DiversiFi

A Simple Protocol for Tokenizing a Portfolio of Pegged Assets

DiversiFi Foundation

Abstract

On-chain pegged assets (mostly stablecoins) are a cornerstone of the DeFi ecosystem, they have also been the source of some of the most serious market disruptions and failures due to the combination of their risk profiles and their foundational role. In this paper, we introduce DiversiFi: a simple protocol for tokenizing a portfolio of pegged assets (primarily stablecoins). By tokenizing a portfolio of stablecoins, we can create a token with significantly dampened downside risk, which is much better suited to be the foundation of the DeFi ecosystem than any individual stablecoin.

1. Introduction

Stablecoins are arguably the most important collateral asset in DeFi. They are the only consistently stable asset in an extremely volatile market, which is what makes them excellent collateral. Unfortunately, they suffer from the same risk profile as most pegged assets: they are stable 99.9% of the time, but on rare occasions they can experience extreme downside volatility. This makes risk calculations for traders using them as collateral extremely complicated—so much so that most traders simply ignore the risk of a depeg scenario entirely. This dynamic has lead to some of the most catastrophic failures in DeFi history, such as the collapse of Terra's UST stablecoin, and the temporary depegging of USDC which also led to the depegging of DAI via a 1:1 exchange portal.

With DiversiFi, we aim to create a service that allows users to delegate this complex risk mitigation strategy to a DAO. DAO members stake governance tokens on what they believe to be the optimal collateral configuration. If a staker's selected configuration performs well, they are rewarded. If it performs poorly, they are penalized through a system of economic incentives. *Note:* This paper does not go into staking economics, as that system is still in development. The purpose here is to outline the mechanisms required to facilitate a collateral index token.

Our ultimate goal is to reduce the severity of depegs through automated diversification of stablecoins and other pegged assets. We aim to keep the core protocol as simple as possible, because we believe simplicity is security. That simplicity starts with this whitepaper. While there are a few seemingly complex proofs in some sections, we also explain everything in plain English that anyone can intuitively understand. Our goal is not to use a bunch of mathematical symbols to make this sound smarter or more complicated than it really is. At its core, this is an extremely simple system.

2. Core Functionality

2.1. Definitions

Before we begin to describe the protocol, we must define some basic terms.

- **Allocation**: The ratio of a specific reserve asset relative to the total reserves in the pool. The sum of allocations for all assets in the pool is 1.
- **Reserve Asset**: The asset held in the pool, in proportion to its allocation.
- **Index Token**: The token that represents the pool's reserves and is used to interact with the pool. It is minted when assets are deposited and burned when assets are withdrawn.

In a perfect world where nothing ever goes wrong, all we need to create a portfolio is a data structure to represent the allocations of each asset and a portal to wrap and unwrap reserve assets for index tokens according to those allocations. We will start with this simple case and expand our functionality to handle more complex scenarios later in this paper.

2.2. Allocation Configuration

We define a portfolio configuration as a vector of allocations:

$$\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}, \quad \text{where } a_i \in [0,1] \text{ and } \sum_{i=1}^{n} a_i = 1$$

Each a_i represents the **target allocation** of the i^{th} reserve asset. The sum of all allocations must equal 1.

To mint N units of the index token, the user must deposit reserve assets such that the deposited amounts $r_1, r_2, \ldots, r_n$ satisfy:

$$r_i = a_i \cdot N, \quad \text{for all } i = 1, \ldots, n$$

This is a fancy way of saying: if the allocation configuration is .25 A, .25 B, .25 C, and .25 D, then you need to deposit 25 tokens of each reserve asset A, B, C, and D to mint 100 index tokens.

2.3. Encoding

Allocations are stored in an array of `AssetParams` structs, as defined in `DataTypes.sol`:

```
struct AssetParams {
    address asset;
    uint88 allocation; // 88 fractional-bit fixed-point number
    uint8 decimals;
}
```

The `allocation` value represents the share of the total reserves to be held in a given asset. It is encoded as a fixed-point number with 88 fractional bits, meaning the actual value is $\frac{n}{2^{88}}$, where n is the stored integer. 88 bits were chosen specifically so that this data type uses only one 256-bit EVM slot, for maximum gas efficiency.

