Staking proposal

Goals

- Simple design to be understood by many people
- Native support for historical tier logic
- 1:N support for a single staking contract to have many different tier views on the same historical data
- Non-interactive rewards, users should passively accrue value until they exit the system
- Gas efficiency for both reads and writes
- No admin keys required to manage or rescale rewards over time
- Support third party tokens (but not necessarily "exotic" tokens, can be interactive)
- Support "same token" rewards as "revenue share" style distributions

Current situation

We have a TransferTier contract that complects handling tokens (depositing, withdrawing, recording blocks) with reporting (creating a standard uint256 report over recorded block history).

There are some nice things about this:

- It's very simple to understand conceptually
- Users know that when they deposit/withdraw they are guaranteed to move up/down exactly the tiers they specify
- Very gas efficient as we can simply record the tiers rather than the history of every transaction that goes into calculating the tiers

There are some bad things about this:

- There's no internal mechanism for value accrual/distribution based on tokens locked in the contract, all rewards must be downstream based on tiers held
- There's no way to rescale the tier values over time which can be a big problem if the market value of the locked tokens drastically changes relative to when the tier levels were established
- There can only be a single set of tier values associated with deposits so it can be very difficult to create a fair scaling when the nature of rewards may be very different across contexts
- Users cannot do partial deposit/withdrawals so even if we added admin keys to the contract to allow tiers to rescale, users would all have deposited the incorrect values if the tiers did change

Prior art

We can look around and see examples of both interactive and non-interactive claim mechanisms.

Generally interactive claim mechanisms are popular, hence the proliferation of "autocompounder" platforms such as harvest finance. These exist because a common reward scheme looks like this:

- Some reward schedule is decided by the distributor, e.g. "1 million tokens per month"
- End users stake a token and the staking contract tracks a pro-rata share per block/second of the total rewards being distributed
- Users send a transaction to claim the reward and the staking contract logs the time of the claim (to be the new baseline for the next claim) and sends back the reward to the user

Typically the rewards are paid in a different token (e.g. LM mining where the LP token is locked and the reward token is something else) so with the above scheme we almost always see the following play out:

- The users value the token they locked more than the token being rewarded, which makes intuitive sense as they already own the token they are locking, so it was likely their preferred token to begin with (e.g. "putting your X to work", "gresham's law", etc.)
- Users can't compound their gains natively due to the linear distribution, interactive claims, and 2-token stake/reward system, so they dump the reward token to buy more of the staked token, then restake to manually compound their gains at the expense of the reward token
- The reward token faces intense sell pressure due to being distributed directly to users who primarily want more of the staked token, and away from users who might want to buy it, so the token methodically dumps at literally any price, the moment the price goes too low stakers don't hold the reward token, they take their staked token and exit the ecosystem entirely to the next project doing exactly the same thing

The above is the exact business model of platforms like harvest, it's codified mercenary capital. As well as turning the reward tokenomics into an uphill battle, it's very gas inefficient for end-users, with a single claim/restake cycle typically costing $100-500+ on L1.

It's also exactly how our current emissions/vesting contract works, with the caveat that the opcodes on the emissions contract allow for non-linear distribution schemes that can be auto-compounder resistant (e.g. bonuses for claiming less often, or writing the compound interest formula straight into the emissions logic).

As we already support interactive tier based rewards via. Emissions, what we need to achieve next is:

- Arbitrary deposit/withdrawals to allow users to be staking whatever they have on-hand, rather than tied to a fixed tier-diff
- Non-interactive rewards that give every staker the maximum they are entitled to

There are examples of non-interactive rewards systems, Lido is probably the largest.

Lido does ETH 2.0 staking and other managed staking products.

They have a governance token that has an interactive rewards scheme for LPing their staked tokens against native tokens, but that's not relevant to the non-interactive rewards in their system.

stETH is the "staked ETH" token that is minted whenever someone deposits ETH into the staking contract.

stETH has a dynamic balance function and so is easiest to understand but also not compatible with most defi (including Rain's systems) because dynamic balances generally break the internal ledgers of contract that hold tokens.

Every day more 2.0 ETH is minted for Lido validators and deposited in the staking contract, Everyone who holds stETH sees their balance go up pro-rata according to the newly deposited 2.0 ETH.

In theory this means that 1 stETH is worth exactly 1 ETH. In practice there are all kinds of liquidity concerns and other risks associated with the ETH2.0 rollout, but that is out of scope for this document.

Relevant to us is the associated wstETH token which has static balances and so is fully defi compatible. Importantly, holding wstETH over some time period is equivalent functionally to holding stETH over the same time period.

How is this achieved with a static balance? It's simple, when stETH is deposited and withdrawn to mint/burn wstETH, there is no guarantee that the 1 wstETH will be worth 1 ETH, in fact 1 wstETH is continually increasing the amount of ETH that it represents a claim on as ETH is continually being deposited into the relevant contract.

