GYSR Core v2

June 14, 2021

Devin Conley
GYSR
devin@gysr.io

Alex Koren
GYSR
alex@gysr.io

Iain McCown
GYSR
iain@gysr.io

Ben Roy
GYSR
ben@gysr.io

ABSTRACT

This technical whitepaper introduces *GYSR Core v2*, which includes major architectural, functional, and economic improvements to the protocol. Specifically, it covers the new modular architecture, enhancements to $GYSR spending economics, and the addition of a *friendly* incentive mechanism.

1 INTRODUCTION

GYSR is an open platform for on-chain incentives made up of several distinct components. The core staking protocol is a contract between creators and users to incentivize and reward a particular tokenized behavior. The factory contract allows any creator to easily configure and deploy a staking pool for their own project. Finally, at the global level, the platform facilitates discovery, investment, and value flow between projects and users.

GYSR Core v2 [1] introduces many significant improvements to the GYSR platform. The primary staking pool has been entirely refactored to use a modular architecture. This allows for more flexibility and the easy addition of new capabilities. Through this modular architecture, an entirely new reward mechanism has been implemented with an emphasis on friendly staking dynamics, ease of use, and predictable returns. This protocol update also improves on platform economics through more efficient $GYSR pricing and the introduction of a global fee system.

2 BACKGROUND

2.1 GYSR Core v1

GYSR v1 [2] is the predecessor to the protocol described in this paper. It is a configurable system of smart contracts that provides a general solution for token distribution and incentive programs. This allows creators to promote useful behaviors by rewarding users for long-term participation. The protocol includes a native token $GYSR, which was introduced to further align incentives, act as a diversified investment asset, and provide a source of continuous funding for projects.

GYSR v1 was launched on the Ethereum blockchain in October of 2020. Since then, 91 pools have been created for a variety of use cases including token distribution, promoting liquidity on automated market makers (AMMs), and enforcing vesting schedules.

One distinguishing feature of GYSR v1 is that staking mechanics are competitive. The staking contract implements bonus multiplier mechanics to reward users for long term commitments (time bonus) and project support ($GYSR bonus). These work by increasing the user's share of the reward pool and effectively decreasing the claimable amount for all other users. Therefore, it is important to be aware of user activity, funding schedules, and bonus configuration in order to maximize rewards.

2.2 Upgradeability in DeFi

A common philosophical dilemma when building decentralized applications is the inherent tradeoff between upgradeability and security. For a thorough review on the issue, see *The State of Smart Contract Upgrades* from OpenZeppelin [3].

A standard smart contract is immutable; the code cannot be modified after it has been deployed to the blockchain. While this provides explicit guarantees on functionality, it means that bug fixes are impossible and new features can never be added.

One approach is to use the *upgradeable proxy* pattern [4] so that the underlying implementation logic can be completely swapped out. This results in a contract that is highly flexible to support patches and new features. However, this can also introduce significant security vulnerabilities through added complexity, increased attack surface, and privileged points of centralization.

In other instances, projects have simply built a separate product altogether and focused on promoting migration. Unfortunately, this is a manual process, and it can both lead to bifurcation of the user base and cause confusion around which platform to use.

A wide variety of approaches have been taken to address the issue of upgradeability, and the optimal strategy is dependent on the nature of the project in question.

2.3 Scalable reward calculations

Another common challenge when building DeFi applications is achieving efficient bookkeeping that scales to a high number of users and operations. In the post *Scalable Reward Distribution with Changing Stake Sizes* [5], the author describes an incremental method of accounting that handles variable stake sizes for an arbitrarily high number of users, operations, and reward events. Further, the algorithm maintains all earned rewards exactly, even before the pull-based distribution has occurred.

3 MODULAR ARCHITECTURE

One of the key advancements in GYSR v2 is the introduction of a modular *Pool* architecture. With this redesign, the staking logic and reward logic are each abstracted out into their own contracts, which are referred to as *modules*. The Pool contract acts as an interface, wrapper, and manager for the associated staking and reward modules. Different combinations of modules can be used to implement a variety of incentive mechanisms.

