Superpositive Non-Custodial Yield Generation Protocol

Elliott G. Dehnbostel

June 2025

Contents

Abstract

We present a non-custodial token wrapper that supports *concurrent* staking of a single balance across multiple on-chain yield programs ("protocols") while maintaining snapshot-safety, deterministic gas costs, and explicit opt-in haircut mechanics. The design eliminates trusted custody, allows up to eight parallel protocol memberships per account, and exposes a flash-loan interface with zero fixed fee. Rigid invariants and conservative state transitions minimise re-entrancy and accounting risk. All monetary units are fixed-point with six on-chain decimal places (10^{12} wei per wrapped token), enforcing a hard global supply ceiling of $2^{64}-1$ tokens. This paper is intended to be *implementation-complete*; it avoids language-specific details and provides platform-agnostic pseudocode for every critical routine.

1 Introduction

1.1 Motivation

Layer-1 assets that participate in multiple yield mechanics typically incur fragmentation (liquidity split across wrappers) or custodial risk (delegated intermediaries). Our goal is a single, fungible wrapper that:

- Allows simultaneous participation in *several* external protocols without duplicating the underlying collateral.
- Guarantees non-custodial ownership: withdrawals always redeem the native asset directly.
- Preserves fungibility even under protocol failure via deterministic haircut logic.
- Requires no per-protocol loops proportional to global user count (gas cost is constant or logarithmic).
- Offers built-in flash loans for capital efficiency and attack surface testing.

1.2 Design Goals

Safety before yield Security invariants dominate APY optimisation. Any feature that jeopardises balance integrity is rejected.

Predictable gas Transitions must have worst-case costs bounded by a compile-time constant.

Explicit dilution Protocol-level losses ("haircuts") are opt-in, signed by the protocol controller, and front-run-resistant.

Skeptical assumptions Counter-parties may be malicious; flash borrowers may default; network re-ordering is normal.

1.3 Non-Goals

- Automatic governance or DAO approval.
- Arbitrary user-supplied smart-contract hooks.
- Unlimited concurrent protocols (we cap at eight for packing efficiency).

2 System Model

2.1 Actors

User Holds wrapped tokens, optionally joins protocols, triggers harvests.

Protocol Controller Off-chain or on-chain entity authorised to inject yield, signal haircuts, and collect burned balances for a given protocol identifier *pid*.

Lender Any contract invoking the flash-loan interface.

2.2 Units and Fixed-Point Arithmetic

- One wrapped token represents 10^{12} wei of the native currency.
- All balances are stored as $u64$. The maximum supply is therefore $2^{64}-1 \approx 1.84\times10^{19}$ tokens.
- Yield accumulators use unsigned Q64.64 [1] to avoid rounding drift.

2.3 State Objects

Account

- **bal**: current liquid token balance ($u64$).
- **ptr[0..2]**: indices into the *membership array* identifying at most eight concurrent protocol stakes.

Protocol

- Controller address, minimum stake, lock window (blocks).
- *inBal*: aggregated stake currently contributed by members.
- *outBal*: total haircut signalled but not yet burned.
- *burned*: tokens actually removed from members.
- *collected*: burned tokens already redeemed by the controller.
- *yAcc*: cumulative yield per staked token, Q64.64.

Membership

- *pid*: associated protocol.
- *resPtr*: index into the *reserve snapshot*.
- *unlock*: block height after which the user may leave.
- *stake*: cached user balance *at last snapshot*.

[1] A 128-bit fixed-point format with 64 integer bits and 64 fractional bits.

Reserve Snapshot Holds the protocol-level counters (*inStart*, *outStart*, *yStart*) captured when the user last interacted and the *joinMin* then in effect.

2.4 Concurrency Bound

Exactly eight membership slots keeps the per-account struct within a single 256-bit EVM slot, substantially reducing SSTORE gas while preserving enough parallelism for practical diversification.

3 Lifecycle Operations

This section specifies every externally visible transition in precise pseudocode. All routines are wrapped in an implicit `nonReentrant` guard.

