Nirn: A Compositional, Permissionless, Extensible Yield Aggregator

Dr Laurence E. Day & Dillon Kellar
Indexed Finance

{laurence, dillon}@indexed.finance

V1.0

2 August 2021

Abstract

In this paper we present Nirn, a DeFi yield aggregator protocol that interfaces with several of the largest lending platforms on the Ethereum mainnet. Capital within a Nirn vault for a given asset can be allocated in chunks across several such platforms simultaneously in such a way as to maximise the average returns for vault depositors.

Rather than relying on whitelisted accounts to determine how and when a Nirn vault should rebalance, *anyone* can propose a new weighting that produces returns higher than the present rate at any time, which is checked on-chain for validity before being executed.

Nirn is designed to be modular and future-proof: the introduction of additional lending protocols can easily be accommodated through the deployment of a protocol adapter, which subsequently exposes new lending markets to the existing vaults without any need for migrations.

Here we describe - in brief - the technical fundamentals of Nirn's implementation, detail the vault rebalancing process, show how an offchain vault optimiser can be implemented as an instance of the {0, 1}-knapsack problem, and discuss access controls and security.

Contents

1 Motivation

The average DeFi user who wishes to use their assets productively is spoilt for choice in terms of lending platforms: Compound, Aave, and Fulcrum to name a few. Most - if not all - of these platforms support lending markets for all of the major stablecoins and leading DeFi protocol tokens, such as `DAI` and `UNI`.

These various protocols can be - and often are – viewed as black boxes from the perspective of the average lender: security concerns notwithstanding, they will deposit their funds with the protocol promising the best returns.

However, the nature of these lending protocols is such that the rate of return on loaned assets is inversely proportional to the size of the capital pool available to borrowing counterparties. More bluntly: in the presence of an excess of available capital, borrowing rates - and returns on loaned assets - decrease.

These protocols are mostly siloed from each other, leading to the situation whereby lending a given amount of capital to a particular protocol decreases the rate of return relative to others. This is a problem that was first tackled by Yearn, which automatically migrated capital to whichever protocol was offering higher returns at the time of calling.

Nirn is an evolution of this concept: a wholly-automated yield aggregator, operating across approved protocols via a common interface and deferring the choice of how yield is generated to the black-boxes of the protocols themselves.

Nirn is *compositional*: each vault is associated with an array of weightings, which corresponds to the proportions of the pool to deposit with each supported protocol, so as to minimise the reduction in the rate of return across the weighted average of all exposures.

Nirn is *permissionless*: it does not rely on whitelisted accounts that determine how and when a vault should be rebalanced: *anyone* can propose a new weighting of funds for a given vault across its various supported token adapters at any time, however Nirn will only accept and implement a reweighting if the resulting rate of return is verified as being higher than the current rate.

Nirn is *extensible*: new lending protocols can be integrated by simply deploying a suitable protocol adapter. All lending markets supported by the new protocol are thereafter deployed as token adapters and logged by an adapter registry, becoming immediately available for usage by the relevant asset vaults.

1.1 Supported Protocols

At the time of launch, Nirn supports the following protocols:

- Aave V1 & V2
- Compound
- C.R.E.A.M. & Iron Bank
- Fulcrum (bZx)

The following protocols are ready for integration, pending final tests:

- dYdX
- Rari Fuse

1.2 Supported Assets

Asset-wise, Nirn supports any ERC20 token that is listed by at least one of the supported protocols above. However, Nirn is best utilised for those assets which have lending markets on multiple protocols, as its primary benefit is in the distribution of assets between markets. At the time of writing, Nirn supports 98 assets across 169 token adapters, with:

- 71 assets supported by a single protocol,
- 9 assets supported by two protocols,
- 18 assets supported by three or more, and
- 12 assets supported by at least four protocols.

The assets most suited to Nirn are those in the latter categories, containing:

```
DAI, TUSD, USDC, USDT, sUSD, BAT, LINK, KNC, MKR,
ZRX, COMP, WBTC, WETH, BUSD, YFI, AAVE, UNI, SNX
```

1.3 Purpose & Code Repository

Nirn is a standalone, low-fee protocol: we encourage you to use it if you wish to engage in *pure lending* without having to monitor yield rates across several platforms. However, Nirn was conceived - and built - for integration into the Indexed Finance platform for passive portfolio management, enabling yield-bearing ETF tokens with a single wrapped proxy representing a given index constituent across multiple lending protocols simultaneously.

