Automatic Value Distribution in Decentralized Finance Protocols

Xavier Negron

September 12, 2023

Abstract

Decentralized finance protocols consist of tamper-proof verifiable code that governs the use of digital assets on a blockchain network. In order for these protocols to operate as intended, there needs to be sufficient liquidity deposited. However, users are unlikely to deposit liquidity in a decentralized finance protocol without sufficient incentives to compensate for the risks.

The concept of yield farming was invented to reward users for depositing liquidity in a protocol. Users typically receive a share of a protocol's governance token proportional to the amount of liquidity they deposit in the protocol. Existing yield farming systems depend on frequent updates from developers to distribute rewards, creating a single point of failure.

In this paper, we propose a decentralized self-sustaining yield farming system. We describe the design principles behind this system, a method for tracking performance over discrete periods, and the smart contracts for each component of the system. We also describe the algorithms used for unlocking tokens, updating weights, and distributing rewards. Next, we categorize malicious actors and describe methods to prevent them, and analyze how rewards are distributed under different edge cases. Finally, we describe how developers can integrate the farming system into their own projects.

1 Introduction

A decentralized finance - DeFi - protocol usually needs liquidity deposited to operate as intended. Due to the inherent risks of DeFi protocols, such as smart contract exploits, users are unlikely to deposit capital in a DeFi protocol unless there are sufficient incentives to compensate for the risks. One of the first DeFi protocols, Compound, invented the concept of yield farming as a way to incentivize users to deposit liquidity in the protocol.

Under Compound's yield farming system, users would receive 'LP' (liquidity provider) tokens whenever they deposit liquidity in the protocol. These tokens represent a user's share of a liquidity pair (such as ETH-USDC) and act as proof that a user deposited liquidity in the protocol. The LP tokens can be deposited into 'farms' that distribute Compound's governance tokens to users proportional to their share of a farm.

This system helped Compound bootstrap liquidity while distributing their governance token to a wider user base, leading to greater decentralization. Although this system proved successful in growing the Compound protocol, as well as other protocols using similar farming systems, it also introduced potential risks due to developers needing to manually update pool weights on a regular basis. Some of these risks include:

- Only a small subset of pools are rewarded: Existing farming systems only select the top pools to receive rewards, which may be only a small subset of all pools. This leads to a protocol's governance token being distributed to a subset of users - users invested in top pools - which can lead to monopolization of rewards. If users can stake the governance token to vote on reward distribution, they would likely vote on a high weight for the pools they're invested in, creating a self-feeding cycle. There may also be other pools in the protocol that are more qualified to receive rewards (such as having higher liquidity or being more popular amongst users) but end up not receiving rewards because a small subset of existing pools are monopolizing the rewards.

- Criteria/distribution might be subjective: There are separate criteria for selecting pools for rewards and distributing rewards amongst eligible pools. Although the criteria might be public for some protocols, they might not end up being enforced since it ultimately depends on protocol developers. This introduces the possibility of bias or censorship.
- Reliance on developers for maintenance: Developers need to manually update weights in the farming system, and refresh rewards, on a regular basis. If developers miss an update, rewards would be temporarily halted, leading to users withdrawing liquidity from the protocol. Relying on developers to regularly update a protocol creates an additional point of failure.

We propose an alternative farming system that is tailored to asset management protocols and builds upon existing farming systems. This farming system improves upon existing farming systems by being fully decentralized, tamper-proof, self-sustaining, and inclusive of all pools registered under the system. This decentralized approach allows the system to keep operating as intended without intervention from developers, while ensuring that the criteria used for selecting pools and distributing rewards will always be adhered to.

2 System Overview

The farming system consists of contracts for distributing reward tokens, a contract for registering pools and updating pool weights automatically, and a yield farming contract for each pool registered in the system by an external protocol.

