1

Procedural approaches towards Maximal Extracted Value

Alexander Sandy Bradley[1,2], Sam Bacha[1,2], and et al[1]

[1] Manifold Finance, Inc
[2] CommodityStream, Ltd.
{sandy,sam}@manifoldfinance.com
janitor@manifoldfinance.com

Keywords: MEV, · Miner Extractable Value · trade optimization, · EVM · uniswap · sushiswap · ethereum · formal verification

⋆ Thanks to Johannes, Kitsuana, et al

Table of Contents

2

Protocol Specification

Abstract.

MEV is sucks.
MEV suck, **Sandwiches sucks**,

use OpenMEVs.[3]

1 Motivation

see ***Abstract***.

1.1 Design

The Original design for the router was to use flashloans only to arbitrage then immediately distribute profits to hard coded addresses. Apart from the issue of hard coded addresses, this setup was inefficient because of small amounts frequently being split and transferring to multiple addresses being expensive. This opened the possibility of leaving profits to accumulate on the router. Furthermore it provides a way to arbitrage without a flashloan, saving gas and the loan fee (i.e. more profit, less gas). Additionally, the harvesting profits means ownership control of the router. A more robust 2 step process was chosen to control and transfer router ownership and harvest control. Ideal setup would be multisig consensus ownership.[4]

[3] Bacha, Sam "MEV is essentially looking for the reachable state where their balance is maximized. Given any arbitrary re-ordering, insertion or censorship of pending or existing transactions, this can suck, suck real hard."
[4] Ownership control of the router is not an issue, as a new router can be used or fallback to the legacy router contract.

2 Requirements

2.1 Security properties

In following the standards set forth by the UniswapV02/SushiswapV01 router contracts, the SushiswapV02 router contracts are intended to be safe to use with:

Potentially re-entrant tokens Tokens that do not return from transfer Pathological Tokens The SushiswapV02 router contracts are not intended to be used tokens that exhibit the following behavior Tokens that exhibit a discretizing inflation curve Tokens that exhibit an 'unowned' supply Tokens that implement user defined types for standard mathematical operations Numerical error analysis The engineering team would like to request a review of the numerical error incurred during contract execution, with a focus on the desirable invariant proposed by both the development team and auditors. Examples include any significant rounding error, if any, in a swap, favors the pool. etc.

2.2 Conformant Algorithms

Conformant Algorithms Our requirements are phrased in the imperative, as such, part of algorithms (such as "strip any leading space characters" or "return false and abort these steps") are to be interpreted with the meaning of the RFC/EIP defined key words ("must", "should", "may", etc) used in introducing algorithms

Conformance requirements phrased as algorithms or specific steps that can be implemented in any manner, so long as the end result are equivalent. In particular, the algorithms defined in this specification are intended to be easy to understand and are not necessarily intended to be performant. Implementers are encouraged to either: switch to an L2, optimize, or use Vyper[5].

3 Swap Execution

Presented herein, a derivation of optimal arbitrage between 2 Constant Product Automated Market Maker (CPAAM), Decentralized Exchanges (DEXs) for usage in procedural processes that enable value extraction (i.e. MEV). This math is applied in the new Sushiswap router for at the source of Miner Extractable Value (MEV). These trades post user swaps atomically. Profits are distributed to liquidity providers, in turn giving users better rates. Extracting MEV at source protects user trades from front-run attacks inherently and helps prevent fee spikes from attackers.

Slippage: The amount of price impact that a liquidator engenders when trying to sell collateral. Slippage is denoted $\Delta p(q)$ and is formally defined as the

[5] Vyperlang: https://vyper.readthedocs.io/en/stable/

difference between the midpoint price at time $t, p_{\text{mid}}(t)$ and the execution price, $p_{\text{exec}}(q,t)$ for a traded quantity q at time

$$t, \Delta p(q,t) = p_{\text{mid}}(t) - p_{\text{exec}}(q,t)$$

. This quantity is usually a function of other variables, such as implied and realized volatility. [6]

3.1 Router Implementation

As a general approach to security, deviation from current UniswapV2Router was kept to a minimum. Pair contract calls should be consistent with the existing router. Reliance on the new router storing and transferring tokens brings in a new attack vector. A robust Ownership setup, as above, was chosen to mitigate this threat along with reduction of functions accessing the funds. 2 helper libraries were also chosen from solmate to supersede UniswapV2Helper libraries for security.

safeTransferLib ERC20 Since UniswapV2Router was not designed to store tokens, some functions are not compatible and had to be changed. E.g.