This new design results in a highly flexible and extensible system of smart contracts. Most notably, it allows the GYSR platform to rapidly innovate and develop new capabilities without compromising any trust or security guarantees.

3.1 Pool

The *Pool* contract is the end user's primary point of interaction. It handles all information flow and manages interactions with the underlying modules. The contract only exposes a few simple methods that the end user needs to be aware of.

The references to the reward module, staking module, GYSR token, and factory contracts are configured at deployment time, and are immutable beyond that point. This allows the Pool to support a wide variety of future incentive mechanisms for new deployments, while ensuring that existing contracts remain immutable and decentralized.

One notable decision in this version of the Pool contract is that we diverge from the EIP-900 standard [6]. Unfortunately, the *IStaking* interface is not sufficient to describe interactions with the new modular architecture. We did not want to compromise on our design for the sake of conforming, nor did we want to waste bytecode on half-functional wrapper methods.

The highly extensible design of GYSR v2 aims to establish a new common standard. The *IPool* interface and methods are described throughout the remainder of this section.

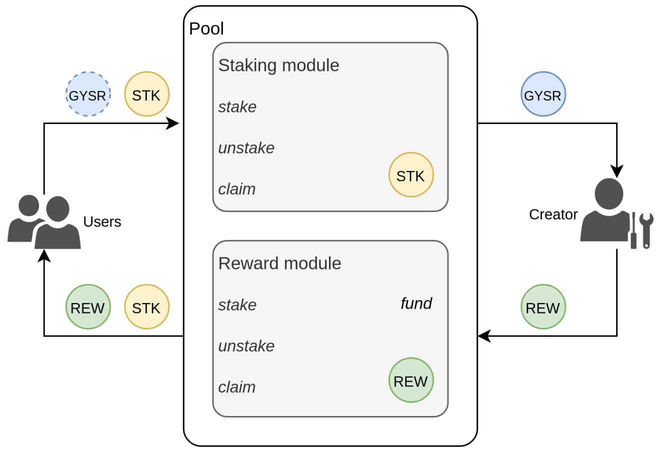

Figure 1: Modular Pool architecture

3.1.1 Stake. The stake method is called by the user to deposit an amount of a given staking asset into the Pool's staking module. It also registers this stake operation with the reward module to allow relevant accounting to occur.

3.1.2 Unstake. The unstake method is called by the user to withdraw an amount of a given staking asset from the Pool's staking module. It also triggers the reward module to distribute any relevant rewards a user has earned.

3.1.3 Claim. The claim method is called by the user in order to receive rewards without unstaking their staked asset.

3.1.4 Withdraw. The withdraw method is called by the controller of the Pool to collect any accrued $GYSR earnings.

3.1.5 $GYSR processing. The Pool is responsible for the actual transfer and processing of $GYSR that is used in the reward module. When a user spends $GYSR for a bonus multiplier, those tokens are transferred to the Pool contract. Once those tokens have fully vested, they are eligible for withdrawal. Note that spending and vesting can occur in the same operation.

3.2 Staking module

The staking module encapsulates and handles all Pool logic dealing with staking assets. This includes token valuation, balance management, and transfers. Below, we describe the *IStakingModule* interface, which any staking module must implement to be compatible with the modular system.

3.2.1 Stake. The stake method is called by the Pool contract to execute the actual deposit of staking assets. It receives a user address, a token amount, and a flexible bytes parameter for additional data. It returns an account address to credit along with the number of shares that the user deposit was worth.

Bookkeeping is done in shares rather than token amounts for generalization across asset types and to support elastic supply or interest bearing tokens.

The staking module is given the flexibility to specify a credited account (rather than just assuming the user) in order to support a wider variety of staking designs in the future. For example, we could use this field to implement a staking module which issues tokenized positions.