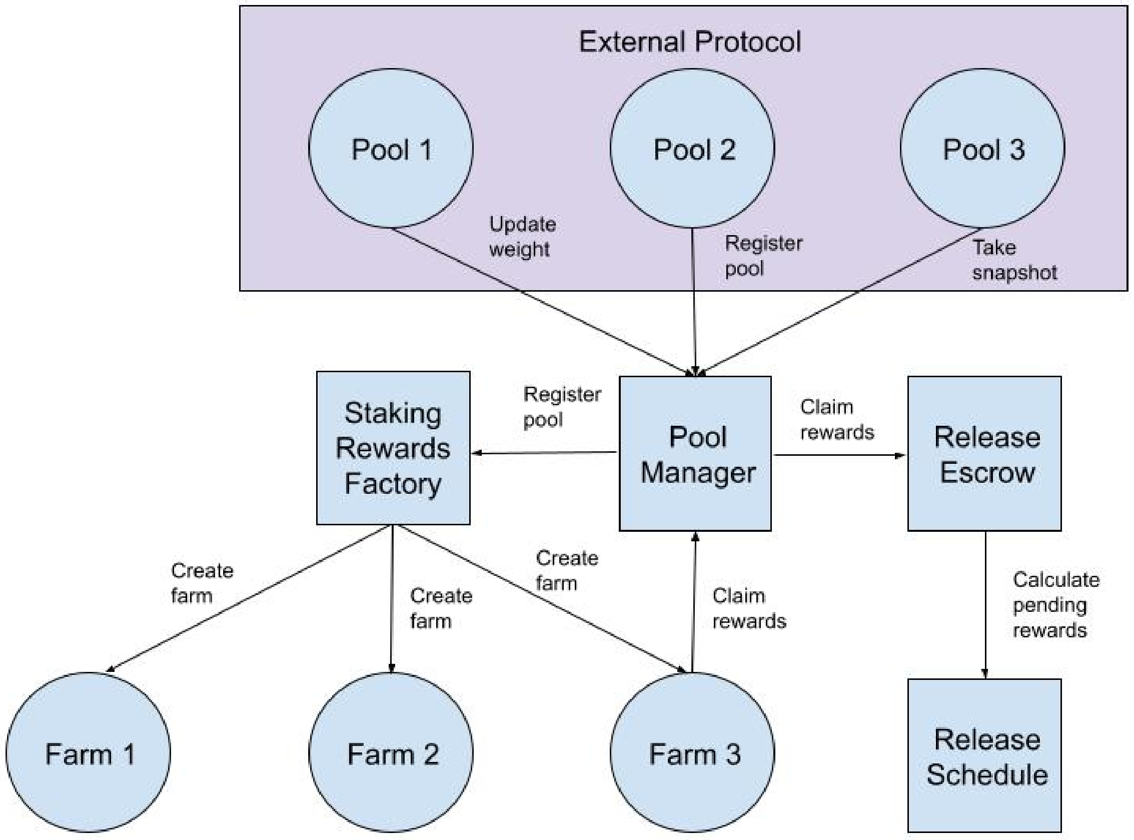

Figure 1: High-level Design of the Farming System

2.1 Design Requirements

- Incentives are self-sustaining: Tokens are distributed on a halving schedule, ensuring that there will always be incentives for registered pools.
- System is self-sustaining: Once the system is deployed, developers will not need to make updates to the system for it to function as intended, since pool weights are updated automatically after each transaction in the external protocol and rewards are distributed automatically after a pool/user transaction.
- Pools are rewarded based on merit: Whether a pool receives rewards, and the amount of reward tokens a pool receives, depends on a pool's unrealized profits and price performance over the last 2 periods relative to the global average.
- All pools meeting minimum criteria can be rewarded: The reward allocation for a given period is distributed to all pools that meet the minimum criteria and the amount that each eligible pool receives is proportional to its weight.
- Reward distribution cannot be manipulated: The combination of minimum criteria and a sliding window for pool weights make Sybil attacks and excessively risky strategies less feasible.

2.2 Internal Contracts

- Pool Manager: The Pool Manager contract is the main contract in the farming system. It is responsible for registering pools, marking pools as eligible to receive rewards after meeting minimum criteria, updating pool weights after each external transaction, and withdrawing unlocked tokens from escrow.
- Staking Rewards Factory: The Staking Rewards Factory contract deploys a Staking Rewards contract whenever a pool is registered in the farming system. This contract is necessary because trying to deploy a Staking Rewards contract directly from the Pool Manager contract would exceed the contract size limit (introduced to prevent denial-of-service attacks) in EVM-compatible blockchains [2].
- Staking Rewards: Each registered pool has a unique Staking Rewards contract that represents a 'farm' in existing yield farming systems. This contract receives rewards from the Pool Manager contract whenever its associated pool is updated through an external transaction (such as depositing or withdrawing). Users can stake their pool tokens (distributed by a project integrating this system) in this contract to earn a share of the pool's rewards proportional to their stake.
- Release Schedule: The Release Schedule contract is used for calculating the number of tokens unlocked based on the time elapsed since the contract was deployed. Tokens are unlocked on a halving schedule to ensure that there will always be tokens to distribute. The schedule consists of an infinite number of cycles, each lasting 26 weeks. Each cycle distributes half as many tokens as the previous cycle, creating artificial scarcity for the reward token.
- Release Escrow: The Release Escrow contract stores the lifetime supply of tokens to distribute. It uses the Release Schedule contract to determine how many tokens to unlock at any given time. The contract is called by the Pool Manager contract whenever a pool updates weights or a user claims rewards.

2.3 External Contracts

- Pool: The Pool contract represents a decentralized hedge fund that's part of an external asset management protocol. The asset management protocol integrates the farming system by calling the Pool Manager contract's 'update pool weight' function whenever a user deposits/withdraws from a pool or the pool's owner executes a transaction on behalf of the pool.

2.4 Partitioning Into Periods

The farming system uses 14-day periods, starting from the timestamp at which the system was deployed, to track pool performance over time and help prevent malicious actors. Using pool periods when calculating pool weight allows pools that outperform over recent periods to receive rewards regardless of their lifetime performance. It also filters out inactive pools, since the periods act as a 'sliding window' on pool performance history.

