This research has received funding from the Uniswap Foundation Grants Program. Any opinions in this post are my own.

The long-anticipated release of Uniswap V4 is upon us. This blog post sketches a straightforward combination of a singleton pool and hooks within the new V4 framework to tackles cross-domain MEV at the source: the block producer, or searchers paying for that privilege.

Recently, I've been doing research on cross-domain MEV sources within the DEX ecosystem, and it almost always came back to the same term: Loss-versus-rebalancing

or LVR

. LVR put a name to the face of one of the primary costs incurred by DEX liquidity providers. Block builders on any chain are being goose-stepped into profit-maximizing machines. To do this requires a knowledge of the most recent state of the primary exchanges and market-places. Typically, these take the form of centralized exchanges and other large-volume DEXs, not to mention the swaps in the mempool (discussion for another post).

The first, guaranteed

cost that a DEX must pay each block is that of arbitraging the DEX's stale reserves to line up with the block builders best guess of where the underlying price is. From here, the builder then sequences the DEX swaps in a way so as to maximize the builders back-running profits. This proceeding sequence of back-runs pays fees to the pool, but are not guaranteed to take place.

It is only the arbitrage of the pool reserves that is guaranteed. Luckily, this can be addressed with hooks

.

The exact implementation of hooks hasn't been nailed down yet, but in this article we assume, as in the whitepaper, that a hook implements custom logic before or after 4 key phases in the pool contract:

1. Initialize

: When the pool is deployed

1. Modify Position

: Add or remove liquidity.

1. Swap

: Execute a swap from one token to another in the V4 ecosystem.

1. Donate

: Donate liquidity to a V4 pool.

The Solution: Low-Impact re-addition of retained LVR into the liquidity pool

Our proposed solution is based on techniques formalized as the Diamond protocol, with similarities to another ethresearch post. We are only interested in hooks before and after swaps. For a particular pool, we need to make a distinction between the first swap interacting with the pool in a block and all other swaps.

For our solutions we introduce an LVR rebate function $\beta: \{1,...,Z\} \rightarrow [0,1]$

. It suffices to consider $\beta()$

as a strictly decreasing function with $\beta(Z)=0$

, for some $Z \in \mathbb{N}$

. Whenever we call a hook, let B_{current}

be the current block number when the hook is called, and B_{previous}

be the number of the block in which the most recent swap occurred. We also need to introduce the idea of a vault contract $\texttt{vault}$

, and a hedging contract $\texttt{hedger}$

.

Depositing x_A

token As to \texttt{hedger} increases a contract variable \texttt{hedgeAvailableA}

by x_A

(likewise \texttt{hedgeAvailableB}

for token B deposits). At all times, the block builder can submit a transaction removing some amount of tokens x_A

from \texttt{hedger} if at the end of the transaction \texttt{hedgeAvailableA}>x_A

. If the builder withdraws x_A

tokens, reduce \texttt{hedgeAvailableA}

by x_A

.

Solution Description

In this solution, consider the following logic(described algorithmically as \texttt{beforeSwap()}

and \texttt{afterSwap()}

hooks in Algorithms 1, 2, and 3):

- \textbf{If} \ B_\text{current}-B_\text{previous}>0

, the swap is the first swap in this pool this block. Send any remaining tokens in \texttt{hedger}

to pool. After this, add some percentage of the tokens in \texttt{vault}

back into the pool. The correct percentage is the subject of further research, although we justify approximately 1% later in this post. Set the \texttt{hedgeAvailable}

variables to 0.

Execute 1-\beta(B_\text{current}-B_\text{previous})

of \texttt{swap}_1

, and remove the required amount of token A

from the pool so the implied price of the pool is equal to the implied price given \texttt{swap}_1

was executed. This is necessary because if only 1-\beta(B_\text{current}-B_\text{previous})

of \texttt{swap}_1

is executed the price of the pool will not be adjust to reflect the information of \texttt{swap}_1

. Add the removed tokens to \texttt{vault}

.

- \textbf{Else}

, it must be that B_\text{current}-B_\text{previous}==0

, which implies the swap is a \texttt{swap}_2

. Let \texttt{swap}_2

be buying some quantity x_A

of token A

. One of the following three conditions must hold:

- \textbf{If }\texttt{hedgeAvailableA}\geq x_A

AND x_A>0

, then execute \texttt{swap}_2

and decrease \texttt{hedgeAvailableA}

by x_A

, but do not remove any tokens from \texttt{hedger}

. Increase \texttt{hedgeAvailableB}

by x_B

.

- \textbf{Else if } \texttt{hedgeAvailableB}\geq x_B

AND x_B>0

, then execute \texttt{swap}_2

and decrease \texttt{hedgeAvailableB} by x_B

, but do not remove any tokens from \texttt{hedger}. Increase \texttt{hedgeAvailableA} by x_A

.

- \textbf{Else}

there is not enough tokens deposited to perform the swap, then revert.

- \textbf{If }\texttt{hedgeAvailableA}\geq x_A

AND x_A>0

, then execute \texttt{swap}_2

and decrease \texttt{hedgeAvailableA}

by x_A

, but do not remove any tokens from \texttt{hedger}

. Increase \texttt{hedgeAvailableB}

by x_B

.

- \textbf{Else if } \texttt{hedgeAvailableB}\geq x_B

AND x_B>0

, then execute \texttt{swap}_2

and decrease \texttt{hedgeAvailableB} by x_B

, but do not remove any tokens from \texttt{hedger}. Increase \texttt{hedgeAvailableA} by x_A

.

- \textbf{Else}

there is not enough tokens deposited to perform the swap, then revert.

What does this solution solve?