Proposal

A new factory contract that deploys configurable staking contracts. Each deployed staking contract is bound to some specific token X that it will accept deposits for and mint stX (staked X) tokens.

Anon users deposit token X and receive newly minted stX tokens.

The stX supply increases by the same % as the X deposited % increase.

For example, if 1000 X is already in the contract and 2000 stX are in circulation, then a user deposits 100 X they have increased the deposits by 10% and so can mint a 10% increase in stX supply for themselves, i.e. 200 stX tokens.

The initial minting ratio is arbitrary so can be configured to anything by the factory deployer, can be 1:1 or 1:2 or anything else.

If no outside funds are deposited then the staking contract looks exactly like an ERC20Wrapper contract from Open Zeppelin, where a minted token "wraps" a deposited token in some ratio (Open Zeppelin is 1:1 but the ratio is superficial, it's like saying "i have 1 dollar" or "i have 100 cents" and meaning the same thing) https://docs.openzeppelin.com/contracts/4.x/api/token/erc20#ERC20Wrapper.

We will build 2 additional features on top of a vanilla wrapper:

- We calculate withdrawals based on the current X token supply of the staking contract rather than the initial ratio
- We will track the stX mint/burn logs much like Open Zeppelin Checkpoints https://docs.openzeppelin.com/contracts/4.x/api/utils#Checkpoints

Balance based withdrawals and deposits

The initial ratio configured for a staking contract will only be respected when the X token balance of the contract is 0. After that point deposits and withdrawals will mint and burn according to the balance of X in the contract and total stX supply.

This means that if some X token is deposited to the staking contract without calling the stX mint function (i.e. a normal token transfer) then it is effectively being transferred pro rata to all stX token holders non-interactively.

Mints (deposit X):

stXminted / stXtotalSupply* = Xdeposited / Xpool*
=> stXminted = (stXtotalSupply x Xdeposited) / Xpool

Burns (withdraw X):

stXburned / stXtotalSupply* = Xwithdrawn / Xpool*
=> Xwithdrawn = (stXburned x Xpool) / stXtotalSupply

* total supply and pool amounts before the mint/burn operation.

An example to show this. Say 1000 X is already in the contract and 2000 stX are in circulation then as above, a 100 X deposit will yield a 200 stX mint. Similarly a 200 stX burn will withdraw 100 X. This is linear all the way to zero, if 200 stX is burned then the balances will be 900 X and 1800 stX, so the ratio never changes for minting or burning.

Now say 1000 X is deposited directly to the staking contract with no minting. In this case the new balance is 2000 X deposited with 2000 stX outstanding. Now the ratio is 1:1 not 1:2, which means that burning 200 stX tokens yields 200 X tokens, not 100 X tokens. It also means that the ratio is 1:1 for further deposits, so depositing 100 X tokens after the external deposit now only mints 100 stX tokens, not 200. If the deposited X tokens are doubled without minting again to 4000 X deposited then the ratio is 2:1 which means burning 200 stX tokens yields 400 X tokens and depositing 100 X tokens only mints 50 stX tokens.

That looks like:

stXminted = (stXtotalSupply x Xdeposited) / Xpool1
Some distribution event => Xpool2 = Xpool1 + Xdistribution
Burn all minted stX => stXburned = stXminted
Final X claim => Xwithdrawn = (stXminted x (Xpool1 + Xdistribution)) / stXtotalSupply

From this we can see:

- Rewards follow distribution amounts exactly proportionally, whether linear, exponential, ad-hoc, etc.
- As more stXtotalSupply increases due to other people minting, future rewards are diluted (although current entitlements are preserved due to "buy in" requirements on minting)
- If the user could gain more stX they could withdraw more X (but in reality they cannot do this due to the internal ledger restrictions, see below)

Note that this means that unless every last X token is withdrawn, which would reset the ratios back to the start ratio configured on the contract, the ratio between X tokens and stX tokens can only ever increase and only for non-minting deposits. For example, in the above scenario with 4000 X deposited at a 2:1 ratio with 2000 stX tokens circulating, let's say 1995 stX tokens are burned such that only 10 X tokens remain in the contract and 5 stX tokens are in circulation. The ratio is still 2:1 so if someone were to then deposit 1000 X tokens they'd receive 500 stX tokens ((1000 / 10) x 5). This preserves everyone's stX token claim on X tokens deposited during their staking period in addition to their own without requiring an internal ledger. 5 stX tokens still withdraws 10 X token ((5 / 505) x 1010) and 500 stX tokens withdraws 1000 X tokens ((500 / 505) x 1010).