Each pool has a 'latest recorded period' and a 'previous recorded period'. The 'latest recorded period' is the most recent period in which the pool was updated. The 'previous recorded period' is the most recent period before the 'latest recorded period' in which the pool was updated. In cases where a pool has only been updated in one period, 'latest recorded period' will be the same as 'previous recorded period'. The system keeps track of two periods for each pool in order to calculate the percentage change in pool token price, which is used when calculating pool weight.

Pool period index can be calculated as follows:

$$i = (t_c - t_s)/d. \tag{1}$$

The starting timestamp of a period can be calculated as follows:

$$t_i = t_s + (i * d). \tag{2}$$

2.5 Minimum Criteria for Pools

If it were too easy to be eligible for rewards, users would be more likely to take excessive risks to try and get a higher weight for their pool. Excessive risk-taking can be detrimental to investors since it is more likely to have negative expected value.

There is also the potential for Sybil attacks, where a user creates multiple accounts to interact with a protocol with malicious intent. In the context of the farming system, a Sybil attack would involve a pool creating multiple wallets to bypass the pool-per-user limit on protocols integrating the farming system [3]. They could take excessive risks on each of their pools in hopes that the risk will generate large returns for one of the pool. The pool with the large return would have a larger weight in the farming system, leading to higher APR for the pool's farm, which would incentivize other users to invest in the pool just to deposit the pool tokens in the pool's farm.

To help discourage excessive risk-taking and reduce the possibility of a Sybil attack, as well as the potential damage to the protocol if a Sybil attack is conducted, the farming system introduces a set of minimum criteria that a pool must meet before it can be eligible for rewards.

The following criteria are proposed:

- Pool has existed for at least 30 days
- Pool has at least 10 unique investors
- Pool has at least $1000 in value

The first criteria encourages pool managers to establish a track record to prove that they are consistently performing well. By having a track record of consistent outperformance, pool managers can demonstrate to potential investors that their strategy has positive expected value. It's still possible for pool managers to take excessive risk and wait the 30 days without making additional trades to maintain an artificially high lifetime return, but the profitable trade will not impact the pool's weight after 2 periods.

The second and third criteria combined help to discourage Sybil attacks. A user wanting to bypass these requirements would need to deposit at least $1000 of their own funds across multiple accounts to simulate multiple unique investors in the pool. It's unlikely that the extra rewards from having a higher pool weight would outweigh the value at risk.

3 Pool Manager Contract

The Pool Manager contract is responsible for most of the administrative work in the system. It registers pools, checks pools against the minimum criteria, updates weights, and distributes rewards to farms.

This contract also stores the state (such as weight, pool token price, and unrealized profits) of each pool for each period.

3.1 Registering Pools

When registering a pool, a transaction is initiated by the external pool contract to call the Pool Manager contract's "registerPool()" function. The pool would initially not be eligible for rewards, but its average price change and unrealized profits per period would be updated in the farming system after each pool transaction so that the pool's weight can be calculated after it is marked as eligible. The pool contract would have to initiate a transaction to mark the pool as eligible after it meets the minimum criteria.

3.2 Calculating a Pool's Share of Global Rewards

The current global scaled weight can be calculated by using Equation 8 with global weight in current period for w, global weight in previous period for w_0, the current timestamp for t, and the starting timestamp for the current period for t_0.

Once the scaled weight has been calculated, the current global reward per token can be calculated with:

$$r = \left\{ \begin{array}{ll} r_0 + \frac{x}{w}, & \text{if } w > 0 \\ r_0, & \text{if } w = 0 \end{array} \right\}, \tag{3}$$

where r_0 is the current global reward per token stored, x is the number of tokens unlocked in the release schedule, and w is the global scaled weight calculated previously.

After the global reward per token stored and the global scaled weight are calculated, the pool's scaled weight can be calculated by using Equation 8 with the pool's weight in the current period for w and the pool's weight in the previous period for w_0. When calculating pool's scaled weight and global scaled weight, the w_0 is set to 0 if this is the first period since the release escrow started.

The pool's share of rewards could then be calculated with:

$$x = (w * (r - r_{pool})) + x_0, \tag{4}$$

where x is the number of tokens unlocked for the pool, w is the pool's scaled weight, r is the reward per token calculated previously, r_{pool} is the reward per token when the pool last claimed rewards, and x_0 is the number of tokens accumulated when the pool last updated its weight.

Note that both the scaled pool weight and the scaled global weight are multiplied by the same scalar, and the global weight changes by $\triangle \mathrm{w}_{pool}$ whenever a pool is updated, so the sum of pool weights will always equal the global weight.

3.3 Updating Rewards

Before a pool's weight is updated, the farming system first updates the pool's available rewards and withdraws unlocked tokens from the Release Escrow contract on behalf of all pools registered in the farming system.

