Uniswap v2 Core

Hayden Adams
hayden@uniswap.io

Noah Zinsmeister
noah@uniswap.io

Dan Robinson
dan@paradigm.xyz

March 2020

Abstract

This technical whitepaper explains the design decisions behind the Uniswap v2 core contracts. It covers the contracts' new features—including arbitrary ERC20<>ERC20 pairs, a hardened price oracle that allows other contracts to estimate the time-weighted average price over a given interval, "flash swaps" that allow traders to receive assets and use them elsewhere before paying for them later in the transaction, and a protocol fee that can be turned on in the future. It also re-architects the contracts to reduce their attack surface. This whitepaper describes the mechanics of Uniswap v2's "core" exchange contract—the contract that stores liquidity providers' funds—as well as the factory contract used to instantiate exchange contracts.

1 Introduction

Uniswap v1 is an on-chain exchange on the Ethereum blockchain, implementing an automated market maker known as the "constant product market maker" [1]. Each Uniswap v1 exchange stores pooled liquidity from two assets, and makes a market in those two assets, maintaining the invariant that the product of the reserves cannot decrease. Traders pay a 30-basis-point fee on trades, which goes to liquidity providers. The contracts are non-upgradeable.

Uniswap v2 is a new implementation of the same automated market maker, with several new highly-requested features. Most significantly, it enables the creation of arbitrary ERC20<>ERC20 pairs, rather than supporting only pairs between ERC20 and ETH. It also provides a hardened price oracle that accumulates the relative price of the two assets at the beginning of each block. This allows other contracts on Ethereum to estimate the time-weighted average price for the two assets over arbitrary intervals. Finally, it enables "flash swaps" where users can receive assets for free and use them elsewhere on the chain, only paying for (or returning) those assets at the end of the transaction.

While the contract is not generally upgradeable, there is a private key that has the ability to update a variable on the factory contract to turn on an on-chain 5-basis-point fee on trades. This fee will initially be turned off, but could be turned on in the future, after which liquidity providers would earn 25 basis points on every trade, rather than 30 basis points.

As discussed in section 3, Uniswap v2 also fixes some minor issues with Uniswap v1, as well as rearchitecting the implementation, reducing Uniswap's attack surface by minimizing the logic in the "core" contract that holds liquidity providers' funds.

This white paper describes the mechanics of that core contract, as well as the factory contract used to instantiate exchange contracts. Actually using Uniswap v2 will require calling the exchange contract through a "helper" contract that computes the trade or deposit amount and transfers funds to the exchange contract.

2 New features

2.1 ERC-20 pairs

Uniswap v1 used ETH as a bridge currency. Every exchange contract is a pair between some asset and ETH. This makes routing simpler—every trade between ABC and XYZ goes through the ETH<>ABC pair and the ETH<>XYZ pair—and reduces fragmentation of liquidity.

However, this rule imposes significant costs on liquidity providers. All liquidity providers have exposure to ETH, and suffer impermanent loss based on changes in the prices of other assets relative to ETH. When two assets ABC and XYZ are correlated—for example, if they are both USD stablecoins—liquidity providers on a Uniswap pair ABC<>XYZ would generally be subject to less impermanent loss than the ABC<>ETH or XYZ<>ETH pairs.

Using ETH as a mandatory bridge currency also imposes costs on traders. Traders have to pay twice as much in fees as they would on a direct ABC<>XYZ pair, and they suffer slippage twice.

Uniswap v2 allows liquidity providers to create exchanges for any pair of ERC-20s. There is one canonical exchange for each such pair.

A proliferation of exchanges between arbitrary ERC-20<>ERC-20 pairs could make it somewhat more difficult to find the best path to trade a particular pair, but routing can be handled at a higher layer (either off-chain or through an on-chain helper or aggregator).

2.2 Price oracle

The marginal price offered by Uniswap (not including fees) at time t can be computed by dividing the reserves of asset a by the reserves of asset b.

$$p_t = \frac{r_t^a}{r_t^b} \tag{1}$$

Since arbitrageurs will trade with Uniswap if this price is incorrect (by a sufficient amount to make up for the fee), the price offered by Uniswap tends to track the relative market price of the assets, as shown by Angeris et al [2]. This means it can be used as an approximate price oracle.

However, Uniswap v1 is not safe to use as an on-chain price oracle, because it is very easy to manipulate. Suppose some other contract uses the current ETH-DAI price to settle a derivative. An attacker who wishes to manipulate the measured price can buy ETH from the ETH-DAI exchange, trigger settlement on the derivative contract (causing it to settle based on the inflated price), and then sell ETH back to the exchange to trade it back to the true price.[1] This might even be done as an atomic transaction, or by a miner who controls the ordering of transactions within a block.