This noninteractive system is easy to reason about in terms of percentages and has the advantage that any autocompunder-like interactions for a 1-token contract can only disadvantage a user by wasting gas to get an equivalent or worse result. This is because withdrawing/claiming any rewards can only re-mint the same or fewer stX tokens that

were burned for the withdrawal. For example, assume a 1:1 system with 1000 X token and 1000 stX tokens and a user withdraws 10% of the stX (100) tokens to withdraw 10% of the X (100) tokens. Now there are 900 of each associated with the contract. If the user then redeposits their 100 X token they increase the X supply by 1/9th and so also increase the stX token supply by 1/9th, i.e. minting the same 100 stX tokens once more.

Noninteractive staking means that every staked user is guaranteed to receive their maximum possible entitlement of all distributed X tokens.

Noninteractive multi-token systems for a single contract are NOT supported. This is because a user could repeatedly deposit and withdraw the same X tokens to claim a share on third-party arbitrarily many times, thus draining the contract of external rewards without changing the overall X token balance. This attack works because the user is not required to deposit a corresponding share of the third party tokens to mint a claim on what is already deposited.

A combination of non-interactive and interactive rewards is recommended for long term staking campaigns where the non-X token rewards are based on the staking contract deposit logs and emissions/vesting contract(s).

Another option is to use an aggregator/routing service like 0x protocol to convert non-X to X tokens automatically onchain before transferring them to the staking contract. A wrapper contract can be written as the recipient of Rain platform fees to feed directly into a staking contract. In this way the non-X token fees are "pre-dumped" in a sense, so it may or may not be appropriate situationally (it puts protocol-level buy pressure on X token at the same time as distributing it to existing stakers, at the expense of all non-X tokens).

Deposit and withdrawal block logs

Every deposit and withdrawal should be logged against the depositing/withdrawing account to create a simple history that can build a report dynamically.

Each deposit should log a block number and the amount of stX token minted (not X token deposited). Logging the stX token minted and burned rather than X token deposited guarantees that the ledger remains consistent across arbitrary deposits and withdrawals.

Consider the case where we try to lodge X token deposits. As there may be external X token deposited, increasing the amount that a user can withdraw over time (see above), it would be entirely possible (very likely by design) that if a user burns all their stX token they will withdraw MORE X token than they ever deposited. This additional withdrawal would mean that more ledger entries are deleted (see below) than for the stX tokens that created them (impossible, or at least very undesirable).

Note that as the stX minting ratio monotonically increases over the lifespan of the staking contract (more X token must be deposited for the same number of stX token minted), it becomes more difficult to achieve a given tier value over time, all else equal. It is expected that new report view contracts will be deposited from time to time to rescale this difficulty, thus providing long time stakers an additional "soft" boost to their tier over time (i.e. when a rescaled contract is deployed they may retroactively find themselves at a higher tier).

Summary: we create a ledger of mints/burns of stX token which are guaranteed to always sum to 0 when everyone fully exits the system. A ledger of X token can't work because we're intending to have everyone withdraw MORE X than they deposit, which will cause the ledger to go negative on full exit (impossible).

As we want to support arbitrary withdrawals, the deposits ledger should be wound back in FILO (first in last out) order, to maximise block ages for depositors in subsequent tier calculations.

For example, a user mints 500 stX tokens at block 5, 250 tokens at block 10, 300 tokens at block 20. They would have a ledger like:

05 | 500
10 | 750
20 | 1050

Note the ledger is cumulative which increases gas on writes and allows less calculations on reads.

If the user burns 400 stX tokens it would consume the full 300 token diff on block 20 and 100 tokens from block 10. The burn action itself is not logged, it is simply a destructive action that consumes prior deposit logs. The ledger would then look like:

05 | 500
10 | 650

If the user then mints 500 stX tokens at block 30, as the burn is an opaque and destructive operation, it would simply appear as a new entry:

05 | 500
10 | 650
30 | 1150

This system is destructive but simple as to be (hopefully) gas efficient and also guarantees that any log entries visible are contiguous (uninterrupted) which is important for calculating claims downstream. We do NOT want lost tiers to be reinstated, unless at a newer block of course.

From the basic logs a tier report view system can be overlaid, where an array of up to 8 values is passed in and a standard uint256 report is produced representing 8 blocks.

For example, the final log above could have several value arrays passed and produce the following block outputs (using 3 values instead of 8 for illustrative purposes):

[100, 200, 300] => 05, 05, 05
[300, 600, 900] => 05, 10, 30
[500, 1000, 1500] => 05, 30, n/a

In this way a single staking log can power many downstream rewards systems, even allowing for rescaling of tiers by deploying new views over existing data, rather than using admin keys to try to rescale an existing tier contract in situ.