Algorithm 1 Updating a pool's rewards

```
1: $r_0 \leftarrow r$
2: $r \leftarrow rewardPerToken()$
3: if $r_0 = r$ and releaseEscrow.hasStarted() then
4:     Transfer unlocked tokens from the Release Escrow contract to the Pool Manager contract
5:     Transfer unlocked tokens from the Pool Manager contract to the dedicated recipient
6: end if
7: $lastUpdateTime \leftarrow currentTime$
8: $rewards[pool] \leftarrow earned(pool)$
9: $poolRewardPerTokenStored[pool] \leftarrow r$
```

Operation 1 creates a temporary variable with value set to "reward per token stored". Operation 2 calculates the new global "reward per token stored" based on the current global scaled weight and the available rewards in the release schedule. Operation 3 compares the initial global "reward per token stored" to the new global "reward per token stored" to check whether the current global scaled weight is 0. If this occurs after the release schedule has started, then the unlocked tokens need to be sent to a dedicated recipient (an address provided by the protocol owner when the system is deployed) to avoid getting lost, since pools will not earn rewards when the total weight is 0. Operations 4-5 withdraws unlocked reward tokens from the Release Escrow contract and transfers them to the protocol's dedicated recipient. Operation 7 sets the "lastUpdateTime" timestamp to the current timestamp to prevent inflated values in the "rewardPerToken()" function. Operation 8 updates the pool's accumulated rewards to ensure that they do not decrease when the pool's weight is subsequently changed. Operation 9 sets the pool's "reward per token stored" to the global "reward per token stored" so that the pool's pending rewards is 0. This prevents double counting, since any additional rewards earned since the last update would already be accounted for in Operation 8.

3.4 Average Price Change

A pool's 'average price change' (APC) tracks the current pool token price in the current period vs. the last recorded pool token price in the latest period before the current period in which the pool was updated. The APC can be calculated by

$$APC = \frac{1000 * (p - p_i)}{p_i * max(1, t - t_i)}, \tag{5}$$

where p is the current pool token price in the current period, p_i is the last recorded pool token price in the latest period before the current period, t is the index of the current period, and t_i is the index of the latest period before the current period.

APC is scaled by a factor of 1000 to preserve fractional percent changes, since fractional values are not allowed in smart contracts. A 100% price increase over 2 periods would result in an APC of 1000, and a 0.1% price increase over 2 periods would result in an APC of 1.

Max is used to account for the edge case where t = $t - i$ when the current period is the first period, which would lead to division by 0. In this case, the value is hard-coded to 1.

It's important to note that the latest period index may not always be the period before the current one. For instance, if the current period is at index 5 and a pool was last updated in period 1, then the latest period index would be 1 instead of 4. By using the latest period index instead of the previous period index, the farming system can account for inactivity between 2 periods while ensuring that a pool's change in token price can always be calculated. If the previous period index is used instead, then a pool's change in token price would be infinite if the pool was updated in the current period but not in the previous period.

When designing the farming system, we considered using a pool's lifetime returns in lieu of APC, but decided against it because a pool's weight would stay at 0 until the lifetime return is positive. This could potentially break the farming system during bear markets. Using APC also prevents pools with inflated lifetime returns (which may result from an excessive risk that paid off) from monopolizing the global weight for more than 2 periods.

3.5 Updating Pool Weights

When deciding on the equation for updating pool weights, we wanted all pools with positive weight to be eligible for rewards, but also wanted to give more rewards to pools that are outperforming. The weight of a pool is based on the pool's APC relative to the average APC across all eligible pools in the farming system. All pools with $APC > 0$ (any amount of unrealized profits over the current period or previous period) will have a positive weight and thus be eligible for rewards. Pools performing below average will still have a positive weight but the weight would be lower than it would be if the same performance was above average.

A pool's weight is calculated by

$$w = \left\{ \begin{array}{ll} p * \sqrt{APC - aAPC + 1}, & \text{if } APC \geq aAPC \\ \frac{p * \log_2 APC}{\sqrt{aAPC - APC}}, & \text{if } APC < aAPC \end{array} \right\}, \tag{6}$$

where p is the pool's unrealized profits, APC is the pool's average price change, and aAPC is the average APC across all pools in the farming system.

The average APC is calculated by

$$aAPC = \frac{\sum_i^n APC_i}{n}, \tag{7}$$

where APC_i is the average price change of pool i and n is the number of eligible pools in the farming system.

Algorithm 2 Updating a pool's weight

```
 1: Use Algorithm 1
 2: if $u < 0$ or $p \leq 0$ then
 3:     Throw error
 4: end if
 5: $w_0 \leftarrow w[pool, i]$
 6: if $currentperiodindex > pool'slatestrecordedperiodindex$ then
 7:     $previousRecordedPrice_{pool} \leftarrow latestRecordedPrice_{pool}$
 8:     $previousRecordedPeriodIndex_{pool} \leftarrow latestRecordedPeriodIndex_{pool}$
 9: end if
10: $u_{pool} \leftarrow u$
11: $latestRecordPrice_{pool} \leftarrow p$
12: $latestRecordedPeriodIndex_{pool} \leftarrow i$
13: if pool is not eligible then
14:     $lastUpdated_{pool} \leftarrow currentTime$
15:     Return early
16: end if
17: Subtract from totalWeightedAPC and totalDuration
18: $APC_{pool} \leftarrow calculateAveragePriceChange(pool)$
19: $lastUpdated_{pool} \leftarrow currentTime$
20: Add to totalWeightedAPC and totalDuration
21: $w \leftarrow calculatePoolWeight(pool)$
22: $poolPeriods[pool, i] \leftarrow w$
23: $globalPeriods_i \leftarrow globalPeriods_i - w_0 + w$
24: Use Algorithm 3
```