Nirn has been open-sourced under an MIT license - the code can be viewed at:

`http://www.github.com/indexed-finance/nirn`

2 Components

In this section, we present a brief rundown of the core mechanics of Nirn; the adapter registry, protocol and token adapters, and vaults. Throughout this section we will periodically refer to certain functions to illustrate key concepts: however the material presented here is not exhaustive, and for a definitive specification, we direct the reader to the GitHub repository referenced at the end of the previous section.

2.1 Adapter Registry

Central to Nirn is the *adapter registry*, a contract which records the addresses and metadata of all of the adapters through which Nirn allocates capital within vaults in a standardised fashion.

The registry itself serves two primary purposes: registering token adapters when they are deployed through an overarching protocol adapter, and verifying that a token adapter is both approved and active when validating a proposed reweighting via a Nirn vault.

2.2 Adapters

Adapters are instances of the standard interface that Nirn utilises to abstract over the various operations that must be exposed in order for vaults to operate. Nirn has two 'classes' of adapter: protocol and token.

2.2.1 Protocol Adapters

Protocol adapters are factories that are used to launch and record the token adapters utilised by Nirn vaults. They are quite light in terms of the functions they must support:

```
interface IProtocolAdapter {
  function protocol() external view
    returns (string memory);

  function getUnmapped() external view
    returns (address[] memory tokens);

  function map(uint256 max) external;
}
```

The role of protocol adapters is fundamentally that of administration: mapping out the list of assets supported by a protocol, determining which of them currently have active lending markets, and deploying token adapters as appropriate.

Assets with inactive lending markets are recorded, pending such time as they are reactivated, at which point an 'unfreeze' function can be called that verifies the market is indeed active and subsequently deploys a new token adapter if one does not already exist.

Similarly, in the event that lending markets for *existing* token adapters are frozen or unfrozen, it is the protocol adapter that is responsible for updating the adapter registry as necessary.

2.2.2 Token Adapters

Token adapters are the points of contact between a Nirn vault - or any other contract using Nirn - and the supported lending protocols. They create an interface through which any contract can use any supported protocol without understanding the details of how it handles deposits, withdrawals or conversions.

Alongside the functions you would typically expect to see in an interface for lending protocols - such as withdrawing, depositing and querying balances - two key functions that token adapters must support are:

- `getAPR() returns (uint256)`
- `getHypotheticalAPR(int256 delta) returns (uint256)`

`getAPR()` returns the annualised interest rate that the wrapper token reports, assuming that all pending interest from borrowers of the asset has been accrued, and is inclusive of any additional sources of yield, such as governance token rewards (i.e. `COMP` when lending on Compound).

`getHypotheticalAPR(delta)`, on the other hand, returns the annualised interest rate of the wrapper token if `delta` tokens are added or removed from the pool. For lending protocols, this can be directly determined by referencing the underlying interest rate model of the protocol, and for yield aggregators this can be *estimated* by adjusting their growth.

The latter function acts as the skeleton upon which a vault reweighting is validated: when a new weighting is proposed, the current liquidity deltas are used in calls to `getHypotheticalAPR` to determine the *current* yield, the necessary liquidity deltas are calculated and used to retrieve the *proposed* yield, and if the latter exceeds the former, the reweighting is executed.

All lending protocols are slightly different: some choose not to utilise wrapper tokens (e.g. dYdX), whereas others have governance rewards that can only be claimed by the depositor (Aave V2). In all cases, token adapters are designed to abstract away implementation details such that depositors receive their full interest without ever interacting with the lending markets directly. This sometimes requires that they extend their scope beyond wrapping function calls.

Adapters for lending markets which do not have tokens implement ERC20 token wrappers for the markets in question to preserve the assumption that all adapters have both an underlying and wrapper token. When a lending market only allows the owner of a token to execute disbursal of payments to the owner, such as with `stkAave` rewards for Aave V2, the adapter holds deposits on behalf of users. All adapters for lending markets with secondary reward tokens handle disbursals at the end of withdrawal transactions.

One thing that token adapters do not handle is the conversion between reward token and underlying token. This is considered to be outside of their scope: they are intended only to abstract the specific behaviour needed to deposit, withdraw and claim interest from lending markets.