Note also that the log is REQUIRED for a user to unstake. If a user attempts to withdraw X tokens by burning stX tokens, they cannot do so if they do not have a corresponding internal ledger that covers at least the amount of stX tokens being burned. The withdrawals are locked to the user that deposited, and this means that transferring stX tokens does NOT transfer the ability to withdraw X tokens. Additionally, stX tokens do NOT (probably, unless a second-order staking tier contract is created for ststX tokens) give any interactive rewards from downstream tier contracts, only the underlying ledger of mints/burns will inform the report logic.

This means that by default stX tokens really have no value to anyone other than their original minter, but because they can be freely transferred they can also be traded, which implies that there MAY be some market for them, but it is beyond the scope of this document to consider the implications or implementation of achieving a liquid market for stX tokens. UNLIKE Lido (see above) there is no reason to try to achieve a liquid stX/X token market (e.g. like stETH/ETH) because it is always possible for minters to burn and remint stX tokens at will, just like there is no need to establish a wETH/ETH market to enable people to access either.

The market value of a stX token, assuming no external factors, indicates desire for minters to want to burn at that moment to retrieve their X token, but cannot because they previously sold their stX tokens (leverage?). This is impossible to estimate in the general case as every project has its own rewards and tokenomics.

Of course, maintaining this log in solidity begs the gas question. We can look to Open Zeppelin for the Checkpoints struct which feeds into their voting system, that calculates voting power at a particular block very similar to calculating an array of thresholds as we do with tiers.

Stripping back their convenience methods and structs, the fundamentals are simple and aligned with what we already do for tiers; they reserve the 32 high bits for a block number for each checkpoint, and the remaining 224 bits for values. This allows 2 ^ 224 tokens to be deposited/withdrawn for each ledger entry, which is ~2.7 x 10 ^ 67 which for a standard 18 decimal ERC20 gives up to ~2.7 x 10 ^ 49 tokens per deposit/withdrawal…

which should be plenty for almost any project. With this approach each ledger entry can fit in a single uint256.

Reads will be happening a LOT relative to writes, and storing the ledger in blockchain storage automatically makes reads as expensive as possible (bad). To a large degree this is unavoidable in order to achieve a flexible internal ledger, and so users should be advised NOT to make many deposits as it will make all their subsequent reads more expensive. As a design principle the cost of downstream reads on a user's staking should also be borne by the user, this will encourage them to optimise their ledger (keep it short). All pull-based claim mechanisms do this, but any kind of push-based distribution mechanism could be griefed by a user creating a very long ledger with dust size deposits.

We can likely optimise the storage reads by looping over the ledger directly with assembly rather than loading the entire ledger from storage before attempting to calculate a view over it. In the examples above, we know the user has reached all 3 tiers by block 5 after viewing a single ledger entry, so we can short-circuit further expensive storage reads on the underlying deposits ledger, having fulfilled all the tier values of the view calculation.

It's still not ideal that we are looping over storage reads, so it is important that we measure and review gas costs in practice, keeping an eye on overall feasibility. As mentioned above, crypto in general is littered with staking mechanisms that are so expensive to maintain that most users are simply priced out of rewards that are meaningful compared to the size of their staked tokens. Hopefully the combination of noninteractive staking, compressed log entries and assembly/short circuiting view logic is enough to sufficiently mitigate this pain, and provide a foundational building block for native project token staking.

Impact to the codebase

Contracts

The following contracts may all be deleted or heavily reworked to fit this paradigm:

- Balance tier
- Transfer tier
- Value tier

Overall the final size of the codebase should be comparable after staking is added to the current LOC.

Interfaces

The ITier interface may be extended to support an arbitrary bytes parameter for report.

This would be a breaking change for existing ITier ecosystems, so maybe manifest as an ITierV2 or similar. Perhaps there is a backwards compatible implementation that preserves the current report functionality.

In this case the additional bytes could be a value list that the staking contract can directly return reports for, this would make view contracts very simple to implement, while allowing the staking contract itself to be ITier compatible.

If we don't change ITier in this way then the staking contract will have to expose a non-standard report function to handle values.

Note that adding data to ITier will likely be necessary in the future anyway to support NFT ID based reporting (rather than NFT balance).

Real world example: PGEN staking

Consider polygen PGEN token.

Polygen can set fees for each sale with the existing Sale contract.

Fees are paid in the reserve currency of the raise so will need converting.

Polygen can write a wrapper contract as the recipient of fees from their website.

Anon can call a function on the wrapper to sweep fees into the wrapper.

Anon can call a function on wrapper to trade collected fees in aggregate through 0x or similar to convert reserve to PGEN via. AMM (effectively a "buyback").

PGEN tokens post-conversion can be deposited without minting stPGEN into the PGEN staking contract (effectively revenue share from fees).

All the above can be implemented trustlessly without admins on the wrapper contract.

In addition to revenue share, staking internal ledger can feed into tier based rewards for PGEN stakers. Examples include exclusive access and discounts on future raises on the Polygen platform.