FOR REVIEW

UniswapV2

Tiz

1. UniSwapV2

In the second iteration of Uniswap many changes were implemented to the protocol, mainly focusing on compatibility, longevity and security. Reducing the attack surface of the pool's contract a UniswapV2 "core" set of contracts was created and they are responsible for providing the reserve holding and swapping logic only while keeping the same constant product AMM system that was used in V1, the other main change was a switch in programming language from Vyper to Solidity.

2. UQ112x112

UQ112x112 is used to represent an unsigned fixed point number with 112 bits to represent the integer part of the number and 112 bits to represent the fractional part of the number. The integer part of the number can count from 0 to $2^{112}-1 = 5,192,296,858,534,827,628,530,496,329,220,095$ and it can have a precision of up to $\frac{1}{2^{112}} \approx 0.000000000000000000000000000000019259 \approx 1.925 \times 10^{-`34}$, it is commonly visualized as a binary number where a single place is defined as the 'point' where the integer representation ends and the fractional one begins, in this case there are 112 bits to the left and 112 bits to the right but it can also be 2 and 2 or any other combination, for some visual input say we have a 16 bit system and an arbitrary 16 bit binary number:

$$0101101010010001 \tag{1}$$

This can be commonly converted as a pure integer by computing each 1 as a power of two and summing like so:

$$1 * (2^{14}) + 1 * (2^{12}) + 1 * (2^{11}) + 1 * (2^{9}) + 1 * (2^{7}) + 1 * (2^{4}) + 1 * (2^{0}) = 46369 \tag{2}$$

NOTE: the rest of the bits have been omitted as they are being multiplied by 0 but they would still be present in a binary representation.
We can also expand this representation by using some of our 16 bits for precision, as an example taking 8 bits as the integers and 8 bits as the rationals, using the same number 0101101010010001, separating the bits in half for each representation

$$01011010|10010001 \tag{3}$$

$$1 * 2^{6} + 1 * 2^{4} + 1 * 2^{3} + 1 * 2^{1} + 1 * 2^{-1} + 1 * 2^{-4} + 1 * 2^{-7} = 90.5703125 \tag{4}$$

Therefore we could now represent the number 90.5703125 using the exact same number of bits and without the need for an explicit *float* class of numbers in our system.

3. Adding liquidity

Similar to the original UniSwap contract, the user who creates a liquidity pool must choose by themselves a ratio with which to add liquidity to the protocol, the main difference now is that the amount of liquidity tokens minted are not dependent on the amount of ETH that the user deposited now using the geometric mean to support tokens made up of two arbitrary tokens, the geometric mean of two number is defined as the square root of their product, $\sqrt{xy}$, this is to ensure that the

seemingly arbitrary ratio that was initially deposited has less weight on the value of a share and puts more weight on the size and balance of the pool being created. The geometric mean is similar to the arithmetic mean (average) but it applies better to variables that are multiplied and need to find a number that multiplied by itself n times is the number you're looking for, due to the multiplicative properties rooted in the protocol it makes sense to implement for liquidity calculations, some concrete examples using different pool assets are a resource to visualize this and see the effects this mean has on the amount of liquidity minted depending on the various assets and amounts (Price as of February 27 2025).

1. Pool 1: 1 WETH and 2,261 DAI
   Calculating the geometric mean:

$$\sqrt{1 * 2261} \approx 47.54 UNI \tag{5}$$

   Meaning the depositor would get 47.54 shares that can be burnt for 1 WETH and 2261 DAI
2. Pool 2: 1 WETH and 3,500,175,008.75 MOG
   Calculating the geometric mean:

$$\sqrt{1 * 3,500,175,008.75} \approx 59162.2 UNI \tag{6}$$

   Meaning the depositor would get 59162.2 shares that can be burnt for 1 WETH and 3,500,175,008.75 MOG
3. Pool 3: 3,500,175,008.75 MOG and 2,261 DAI
   Calculating the geometric mean:

$$\sqrt{3,500,175,008.75 * 2,261} \approx 2813164.7 UNI \tag{7}$$

   Meaning the depositor would get 2813164.7 shares that can be burnt for 2261 DAI and 3,500,175,008.75 MOG