3.2.2 Unstake. The unstake method is called by the Pool contract to execute the withdrawal of staking assets. It receives a user address, a token amount, and a flexible bytes parameter for additional data. It returns an account address along with the number of shares that should be burned.

3.2.3 Claim. The claim method is called by the Pool contract to quote the value of tokens, without actually unstaking them. It receives a user address, a token amount, and a flexible bytes parameter for additional data. It returns an account address along with the number of shares that the claim amount is worth.

3.3 Reward module

The reward module encapsulates and handles all Pool logic dealing with reward assets. This includes funding, $GYSR bonus, reward calculation, and distribution. Below, we describe the *IRewardModule* interface, which any reward module must implement to be compatible with the modular system.

3.3.1 Stake. The stake method is called by the Pool contract to perform any relevant bookkeeping for a new stake. It receives an account address, a user address, the number of newly minted shares, and a flexible bytes parameter for additional data. It returns the amount of $GYSR spent and the amount of $GYSR vested during the operation.

The reward module is responsible for handling $GYSR multiplier mechanics. This includes reporting the amount of $GYSR consumed back to the Pool contract in order to do the actual token transfer.

3.3.2 Unstake. The unstake method is called by the Pool contract to distribute earned rewards and perform any other bookkeeping for the removed stake. It receives an account address, a user address, the number of burned shares, and a flexible bytes parameter for additional data. It returns the amount of $GYSR spent and the amount of $GYSR vested during the operation.

3.3.3 Claim. The claim method is called by the Pool contract to distribute rewards without burning staked shares. It receives an account address, a user address, the number of shares claimed against, and a flexible bytes parameter for additional data. It returns the amount of $GYSR spent and the amount of $GYSR vested during the operation.

3.4 Access controls

The module contracts utilize an owner-controller access model to secure function calls. The owner of the module is the Pool contract and the controller is the creator of the Pool.

The stake, unstake, claim, clean, and update methods are restricted as *owner only*. The system is designed such that all core logic must be orchestrated by the primary Pool contract and can never be triggered directly.

A module may also designate *controller only* methods for any relevant administrative tasks. For example, this might include supplying funding to a reward module.

The owner of a Pool can also transfer ownership or control to another account. When the control of a Pool is transferred, the control of the associated modules is transferred as well.

3.5 Factory system

In order to extend the principles of modularity and extensibility to the Pool factory, we introduce the *IModuleFactory* interface. This sub-factory is responsible for the construction of its respective module and is managed by the primary Pool factory. Each module factory must be whitelisted by the Pool factory controller before it can be used.

The Pool factory *create* method can be called by any user to configure and deploy a new Pool. The desired module types are specified by passing the corresponding factory addresses. Additionally, the constructor data for each sub-factory is passed through a pair of flexible bytes parameters. This primary factory method calls each module factory and assembles the overall Pool contract.

4 $GYSR SPENDING MECHANICS

GYSR v2 includes various improvements to spending mechanics. These changes primarily focus on the efficiency of $GYSR pricing and on platform-level economic improvements.

4.1 Global fee system

Previously, the entire amount of $GYSR spent would go to the creator of that Pool. While this aligns with our principles of decentralization, it neglects a large opportunity to grow the broader ecosystem and platform.

GYSR v2 introduces a global fee system which sends a small portion of all $GYSR spent to the protocol treasury. The fee amount starts at 20% and can be lowered down to zero, but never raised above 20%. Both the treasury address and the fee amount are set globally by the factory controller.

This change unlocks a huge number of opportunities to support ecosystem growth through grant programs, token redistribution, community contributor rewards, new feature development, etc.

Notably, this fee system also disrupts the circular effect that can occur when Pool owners immediately sell earned $GYSR back into the market. By diverting some portion of that spent $GYSR to productive initiatives, it can improve overall economic stability and growth of the platform.

4.2 $GYSR multiplier

The $GYSR multiplier is a critical piece of the platform. It facilitates a mutually beneficial relationship between investors and pool owners. Investors can further increase their returns while proportionally funding the project for their development efforts.