Some points of interest related to the implementation of token adapters:

- *Conversions*: token adapters report precise values for conversion rates. Queries to `toWrappedAmount` always return the exact amount of the wrapper token that can be minted for a given amount of underlying tokens, and queries for `toUnderlyingAmount` always return the exact amount of underlying tokens that can be redeemed by burning a given amount of wrapper tokens.

  This often requires token adapters to perform calculations that are typically handled internally by the underlying protocol. For example, the exchange rates for wrapped Compound tokens ('cTokens') are dependent upon the accrual of interest within the token contract, but the Compound external view functions do not account for pending interest. As such, token adapters for cTokens calculate this interest in order to obtain the precise exchange rate.

- *Dust*: if it is impossible to avoid dust without a second transaction, a maximum of 1 wei of dust is accumulated per deposit or withdrawal.

- *Rates*: APRs are inherently imprecise because they are annualised from short-term interest rates. As such, adapters use precise values for these interest rates before annualising wherever possible.

  When necessary, precision in the short-term rate is sacrificed to preserve accuracy. For example, the precise per-block interest for lenders can be queried from Compound, but Compound also distributes rewards in the form of the governance token `COMP`. Nirn vaults do not have the ability to instantaneously sell `COMP` as it is earned: this - along with the fact that there is no direct conversion rate between `COMP` and any other underlying asset - makes it impossible to obtain a *precise* APR for `COMP` rewards. A more accurate result can be obtained by including these rewards in the reported APR of the adapter by using a price oracle.

There are two 'categories' of token adapter: ERC20 adapters and Ether adapters. In the latter case, `WETH` is always used as the underlying, regardless of whether the protocol in question lends Ether or Wrapped Ether.

The behaviour of the withdraw and deposit functions of an Ether token adapter is consistent with that of the ERC20 adapters, transferring `WETH` to/from the caller. Corresponding functions are also defined for each deposit/withdraw method which allow funds to be deposited or withdrawn using Ether – these functions use the `AsETH` suffix.

2.3 Vaults

A Nirn vault is a pooled collection of a particular asset which users have deposited their assets into, which utilises the token adapters of said asset in order to migrate capital across lending protocols in a way that maximises their return.

2.3.1 Initialisation

When a vault is initialised, the adapter registry is queried to find the token adapter offering the highest APR at current levels. This adapter is then marked as having maximum weight (`1e18`) - however, as there is no capital in the vault at the time of creation, this is a placeholder value put forward so that the first depositor can immediately call a rebalance and 'start' yield generation.

2.3.2 The Reserve Ratio/Balance

The reserve ratio is the amount of capital deposited to a Nirn vault that is *not* allocated to any lending protocols, in order to ensure a degree of liquid capital for withdrawals: i.e. funds that can be readily withdrawn without having to first pay the gas to withdraw it via a token adapter. By default, the reserve ratio of a newly created vault is 10%.

Note that deposits and withdrawals will adjust the amount of unallocated capital - the reserve *balance* - within the vault in between rebalances:

1. Newly deposited assets are not immediately transferred to lending protocols via their token adapters. Rather, the assets remain in reserve until the next rebalance.

2. Withdrawing an amount greater than the current liquid reserves invokes a chain of withdrawals from token adapters - in the order that they were proposed - until the withdrawal can be honoured. In the event that this happens, the vault reserves are empty until the next deposit or rebalance.

Note that rebalances will always shift capital such that immediately afterwards, the appropriate amount is once again in reserve.

2.3.3 Vault Fees

Nirn vaults have a configurable fee parameter `performanceFee`, set by default to 10% of the *yield* earned by productive assets. Fees are collected whenever a deposit or withdrawal is made from a vault, or when a `claimFees` function is called. Fees are taken as a percentage of the interest generated by a vault, converted to vault tokens.

A value `priceAtLastFee` is updated each time fees are collected: this value is used to calculate the unencumbered interest, defined as the total supply multiplied by the difference between the current price and `priceAtLastFee`. Multiplying this interest by the performance fee yields the pending fees in terms of the underlying asset, which - when claimed - are treated as a deposit by the fee recipient to a vault with a balance reduced by `pendingFees`.