3.1 Deposit Native Asset

```
require(msg.value > 0 && msg.value mod 1e12 == 0)
amountTokens <- msg.value / 1e12
harvest(sender)
addBalance(sender, amountTokens)
```

The contract mints wrapped tokens at 1:1e12 and holds the underlying currency.

3.2 Withdraw Native Asset

```
require(requestedTokens > 0)
harvest(sender)
require(balance[sender] >= requestedTokens)
enforceMinStake(sender, balance[sender] - requestedTokens)
subtractBalance(sender, requestedTokens)
sendNative(sender, requestedTokens * 1e12)
```

A withdrawal fails if any active membership would fall below its recorded minimum stake.

3.3 Create Protocol

```
require(controller != 0x0)
index <- protocols.push(new Protocol(...))
emit ProtocolCreated(index,...)
```

The creator defines *lockWindow* (minimum commitment in blocks) and *minStake*.

3.4 Join Protocol

```
require(account.balance >= protocol.minStake)
slot <- firstEmpty(account.ptr)
require(slot != NONE)
protocol.inBal += account.balance
membership <- allocate()
reserve    <- allocate()
reserve.snapshot(protocol)
account.ptr[slot] <- membership.id
```

The join action implicitly stakes *the caller's entire balance*. Users seeking per-protocol sizing should split holdings across multiple wallets.

3.5 Leave Protocol

```
harvest(sender)
require(block >= membership.unlock)
protocol.inBal -= membership.stake
free(membership, reserve)
account.ptr[slot] = 0
```

Leaving restores full liquidity and lifts the min-stake restriction for that protocol.

3.6 Transfer Tokens

Before altering balances, the contract:

1. Harvests both sender and recipient.
2. Confirms the sender is not inside the lock window of any protocol.
3. Checks the sender's post-transfer balance against each recorded *joinMin*.

If all conditions succeed, balances are updated atomically.

3.7 Add Yield

Anyone can inject yield into any protocol.

```
require(yieldTokens > 0)
require(protocol.inBal > 0)
moveTokens(controller -> contract, yieldTokens)
q <- (yieldTokens << 64) / protocol.inBal
protocol.yAcc += q
```

The Q64.64 accumulator ensures high-precision proportional distribution during the next harvest.

3.8 Harvest (User-Triggered or Batch)

Algorithm 1: HARVEST(user)

```
for each membership in user.activeSlots:
    protocol <- protocols[membership.pid]
    reserve  <- reserves[membership.resPtr]

    -- Haircut --
    if protocol.outBal > reserve.outStart:
        delta  <- protocol.outBal - reserve.outStart
        base   <- max(reserve.inStart - reserve.outStart, 0)
        cut    <- (user.balance * delta) / base if base > 0 else 0
        cut    <- min(cut, user.balance)
        user.balance -= cut
```

```
        protocol.inBal -= cut
        protocol.burned += cut

    -- Yield --
    if protocol.yAcc > reserve.yStart and user.balance > 0:
        dy  <- protocol.yAcc - reserve.yStart
        owe <- (user.balance * dy) >> 64
        owe <- min(owe, contract.balance)
        user.balance += owe
        contract.balance -= owe

    -- Update Snapshots --
    reserve.inStart  = protocol.inBal
    reserve.outStart = protocol.outBal
    reserve.yStart   = protocol.yAcc
    membership.stake = user.balance
```

This routine is $O(s)$ where $s \leq 3$ is the membership slot count.

3.9 Haircut Signal and Collect

A haircut reflects a realised loss or insolvency in the external protocol.

- **Signal:** Controller specifies an absolute token amount. The system simply records the delta in *outBal*.
- **Burn:** During each affected member's next harvest, a proportional slice is destroyed (burned).
- **Collect:** Once the cumulative burned amount exceeds prior collections, the controller may mint an equal quantity to itself.