Operation 1 calls Algorithm 1 to update the pool's rewards. Operations 2-4 revert the transaction if the pool has an unrealized loss (u) or the pool token price (p) is below 0. These values are provided by the external protocol (project integrating this farming system) when calling the function "Pool-Manager.updateWeight()". Operation 5 creates a temporary variable that stores the pool's weight for the current period (before updating). Operations 6-9 set the pool's "previous recorded price/index" to the "latest recorded price/index" if the current period index (global) is greater than the pool's "latest recorded period index". This is done to keep track of the two periods needed for calculating APC. Operation 10 sets the pool's unrealized profits to the unrealized profits provided by the external protocol. Operation 11 sets the pool's latest recorded price to the pool token price provided by the external protocol. Operation 12 sets the pool's "latest recorded period index" to the current global period index. Operations 13-16 set the pool's set the pool's "last updated" timestamp to the current timestamp and ends the algorithm early if the pool is not eligible for rewards. Operation 17 subtracts the pool's time-scaled APC and duration from the "totalWeightedAPC" and "totalDuration" global variables before updating the pool's APC and "last updated" timestamp. Operation 18 sets the pool's APC to the value calculated using Equation 5. Operation 19 sets the pool's "last updated" timestamp to the current timestamp (global). Operation 20 adds the pool's new APC and duration (number of seconds elapsed since the pool was registered in the farming system) to the "totalWeightedAPC" and "totalDuration" global variables. Operation 21 creates a temporary variable (w) that stores the pool weight calculated using Equation 6. Operation 22 sets the pool weight for the current period to the value calculated in Operation 21. Operation 23 updates the global weight for the current period by the difference between the value calculated in Operation 21 and the value calculated in Operation 5. Operation 24 calls Algorithm 3 to get the latest rewards.

3.6 Determining a Pool's Share of Available Rewards

Once a pool's weight for the current period is calculated, the pool's share of available rewards can be calculated by taking the ratio of pool's weight in the current period to global weight in the current period. To prevent weights from starting at 0 whenever a new period starts, linear time scaling is applied to weights in the current period and the previous period.

The time-scaled weight is calculated by:

$$w_c = \frac{[w * (t - t_0)] + [w_0 * (d + t_0 - t)]}{d}, \tag{8}$$

where w_c represents the time-scaled weight, w represents the weight in the current period, w_0 represents the weight in the previous period, t represents the current time, t_0 represents the start time for the current period, and d represents the period duration.

There is an edge case where time-scaled weights are strictly lower in the first period of the farming system, since the weights for the previous period would be 0 by default, but both pool and global weights would be multiplied by the same scalar so the reward distribution is not affected.

Time scaling also penalizes pools that were inactive/underperforming in either the current period or previous period without eliminating their share of rewards entirely. Their rewards are reduced based on the time elapsed in the underperforming period vs. the time elapsed in the other period.

4 Staking Rewards Contract

The Staking Rewards contract represents a farming contract that users can deposit pool tokens (distributed by an external protocol integrating this system) into to get a share of the pool's rewards. This contract operates similarly to the farming contracts of existing farming systems, where the amount of rewards a user receives is based on the ratio of their deposited tokens to the total supply. But unlike in existing farming systems, where developers manually allocate reward tokens to each farm, these farms receive tokens automatically from the Pool Manager contract based on the weight of the associated pools.

4.1 User's Share of Rewards

The number of reward tokens available for a user can be calculated by:

$$x = (b_{user} * (r - r_{user})) + x_{user}, \tag{9}$$

where x is the number of reward tokens available for the user, b_{user} is the number of pool tokens the user has deposited in the farm, r is the current reward per token stored, r_{user} is the reward per token stored when the user last claimed rewards, and x_{user} is the number of unclaimed reward tokens the user accumulated when they last deposited/withdrew from the farm.

The global reward per token stored is strictly increasing to prevent unclaimed user rewards from decreasing whenever the total supply of deposited pool tokens changes. Whenever a user deposits/withdraws from the farm, r_{user} is set to r and pending rewards are added to x_{user} to prevent the change in the number of deposited pool tokens from affecting the user's pending rewards. Whenever a user claims rewards, x_{user} is set to 0 and r_{user} is set to r.

4.2 Getting Rewards

Algorithm 3 describes how the Pool Manager contract transfers pending rewards to a pool's farm. This algorithm is triggered by a farm contract whenever a user claims rewards.