Fees generated by Nirn are considered protocol revenue for the Indexed Finance platform, and as such the default fee recipient for all Nirn vaults is the Indexed DAO Treasury. These fees will ultimately be distributed to stakers of NDX, the Indexed Finance governance token.

2.3.4 Strategy Constraints

It is important to note that Nirn vaults do not currently support complex strategies such as leveraging up or staking liquidity pool tokens for additional yield: the choice to initially restrict vault activity to pure lending and aggregation of governance rewards has been made to minimise the surface area for exploits.

It is *possible* for Nirn to integrate vaults such as those provided by Yearn as token adapters, however we are yet to convince ourselves that tracking conversion rates of wrapped assets over time is a reliable way of extrapolating APR.

3 The Vault Reweighting Process

At any point, an EOA can - either directly, or through the safe caller - rebalance a Nirn vault in one of three ways; a 'clean' rebalance against the current weights and adapters, a new weighting against the current token adapters, or a new weighting against an adjusted set of token adapters. In terms of exposed functions, these options correspond to the following:

```
rebalance()
    external onlyEOA

rebalanceWithNewWeights(
    uint256[] memory proposedWeights
    )
    external onlyEOA

rebalanceWithNewAdapters(
    IErc20Adapter[] memory proposedAdapters,
    uint256[] memory proposedWeights
    )
    external onlyEOA
```

Each of the above options is subsumed by those that come after it: a `rebalance()` can be invoked as a `rebalanceWithNewAdapters(adapters, weights)` provided that the two arguments provided are unchanged from the status quo.

In the interests of reducing duplication, we use this section to present a set-theoretic explanation of the rebalancing process at its most generic, abusing some notation as we go: we refer to (multi)sets to indicate the potential for duplicate values, despite everything being an array in Solidity.

The next page contains a table of notation and function aliases referred to throughout the process, alongside any relevant observations/constraints associated with them:

Symbol	Denotation	Notes
u	Underlying token of vault	Stored in vault
A	Set of active token adapters [currently being used by vault]	Stored in vault
W	Multiset of active token adapter weights [indices correspond across A and W]	Stored in vault
a_u	Underlying token of a token adapter	Queried during validation, must be equal to u
r	Reserve ratio of vault	$0 < r < 1$
v^σ	Total capital within vault	Includes all token adapter deposits and reserve balance
v^p	Total *productive* capital within vault	$v^p = v^\sigma \cdot (1 - r)$
b_r	Reserve balance within vault	$b_r \neq v^\sigma \cdot r$ if deposits or withdrawals have taken place since last rebalance
MIN_W	Minimum active token adapter weight	$0 < MIN_W \leq 1$ Default 0.05
MIN_I	Minimum improvement a rebalance must lead to over current APR to be accepted	$0 < MIN_I$ Default 0.05
Alias	**Function**	**Notes**
$R(u)$	`registry.getAdaptersList(u)`	Returns the registered token adapters for u
$b(a)$	`adapter.balanceUnderlying()`	Returns the value of the vault balance in the lending market for token adapter a
$w(a, n)$	`adapter.withdrawUnderlyingUpTo(n)`	Withdraws up to n underlying tokens from token adapter a and returns m, the amount withdrawn
$d(a, n)$	`adapter.deposit(n)`	Deposits n underlying tokens to token adapter a
$apr(a, n)$	`adapter.getHypotheticalAPR(n)`	Returns the APR of the lending market for token adapter a were a deposit or withdrawal of n underlying tokens made

Table 1: Common Rebalancing Notation & Function Aliases

3.1 Equations

Target Balance

The *target balance* $t(w)$ is the amount of capital that should be held by a vault in a token adapter with weight w, assuming that all of v^p is liquid -

$$t(w) = v^p \cdot w$$

- where v^p is calculated using the *stored* (i.e. current) adapters, rather than a proposed set for which a reweighting is being validated.

