delta_V$, resulting in a final value of $localMint_V$ of $localMint_V - q < localMint_V + \Delta_V$, which in turn violates mint-holding preservation as $localMint_V - q < circulation_V$. Conceptually, a hypothetical single entity holding $circulation_V$ minted tokens could have a subset of them converted to surplus (i.e., non-minted tokens), implying that holding minted tokens does not guarantee collection of yield.

RemintSend debits the surplus from Δ_M to transfer it into the packet, then credits the corresponding deficit to Δ_V to transfer it into the packet. Note that neither M nor V is allowed to be θ. The surplus and deficits can no longer be leveraged in any manner once they have been moved to the packet domain, as these specific delta values are permanently eliminated after being moved into the vault. No tokens are actually burned, minted, nor moved in RemintSend (this occurs in RemintReceive), trivially fulfilling asset-circulation equivalence. The remint of the surplus and slashing of the deficit is reflected instantly on the source chain with the implicit assumption that the remint request will eventually be applied to the vault.

The delta-zero invariant is fulfilled by definition as no additional divergence (increase in magnitude of Δ) is introduced and the RemintSend packet must include a surplus and a corresponding deficit. RemintSend explicitly prevents the change of any $\Delta_{\mathtt{C}}$ from negative to positive or vice-versa; this property is crucial in preserving mint-holding as allowing a non-positive Δ to increase past 0 is congruent to converting $localMint$ to surplus.

RemintReceive debits the surplus from the packet context and credits it to the mint on the vault. In the same transaction, the deficit is similarly transferred from the packet domain into the vault (the packet is implicitly discarded rather than explicitly zeroed out) where it is added to the mint. The delta-zero packet transitively guarantees the delta-zero invariant on the destination chain, as the entire contents of the packet are moved into the destination token contract. No tokens are moved, minted, or redeemed, thus trivially fulfilling the global supply invariant.

However, due to the asynchronous nature of packet transmission, it is possible, in rare cases, for the system to reach a situation where a remint packet cannot be applied to the vault without reducing one of the mints below zero. There are two solutions to this problem–the first is to use a vector clock to enforce gapless happens-before execution of remint requests. The second is to aggregate the un-executable remint requests into a shared pool to be applied after the aggregated remints result in a consistent state on the vault, but due to the network overhead of vector clocks, we opt to use this second solution. An example of the above scenario is as follows:

1. Initially, $mint_{\mathtt{B}} = 100$, $mint_{\mathtt{G}} = 0$, $mint_{\mathtt{R}} = 0$
2. Recolor $\mathtt{B}|100$ to $\mathtt{G}|100$
3. Remint $\Delta_{\mathtt{G}}$ 100 against $\Delta_{\mathtt{B}}$-100 (remint **R1**)
4. Recolor $\mathtt{G}|100$ to $\mathtt{B}|100$
5. Remint $\Delta_{\mathtt{B}}$ 100 against $\Delta_{\mathtt{G}}$-100 (remint **R2**)
6. Recolor $\mathtt{B}|100$ to $\mathtt{R}|100$
7. Remint $\Delta_{\mathtt{R}}$ 100 against $\Delta_{\mathtt{B}}$-100 (remint **R3**)

Due to crosschain network asynchrony, it is not guaranteed that **R1** will be delivered to the vault before **R2**, but **R2** cannot be applied to the vault before **R1** due to the initial state $mint_{\mathtt{G}} = 0$ and **R2** requiring $mint_{\mathtt{G}} \geq 100$. In addition, if **R3** is delivered and executed before **R1** and **R2**, neither **R1** nor **R2** can ever be executed because executing **R3** results in $mint_{\mathtt{G}} = 0$ and $mint_{\mathtt{B}} = 0$, necessitating the remint pool.

An alternative design to the remint operation of the synchronization layer is to remove the ability to remint directly from secondary chains, and instead require minters to directly remint deltas from the remint pool. Reminting becomes a two-step operation under this design, with the first step synchronizing deltas to the remint pool, and the second step atomically applying a delta-zero remint request to the vault.