`removeLiquidityETHSupportingFeeOnTransferTokens`

```
(, amountETH) = removeLiquidity(
    token,
    WETH,
    liquidity,
    amountTokenMin,
    amountETHMin,
    address(this),
    deadline
);
TransferHelper.safeTransfer(token, to, IERC20(token).balanceOf(address(this)));
```

Changes to

```
uint256 balanceBefore = ERC20(token).balanceOf(address(this));
(, amountETH) = removeLiquidity(token, weth, liquidity, amountTokenMin, amountETHMin,
address(this),
deadline);
ERC20(token).safeTransfer(to, ERC20(token).balanceOf(address(this)) - balanceBefore);
```

[6] Slippage is also known as market impact within academic literature.

3

Mathematical Model

The following sections describe the derivation of the optimal sizes for post user swap arbitrage between UniswapV2 style exchanges.
[7]

4 Constant Product Automated Market Maker

Constant Product Automated Market Makers (CPAMMs) are smart contracts for token liquidity pairs. UniswapV2 and SushiswapV1 are all governed by the constant product formula given in equation 1.

$$k = R_\alpha R_\beta \tag{1}$$

Where R_α corresponds to the Reserves of token α, R_β to the Reserves of token β within the pair contract and k the constant invariant.

A swap trading $\Delta\beta$ tokens for $\Delta\alpha$ must satisfy equation 2.

$$k = (R_\alpha - \Delta\alpha)(R_\beta + \gamma\Delta\beta) \tag{2}$$

$$\gamma = 1 - fee \tag{3}$$

Where the fee on UniswapV02 and SushiswapV01 is 0.3% and 0.25% respectively. For big integer math, equation 3 can be written in the form of equation 4.

$$\gamma = \frac{997}{1000} \tag{4}$$

From equations 1 and 2 we can derive an equations for the expected amounts out and in, given in equations 5 and 6.

$$amountOut : \Delta\alpha = \frac{997 R_\alpha \Delta\beta}{1000 R_\beta + 997\Delta\beta} \tag{5}$$

$$amountIn : \Delta\beta = \frac{1000 R_\beta \Delta\alpha}{997(R_\alpha - \Delta\alpha)} \tag{6}$$

[7] Benchmarking contracts for establishing a baseline can be found here: https://github.com/manifoldfinance/v2-periphery/tree/master/contracts

Post swap, the new liquidity reserves are modified as shown in equations 7 and 8.

$$R_\alpha new = R_\alpha old - \Delta\alpha \tag{7}$$
$$R_\beta new = R_\beta old + \Delta\beta \tag{8}$$

Therefore sequential swaps can be simulated off-chain in a deterministic way, given the current liquidity state.

5 Minimal Procedural DEX Arbitrage

Establishing a minimal swap for DEX arbitrage consists of a single swap on one DEX followed by the reverse swap on another.

Token amount swap path:

$$DEX0: \ \Delta\beta_0 \Rightarrow \Delta\alpha_0 \tag{9}$$
$$DEX1: \ \Delta\alpha_0 \Rightarrow \Delta\beta_1 \tag{10}$$

6 Optimal simple DEX arbitrage size

From equation 5, the definition of a simple DEX arbitrage for CPAMMs can be written in the form of equations 11 and 12.

$$\Delta\alpha_0 = \frac{997 R_{\alpha 0}\Delta\beta_0}{1000 R_{\beta 0} + 997\Delta\beta_0} \tag{11}$$
$$\Delta\beta_1 = \frac{997 R_{\beta 1}\Delta\alpha_0}{1000 R_{\alpha 1} + 997\Delta\alpha_0} \tag{12}$$

Profit of the arbitrage is simply the amount out of the second trade minus the amount in of the first, shown by equation 13.

$$profit: y = \Delta\beta_1 - \Delta\beta_0 \tag{13}$$

Substituting equation 11 into equation 12, we get equation 14.