Liquidity Delta

The *liquidity delta* $t^\Delta(a, w)$ of a token adapter a is an integer representing the underlying tokens that a vault should deposit or withdraw to or from a in order to reach its target balance:

$$t^\Delta(a, w) = t(w) - b(a)$$

Current Hypothetical APR

The *current hypothetical APR* $apr^c(a, w)$ is the rate that will be earned by a token adapter a if its liquidity delta is resolved for weight w:

$$apr^c(a, w) = apr(a, t^\Delta(a, w))$$

Net Hypothetical APR

The *net hypothetical APR* $napr(A', W')$ of a given set of token adapters A' and corresponding weights W' is defined as the sum of each token adapter's current hypothetical APR multiplied by its weight, minus the reserve ratio (which does not produce interest):

$$napr(A', W') = \left(\sum_{(A'_i, W'_i) \in (A', W')} apr^c(A'_i, W'_i) \cdot W'_i \right) \cdot (1 - r)$$

3.2 Arguments

When proposing a rebalance that alters the token adapters used by a vault – i.e. invoking `rebalanceWithNewAdapters` –, a caller must provide both the set A' of proposed token adapters and the multiset W' of weights, where the value of W'_i corresponds to the desired weight of token adapter A'_i.

A rebalance that simply proposes a shift in the weights of the current token adapters in use – i.e. `rebalanceWithNewWeights` –, requires only the multiset W' to be passed in, under the assumption that ordering of the elements of W' corresponds to that of the token adapters in the current set A.

3.3 Validation

Prior to executing a rebalance, the vault checks the following conditions with the help of the adapter registry, reverting if *any* of them fail:

1. $A' = \texttt{Supp}(A')$ - A' is its own root set, containing no duplicates.
2. $|A'| = |W'|$ - the token adapter and weight (multi)sets have equal length.
3. $\sum |W'| = 1$ - the weights sum to unity (defined in Nirn as `1e18`).
4. $A' \subseteq R(u) \wedge \forall\ a' \in A'\ .\ a'_u = u$ - all proposed token adapters are registered and have underlying token u.
5. $\forall w \in W'\ .\ w \geq MIN_W$ - all proposed weights are at least the configured minimum value.
6. $\frac{napr(A',W') - napr(A,W)}{napr(A,W)} \geq MIN_I$ - the proposed adapters and weights would improve the net APR by at least the configured minimum value.

3.4 Execution

If the validation phase succeeds, the vault will execute the rebalancing as follows:

1. Add all adapters which will be removed to A' with a weight of zero:
   (a) Define $A^\Theta = A \setminus A'$
   (b) Set $A' = A' \ \cup \ A^\Theta$
   (c) Set $W' = W' \ \cup \ \mathbf{0}^{|A^\Theta|}$ (may be a multiset, guaranteed if $|A^\Theta| > 1$)
2. Construct the multiset of target balances for A' and W':
   $T^\Delta = \{\ t^\Delta(A'_i, W'_i) \mid (A'_i, W'_i) \in (A', W')\ \}$
3. For all adapters A'_i where $T^\Delta_i < 0$, withdraw as much of the liquidity delta as is available and record the amounts withdrawn:
   (a) Define $M = \left\{ \begin{cases} w(A'_i, T^\Delta_i) & \text{if } T^\Delta_i < 0 \\ 0 & \text{otherwise} \end{cases} \;\middle|\; A'_i \in A' \right\}$
   (b) Define $m^\sigma = \sum M$
4. For all adapters A'_i where $M_i = -T^\Delta_i \wedge W'_i = 0$, remove the adapter:
   (a) Set $W' = \{W'_j \mid \ j < |W'| \wedge \ j \neq \ i\}$
   (b) Set $A' = \ \{A'_j \mid \ j < |A'| \ \wedge \ j \neq \ i\}$
5. For each token adapter A'_i where $T^\Delta_i > 0$, and while $m^\sigma > 0$:
   (a) Set $n = \texttt{max}(T^\Delta_i, m^\sigma)$
   (b) Execute $d(A'_i, n)$
   (c) Set $m^\sigma = m^\sigma - n$
6. Set $A = A'$ and $W = W'$ within the vault.

3.5 Constraints

Calls to any rebalance functions that do not change the weights of any token adapters - i.e. a plain `rebalance()` - will always execute, as these are used to adjust reserve ratios and/or put newly deposited funds to work via the appropriate token adapters.

We call attention to the fact that rebalances that *do* constitute a change in the vault weightings are subject to two constraints:

1. The MIN_I constraint, which defaults to 0.05: the proposed rebalance must improve the APR of a vault by at least 5% of the current value.
2. At least an hour must have passed since the last composition-changing rebalance.

These restrictions are in place to reduce the attack surface on vaults, ensuring that vaults are not griefed by a barrage of small shifts.