These examples present some insight into the magnitude the geometric yields when minting liquidity, where it presents a sense of magnitude not only of the size of the deposit but how many tokens a single share of liquidity is entitled to claim, a much better solution to using one of the assets to dictate directly how much one share was worth like the previous iteration.
Once a pool has been created the user needs to deposit using the existing ratio

$$\frac{UNI_{mint}}{UNI_{res}} = \frac{x_{mint}}{x_{res}} \tag{8}$$

A possible DOS attack was taken into account on the contract that mitigates the possibility that adding liquidity to a pool would be deemed too expensive for users looking to add small amounts of liquidity, and where the minimum amount to add 1×10^{-18} could inflate and become unattainable, this is achieved by burning 1×10^{-15} of 1000 minimum shares of liquidity every time a new pool is created, permanently locking it up , this is known as an inflation attack (Team RareSkills (2013)). Starting on the example without any liquidity being initially burned, thus the creator of any pool is free to dictate the size of the pool they are creating, assuming they are adversarial and their goal is to handicap the protocol they create the smallest pool they can so they deposit enough ETH and DAI to get $1e^{-18}$ liquidity tokens thus the geometric mean must equal that number $\sqrt{xy} = 1 \times 10^{-18}$ since the minimum deposit is arbitrary the user opts for a simple $1e^{-18}$ WETH and DAI in order to get the same amount of shares (since the square root of a number squared is that same number). Now they have a tiny amount of liquidity that others may deposit into, but this user (or any other) does not want to support the network they want to destroy it thus they send 1 ETH and 2000 DAI directly

into the pool contract, altering the balances but not minting new liquidity tokens, thus meaning that now if another user who actually wants to support the protocol wanted to add liqdidity into the protocol they would need to add around 4 thousand dollars for each $1e^{-18} = 0.000000000000000001$ shares of liquidity, making small deposits impossible thus heavily handicapping the decentralization and participation of the various liquidity pools, since no new ones can be created for a single token pair, if this was to be done on an important pair like $\frac{DAI}{WETH}$ provided in this example this present a big problem for the protocol, by burning 1000 minimum shares if the user were to deopsit the minimum in order to get $1e^{-15}$ shares of liquidity and they then deposited the same 1 ETH 2000 DAI into the contract each minimum share may be computed using the rule of three

$$\frac{4000 * 1 \times 10^{-15}}{1 \times 10^{-18}} = 4000 * 1 \times 10^{-3} = 4 \tag{9}$$

Meaning that the minimum amount of liquidity can at least be minted for 4 dollars, greatly improving the resistance to the inflation attack vector, in order to make a single minimum share worth $1000 the user would need to deposit 250 times more $(4000 * 250 = 1,000,000)$ meaning the user would need to be willing to give up a million dollars and still would not make it completely unusable since those funds would still be burned into the pool thus incentivizing more users to deposit and trade, ultimately doing more good than harm.

4. Protocol Fees

A major difference with the protocol fees using UniSwapV2 is that, unlike the original iteration, the 0.3% protocol fee is calculated inside the contract but there is an additional fee that can be turned on by a privileged user in the protocol when it is active, 0.05% will go to the protocol instead of liquidity providers, for gas efficiency, this protocol fee can be calculated by measuring the growth of the geometric mean of the reserves $\sqrt{k}$ and dividing by 6 in order to get $\frac{0.3\%}{6} = 0.05\%$, this change in $\sqrt{k}$ can be measured by calculating the ratio between the current k and the initial k using the relative change formula denoted as $f_{1,2}$:

$$f_{1,2} = \frac{\sqrt{k_2} - \sqrt{k_1}}{\sqrt{k_2}} = 1 - \frac{\sqrt{k_1}}{\sqrt{k_2}} \tag{10}$$