4.2.2 Δ_θ synchronization

As mentioned in Section 4.2.1, Δ_θ cannot be used as the counterparty for reminting, necessitating some mechanism to balance nonzero Δ_θs to ensure sufficient colored deficits are available on the local chain to facilitate reminting of surpluses. Δ_θ synchronization is used to move colored deficits from chains that have more colored deficit than surplus to chains that have more surplus than colored deficit. Without synchronization, it is possible for surpluses to be un-remintable due to lack of colored deficit on the local chain; Δ_θ is an indicator of which chains have a net colored deficit ($\Delta_\theta > 0$), and which have a net colored surplus ($\Delta_\theta < 0$).

```
1: procedure SyncSend($C, q$)
2:     if $q > 0 \wedge \Delta_{\mathtt{C}} + q \leq 0 \wedge \Delta_\theta - q \geq 0$ then
3:         $\Delta_{\mathtt{C}} \leftarrow \Delta_{\mathtt{C}} + q$
4:         $\Delta_{\mathtt{C}_{pkt}} \leftarrow -q$
5:         $\Delta_\theta \leftarrow \Delta_\theta - q$
6:         $\Delta_{\theta_{pkt}} \leftarrow q$
```

```
1: procedure SyncReceive($pkt$)
2:     $\Delta_{\mathtt{C}} \leftarrow \Delta_{\mathtt{C}} + \Delta_{\mathtt{C}_{pkt}}$
3:     $\Delta_\theta \leftarrow \Delta_\theta + \Delta_{\theta_{pkt}}$
```

Figure 6: SyncSend moves q deficit from color C into the packet, and SyncReceive correspondingly moves it from the packet into the destination token contract state.

At a high level, Δ_θ synchronization can be thought of as a separate abstraction layer to the main recoloring propagation layer. For example, it is feasible to allow direct synchronization of Δ_θ and colored Δ to the remint pool instead of between secondary chains, though we opted for the latter option for economic reasons.

SyncSend transfers d tokens worth of deficit from a color C from a source chain with a negative Δ_θ into the packet such that $d \leq \Delta_\theta \wedge d \leq |\Delta_{\mathtt{C}}|$. Because SyncSend is the source transaction it achieves instant finality. By requiring $\Delta_\theta > 0$ (Figure 6 SyncSend line 2), and only permitting colored *deficits* to be synchronized, ColorTrace guarantees *source chain convergence* where Δ_θ always approaches zero as a result of SyncSend. The delta-zero invariant is fulfilled as $\Delta_{\mathtt{C}} + d + \Delta_\theta - d = \Delta_{\mathtt{C}} + \Delta_\theta = 0$, guaranteeing that both the source chain and packet domains are delta-zero. No tokens are moved nor recolored in SyncSend, trivially fulfilling the global supply invariant. SyncSend explicitly forbids the increase in magnitude or change in sign of any $\Delta_{\mathtt{C}}$ (Figure 6 SyncSend line 2) to fulfill the delta-influence invariant. For simplicity, we define SyncSend and SyncReceive in the context of a single deficit taken from a single color, but because the delta-zero invariant is commutative a single packet may contain many composed SyncSend requests.

SyncReceive accepts the delta-zero payload from a SyncSend packet and applies it to the local token contract, increasing Δ_θ by d and increasing the deficit of $\Delta_{\mathtt{C}}$ by d. SyncSend guarantees the delta-zero and delta-influence invariants in the packet domain, so atomically moving this set of deficits and Δ_θ from the packet to the destination chain domain transitively maintains the delta-zero and delta-influence invariants on the destination chain. No tokens are moved or recolored in SyncReceive, trivially fulfilling the global supply invariant. The packet payload is unconditionally applied to the destination chain state, so finality is guaranteed so long as the packet is delivered by the network.