As a recap from the original paper [2], $GYSR is a universal multiplier of shares representing other assets. These reward assets have highly variable value, total supply, and unlocking schedules. Further, these numerous token markets are completely independent and unaware of each other. All that said, $GYSR must still converge to some common market value. This introduces a fairly complex design problem.

- The multiplier mechanics must be agnostic to the value and supply of relevant assets
- There must be some natural limitation on the multiplier to reduce exploitation
- The multiplier must be responsive to meet Pool-specific usage
- The multiplier must be resilient to manipulation by bad actors

The updated multiplier design stays committed to the original goals, and aims to improve on a few specific focus areas.

- *Responsiveness.* Increase the responsiveness of the usage ratio to be more adaptive to changing asset prices and economic conditions.
- *Fairness.* Section 6.8 of the original whitepaper [2] describes the limitation that a $GYSR bonus is applied to the entire unstake. In v2, we normalize pricing with respect to the size of the stake.
- *Smoothness.* Section 6.7 of the original whitepaper [2] describes the limitation that $GYSR cannot be spent at an amount between 0 and 1. This limitation is removed.

With the above criteria and goals in mind, the following function was designed.

Let

x : number of $GYSR applied to an operation

s : number of staking shares

S : total number of staking shares (including s)

U : usage ratio of $GYSR within a particular Pool

Note that the value U is provided by the underlying reward module and will differ between module types. See section 5 for a definition of the usage ratio calculation for each of v2's original reward modules.

Define the multiplier M_{GYSR}

$$M_{\text{GYSR}}(x, s, S, U) = 1 + \log_{10}\left(1 + \frac{\frac{0.01 \cdot S}{s} \cdot x}{0.01 + U}\right) \quad (1)$$

Note that if $s < 0.01 \cdot S$, we do not scale the \$GYSR input amount with respect to share amount. The result is a simplified equation of the following form.

$$M_{\text{GYSR}}(x, U) = 1 + \log_{10}\left(1 + \frac{x}{0.01 + U}\right) \quad (2)$$

This value, M_{GYSR}, is used in the reward calculation for each reward module, as described in Section 5.

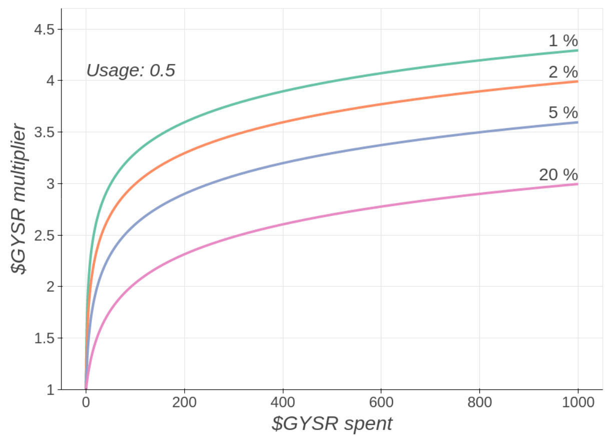

Figure 2: \$GYSR multiplier normalized with respect to share amount

This update achieves the desired improvements through a few specific changes. By calculating usage at the module level, the value can be defined with a tailored and responsive equation. The specific definition for each reward module can be found in section 5.

Additionally, the \$GYSR input amount is now normalized with respect to share amount. This is done by considering 1% of the total staked shares ($0.01 \cdot S$) as a peg and scaling the input (x) inverse proportionally to the share amount (s). This ensures that the value of any \$GYSR spent remains consistent and fair, even as the staked amount increases.

Finally, the multiplier equation has been modified to smoothly handle all \$GYSR input amounts greater than or equal to 0. This includes the 0 to 1 range which was previously restricted.

5 INITIAL MODULES

This section will describe the initial set of modules available immediately with the launch of GYSR v2.