Operation 1 updates the pool's rewards using Algorithm 1 to prevent double counting. Operation 2 withdraws reward tokens on behalf of all pools in the farming system. The pool's share of those rewards is calculated by multiplying the total number of tokens unlocked by the ratio of pool's scaled weight to global scaled weight. Operation 3 creates a temporary variable with value set to the pool's stored rewards. This is done to keep track of the distributed reward after the stored rewards have been reset to 0. Operation 5 resets the pool's stored rewards to 0. Operations 7-9 handle the edge case where the farm has no pool tokens deposited. In this case, rewards are transferred to the dedicated

Algorithm 3 Getting rewards

```
 1: Update the pool's rewards
 2: Withdraw unlocked tokens from the ReleaseEscrow contract and store them in the PoolManager
    contract
 3: $x \leftarrow x_{pool}$
 4: if $x > 0$ then
 5:     $x_{pool} \leftarrow 0$
 6:     Transfer $x$ tokens to the pool's farm
 7:     if the farm's total supply $= 0$ then
 8:         Transfer pool's share of tokens to the dedicated recipient
 9:     end if
10:     $r_{farm} \leftarrow r_{farm} + \frac{x}{s_{farm}}$
11: end if
```

recipient (most likely a staking contract, but it can be any address the protocol owner chooses) instead of the farm. This prevents division by 0 and ensures that reward tokens don't get stuck in the Pool Manager contract. Operation 10 updates the farm's "reward per token stored". The symbol S_{farm} represents the farm's total supply of pool tokens deposited.

4.3 Claiming Rewards

Algorithm 4 describes the process for claiming rewards from a pool's farm contract. A user with pool tokens staked in the farm can initiate this process by calling the farm contract's function for claiming rewards.

Algorithm 4 Claiming rewards from a farm

```
 1: if pool is not marked as eligible then
 2:     Throw error
 3: end if
 4: Withdraw unlocked tokens from the Release Escrow contract on behalf of all pools
 5: Transfer the pool's share of tokens to the pool's farm
 6: if the farm's total supply $= 0$ then
 7:     Transfer pool's share of tokens to the dedicated recipient
 8: end if
 9: $r_{farm} \leftarrow r_{farm} + \frac{x}{s_{farm}}$
10: $x_{user} \leftarrow earned(user)$
11: $r_{user} \leftarrow r_{farm}$
12: Transfer $x_{user}$ tokens to the user
13: $x_{user} \leftarrow 0$
```

Operations 1-3 revert the transaction if the pool is not eligible for rewards. Operation 4 withdraws all pending tokens for the system from the Release Escrow contract (according to the release schedule) and stores them in the Pool Manager contract. This keeps the Release Escrow contract agnostic to the pools. Operation 5 calculates the pool's share of tokens and transfers them to the farm. The pool's pending rewards is set to 0 in the process. Operations 6-8 handle the edge case where a pool receives rewards but no one has deposited pool tokens into the pool's farm. In this case, the reward tokens that would have gone to the farm are instead transferred to the protocol's dedicated recipient to ensure that all reward tokens are accounted for. Operation 9 updates the farm's "reward per token stored". In operation 10, the term x_{user} is the user's stored rewards, separate from the rewards calculated by the difference in "reward per token stored" and time elapsed since the last claim. Setting this value to "earned(user)" ensures that updating both the global and user "reward per token stored" does not reduce the user's rewards. Operation 11 sets the user's "reward per token stored" to the farm's "reward per token stored". Operation 12 sends the stored rewards to the user, and Operation 13 resets the stored rewards to 0 to prevent double-counting.

5 Release Schedule and Release Escrow Contracts

The Release Schedule and Release Escrow contracts are responsible for storing and distributing reward tokens. When the system is deployed, the project's developer transfers the lifetime supply of reward tokens to the Release Escrow contract. The Release Escrow contract stores a Release Schedule (with halving cycles) that specifies how many tokens to distribute at a given time and the timestamp at which distribution will start.

5.1 Release Schedule

The release schedule used in the farming system is inspired by Bitcoin's token release schedule, which has a finite lifetime token supply released through an infinite number of halving cycles [1]. The schedule consists of 26-week cycles, each of which has half the reward rate of the previous cycle. This creates artificial scarcity for the reward token while ensuring that rewards will always be distributed to pools without intervention from developers.

The reward rate can be calculated by:

$$r = \left\{ \begin{array}{ll} \frac{n}{2^{i-1} * d}, & \text{if } i > 0 \\ 0, & \text{if } i = 0 \end{array} \right\}, \tag{10}$$

where r is the reward rate, n is the number of tokens distributed in the first cycle, i is the index of the cycle, and d is the duration of the cycle (in seconds).

The available rewards can be calculated by:

$$x = \left\{ \begin{array}{ll} [(t_s - t_l) * 2r] + [(t_c - t_s) * r], & \text{if } t_l < t_s \\ (t_c - t_l) * r, & \text{if } t_l \geq t_s \end{array} \right\}, \tag{11}$$

where x is the amount of rewards available, t_s is the start time of the current cycle, t_l is the last time rewards were claimed, t_c is the current time, and r is the reward rate calculated previously.

5.2 Release Escrow

The Release Escrow contract stores the lifetime supply of reward tokens and distributes them according to the release schedule. The contract expects to have the lifetime token supply stored by the start time given in the release schedule, so developers looking to integrate the farming system would need to transfer tokens to the contract shortly after deploying the system.

6 Security

Security is especially important in decentralized finance protocols since they work directly with users' funds and have almost immutable code once deployed. Unlike in existing farming systems, where developers manually adjust the weight of pools after each period, this farming system adjusts the weights automatically so it is essential that the algorithms behind this system are mathematically sound. This section outlines the proof behind the security of these algorithms.