Example: Suppose we want to multiply a reserve amount of 500 (a regular integer) by a fixed-point allocation of 0.25. First, we encode the allocation as a fixed-point number:

$$\text{allocation\_fixed} = 0.25 \times 2^{88} = 7{,}922{,}816{,}251{,}426{,}433{,}759{,}354{,}395{,}136$$

To perform the multiplication:

$$\text{result} = \frac{500 \times \text{allocation\_fixed}}{2^{88}} \approx 125$$

Rather than dividing by 2^{88}, which is costly, we can perform a bit shift:

$$\text{result} = (500 \times \text{allocation\_fixed}) \gg 88 \approx 125$$

This is both mathematically equivalent and far more efficient in Solidity due to the lower gas cost of bit shifts compared to division.

This encoding ensures that allocations can be represented with extreme precision, while preserving computational efficiency during operations such as minting and redemption.

2.4. Minting Index Tokens

To mint index tokens, a user deposits reserve assets proportional to the configured allocations. The total required deposit for each asset is calculated by multiplying the target allocation by the mint amount and scaling by decimals. The simplified implementation is as follows:

```
function mint(uint256 mintAmount) external {
    for (uint i = 0; i < targetAssetParamsList.length; i++) {
        AssetParams memory params = targetAssetParamsList[i];
        depositAmount = params.targetAllocation * mintAmount;
        specificReserves[params.assetAddress] += depositAmount;
        IERC20(params.assetAddress).transferFrom(msg.sender, address
           (this), depositAmount);
    }
    totalReserves += mintAmount;
    indexToken.mint(recipient, mintAmount);
}
```

Note: Decimal scaling, fixed-point conversions, and many other details have been removed for simplicity in this example.

2.5. Burning Index Tokens

To burn index tokens, the pool undergoes the process of minting in reverse. However, instead of referencing the `targetAssetParams`, it uses `currentAssetParams` (because `targetAssetParams` may be missing assets that are targeted for removal but haven't been removed yet). It also uses the current allocations instead of target allocations, which are calculated on the fly. The simplified implementation is as follows:

```
function burn(uint256 burnAmount) external {
    for (uint i = 0; i < currentAssetParamsList.length; i++) {
        AssetParams memory params = currentAssetParamsList[i];
        uint256 currentAllocation = specificReserves[params.
           assetAddress] / totalReserves;
        withdrawAmount = currentAllocation * burnAmount;
        specificReserves[params.assetAddress] -= withdrawAmount;
        IERC20(params.assetAddress).transfer(msg.sender,
           withdrawAmount);
    }
    totalReserves -= burnAmount;
    indexToken.burn(recipient, burnAmount);
}
```

3. Complex Functionality

This section describes functionality for handling more complex scenarios than simple minting and burning.

3.1. Changing Allocations

Now that the reader is thoroughly confused as to why burning behaves so much differently than minting, we must discuss the reason: **changing allocations**. What if we want to change the allocations of the pool? We can't simply set them to new values—otherwise, when users want to burn their index tokens for reserve tokens, the appropriate reserves may not be available.

To solve this, we determine burn outputs not by target allocations, but by actual allocations. Actual allocations are calculated on the fly rather than being stored like target allocations.

This approach ensures that:

- Whenever liquidity is **added** to the pool, the allocation drifts **toward** the new target allocation.
- Whenever liquidity is **removed** from the pool, the allocation drifts **away** from the current allocation, which may no longer match the target.

While this approach helps steer the pool in the right direction of the target allocations, it doesn't allow us to perfectly align with them. Since we don't want to be endlessly tracking small amounts of dust tokens that won't evaporate from the pool until the heat death of the universe, another mechanism is needed to totally align the pool—**arbitrage opportunity**. Arbitrage mechanisms are discussed in the next section.

See **Proofs Section 5.1** for a proof that the pool will converge to the target allocations via normal minting and burning.

3.2. Swapping Towards Target Allocation

If an asset is above or below its target allocation, a trader may trade it with the pool directly instead of as part of the entire basket of reserve assets, but only in the direction that brings it in line with the target allocation.

If this is not enough, governance may also commit its fee revenue toward a discount or premium to incentivize this exchange even further, or arbitrage it themselves using the treasury.

See **Proofs Section 5.2** for a proof that these swaps always move the pool closer to convergence with the target allocations.

3.3. Equalization

Equalization refers to the action of bringing the pool's current allocations completely in line with its target allocations. This is done by computing an **equalization vector**, a vector of deltas between each asset's current reserves and target reserves, and applying it to the pool via transfers to and from the caller's account.