5.1 ERC20 staking module

The *ERC20StakingModule* implements the *IStakingModule* interface as described in section 3.2. This module allows users to deposit an amount of ERC20 token in exchange for shares credited to their address. When the user unstakes, the tokens will be returned to the user and the associated shares will be burned.

5.1.1 Construction. The constructor for this module takes an ERC20 staking token address and the module factory address. These two values are immutable after construction.

5.1.2 Stake. The stake method transfers a specified amount of ERC20 staking token from the user to the module contract, mints a proportional number of shares, and updates user and global positions accordingly. It returns the user address as the credited account and the associated number of minted staking shares.

5.1.3 Unstake. The unstake method transfers a specified amount of ERC20 staking token from the module contract back to the user, burns a proportional number of shares, and updates user and global positions accordingly. It returns the user address as the account and the associated number of burned staking shares.

5.1.4 Claim. The claim method returns the proportional share value for a specified amount of ERC20 staking token without unstaking. It does not modify the user position or global state.

5.2 ERC20 competitive reward module

The *ERC20CompetitiveRewardModule* implements the *IRewardModule* interface as described in section 3.3. This reward module was designed to emulate the behavior of the original v1 Geyser contract. [2]

When a user stakes, they receive credit for the associated number of shares, and immediately begin to accrue share seconds. Share seconds are the primary unit of accounting in this module, and they are burned during reward distribution.

This module distributes a single ERC20 token asset as the reward. The overall reward amount and rate is fixed, but individual earning rates can vary based on participation.

The *ERC20CompetitiveRewardModule* allows users to earn both a time multiplier and a \$GYSR multiplier, which are described in more detail below. These are applied to the user's share seconds when computing their final reward distribution, and allow them to claim a larger portion of the overall unlocked reward pool. This naturally introduces a competitive dynamic to the module's staking and reward mechanics.

5.2.1 Construction. The constructor for this module takes an ERC20 reward token address, the minimum time bonus, the maximum time bonus, the time bonus period, and the module factory address. These values are all immutable after construction.

5.2.2 Fund. The fund method allows the controller to lock up a supply of the designated ERC20 token to be distributed as a reward. When a funding operation is executed, the controller will specify the amount of reward token to deposit, the period over which that reward will be unlocked, and optionally, a time offset to begin the unlocking period.

The module may be funded multiple times, but there is a hard limit on the number of active funding schedules. If the limit is reached, the owner must wait until an older schedule expires before funding again.

5.2.3 Stake. The stake method creates a new position for the account, which stores the share amount and the time of staking. This is used later to compute share seconds, time bonus, and overall rewards.

5.2.4 Unstake. The unstake method updates the account position to remove burned shares and distributes the earned reward to the user.

For each stake, the earned share seconds and time multiplier is calculated based on the timestamp. Similarly to the Geyser v1 contract, the user can optionally spend $GYSR during the unstake operation for an additional multiplier. This amount is passed in through the flexible bytes parameter.

The following equation is used to calculate the earned reward:

$$
\begin{aligned}
&h : \text{user share-seconds burned}\\
&H : \text{total share-seconds (including } h)\\
&M_{\text{time}} : \text{time bonus (defined in section 5.2.7)}\\
&M_{\text{GYSR}} : \text{\$GYSR bonus (defined in section 4.2)}\\
&K : \text{total unlocked rewards}
\end{aligned}
$$

$$R = K \cdot \left(\frac{M_{\text{time}} M_{\text{GYSR}} \cdot h}{H - h + M_{\text{time}} M_{\text{GYSR}} \cdot h} \right) \tag{3}$$

5.2.5 Claim. The claim method is implemented as a wrapped call to unstake and stake. The optional $GYSR amount is passed in through the flexible bytes parameter to the unstake method.

5.2.6 $GYSR usage. As discussed in Section 4.2, the usage ratio should be highly responsive to updated asset pricing and user behavior. Some obvious approaches were to compute usage over a sliding window, defined by a number of operations or by an explicit time period. Unfortunately, using a set number of operations would be vulnerable to manipulation by many small transactions. Similarly, a fixed time window can easily be monitored and manipulated in lower activity Pools.