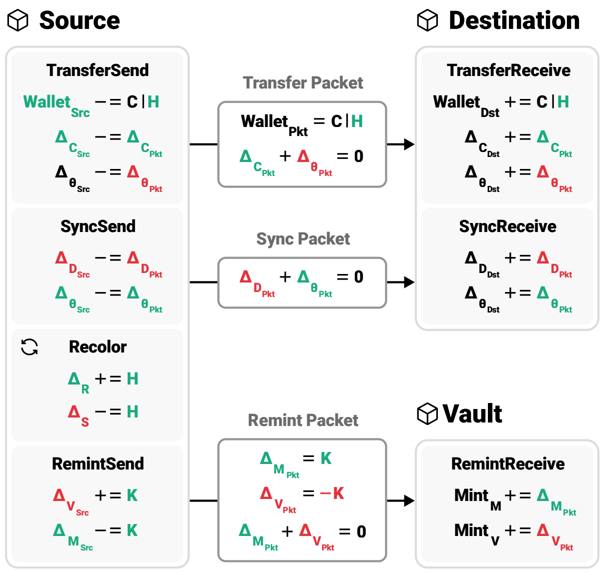
Source
Destination
TransferSend
$Wallet_{Src} -= C \mid H$
$\Delta_{C_{Src}} -= \Delta_{C_{Pkt}}$
$\Delta_{\theta_{Src}} -= \Delta_{\theta_{Pkt}}$
SyncSend
$\Delta_{D_{Src}} -= \Delta_{D_{Pkt}}$
$\Delta_{\theta_{Src}} -= \Delta_{\theta_{Pkt}}$
Recolor
$\Delta_R += H$
$\Delta_S -= H$
RemintSend
$\Delta_{V_{Src}} += K$
$\Delta_{M_{Src}} -= K$
Transfer Packet
$Wallet_{Pkt} = C \mid H$
$\Delta_{C_{Pkt}} + \Delta_{\theta_{Pkt}} = 0$
Sync Packet
$\Delta_{D_{Pkt}} + \Delta_{\theta_{Pkt}} = 0$
Remint Packet
$\Delta_{M_{Pkt}} = K$
$\Delta_{V_{Pkt}} = -K$
$\Delta_{M_{Pkt}} + \Delta_{V_{Pkt}} = 0$
TransferReceive
$Wallet_{Dst} += C \mid H$
$\Delta_{C_{Dst}} += \Delta_{C_{Pkt}}$
$\Delta_{\theta_{Dst}} += \Delta_{\theta_{Pkt}}$
SyncReceive
$\Delta_{D_{Dst}} += \Delta_{D_{Pkt}}$
$\Delta_{\theta_{Dst}} += \Delta_{\theta_{Pkt}}$
Vault
RemintReceive
$Mint_M += \Delta_{M_{Pkt}}$
$Mint_V += \Delta_{V_{Pkt}}$

Figure 7: Positive values are notated in green and negative values in red. The transfer example sends tokens of a surplus color from the source chain to the destination chain. The synchronization example illustrates synchronizing a colored deficit and positive Δ_θ. The reminting example demonstrates reminting an M-colored surplus collateralized against a V-colored deficit.

An alternative way to design the Δ_θ synchronization layer is to restrict the synchronization of deltas to be unidirectional to the remint pool on the primary chain. Under this design, it is unnecessary to guarantee source chain convergence on SyncSend and divergence or imbalance can be directly handled in the remint pool.

5 Discussion

The full architecture of ColorTrace is outlined in Figure 7, which illustrates the crosschain transfer of a surplus, Δ_θ synchronization, local recoloring, and reminting in a system of two secondary chains (source and destination), the vault, and three packets.