4 Optimisation

As discussed prior, Nirn is open-ended about the *way* in which capital is allocated - the primary constraint imposed upon a reweighting is that it must produce an average rate of return that is higher than the current one. Whilst there are several techniques one can take in attempting to produce an *optimal* allocation, in this section we present one potential approach.

4.1 Formulation

Calculating the optimal allocation of a pool of funds v^p for a vault can be viewed as an variation on the $\{0, 1\}$-knapsack problem (ZOKP), which states:

"Given a maximum capacity W and the weights w and values v of an array of n items, find the maximum value subset $n' \subseteq n$ such that the sum of the corresponding weights is less than or equal to W."

4.2 Translation

Recalling the notation given in Table 1, we assume that $MIN_W = 0.05$, both to shrink the search space and to reduce the amount of gas spent moving small amounts of funds across multiple adapters.

With this assumption, n in the context of the problem statement is the array $[5_i, \ldots, 99_i, 100_i, 5_j, \ldots, 99_j, 100_j, \ldots]$, with each element corresponding to the choice to deposit, e.g. 5% of the productive capital to the token adapter for protocol i, 10% to protocol j and so on.

The weight array w associated with n is therefore $[5, \ldots, 99, 100, 5, \ldots]$, and the maximum capacity W is 100.

The value elements v_i associated with each $n_i \in n$ are defined as -

$$v_i = w_i \cdot apr(a, t^{\Delta}(a, w_i))$$

The outer multiplication by w_i is performed to ensure that the resulting elements of v are components of the net hypothetical APR.

4.3 The Solving Process

In this section, we briefly explain – in prose – the process by which the ZOKP algorithm selects maximum value subsets, corresponding in our case to optimal capital allocations. For those that are more code inclined, the algorithm itself is presented in section 4.6.

Given a maximum capacity W and the length L of the arrays n, w and v, we first create a $(L+1) \times (W+1)$ matrix K, the elements of which - $K[i][j]$ - correspond to the maximum net APR achievable using the first i members of n against a capacity of j. The first row and column correspond, respectively, to the cases where the input arrays are empty and the maximum capacity is zero.

K is initially filled with zero values, and then fleshed out iteratively. For non-zero indices i and j, and with j as the inner loop:

1. Define w' and v' as the weight and value associated with n_i.
2. If w' is less than or equal to j, set $K[i][j]$ as the greater of:
    - $K[i-1][j]$ – the maximum net APR reachable without considering n_i for capacity j, or
    - $v' + K[i-1][j-w']$ – the net APR component associated with n_i, added to the maximum net APR reachable without considering n_i for the difference in capacity.
3. If w' is greater than j, $K[i][j]$ is set to $K[i-1][j]$ (i.e. the item is excluded, as its inclusion would violate maximum capacity).

Following this, $K[L][W]$ represents the maximum net APR that can be obtained across all combinations presented by the array n, with a sum of weights that is as close to W as possible.

We now use K to select members of a solution subset $n' \subseteq n$ – the allocations for the vault – as follows:

1. Set tracking variables `res_v` $= K[L][W]$ and `res_w` $= W$.
2. Looping down from $i = L$ to 0, while `res_v` > 0:
    - If `res_v` is equal to $K[i-1]$[`res_w`] then skip to the next (lower) value of i: this means that the same set could be constructed without n_i.
    - If not, then add n_i to n' and subtract both the corresponding v' from `res_v` and w' from `res_w`.

4.4 Solution Constraints

The algorithm described in the previous section will produce *at least one* solution subset $n' \subseteq n$ for the input array n and corresponding weight and value arrays given, assuming that they are well formed (i.e. at least one of the weights is less than or equal to the maximum capacity). By construction, this assumption holds in our case.

However, given the nature of the problem that we are solving, there are certain restrictions that we must impose as sanity-checks.

Due to its nature as a purely value-optimising algorithm, solutions *can* be produced which have a total sum of weights that is lower than 100. In our formulation, we require a valid solution to fill out the provided 'space': a solution with a sum of weights equal to 99 is invalid for us in the sense that it does not allocate the full productive balance v^p of a vault.

Moreover, by construction, it is *possible* for a solution to contain multiple weighting elements drawn from the same protocol: i.e. $n' = [5_i,\ 45_i,\ 50_j]$. In the context of allocating capital, this is equivalent to $n' = [50_i,\ 50_j]$.