Because burns happen lazily, a user that never interacts again will not subsidise later withdrawals; this is a deliberate trade-off favouring gas over liveness. Batch harvesting is possible by off-chain "keepers" to enable protocols to self-ensure necessary liveness.

3.10 Flash Loan

The flash-loan interface conforms to ERC-3156. Key invariants:

1. Borrowed supply may not exceed the unused portion of the global cap.
2. The borrower must return exactly the principal within the same transaction via `approve`.
3. The borrower may *not* hold a positive wrapped balance or active membership before or after the loan.

The protocol charges no fixed fee; implementers should layer slippage or dynamic premiums if required.

4 Security Considerations

4.1 Re-Entrancy

All externally callable functions are implicitly non-re-entrant. Internal calls that adjust user and protocol balances occur strictly after the re-entrancy guard has set the entered flag.

4.2 Min-Stake Enforcement

Transfers and withdrawals compute the prospective balance and compare it against each stored *joinMin*. Edge cases such as partial burns or yields cannot lower the threshold without the user explicitly leaving.

4.3 Overflow Boundaries

- $u64$ balance fields saturate at $2^{64}-1$. Mint, deposit, and yield routines each guard against overflow.
- Protocol accumulators use $u128$ and are similarly bound.

4.4 Haircut Abuse

Controllers could grief users by signalling excessive haircuts then injecting equal yield, causing churn. Users should vet controllers; the contract deliberately offers no governance escape hatch.

4.5 Flash-Loan Attacks

Because the borrower cannot enter memberships or leave stray allowances, traditional reserve-drain vectors are ineffective. However, integrators should assume flash-loaned liquidity can manipulate off-chain pools within the transaction.

5 Gas and Storage Optimisation

- All account data fits in one storage slot; membership and reserve rows are recycled via freelists to avoid unbounded growth.
- Propagation of balance deltas touches at most eight protocols, each updating one 256-bit word.
- Yield calculations use bit-shifts rather than division where possible.

6 Parameterisation

6.1 Lock Window

The controller sets $lockWindow$ (in blocks) during protocol creation. Short windows improve liquidity but encourage adversarial withdrawal right before a haircut; implementers should choose a value reflecting external settlement latency.

6.2 Minimum Stake

minStake can be raised or lowered for *future* joiners only; existing members retain the snapshot at join time. Very small minimums increase state bloat; very high minimums reduce inclusivity.

7 Limitations and Future Work

- The concurrency bound of eight simplifies storage packing but may be insufficient for highly diversified treasuries.
- Lazy haircut application means dormant wallets either never pay or must be harvested by a keeper. A check–pointing incentive (e.g., block-subsidised harvest) could enforce eventual settlement.
- No permit (EIP-2612) support is included; adding typed-data signatures would improve UX.
- Integrating dynamic flash-loan fees could turn the wrapper into a revenue source.

8 Implementation Checklist

1. Unit tests for every arithmetic branch, including edge overflows at $2^{64}-1$.
2. Fuzz harness covering random joins/leaves under simultaneous haircuts.
3. Static-analysis for re-entrancy, approval race conditions, and storage aliasing.
4. Integration tests with at least two external yield protocols.

9 Conclusion

We have specified a compact yet powerful wrapper that supports concurrent non-custodial yield extraction while maintaining predictable gas and strong accounting guarantees. By enforcing explicit snapshot boundaries and rejecting implicit custody, the design aims to stay robust under adversarial market conditions. The provided algorithms and invariants should enable straightforward re-implementation in any deterministic smart-contract environment.

A Detailed Pseudocode

Full listings of every helper (Propagate, AddBalance, freelist management, etc.) are available in the project repository. Adaptations to non-EVM chains need only substitute the 256-bit storage packing and overflow checks.

B Glossary

Haircut Forced proportional burn reflecting protocol-level loss.

Harvest User-triggered settlement that applies pending haircuts and credits yield.

Snapshot Immutable record of protocol counters at the time of user interaction.

Controller Privileged entity managing a protocol's parameters and reserves.