6.1 Preventing Malicious Actors

Malicious actors fall into two main categories: pool managers who conduct a Sybil attack in hopes of getting a high pool weight, and pool managers who stay primarily in stablecoins to get a high pool weight with near-zero unrealized profits.

6.1.1 Sybil Attacks

The first category of malicious actors describes pool managers who use a risky strategy across multiple pools (bypassing the "pool per user" limit by creating fake addresses) by creating a new pool for each trade until a profitable trade is made. We define a risky strategy as a strategy where the probability of hitting the profit target is small compared to the probability of hitting the stop loss. Such strategies don't need to have negative expected value to be detrimental to the farming system, as long as there are more unprofitable trades than profitable ones.

Pools with a losing trade will have a negative ROI for the current period, leading to 0 weight. The pool with a profitable trade would have a positive ROI for the current period, leading to a positive weight and a share of the rewards for the current period. Such strategies can be detrimental to the farming system because they lead to a large number of inactive pools, which can make it difficult for potential investors to find reliable pools. In addition to crowding the protocol, these strategies also take away rewards from reliable pools.

6.1.2 Deterring Risky Strategies

A pool manager would be incentivized to use a risky strategy as long as the expected value of the profitable pool's farming rewards exceeds the total expected cost.

The total expected cost of such a strategy is the total loss across unprofitable trades minus the profit from the one profitable trade. Before a profitable trade is made, we can expect

$$n = \frac{1-p}{p} \tag{12}$$

unprofitable trades, where p is the probability of a profitable trade. The total expected cost could then be calculated by:

$$c = m[p_-(\frac{1-p}{p}) - p_+], \tag{13}$$

where m is the minimum investment per pool, p_- is the percent loss on an unprofitable trade, and p_+ is the percent profit on a profitable trade.

The pool's share of rewards is the dollar value of the total rewards distributed over the current period multiplied by the ratio of the pool's weight for the current period to the global weight for the period. The reward rate is defined as the dollar value of rewards distributed per second, and is given by:

$$r = \frac{nv}{2^i n_p d_p}, \tag{14}$$

where n is the lifetime number of protocol tokens that will be distributed, v is the dollar value per token, i is the index of the cycle, n_p is the number of period per cycle, and d_p is the duration of a period in seconds.

The total dollar value distributed per period is given by:

$$R = rd_p, \tag{15}$$

where r is the reward rate and d_p is the duration of a period in seconds. Substituting equation 12 into equation 13 gives:

$$R = \frac{nv}{2^i n_p}, \tag{16}$$

A pool's share of rewards is given by:

$$R_{pool} = \frac{Rw_{pool}}{w_{global}}, \tag{17}$$

where w_{pool} is the weight of the pool for the current period and w_{global} is the total weight in the farming system for the current period. Substituting equation 14 into equation 15 gives:

$$R_{pool} = \frac{nvw_{pool}}{2^i n_p w_{global}}. \tag{18}$$

A Sybil attack is considered profitable if:

$$R_{pool} > c, \tag{19}$$

which can be rewritten as:

$$\frac{nvw_{pool}}{2^i mn_p w_{global}[p_-(\frac{1-p}{p}) - p_+]} > 1. \tag{20}$$

From this inequality, we can see that the probability of a successful attack decreases after each cycle as long as the increase in scarcity causes the token price to increase by a factor less than 2. As more pools join the system, the weight of the attacker's profitable pool will constitute a smaller and smaller portion of the system's total weight, decreasing the profitability of an attack.

The system is expected to be less secure in the beginning, since there are fewer pools registered and more tokens distributed per period. Due to the relative ease of generating a high reward for a pool, we expect a quick increase in the number of registered pools and a decrease in token price as the token rewards get sold. Both factors will decrease the likelihood of a profitable attack.

6.1.3 Staying in Stablecoins

The second category of malicious actors describes pool managers who stay primarily in stablecoins and only trade a small fraction of the pool's funds to maintain a non-zero pool weight. These pool managers are not detrimental to investors, but they can change the nature of the protocol integrating the farming system by transforming it into a stablecoin farming protocol if most registered pools use such a strategy.

Pools are generally encouraged to stay invested in volatile assets to remain competitive, but during a bear market, pools are more likely to have a negative ROI by staying invested in volatile assets than by staying invested in stablecoins. Under such conditions, pool managers may opt to keep the majority of pool funds in stablecoins and only trade a small fraction of the funds to have non-zero unrealized profits (thus earning rewards for the pool). As a new bull market emerges, pools trading assets other than stablecoins will outperform pools staying in stablecoins, which may cause a migration of investors from stablecoin pools. The weight of pools trading volatile assets will increase as they receive more investments, leading to higher rewards. Stablecoin pools would have to start trading volatile assets to remain competitive.

6.2 Edge Cases

In this subsection, we will examine the distribution of rewards under four cases:

1. zero eligible pools
2. one eligible pool with a loss
3. one eligible pool with non-zero weight and \$10000 invested
4. multiple eligible pools with non-zero weight

For simplicity, the calculations are made with a token price of \$0.01 and a weekly reward distribution of 1 million tokens. The example calculations for case 4 are described in table 1.