Equalization is a separate function because `swapTowardsTarget` always has secondary effects, since it affects the total reserves.

See **Proofs Section 5.3** for a proof that applying the equalization vector always results in convergence of the pool's current allocations to its target allocations.

3.4. Depeg Migration and Rebasement

Note that this section is a work in progress and may change during the testnet phase.

In the event of a depeg that cannot be recovered from via insurance, we need a migration strategy to return the pool to parity with its underlying assets in the least disruptive way possible. We deem the least disruptive method to be a gradual rebasement of the index token. That is, balances of all index token holders will be slowly lowered until they reach a point where every index token is backed by the correct amount of reserves.

- The token contract will have a **balance multiplier**. This value multiplies all token base balances to get the actual balance. For example, if the multiplier is 0.5 and my base balance is 2, my actual balance is 1. This allows us to rebase the token back to full collateralization in the event of a depeg without requiring a contract migration. The balance multiplier will start at 1.
- The token contract will have a state variable: **migration mode**.
- **Migration mode** on the token can only be toggled by the current liquidity pool.
- While migration mode is on:
  - The balance multiplier will slowly decrease at a constant rate (e.g., 0.1% per hour).
  - Migration mode ends when the previous liquidity pool signals that all of its reserves have been migrated.
- Upon migration mode ending, the token is pointed at a new liquidity pool.
- A new liquidity pool is created with a new set of assets and allocations in **immigration mode**. While in immigration mode:
  - Tokens cannot be minted.
  - An **immigration portal** allows for funds to be received in exchange for immigration credits.
- The previous liquidity pool enters **emigration mode**. While in emigration mode:

  - Tokens cannot be minted.
  - The pool signals the token contract to enter migration mode.
  - The pool auctions off its reserves in exchange for a decreasing amount of immigration credits from the new liquidity pool.
  - The conversion starts at a 1:1 rate and references the balance multiplier of the index token contract.
  - If the balance multiplier is not 1 at the start of emigration mode, the initial multiplier is saved. The current multiplier is then divided by the initial multiplier to normalize it. For example:
    * If the initial balance multiplier is 0.5, it is saved in the liquidity pool.
    * The conversion rate of immigration credits is calculated as:

$$\text{Conversion Rate} = \frac{\text{Current Multiplier}}{\text{Initial Multiplier}}$$

    * Example:
      · At start: $\frac{0.5}{0.5} = 1$
      · Two hours later: $\frac{0.4}{0.5} = 0.8$

      The pool will sell off reserves equal to 1 unit of liquidity in exchange for 0.8 immigration credits.
  - The migration ends when the previous pool has no reserves left.
- Emigration mode can be activated by the pool's admin.

3.5. Logic Migration

A logic migration refers to upgrading the pool to a new implementation that preserves the existing reserve allocations. If the new pool's target allocations exactly match the previous pool's current allocations, the migration can be completed immediately. In this case, no changes to the index token's balance multiplier are necessary, as the value of each token remains consistent between versions.

3.6. Total Reserves Growth Rate Limiting

Suppose the protocol has committed a fixed amount of revenue to an insurance fund. If a small depeg occurs, the insurance fund might be sufficient to cover up to a 5% loss in the value of the total reserves.

However, if minting remains unrestricted, users could exploit the situation by depositing depegged assets to mint index tokens—effectively draining the insurance fund.

To prevent this, the protocol enforces a rate limit on the growth of total reserves. This gives governance a time window to intervene and disable minting if a depeg is detected, limiting the impact and preserving the integrity of the insurance fund.

4. Additional Considerations

4.1. Token Decimals

Since the protocol compares allocations in terms of raw unit quantities, all tokens must be normalized to the same decimal scale. Internally, the system uses the same decimal scale as the index token, and incoming token amounts are scaled accordingly during minting and burning operations.

4.2. Fees

Fees are charged on mint and burn operations but not on actions that help realign the pool toward its target allocations. All fees are collected in the index token itself. This simplifies treasury management and allows the DAO to use collected fees directly for pool stabilization—such as buying failed assets to rebalance the pool.