This solution allows the producer to move the price of the block to any level with \text{swap}_1

, although only executing 1-\beta(B_\text{current}-B_\text{previous})

of \text{swap}_1

. This \text{swap}_1

can be thought of as the LVR swap, and is such it is discounted. From there, the producer is forced to match buy orders with sell orders. Orders are only executed against the pool if they can also be executed against the tokens in the hedge contract

\texttt{hedger}. If the price does not return to the price set after \text{swap}_1

(the tokens in \texttt{hedger} don't match the \texttt{hedgeAvailable} variables,) there are sufficient tokens in \texttt{hedger} to rebalance the pool, and these tokens in the hedging contract are used to do so in the next block the pool is accessed.

An ideal solution would allow the producer to execute arbitrary transactions, and then repay \beta

of the implied swap between the start and end of the block, as this is seen as the true LVR (the end of block price is the no-arbitrage price vs. external markets, otherwise the producer has ignored a profitable arbitrage opportunity). Our solution does this in a roundabout way, although using hooks. The producer moves the price of the block to the no-arbitrage price in the first swap, and is then forced to return the price here at the end of the block, all through hooks.

Does the solution work?

\texttt{yes}

How? This solution differs from the theoretical proposal of that of Diamond in 2 important functional ways. Firstly, Diamond depends on the existence of a censorship-resistant auction to convert some % of the vault tokens so the vault tokens can be re-added into the liquidity pool. The solution provided above directly readds the vault tokens into the pool. Through simulation, we have identified the solution provided in this post approximates the returns of Diamond and its perfect auction when the amount being re-added to the pool is less than 5% per block.

Don´t just take our word for it!

The simplest solution in Diamond periodically re-adds the retained LVR proceeds from the vault into the pool at the pool price. We include the graph of the payoff of this protection applied to Uniswap V2 pool vs. a Uniswap V2 pool without this protection. The payoff of this protocol is presented below. [

3200×2400 269 KB

](https://ethresear.ch/uploads/default/original/2X/8/825779a183459ee004ee83281fcfa6e67ea402be.png)

.

(The core code used to generate these simulations is[available here](.).)

We ran some simulations to compare our solution to the theoretical optimal of Diamond. We chose a $300M TVL ETH/USDC pool at a starting price of $1844, and a daily volatility of 5%. These simulations were run The expected returns of the Diamond-protected pool relative to the unprotected pool over 180 days (simulated 1,000 times) is 1.057961. This is almost exactly equal to the derived cost from [the original LVR paper](.) of 3.125bps per day, computed as \frac{1}{(1-0.0003125)^{180}} \approx 1.05787.

That's a saving of $17M over half a year!

Unfortunately, 100% LVR retention is unrealistic. Let's instead assume an average LVR retention of 75% (LVR rebate value \beta

of 0.75). Through simulation, the Diamond-protected UniV2 pool gives a relative return vs the unprotected UniV2 pool of 1.0431.

Compare this to the relative returns of the protocol described in this post applied to a UniV2 pool vs the unprotected UniV2 pool. These simulations are plotted below for several re-add percentages (the amount of the vault tokens (retained LVR) to be re-added to the pool each block). The average relative returns are 1.0456, 1.0436, and 1.0335, for re-add percentages of 1%, 5%, and 12.5% respectively. [

3200×2400 256 KB

](https://ethresear.ch/uploads/default/original/2X/d/d10fd96626fd581b32e3ae9c76cca0c4129c1d6c.png)

From this, we can see that the 1% re-add strategy each block actually outperforms

the optimal conversion strategy (1.0456 for % re-adding vs. 1.0431 in Diamond). This is because for simulations with large moves, converting the pool is less profitable. Without fees, HODLing is optimal, and re-adding less tokens approaches some form of HODLing. If we introduce fees related to pool size, we can counteract this out-performance.

In the following graph, we include a representation of the performance of the theoretical conversion protocol of Diamond (pink), the low-impact re-adding protocol of this post with re-add % of 1% (blue), and HODLing (orangey). All of these returns are vs. the corresponding unprotected UniV2 pool.

[

3200×2400 236 KB

](https://ethresear.ch/uploads/default/original/2X/3/3ca157d0afb08050e451b6eee35c2eded92ebbd6.png)

Considerations and Limitations

Re-adding tokens from the pool to the vault creates an expected arbitrage opportunity in the next block, so re-adding less tokens intuitively reduces the losses (increases the profitability of the pool).

To access the pool, someone must submit the initial pool swap, and then deposit tokens to the \texttt{hedger}

contract. However, as we typically expect searchers to be profit maximizing, we should then expect these same searchers to back run any set of buy or sell orders to the no-arbitrage price.

It is possible that the tokens in \texttt{hedger}

can be used for this back-running. The balance of \texttt{hedger}

can also be updated mid-block before all swaps are executed. This may be necessary if tokens are required mid-block, with benefits outweighing the gas cost to exit and re-enter tokens to \texttt{hedger}

.

There are lots of other quirky potential possibilities with hooks on top of this core LVR-retention framework. This post is intended to demonstrate one of the many possibilities that hooks give us.

Algorithm Pseudocode

[

image

970×747 167 KB

](https://ethresear.ch/uploads/default/original/2X/c/cb7bf91d55b56e359a6cfda5ba8b54220bb1a590.png)

[

image

1190×584 88.2 KB

](https://ethresear.ch/uploads/default/original/2X/5/59c7b37dc121efb7d2db48d37892fe71e0fceaed.png)

[

image

957×421 59 KB

](https://ethresear.ch/uploads/default/original/2X/0/0ea3028e27c59698499308513b2e3ee15f8a2a02.png)