In practice, this latter category of 'ill-formed' solutions does not arise: even though the APR functions associated with lending protocols are monotonically decreasing with respect to deposited capital (all other parameters being equal), scaling their outputs by weight renders the elements of v as *increasing* as proportional weight increases, a claim we present without proof.

4.5 Performance

The optimiser we have presented above is a fairly naïve – albeit effective – way of determining the most profitable capital allocation for a given vault. The size of the search space matrix K is relatively small, containing $(95p + 1) \times 101$ elements for p distinct lending protocols that support the underlying asset, and the subsequent selection of a solution subset collapses quickly.

The heaviest lifting is borne via on-chain queries to the token adapters to retrieve the hypothetical APRs at various liquidity deltas, although these figures are obtained through a `view` function that can be batched.

We will release this optimiser to the Nirn GitHub repository shortly after the launch of vaults, in order to enable anyone to propose optimal allocations for their vault of choice (and will reflect this release in the change-log). We welcome any additional optimisers that make use of different techniques: elegant/efficient ones will – with permission – be merged into the Nirn GitHub repository.

4.6 ZOKP Algorithm

An algorithm for solving the ZOKP in JavaScript that has linear complexity in both space and time follows:

```
function Knapsack(W, items, wts, vals, L)
{
    let i, w;
    let selected = new Array(0);

    let K = new Array(L + 1);
    for (i = 0; i < K.length; i++)
    {
        K[i] = new Array(W + 1).fill(0);
    }

    for (i = 0; i <= L; i++) {
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (wts[i - 1] <= w)
                K[i][w] =
                  max( vals[i - 1] + K[i - 1][w - wts[i - 1]]
                      , K[i - 1][w]
                      );
            else
                K[i][w] = K[i - 1][w];
        }
    }

    let res = K[L][W];
    w = W;
    for (i = L; i > 0 && res > 0; i--)
    {
        if (res == K[i - 1][w])
            continue;
        else {
            selected.push(items[i - 1]);
            res -= vals[i - 1];
            w   -= wts[i - 1];
        }
    }
    return selected;
}
```

5 Gas Costs

To illustrate the costs incurred via typical interactions with a vault, below is a table of the gas costs for depositing and querying the balances of selected assets into the adapters with the highest APR at the pinned block we test against.

Asset	**Adapter**	`deposit(n)`	`tokenBal()`	`underlyingBal()`
DAI	Cream	256,184	31,098	63,350
TUSD	Aave V1	828,688	40,972	40,995
USDC	Cream	279,897	31,098	63,350
USDT	Fulcrum	239,389	26,013	60,127
sUSD	Iron Bank	406,585	31,120	75,221
BAT	Compound	336,928	25,877	57,880
LINK	Iron Bank	260,265	31,120	71,303
KNC	Aave V2	395,558	36,689	36,713
MKR	Aave V1	810,761	40,972	40,995
ZRX	Aave V1	808,620	40,972	40,995
SNX	Iron Bank	370,731	31,120	71,303
WBTC	Aave V2	462,488	36,689	36,713
BUSD	Cream	281,554	31,098	63,350
YFI	Iron Bank	264,291	31,120	71,303
AAVE	Cream	524,624	31,098	63,350
UNI	Fulcrum	198,568	26,013	59,142
COMP	Cream	265,468	31,098	63,350

Balance queries have no cost if queried off-chain: however some of the token adapters require non-trivial amounts of data to achieve the precision we require in the underlying balance of a vault, being off by a maximum of one wei.

Note that these are the gas costs involved in depositing assets to underlying protocols - these are incurred during rebalances. The act of simply depositing assets to a Nirn vault is significantly cheaper.

6 Access Controls & Security

We conclude by briefly surveying the ways in which Nirn can be modified, who has the ability to make such modifications, and potential attack vectors outside the scope of Nirn itself which users should be aware of.

6.1 Access Controls

For any given Nirn vault, the following can be changed:

1. The liquid reserve ratio r,
2. The minimum weight MIN_W of an active token adapter,
3. The minimum APR improvement MIN_I of a proposed rebalancing,
4. The percentage of the fee on generated yield (maximum 20%),
5. The address to which fees are transferred, and
6. Whether or not a vault is accepting deposits.