[1] For a real-world example of how using Uniswap v1 as an oracle can make a contract vulnerable to such an attack, see [3].

Uniswap v2 improves this oracle functionality by measuring and recording the price *before* the first trade of each block (or equivalently, *after* the last trade of the previous block). This price is more difficult to manipulate than prices during a block. If the attacker submits a transaction that attempts to manipulate the price at the end of a block, some other arbitrageur may be able to submit another transaction to trade back immediately afterward in the same block. A miner (or an attacker who uses enough gas to fill an entire block) could manipulate the price at the end of a block, but unless they mine the next block as well, they may not have a particular advantage in arbitraging the trade back.

Specifically, Uniswap v2 *accumulates* this price, by keeping track of the cumulative sum of prices at the beginning of each block in which someone interacts with the contract. Each price is weighted by the amount of time that has passed since the last block in which it was updated, according to the block timestamp.[2] This means that the accumulator value at any given time (after being updated) should be the sum of the spot price at each second in the history of the contract.

$$a_t = \sum_{i=1}^{t} p_i \tag{2}$$

To estimate the *time-weighted average price* from time t_1 to t_2, an external caller can checkpoint the accumulator's value at t_1 and then again at t_2, subtract the first value from the second, and divide by the number of seconds elapsed. (Note that the contract itself does not store historical values for this accumulator—the caller has to call the contract at the beginning of the period to read and store this value.)

$$p_{t_1,t_2} = \frac{\sum_{i=t_1}^{t_2} p_i}{t_2 - t_1} = \frac{\sum_{i=1}^{t_2} p_i - \sum_{i=1}^{t_1} p_i}{t_2 - t_1} = \frac{a_{t_2} - a_{t_1}}{t_2 - t_1} \tag{3}$$

Users of the oracle can choose when to start and end this period. Choosing a longer period makes it more expensive for an attacker to manipulate the TWAP, although it results in a less up-to-date price.

One complication: should we measure the price of asset A in terms of asset B, or the price of asset B in terms of asset A? While the spot price of A in terms of B is always the reciprocal of the spot price of B in terms of A, the mean price of asset A in terms of asset B over a particular period of time is *not* equal to the reciprocal of the mean price of asset B in terms of asset A.[3] For example, if the USD/ETH price is 100 in block 1 and 300 in block 2, the average USD/ETH price will be 200 USD/ETH, but the average ETH/USD price will be 1/150 ETH/USD. Since the contract cannot know which of the two assets users would want to use as the unit of account, Uniswap v2 tracks both prices.

Another complication is that it is possible for someone to send assets to the exchange—and thus change its balances and marginal price—*without* interacting with it, and thus without triggering an oracle update. If the contract simply checked its own balances and updated the oracle based on the current price, an attacker could manipulate the oracle by sending an asset to the contract immediately before calling it for the first time in a block. If the last

[2]Since miners have some freedom to set the block timestamp, users of the oracle should be aware that these values may not correspond precisely to real-world times.

[3]The arithmetic mean price of asset A in terms of asset B over a given period is equal to the reciprocal of the *harmonic* mean price of asset B in terms of asset A over that period. If the contract measured the *geometric* mean price, then the prices would be the reciprocals of each other. However, the geometric mean TWAP is less commonly used, and is difficult to compute on Ethereum.

trade was in a block whose timestamp was X seconds ago, the contract would incorrectly multiply the *new* price by X before accumulating it, even though nobody has had an opportunity to trade at that price. To prevent this, the core contract caches its reserves after each interaction, and updates the oracle using the price derived from the cached reserves rather than the current reserves. In addition to protecting the oracle from manipulation, this change enables the contract re-architecture described below in section 3.2.

2.2.1 Precision

Because Solidity does not have first-class support for non-integer numeric data types, the Uniswap v2 uses a simple binary fixed point format to encode and manipulate prices. Specifically, prices at a given moment are stored as UQ112.112 numbers, meaning that 112 fractional bits of precision are specified on either side of the decimal point, with no sign. These numbers have a range of $[0,\ 2^{112} - 1]$[4] and a precision of $1\ /\ 2^{112}$.