$$\Delta\beta_1 = \frac{997 R_{\beta 1}\frac{997 R_{\alpha 0}\Delta\beta_0}{1000 R_{\beta 0}+997\Delta\beta_0}}{1000 R_{\alpha 1} + 997\frac{997 R_{\alpha 0}\Delta\beta_0}{1000 R_{\beta 0}+997\Delta\beta_0}} \tag{14}$$
$$= \frac{997^2 R_{\beta 1} R_{\alpha 0}\Delta\beta_0}{(1000 R_{\beta 0} + 997\Delta\beta_0)1000 R_{\alpha 1} + 997^2 R_{\alpha 0}\Delta\beta_0} \tag{15}$$

Since we are looking for the optimal amount In ($\Delta\beta_0$), we can make the following simplifications.

$$\begin{aligned}
&let\ x = \varDelta\beta_0 && (16)\\
&let\ C_A = 997^2 R_{\beta 1} R_{\alpha 0} && (17)\\
&let\ C_B = 1000^2 R_{\beta 0} R_{\alpha 1} && (18)\\
&let\ C_C = 997000 R_{\alpha 1} && (19)\\
&let\ C_D = 997^2 R_{\alpha 0} && (20)
\end{aligned}$$

Thus equation 15 can be reduced to equation 21.

$$\varDelta\beta_1 = \frac{C_A x}{C_B + x(C_C + C_D)} \tag{21}$$

Therefore the profit (y), from equation 13 can be expressed in terms of the amount In (x), shown in equation 22.

$$\begin{aligned}
y &= \frac{C_A x}{C_B + x(C_C + C_D)} - x && (22)\\
&= \frac{C_A x - x(C_B + x(C_C + C_D))}{C_B + x(C_C + C_D)} && (23)\\
&= \frac{x(C_A - C_B) - x^2(C_C + C_D)}{C_B + x(C_C + C_D)} && (24)\\
&= \frac{x C_F - x^2 C_G}{C_B + x C_G} && (25)
\end{aligned}$$

Where:

$$\begin{aligned}
C_F &= C_A - C_B && (26)\\
C_G &= C_C + C_D && (27)
\end{aligned}$$

Maximum profit occurs at a turning point i.e. where the gradient or differential is zero, shown in equation 28.

$$\frac{dy}{dx} = 0 \tag{28}$$

Since we have a complex equation for differentiating, we can use the quotient rule from equation **??**. Numerator and denominator differentials are shown in equations 32 and 33.

$$\frac{dy}{dx} = \frac{d\frac{f(x)}{g(x)}}{dx} \tag{29}$$
$$f(x) = xC_F - x^2C_G \tag{30}$$
$$g(x) = C_B + xC_G \tag{31}$$
$$\frac{f(x)}{dx} = C_F - 2xC_G \tag{32}$$
$$\frac{g(x)}{dx} = C_G \tag{33}$$

Combining the quotient rule with equation 28, we get equation 34, which expands to equation 35.

$$f'g = g'f \tag{34}$$
$$(C_F - 2xC_G)(C_B + xC_G) = C_G(xC_F - x^2C_G) \tag{35}$$

Equation 35 can be re-arranged to form a generic quadratic equation 36 and so the parameters can be defined for the quadratic solution in equation 37.

$$x^2C_G^2 + x(2C_BC_G) - C_BC_F = 0 \tag{36}$$

Solution to the optimal simple DEX arbitrage size for a given swap is shown in equation 37.

$$x^* = \frac{-(2C_BC_G) \pm \sqrt{(2C_BC_G)^2 - 4(C_G^2)(-C_BC_F)}}{2C_G^2} \tag{37}$$

For positive roots only, this can be reduced to:

$$x^* = \frac{-C_B + \sqrt{C_B^2 + C_BC_F}}{C_G} \tag{38}$$

4

Equivalence Checking

6.1 Backrun placement

By definition, backruns must occur after user to user swap. From a design point of view the simplest place to insert the backrun function would be in the internal $_{s}wap$ function which is called by the other swaps. However, some of the swap variants eg $swapTokensForExactETH$ perform user actions after $_{s}wap$ is called. This is not ideal, as we do not want to interfere with the user swap. Moreover, other swap variants such as $swapExactTokensForTokensSupportingFeeOnTransferTokens$ do not use $_{s}wap$. Backrun functions were therefore placed at the end of each external swap variant. E.g.