A subtle detail in the fee mechanism arises from the recursive nature of mint fees. Since fees are paid in minted tokens, minting a fee requires additional minting, which itself incurs a fee, and so on. This forms a convergent geometric series:

$$\text{Fee} = \sum_{i=1}^{\infty} \text{mintAmount} \times \text{feeRate}^i$$

This series converges to the closed-form solution:

$$\text{Fee}(\text{mintAmount}, \text{feeRate}) = \frac{\text{mintAmount} \times \text{feeRate}}{1 - \text{feeRate}}$$

This formula is implemented in the function `PoolMath.calcCompoundingFee()`.

4.3. Fixed Point Math

The protocol uses 128.128 fixed point math, where 128 bits are reserved for integers and the other 128 bits are reserved for fractions. To convert a number to or from a fixed point simply multiply or divide by 2^{128}.

5. Proofs

We define the pool's **discrepancy** as the token unit difference between the pool's current reserves and its target reserves. Intuitively, if the discrepancy is zero, then the pool's reserves are exactly on target. This section contains proofs that the discrepancy can only decrease or remain the same under publicly available operations, and therefore the pool will eventually converge at its target allocations through normal activity.

5.1. Discrepancy Does Not Increase When Minting and Always Decreases When Burning

Let the pool contain n assets indexed by $i = 1, 2, \ldots, n$. Define the following for each asset i:

- R_i: current reserves of asset i
- a_i: target allocation of asset i, where $\sum_{i=1}^{n} a_i = 1$

Let the total reserves be:

$$T = \sum_{i=1}^{n} R_i$$

Define the discrepancy for each asset as:

$$D_i = |R_i - a_i T|$$

And the **total discrepancy** across all assets as:

$$D = \sum_{i=1}^{n} D_i = \sum_{i=1}^{n} |R_i - a_i T|$$

We will show that under normal minting and burning operations, the total discrepancy D does not increase.

Case 1: Minting

Suppose a user mints M index tokens. The required deposit for each asset is proportional to the target allocation:

$$\Delta R_i = a_i \cdot M$$

New reserves become:

$$R_i' = R_i + a_i M, \quad T' = T + M$$

Then:

$$D_i' = |R_i' - a_i T'| = |R_i + a_i M - a_i (T + M)| = |R_i - a_i T| = D_i$$

So:

$$D' = \sum_{i=1}^{n} D_i' = \sum_{i=1}^{n} D_i = D$$

Conclusion: Minting preserves the total discrepancy.

Case 2: Burning

Suppose a user burns B index tokens. The reserves withdrawn are proportional to current allocations:

$$\Delta R_i = \frac{R_i}{T} \cdot B$$

New reserves become:

$$R_i' = R_i - \frac{R_i}{T}B, \quad T' = T - B$$

Then:

$$D_i' = |R_i' - a_i T'| = \left| R_i - \frac{R_i}{T}B - a_i(T - B) \right| = \left| (R_i - a_i T) + B(a_i - \frac{R_i}{T}) \right|$$

Observe:

$$D_i' = \left| (R_i - a_i T) + B\left(a_i - \frac{R_i}{T}\right) \right|$$

This consists of two components:

- $R_i - a_i T$: the original signed discrepancy for asset i
- $B\left(a_i - \frac{R_i}{T}\right)$: the Burn Correction Term

Now consider the sign of each term:

Case 1: Over-allocation

$$R_i > a_i T \Rightarrow R_i - a_i T > 0$$
$$\frac{R_i}{T} > a_i \Rightarrow a_i - \frac{R_i}{T} < 0 \Rightarrow B\left(a_i - \frac{R_i}{T}\right) < 0$$

In this case, the second term is *negative*, pulling the sum closer to zero.

Case 2: Under-allocation

$$R_i < a_i T \Rightarrow R_i - a_i T < 0$$
$$\frac{R_i}{T} < a_i \Rightarrow a_i - \frac{R_i}{T} > 0 \Rightarrow B\left(a_i - \frac{R_i}{T}\right) > 0$$

Here, the second term is *positive*, again pulling the sum toward zero.

Conclusion In both cases, the correction term has the opposite sign of the original discrepancy term. Therefore, their sum has a smaller absolute value than the original discrepancy. This means:

$$|D_i'| < |D_i| \quad \text{whenever } D_i \neq 0$$

As a result, the total discrepancy $D = \sum_{i=1}^{n} |R_i - a_i T|$ strictly decreases under burning, unless the discrepancy is already zero.