Pool Number	TVL	APC	Unrealized Profits	Weight	Tokens per Week	APR
1	\$10000	200	\$1000	666.67	3412.40	17.74%
2	\$15000	400	\$300	2700	13820.16	47.91%
3	\$300000	200	\$45000	30000	153557.41	26.61%
4	\$1200000	400	\$18000	162000	829210.01	35.93%

Table 1: Reward Distribution for Case 4

In case 1, since there are no eligible pools (global scaled weight is zero), 1 million reward tokens go to the farming system's dedicated recipient (most likely a staking contract for the reward token).

In case 2, the global scaled weight is zero because the only eligible pool in the farming system has zero unrealized profits. Like in case 1, the reward tokens will go to the dedicated recipient.

In case 3, the pool will receive \$10000 worth of reward tokens per week, for an APR of 5200%. The excessive APR will either cause the reward token's price to fall or lead to an influx of investments in the pool until the APR drops.

In case 4, pools have varying weights but similar APR despite different TVL. The APR for outperforming pools is higher. Note that the unrealized profits are lower than expected for the given TVL, since users are depositing into the pools at different times.

From Equation 6, we can see that a pool's weight is directly proportional to its unrealized profits. Since the increase in weight from higher unrealized profits equals a user's decrease in their portion of the pool's rewards, a pool's APR is unaffected by an increase in deposits.

7 Integrating the Farming System

Developers looking to integrate the farming system should first deploy their own copy of the system with a custom release schedule, depending on the token distribution of their project. After deploying the system, the developer needs to transfer the lifetime supply of reward tokens to the Release Escrow contract before the start time provided in the Release Schedule contract.

In the protocol that integrates the farming system, the function that creates a pool should call PoolManager.registerPool() as the last statement. Each function that allows users to interact with a pool (such as deposit, withdraw, and exit) should call PoolManager.updateWeight() as the last statement. There should also be a function that wraps PoolManager.markAsEligible(), which can only be called once a pool is registered.

Although the overall design of the farming system is protocol-agnostic, the metrics used for calculating pool weight are tailored to asset management protocols. Developers should use different metrics, depending on what the pools represent, and update the unit tests accordingly. For instance, an AMM (automated market maker) protocol may use average liquidity for calculating pool weight.

7.1 Gas Usage

Smart contracts use gas as a weighted computational cost per transaction. Each type of computation has a gas value assigned to it, with arithmetic operations having the lowest values and writes to storage having the highest values. Blockchains supporting smart contracts charge a fee per unit of gas, which users pay when a transaction is sent to the blockchain. This helps prevent users from overloading a blockchain through DoS attacks [4].

Table 2 compares the gas cost for various transactions before, and after, integrating the farming system. Integrating the system leads to a 158000-232000 increase in fees, which could prove significant depending on the average gas fee for the blockchain the system is deployed on. Developers looking to integrate the farming system should weigh the benefits of having a self-sustaining reward system against the potential decrease in users due to higher fees.

Transaction	Gas Used Before	Gas Used After
Deposit	293603	450864
Execute Transaction	139294	331039
Take Snapshot	130807	288352
Withdraw	283713	515178

Table 2: Amount of Gas Used by a Pool Transaction Before, and After, Integrating the Farming System

7.2 Improvements

In the current version of the farming system, the minimum criteria for pools and the calculations for pool weights are included in the Pool Manager contract. These could be moved to two separate smart contracts that get called when registering pools or updating weights, allowing for greater customization. Although these changes would make the system more flexible, they may be difficult to implement due to the call stack having too many local variables.

8 Conclusion

In this paper, we propose a yield farming system that is fully-decentralized, tamper-proof, self-sustaining, and inclusive of all pools registered under the system. The main innovation of this farming system is calculating pool weights by tracking the performance of each pool relative to the global average and measuring performance over discrete periods with a sliding window. The system becomes self-sustaining by acting as a 'hook' that gets called at the end of each transaction in projects integrating the system, allowing pool weights to be updated automatically. We believe that the approach to yield farming set forth in this paper will lead to more trust from users while reducing the sources of error and effort needed for project maintenance.

References

[1] Antonopoulos, Andreas M (1 July 2017). Mastering bitcoin: programming the open blockchain (2nd ed.). Sebastopol, California, USA: O'Reilly Media. p. 239.

[2] Waas, M. 2020. Downsizing contracts to fight the contract size limit. Retrieved Sep 11, 2023 from https://ethereum.org/en/developers/tutorials/downsizing-contracts-to-fight-the-contract-size-limit.

[3] Ali SE, Tariq N, Khan FA, Ashraf M, Abdul W, Saleem K. BFT-IoMT: A Blockchain-Based Trust Mechanism to Mitigate Sybil Attack Using Fuzzy Logic in the Internet of Medical Things. Sensors (Basel). 2023 Apr 25;23(9):4265. doi: 10.3390/s23094265. PMID: 37177468; PMCID: PMC10181539.

[4] Smith, C. 2023. Gas and Fees. Retrieved Sep 11, 2023 from https://ethereum.org/en/developers/docs/gas.