Fig. 1. Backrun Placement

```
   function swapExactTokensForTokens(
      uint amountIn,
      uint amountOutMin,
      address[] calldata path,
      address to,
      uint deadline
   ) external virtual override ensure(deadline) returns
   (uint[] memory amounts) {
amounts = UniswapV2Library.getAmountsOut(factory, amountIn, path);
require(amounts[amounts.length - 1]
>= amountOutMin, 'UniswapV2Router: INSUFFICIENT_OUTPUT_AMOUNT');
   TransferHelper.safeTransferFrom(
           path[0], msg.sender,
           UniswapV2Library.pairFor(factory, path[0], path[1]), amounts[0]
       );
       _swap(amounts, path, to);
   }
```

Fig. 2. Backrun Implementation

```
function swapExactTokensForTokens(
    uint amountIn,
    uint amountOutMin,
    address[] calldata path,
    address to,
    uint deadline
) external virtual override ensure(deadline) returns (uint[] memory amounts) {
    amounts = UniswapV2Library.getAmountsOut(factory, amountIn, path);
    require(amounts[amounts.length - 1] >= amountOutMin,
    'UniswapV2Router: INSUFFICIENT_OUTPUT_AMOUNT');
    TransferHelper.safeTransferFrom(
        path[0],
        msg.sender,
        UniswapV2Library.pairFor(factory, path[0], path[1]), amounts[0]
    );
    _swap(amounts, path, to);
    _backrunSwaps(path);
}
```

6.2 Multiple factories

Multiple factories (at least 2) are required for the backrun arbitrage. The adoption of multiple factories within the router, lead to some internal function changes. In particular *pairFor*.

```
// calculates the CREATE2 address for a pair without making any external calls
function pairFor(
address factory,
address tokenA,
address tokenB
)
internal pure returns (address pair) {
(address token0, address token1) = sortTokens(tokenA, tokenB);
pair = address(uint160(uint(keccak256(abi.encodePacked(
    hex'ff',
    factory,
    keccak256(abi.encodePacked(token0, token1)),
    // hard coded factory init code hash
    hex'96e8ac4277198ff8b6f785478aa9a39f403cb768dd02cbee326c3e7da348845f'
    )))));
}
```

Changes to

Fig. 3. New Factories Implementation

```
function pairFor(address factory, address tokenA, address tokenB)
 internal view returns (address pair) {
    bytes memory factoryHash = factory
        == SUSHI_FACTORY ? SUSHI_FACTORY_HASH : BACKUP_FACTORY_HASH;

(address token0, address token1) = _sortTokens(tokenA, tokenB);
   pair = address(uint160(uint(keccak256(abi.encodePacked(
     hex'ff',
     factory,
     keccak256(abi.encodePacked(token0, token1)),
     factoryHash // init code hash
    )))));
}
```

6.3 Fallback factory

Since the extra factory is required for the arbitrage, we can use it, for the user, to check for an available swap on the alternate factory if it would otherwise fail on the default factory through slippage.

```
amounts = UniswapV2Library.getAmountsOut(factory, amountIn, path);
require(amounts[amounts.length - 1]
>= amountOutMin, 'UniswapV2Router: INSUFFICIENT_OUTPUT_AMOUNT');
```

Changes to

Fig. 4. Fallback Factory Implementation

```
address factory = SUSHI_FACTORY;
amounts = _getAmountsOut(factory, amountIn, path);
if(amounts[amounts.length - 1] < amountOutMin){
    // Change 1 -> fallback for insufficient output amount, check backup router
    amounts = _getAmountsOut(BACKUP_FACTORY, amountIn, path);
    require(amounts[amounts.length - 1]
    >= amountOutMin, 'UniswapV2Router: INSUFFICIENT_OUTPUT_AMOUNT');
    factory = BACKUP_FACTORY;
}
```

5

Math Libraries and BigInt support

These following sections describe testing various math libraries to accommodate the additional calculations required of the router.