Final Result: The total discrepancy $D = \sum_{i=1}^{n} |R_i - a_i T|$ never increases when minting and always decreases when burning

5.2. swapTowardsTarget Always Reduces Discrepancy

Let the pool contain n assets indexed by $i = 1, \ldots, n$, with:

- R_i: current reserves of asset i
- a_i: target allocation of asset i, where $\sum_{i=1}^{n} a_i = 1$
- $T = \sum_{i=1}^{n} R_i$: total reserves
- $D_i = |R_i - a_i T|$: discrepancy of asset i
- $D = \sum_{i=1}^{n} D_i$: total discrepancy

Suppose a call to `swapTowardsTarget` modifies a single asset j by $\Delta R_j = \delta$, where:

- $\delta > 0$: deposit
- $\delta < 0$: withdrawal

We now prove that:

$$D' = \sum_{i=1}^{n} |R'_i - a_i T'| < \sum_{i=1}^{n} |R_i - a_i T| = D$$

Step 1: Asset j Moves Toward Target

Let's examine the discrepancy of asset j after the change.

New total reserves:

$$T' = T + \delta$$

New reserve for asset j:

$$R'_j = R_j + \delta$$

New discrepancy for j:

$$D'_j = |R_j + \delta - a_j(T + \delta)|$$

Rewriting:

$$D'_j = |(R_j - a_j T) + \delta(1 - a_j)|, \quad D_j = |R_j - a_j T|$$

Now:

- $R_j > a_j T \Rightarrow \delta < 0 \Rightarrow (1 - a_j)\delta < 0 \Rightarrow$ discrepancy decreases
- $R_j < a_j T \Rightarrow \delta > 0 \Rightarrow (1 - a_j)\delta > 0 \Rightarrow$ discrepancy decreases

In both cases, the correction term moves the discrepancy toward zero:

$$|R'_j - a_j T'| < |R_j - a_j T| \Rightarrow D'_j < D_j$$

Step 2: Other Assets Are Unchanged, But Total Reserves Change

For all $i \neq j$, reserves are unchanged, but total reserves increase or decrease:

$$R_i' = R_i, \quad T' = T + \delta \Rightarrow D_i' = |R_i - a_i(T + \delta)| = |(R_i - a_i T) - a_i \delta|$$

So:

$$D_i' = |D_i - a_i \delta|$$

This means each D_i' changes by at most $|a_i \delta|$. Summing across all other assets:

$$\sum_{i \neq j} (D_i' - D_i) \leq \sum_{i \neq j} |a_i \delta| = |\delta| \cdot (1 - a_j)$$

But the change in discrepancy for asset j is:

$$D_j' - D_j = -|\delta(1 - a_j)|$$

Conclusion

The decrease in discrepancy from asset j is greater than or equal to the total potential increase in discrepancy from all other assets:

$$\sum_{i \neq j} (D_i' - D_i) < -(D_j' - D_j) \Rightarrow D' < D$$

Therefore, `swapTowardsTarget` strictly reduces the total discrepancy D.

5.3. Equalization Eliminates Discrepancy

Let the pool consist of n assets, each with:

- R_i: current reserve quantity of asset i
- a_i: target allocation of asset i, with $\sum_{i=1}^{n} a_i = 1$

Let $T = \sum_{i=1}^{n} R_i$ be the total reserves.
The **target reserves** for each asset are:

$$R_i^{\text{target}} = a_i T$$

We define the **equalization vector** as:

$$\Delta_i = R_i^{\text{target}} - R_i = a_i T - R_i$$

Applying this equalization vector means that the new reserve for asset i becomes:

$$R_i' = R_i + \Delta_i = R_i + (a_i T - R_i) = a_i T$$

The new total reserves become:

$$T' = \sum_{i=1}^{n} R_i' = \sum_{i=1}^{n} a_i T = T$$

Thus, the total reserves remain unchanged, and for every asset i, we have:

$$R'_i = a_i T' \Rightarrow \text{allocation is exactly equal to target}$$

The discrepancy for each asset is:

$$D_i = |R'_i - a_i T'| = |a_i T - a_i T| = 0$$

Therefore, the total discrepancy becomes:

$$D = \sum_{i=1}^{n} D_i = 0$$

Conclusion: Applying the equalization vector results in every asset's reserves exactly matching its target allocation, and the total discrepancy becomes zero.