As a more robust alternative, the portion of total share seconds burned is used to do a weighted update of the usage ratio. This provides a much more secure weighting scheme by tying it directly to claimed rewards, meaning the end user is motivated to act rationally when selecting an amount of $GYSR to spend.

Previously, the entire reward amount from an unstake with $GYSR was considered towards the usage ratio. The usage, u, is now computed on a single transaction based on the portion of *additional* rewards that are due to $GYSR spending.

$$u = \left(\frac{M_{\text{GYSR}} - 1}{M_{\text{GYSR}}} \right) \tag{4}$$

The global usage, U, is then updated as a rolling weighted mean.

$$U = U_{last} - \frac{h}{H} U_{last} + \frac{h}{H} u \tag{5}$$

5.2.7 Time incentive. To incentivize longer-term participation, the module can be configured to give users a bonus multiplier as a function of time staked. This time multiplier is earned linearly over the defined period, and is tracked independently for each stake.

There is no hard limit on the max time bonus, and it can also be removed entirely by setting both the min and max values to 0.

The time multiplier, M_{time} is defined below:

$$
\begin{aligned}
&b_{\text{min}} : \text{minimum time multiplier}\\
&b_{\text{max}} : \text{maximum time multiplier}\\
&b_{\text{period}} : \text{time bonus period}\\
&t : \text{time staked}
\end{aligned}
$$

$$M_{\text{time}} = b_{\text{min}} + \left(\frac{(b_{\text{max}} - b_{\text{max}}) \cdot t)}{b_{\text{period}}} \right) \tag{6}$$

Note: when $t > b_{\text{period}}$, $M_{\text{time}} = b_{\text{max}}$

5.3 ERC20 friendly reward module

The *ERC20FriendlyRewardModule* implements the *IRewardModule* interface as described in section 3.3. This reward module was designed to facilitate a simpler staking process where earned rewards can only increase and cannot be negatively impacted by the actions of others in the Pool.

When a user stakes, they receive credit for the associated number of shares and begin earning at a rate proportional to their share of the entire Pool. As stakes are added or removed from a given Pool, the rate of earnings is adjusted for all stakes in that Pool.

This module distributes a single ERC20 token asset as the reward. The overall reward amount and rate is fixed, but individual earning rates can vary based on participation.

The *ERC20FriendlyRewardModule* allows creators to define a time-based vesting schedule which enforces a prorated penalty for early unstaking. It also lets users earn a $GYSR multiplier, which is applied at staking time so that it can be accounted for predictably. These are both described in more detail below. It is important to note that neither mechanic will ever negatively affect the earned rewards of other users.

5.3.1 Construction. The constructor for this module takes an ERC20 reward token address, the initial vesting multiplier [0-1], the vesting period in seconds, and the module factory address. These values are all immutable after construction.

5.3.2 Fund. The fund method allows the controller to lock up a supply of the designated ERC20 token to be distributed as a reward. When a funding operation is executed, the controller will specify the amount of reward token to deposit, the period over which that reward will be unlocked, and optionally a time offset to begin the unlocking period.

The module may be funded multiple times, but there is a hard limit on the number of active funding schedules. If the limit is reached, the owner must wait until an older schedule expires before funding again.

5.3.3 *Stake.*

The stake method creates a new position for the account. During the stake operation, \$GYSR can be applied as a multiplier on the associated stake amount. Each stake is stored with the raw shares staked, the rewards per staked share prior to the stake, \$GYSR applied, \$GYSR multiplier earned, and the timestamp. The rewards per staked share and the multiplier are each stored because they are dependent on the contract values at the time of stake.

\$GYSR spent during the stake operation is not immediately available for withdrawal by the Pool owner. It is only vested and released at the time of unstake. This avoids a potential economic vulnerability where the same supply of \$GYSR could be spent multiple times in a loop, while still providing value to earlier stakes.