6.4 Uint256 overflow

Optimal arbitrage calculations were overflowing uint256. i.e.

```
uint Cd = reserve0Token1.mul(997).mul(997);
uint Cc = reserve1Token1.mul(997000);
uint Cb = reserve1Token1.mul(reserve0Token0).mul(1000).mul(1000);
uint Ca = reserve1Token0.mul(reserve0Token1).mul(997).mul(997);
uint Cf = Ca - Cb;
uint Cg = Cc + Cd;
uint a = Cg * Cg;
uint b = 2 * Cb * Cg;
uint c = Cb * Cf;
uint d = (b*b) + ( 4 * a * c );
```

would consistently overflow by *uintb*. Found out through individual checks:

```
unchecked {
    uint a = Cg * Cg;
    require(a/Cg == Cg,"a overflow");
    uint b = 2 * Cb * Cg;
    require(b/Cb == 2*Cg ,"b overflow");
    uint c = Cb * Cf;
    require(c/Cb == Cf,"c overflow");
    uint d = (b*b) + ( 4 * a * c );
    require(d/(b*b) == 4*a*c,"d overflow");
}
```

ABDKMath ABDKMath library was used for a time, as it avoided overflow by dropping to floats.[8]

```
bytes16 _Cg = ABDKMathQuad.fromUInt(Cg);
bytes16 _a = ABDKMathQuad.mul(_Cg, _Cg);
```

However we found this lost precision and failed echidna tests.

Fig. 5. Uint256 overflow:5

```
echidna_mulUint:
failed!
  Call sequence:
    setX1(1106235220955)
    setX(9390953368914254812617)

echidna_Uint_conversion:
failed!
  Call sequence:
    setX(10518526264810785485368401065708505)

echidna_divUint:
failed!
  Call sequence:
    setX(10417774989007224423389698535018119)
    setX1(1)
```

[8] *https : //github.com/abdk − consulting/abdk − libraries − solidity*

PRBMath We also tried PRBMath[9] library. These performed better in echidna tests but still suffered overflow issues.

Fig. 6. PRBMath

```
echidna_mulUint:
~ failed!
  Call sequence:
    setX1(1106235220955)
    setX(9390953368914254812617)


echidna_Uint_convertion:
~ failed!
  Call sequence:
    setX(10518526264810785485368401065708505)


echidna_divUint:
~ failed!
  Call sequence:
    setX(10417774989007224423389698535018119)
    setX1(1)
```

Uint512 Ultimately we settled on Uint512 which both passed echidna and overflow issue.[10]

Fig. 7. Uint512

```
echidna_mulUint: ~ passed!
echidna_divUint: ~ passed!
```

7 Conclusions and Future Work

9 https://github.com/paulrberg/prb-math/

10 see $github.com/SimonSuckut/Solidity_Uint512/blob/main/contracts/Uint512.sol$

6

Bibliography

References

1. Sam Bacha, Sandy Bradley. OpenMEV source code, github.com/manifoldfinance/OpenMevRouter. Last accessed 20 April 2022

2. Guillermo Angeris, Tarun Chitra, Alex Evans, Stephen Boyd Guillermo Angeris et al. Optimal Routing for Constant Function Market Makers. arXiv:2204.05238 In *arXiv 1911.03380*, 26 Jul 2021 Optimization and Control (math.OC); Trading and Market Microstructure (q-fin.TR)

3. Guillermo Angeris et al. An analysis of Uniswap markets. 2019. arXiv: 1911.03380 An analysis of Uniswap markets. Mathematical Finance (q-fin.MF); Optimization and Control; Trading and Market Microstructure (q-fin.TR) *arXiv: 1911.03380*,

4. Guillermo Angeris, Alex Evans, Tarun Chitra Replicating Market Makers Mathematical Finance (q-fin.MF); Optimization and Control; Trading and Market Microstructure (q-fin.TR) *arXiv:2103.14769*, 26 Mar 2021.

5. Guillermo Angeris, Alex Evans, Tarun Chitra Constant Function Market Makers: Multi-Asset Trades via Convex Optimization Mathematical Finance (q-fin.MF); Optimization and Control; Trading and Market Microstructure (q-fin.TR) *arXiv:2107.12484* , 26 Jul 2021] https://doi.org/10.48550/arXiv.2107.12484

6. Guillermo Angeris, Tarun Chitra, Alex Evans, Stephen Boyd Optimal Routing for Constant Function Market Makers Optimization and Control (math.OC); Trading and Market Microstructure (q-fin.TR) *arXiv:2204.05238*, 11 Apr 2022] https://arxiv.org/abs/2204.05238v1

7. Suckut, Simon Uint512 Solidity Library *GitHub*, 11 Apr 2022] $https://github.com/SimonSuckut/Solidity_Uint512/blob/main/contracts/Uint512.sol$