which of course is used to compute the change relative to it's own initial magnitude. To find s_m liquidity to mint in accordance to $\phi = \frac{1}{6}$ the change in k defined as $f_{1,2}$, we need to find an amount of new liquidity to mint to the fee address where the ratio of the new liquidity minted to the new total liquidity is equals to $\phi f_{1,2}$ like so:

$$\frac{s_m}{s_m + s_1} = \phi f_{1,2} \tag{11}$$

Solving for s_m

$$s_m = \phi f_{1,2}(s_m + s_1) = \phi f_{1,2} s_m + \phi f_{1,2} s_1 \tag{12}$$

$$s_m - \phi f_{1,2} s_m = \phi f_{1,2} s_1 \tag{13}$$

$$s_m(1 - \phi f_{1,2}) = \phi f_{1,2} s_1 \tag{14}$$

$$s_m = \frac{\phi f_{1,2} s_1}{(1 - \phi f_{1,2})} \tag{15}$$

Substituting $f_{1,2} = \frac{\sqrt{k_2}-\sqrt{k_1}}{\sqrt{k_2}}$

$$s_m = \frac{\phi(\frac{\sqrt{k_2}-\sqrt{k_1}}{\sqrt{k_2}})s_1}{(1-\phi(\frac{\sqrt{k_2}-\sqrt{k_1}}{\sqrt{k_2}}))} = \frac{\phi(\sqrt{k_2}-\sqrt{k_1})s_1}{\sqrt{k_2}\frac{\sqrt{k_2}-\phi(\sqrt{k_2}-\sqrt{k_1})}{\sqrt{k_2}}} = \frac{\phi(\sqrt{k_2}-\sqrt{k_1})}{\phi(\frac{\sqrt{k_2}}{\phi}-(\sqrt{k_2}-\sqrt{k_1}))} = \frac{\sqrt{k_2}-\sqrt{k_1}}{(\frac{1}{\phi}-1)\sqrt{k_2}+\sqrt{k_1}}s_1 \tag{16}$$

Following the same example given in the white-paper, substituting for: $\phi = \frac{1}{6}$.

$$s_m = \frac{\sqrt{k_2}-\sqrt{k_1}}{(6-1)\sqrt{k_2}+\sqrt{k_1}}s_1 = \frac{\sqrt{k_2}-\sqrt{k_1}}{(5)\sqrt{k_2}+\sqrt{k_1}}s_1 \tag{17}$$

In the case that the original depositor added 100 *DAI* and 1 *ETH* into the pool, if they withdrew when there was 96 *DAI* and 1.5 *ETH* in the pool the amount of liquidity to mint can be computed like so:

$$s_m = \frac{\sqrt{1.5*96}-\sqrt{1*100}}{5(\sqrt{(1.5*96)})+\sqrt{1*100}}(10) = 0.\overline{285714} \tag{18}$$

5. Flash swaps and adjustment for fees.

To accommodate the growing complexity of the DeFi world UniSwapV2 implemented the ability to swap one token to another and letting the user make other actions within the same transaction as long as the correct amount of the original token is paid into the pool, this is useful for MeV searchers who are constantly looking for inefficiencies between protocols so that they can arbitrage money and keep the various protocols balanced, the given example is a user looking to buy token ABC for token XYZ, the UniSwap procotol permits the pool to "lend" the XYZ tokens requested by the user that he is then able to use within the same txn to sell into ABC token on another protocol and then use those funds in order to pay the protocol back, this all needs to be done within the same transaction otherwise the entire operation will revert and the pool go back like nothing ever happened, another example that these loans can be used for is by using the XYZ tokens to selling collateral on a lending protocol and they paying uniswap with those same funds.