Note: the first vaults that we deploy will be implemented as upgradable proxies, in order to head off any issues that are detected early on. This means that the DAO can change the entire contract implementation, subject to a successful vote. Once we are satisfied that the existing implementation is secure, we will lock the implementation of the vaults that exist at that point, and utilise non-upgradable proxies going forward.

The ability to burn wrapper tokens and withdraw from a Nirn vault cannot be frozen: depositors are free to retrieve their assets at any time.

Within the wider Nirn ecosystem, the only change that can be made is the white-or-blacklisting of protocol adapters within the adapter registry, enabling the introduction of new lending markets or the removal of existing ones.

As stated in Section 2.3.3, fees generated by Nirn are ultimately routed to members of the Indexed Finance DAO. As goes the benefit, so does the responsibility, and as such the DAO is the only body capable of enacting any of the above changes, subject to a successful Governor Alpha vote via NDX tokens.

To emphasise: there is no EOA that can make unilateral changes to Nirn, although we encourage the security conscious to verify this for themselves.

6.2 Security

There are two primary categories of attack vector on a Nirn vault: critical vulnerabilities within underlying protocols, and the exploitation of incorrectly implemented protocol logic within token adapters.

Example: Underlying Protocol Vulnerability

As an example of the former, consider the following attack that could be executed in the event of a compromised oracle:

- An oracle 'breaks' and reports that one `UNI` is worth a million dollars.
- An attacker deposits `UNI` into the lending market of a protocol that relies on this compromised oracle, and maxes out the borrow limit in `USDC` against this inflated value collateral.
- With the lending rate for `USDC` now artificially inflated, the attacker triggers a rebalance of the `USDC` Nirn vault to maximise the amount of capital allocated to this – now compromised – protocol.
- With borrowing rates reduced on the `USDC` market of the protocol due to the influx of capital from the `USDC` Nirn vault, the attacker can borrow *additional* `USDC` – which is now bad debt – from the protocol.
- Once the oracle failure is resolved, the deposited `UNI` is liquidated, the deposited `USDC` from the Nirn vault is drained and the attacker is free.

More generally, any vulnerability in the underlying protocols that leads to a loss of funds is – by proxy – a vulnerability in Nirn itself, provided that a Nirn vault has a non-zero allocation to an affected protocol during an attack. We have attempted to minimise the chances of such an attack affecting Nirn by initially whitelisting only 'blue-chip' lending protocols but, as with all things DeFi, the risk cannot be completely eliminated.

Example: Incorrect Token Adapter Logic Vulnerability

The second category of attack that Nirn can be exposed to comes about due to the potential that token adapters are implemented incorrectly. As an example, if it is possible to temporarily manipulate the exchange rate between the underlying token of a vault and its corresponding wrapper token *without being exposed to that rate* for the amount that was used to manifest this imbalance, this represents a critical vulnerability within the corresponding Nirn vault.

Consider the following series of actions:

- Deposit x `DAI` into the `DAI` Nirn vault, receiving cx `nDAI` wrapper tokens in exchange. The `DAI` is held in reserve until someone calls a rebalance.
- Transfer y `DAI` to `cDAI` (the Compound wrapper token) without depositing it into the Nirn vault.
- Burn your `nDAI` to receive `DAI` in exchange: the token adapter calls the `cDAI` contract to determine the present value of `nDAI`.

This does not describe an attack in and of itself: this is an action that anyone can undertake, albeit not at a profit. The potential for abuse comes in if it becomes possible to misrepresent the `DAI`/`cDAI` exchange rate - either via flash loan abuse, or by exploiting the underlying token adapter such that Nirn overvalues your cx `nDAI` as being worth more than x `DAI` when factoring out any yield that may have been generated. More specifically, if the second step above can be done without calling a transfer, a vulnerability exists.

The usual *caveat emptor* applies as a result: the token adapters have been written with best security practices in mind, however we cannot – and will not – make blanket statements about infallibility. As such, those readers with the appropriate domain knowledge are encouraged to verify the token adapter logic themselves, and reach out to the authors with any concerns.

7 Changelog

The latest LaTeX file for this whitepaper can be found at:

`https://github.com/indexed-finance/nirn-whitepaper`

- V1.0 - 2 August 2021 - initial release.