ColorTrace makes it possible for holding-agnostic remint batching. In a lossless token coloring scheme, an application can only remint as many tokens as they hold—that is, they must hold a token to burn it in exchange for an asset which is then used to remint a new token. Applications that create demand for very large asset flows but only hold a relatively small quantity of tokens would be forced to remint small quantities of tokens–very often under a lossless scheme–thus resulting in high crosschain messaging overheads and transaction fees. In a ColorTrace deployment, this application can simply recolor tokens as they flow into their wallet under the hope that some portion of their recolorings are preserved in the outflow. The application owner (minter) can then wait an arbitrary amount of time for an arbitrarily large surplus to accumulate before reminting. This allows minters to optimize price:benefit ratio on remint fees and capture yield proportional to their contribution to the ecosystem.

5.1 Attack surfaces and countermeasures

Note that it is possible–in ColorTrace–to violate all invariants if any of the participating blockchains is compromised or malicious. A malicious or faulty network can also potentially violate the global supply invariant, one such example being the loss of network liveness resulting in the indefinite lockup of tokens or deltas in a packet. These issues can be solved by using a permissionless, lossless network such as LayerZero [4], but the choice of network is orthogonal to this paper.

One potential attack vector is what we term *flash reminting*. A minter can gain a disproportionate distribution by acquiring a surplus for a very short period of time via a flash loan and reminting that surplus. This can be mitigated through a combination of timelocks and reminting fees. First, we enforce that tokens added to a surplus or deficit can only be reminted or slashed respectively after a timelock has expired (e.g., 1 block). Second, we tune the reminting fees to be high enough that minters are not incentivized to remint unless they are confident their mint will not be redeemed for at least a certain number of distribution periods. This is not expected to be the case for flash reminters, as their reminted surplus is likely to quickly decay into a deficit.

Another potential attack vector is what we call the fugitive deficit attack. Theoretically, an entity can consolidate their deficits along with a corresponding Δ_θ on a single chain, then SyncSend their deficit upon observing in the mempool a remint request that uses their color's deficit as a counterparty. This prevents the aforementioned remint request from slashing the attacker's deficit and can cause grief due to failed remint requests. This attack requires the entity to hold a significant amount of assets to prevent Δ_θ on the victim chain from becoming negative (and thus blocking their attempt to move deficit), but in theory would allow a single entity that controls two colors to double the yield of their underlying funds. This problem can be fixed using a threshold control mechanism to temporarily restrict the movement of specific-colored deficits, or by restricting the routing of Δ_θ flows to prevent the creation of a tight loop on a small set of low-cost blockchains. In practice, there must also be a threshold for Δ_θ synchronization, as small deficits can be created using a small amount of capital and would cost the synchronizing entity potentially large gas fees for the source and destination transactions on top of the crosschain messaging fees. In addition, the source chain and the destination chain for a synchronization request should be different, to prevent trivial griefing by attackers seeking to hide their deficit. Economic and deficit availability semantics must be carefully defined and implemented by the organization managing the token, but these details are orthogonal to this paper.

6 Conclusion

We present ColorTrace, an algorithm that implements efficient onchain attribution of fungible tokens by solving the token coloring problem. A vault on the primary chain governs the association of underlying value to onchain colored tokens and token contracts deployed to a collection of independent blockchains conduct transactions and periodically synchronize token recolorings to the vault. ColorTrace is divided into a *coloring layer* and *synchronization layer*. The coloring layer implements O(1) token transfers with predictable gas consumption for all user-facing operations, but introduces entropy into the system in the form of token recolorings and chain-local Δ imbalances (nonzero Δ_θ). This divergence in the system is resolved by the synchronization layer, which implements safe and globally consistent synchronization of recolorings to the vault. Both the coloring and synchronization layers are provably safe by ensuring the net error in the system is zero via the *delta-zero* invariant. ColorTrace opens the potential for blockchain applications to integrate more deeply with fungible tokens and enables a more equitable Web3 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] Entriken, W., Shirley, D., Evans, J., and Sachs, N. Erc-721: Non-fungible token standard. https://eips.ethereum.org/EIPS/eip-721.

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

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

[5] Zarick, R., Pellegrino, B., Zhang, I., Kim, T., and Banister, C. ColorFloat: Constant space token coloring. https://layerzero.network/publications/ColorFloat.pdf.