Practically this function is facilitated by the core contracts by checking that the constant product invariance is not broken with each swap, taking into account that x_{in} and y_{in} can both be non zero so the pool needs to handle the fees like so:

$$(x_1 - \frac{3}{1000}x_{in})(y_1 - \frac{3}{1000}y_{in}) = x_0 y_0 = k \tag{19}$$

Once again, as to only use integer numbers multiplying by 1000:

$$1000(x_1 - \frac{3}{1000}x_{in})1000(y_1 - \frac{3}{1000}y_{in}) = (1000x_1 - 3x_{in})(1000y_1 - 3y_{out}) = 1,000,000k \tag{20}$$

6. Price Oracle

In order to use the Uniswap pools as an oracle for pricing (**NOTE:** currently this is not recommended, the industry standard is using a decentralized price oracle that is able to stream a more robust price in a safe and decentralized manner, some examples are pythia and chainlink vrf) the protocol needed a more robust way than using the reserves to get pricing since reserves are manipulated very easily will lead to exploits on protocols that use such a weak pricing oracle, giving an example say a leveraged trading protocol is using a UniSwap pool for their pricing, at a deep enough liquidity and volume going through the pool it may seem like a pretty safe way to know what the price of 1 ETH is in

terms of DAI, say 1 ETH = 2000 DAI and the pool has 1,000 ETH and 2,000,000 DAI in the pool, a user that detects that the protocol is using the UniSwap pool for their pricing can use their own funds or flash loans from lending protocols such as AAVE or MakerDAO to inflate his balance into the thousands of ETH or millions of DAI, use that loan to open a big long or short position, buying or selling enough ETH to manipulate the price in their favor, closing the position and repaying the loans, all in the same transaction, effectively draining the protocol by directly controlling the price of the asset they're trading.
In order to mitigate this, UniSwapV2 is continually storing the price of $\frac{ABS}{XYZ}$ AND $\frac{XYZ}{ABC}$ at the start of each block so that any user is able to call the average price of any pool between two blocks, making it harder for exploiters to manipulate the price but there is still risk that the price can be outdated since the timestamps of the pricing oracle are chosen arbitrarily by the user, the Time-weighed average price or TWAP using timestamps t_1 and t_2 calculated from the accumulated price since the beggining of the pool's creation

$$a_t = \sum_{i=1}^{t} p_i$$

, to calculate the TWAP between two arbitrary timestamps:

$$p_{t_1,t_2} = \frac{\sum_{i=t_1}^{t_2} p_i}{t_2 - t_1} \tag{21}$$

Since the contract only stores the accumulated price from the beginning of the protocol ($i = 1$) so we can get it by subtracting the accumulated price until t_1 from the accumulated price until t_2 thus only leaving the accumulated price between t_1 and t_2 as follows:

$$p_{t_1,t_2} = \frac{\sum_{i=1}^{t_2} p_i - \sum_{i=1}^{t_1} p_i}{t_2 - t_1} = \frac{a_2 - a_1}{t_2 - t_1} \tag{22}$$

Therefore making a better yet still far from optimal price oracle using UniSwapV2's accumulated price.
Something to note is that reserves are stored along with the timestamp which is why UniSwap uses 112x112 bits for their numeral representation. UQ112X112 is a numeral system designed to represent a non-negative rational numbers with good precision that also fit in 224 bits, leaving 32 extra bits in a single 256 bit storage slot that Solidity uses, those extra bits are used to encode the timestamp along with the pool's token reserves (each of them is stored as a *uint*112 for the same reason.) as well as for extra storage for the accumulation of the prices for the price oracle.

6.1 skim and sync functions

Since the pools created in Uniswap are made with any two ERC-20 implementations it is possible that some of them have weird functions that could lead to the pool breaking the invariance and funds being stolen, the main "weird" functions that have been accounted for in the core implementation are tokens whose supply may overflow the 112x112 numerical implementation or ones that can arbitrarily inflate the contract's balance, thus manipulating the contract's balance in order to steal funds from the pool, these functions are implemented as skim() and sync().
skim() is used when a pool's balance overflows the 112x112 bit numerical implementation, providing the caller with the difference as a reward and sync() is used to rebalance the pool by updating the balances back to the saved reserves.

7. References:

1. Team RareSkills (2013), *ERC4626 Interface Explained*, from https://www.rareskills.io/post/erc4626 accessed February 17, 2023
2. Adams Hayden (2020), *Uniswap v2 Core*, from https://docs.uniswap.org/contracts/v2/overview, accessed February 17, 2023