5.3.4 *Unstake.*

The unstake method updates the account position to remove burned shares and distributes the earned reward to the user.

For each stake, the raw rewards earned are calculated based on the number of shares, \$GYSR multiplier (from staking), and reward rate. Then, the vesting coefficient is calculated as a function of staking time and applied to the raw rewards. Unlike the Geyser v1 contract and the competitive module, \$GYSR cannot be spent during this unstake operation.

The following equation is used to calculate the earned rewards:

s : user shares unstaked

M_{GYSR} : \$GYSR bonus (defined in section 4.2)

v_0 : initial vesting multiplier

t_{s} : time at stake

t_{u} : time at unstake

T : total vesting period

r_{s} : global rewards per staked share at time of stake

r_{u} : global rewards per staked share at time of unstake

$$R = \left(v_0 + \frac{t_{\text{u}} - t_{\text{s}}}{T} \cdot (1 - v_0)\right) \cdot (s \cdot M_{\text{GYSR}} \cdot (r_{\text{u}} - r_{\text{s}})) \tag{7}$$

Note that if the time between staking and unstaking is greater than the total vesting period, the equation is reduced to the following:

$$R = (s \cdot M_{\text{GYSR}} \cdot (r_{\text{u}} - r_{\text{s}})) \tag{8}$$

5.3.5 *Claim.*

The claim method is implemented as a wrapped call to unstake and stake. The optional \$GYSR amount is passed in through the flexible bytes parameter to the stake method.

5.3.6 *\$GYSR usage.*

Similarly to the *ERC20CompetitiveRewardModule*, the usage ratio should be highly responsive to updated asset pricing and user behavior. In the *ERC20FriendlyRewardModule*, usage is calculated based on the combined \$GYSR usage of all currently active stakes in the Pool. The following usage formula is defined to represent the portion of total staked shares that comes from \$GYSR multipliers:

S_{raw} : total raw staked shares

S_{GYSR} : total staked shares with \$GYSR applied

$$u = \left(\frac{S_{\text{GYSR}} - S_{\text{raw}}}{S_{\text{GYSR}}}\right) \tag{9}$$

5.3.7 *Vesting schedule.*

The purpose of the vesting schedule is to incentivize longer-term participation and staking in the Pool. The vesting schedule is defined by an initial vesting coefficient and a vesting period. The vesting schedule is the same for the entire module, but is calculated on a per stake basis.

When a user unstakes, the length of time staked is calculated and compared to the total vesting period. If the stake was held for longer than or equal to the vesting period, the stake will receive a 1.0 coefficient. Otherwise, the coefficient is computed linearly between the initial vesting coefficient and 1.0 using the following formula:

v_0 : initial vesting coefficient

t_{s} : time at stake

t_{u} : time at unstake

$$V = \left(v_0 + \frac{t_{\text{u}} - t_{\text{s}}}{T} \cdot (1 - v_0)\right) \tag{10}$$

Any unvested rewards will be returned to the pool and distributed among all other stakers proportionally.

REFERENCES

[1] GYSR. *GYSR core.* URL: https://github.com/gysr-io/core.

[2] Alex Koren and Devin Conley. *GYSR Core v1.* URL: https://www.gysr.io/docs#whitepaper.

[3] Santiago Palladino. *The State of Smart Contract Upgrades.* URL: https://blog.openzeppelin.com/the-state-of-smart-contract-upgrades/.

[4] Gabriel Barros and Patrick Gallagher. *EIP-1822: Universal Upgradeable Proxy Standard (UUPS).* URL: https://eips.ethereum.org/EIPS/eip-1822.

[5] Onur Solmaz. *Scalable Reward Distribution with Changing Stake Sizes.* URL: https://solmaz.io/2019/02/24/scalable-reward-changing/.

[6] Dean Eigenmann and Jorge Izquierdo. *EIP 900 - Staking.* URL: https://eips.ethereum.org/EIPS/eip-900.