The UQ112.112 format was chosen for a pragmatic reason — because these numbers can be stored in a uint224, this leaves 32 bits of a 256 bit storage slot free. It also happens that the reserves, each stored in a uint112, also leave 32 bits free in a (packed) 256 bit storage slot. These free spaces are used for the accumulation process described above. Specifically, the reserves are stored alongside the timestamp of the most recent block with at least one trade, modded with 2^{32} so that it fits into 32 bits. Additionally, although the price at any given moment (stored as a UQ112.112 number) is guaranteed to fit in 224 bits, the accumulation of this price over an interval is not. The extra 32 bits on the end of the storage slots for the accumulated price of A/B and B/A are used to store overflow bits resulting from repeated summations of prices. This design means that the price oracle only adds an additional three SSTORE operations (a current cost of about 15,000 gas) to the first trade in each block.

The primary downside is that 32 bits isn't quite enough to store timestamp values that will reasonably never overflow. In fact, the date when the Unix timestamp overflows a uint32 is 02/07/2106. To ensure that this system continues to function properly after this date, and every multiple of $2^{32} - 1$ seconds thereafter, oracles are simply required to check prices at least once per interval (approximately 136 years). This is because the core method of accumulation (and modding of timestamp), is actually overflow-safe, meaning that trades across overflow intervals can be appropriately accounted for given that oracles are using the proper (simple) overflow arithmetic to compute deltas.

2.3 Flash Swaps

In Uniswap v1, a user purchasing ABC with XYZ needs to send the XYZ to the contract before they could receive the ABC. This is inconvenient if that user needs the ABC they are buying in order to obtain the XYZ they are paying with. For example, the user might be using that ABC to purchase XYZ on another on-chain exchange, in order to arbitrage a price difference between that exchange and Uniswap, or they could be unwinding a position on Maker or Compound by selling the collateral to repay the debt.

Uniswap v2 adds a new feature that allows a user to receive and use an asset *before* paying for it, as long as they make the payment within the same atomic transaction. The `swap` function makes a call to an optional user-specified callback contract in between transferring out the tokens requested by the user and enforcing the invariant. Once the callback is

[4] The theoretical upper bound of $2^{112} - (\frac{1}{2^{112}})$ does not apply in this setting, as UQ112.112 numbers in Uniswap are always generated from the ratio of two uint112s. The largest such ratio is $\frac{2^{112}-1}{1} = 2^{112} - 1$.

complete, the contract checks the new balances and confirms that the invariant is satisfied (after adjusting for fees on the amounts paid in). If the contract does not have sufficient funds, it reverts the entire transaction.

A user can also repay the Uniswap pool using the same token, rather than completing the swap. This is effectively the same as letting anyone flash-borrow any of assets stored in a Uniswap pool (for the same 0.30% fee as Uniswap charges for trading).[5]

2.4 Protocol fee

Uniswap v2 includes a 0.05% protocol fee that can be turned on and off. If turned on, this fee would be sent to a `feeTo` address specified in the factory contract.

Initially, `feeTo` is not set, and no fee is collected. A pre-specified address—`feeToSetter`—can call the `setFeeTo` function on the Uniswap v2 factory contract, setting `feeTo` to a different value. `feeToSetter` can also call the `setFeeToSetter` to change the `feeToSetter` address itself. These two functions represent the only kind of upgradeability built into Uniswap v2.

Once the `feeTo` address is set, the protocol will begin charging a 5-basis-point fee, which is taken as a $\frac{1}{6}$ cut of the 30-basis-point fees earned by liquidity providers. That is, traders will continue to pay a 0.30% fee on all trades; $83.\overline{3}\%$ of that fee (0.25% of the amount traded) will go to liquidity providers, and $16.\overline{6}\%$ of that fee (0.05% of the amount traded) will go to the `feeTo` address.

Collecting this 0.05% fee at the time of the trade would impose an additional gas cost on every trade. To avoid this, accumulated fees are collected only when liquidity is deposited or withdrawn. The contract computes the accumulated fees, and mints new liquidity tokens to the fee beneficiary, immediately before any tokens are minted or burned.

The total collected fees can be computed by measuring the growth in $\sqrt{k}$ (that is, $\sqrt{x \cdot y}$) since the last time fees were collected.[6] This formula gives you the accumulated fees between t_1 and t_2 as a percentage of the liquidity in the pool at t_2:

$$f_{1,2} = 1 - \frac{\sqrt{k_1}}{\sqrt{k_2}} \tag{4}$$

If the fee was activated before t_1, we want the `feeTo` address to capture $\frac{1}{6}$ of fees that were accumulated between t_1 and t_2. Therefore, we want to mint new liquidity tokens to the `feeTo` address that have an ownership interest of $\phi \cdot f_{1,2}$ of the pool, where ϕ is 0.2.

That is, we want to choose s_m to satisfy the following relationship, where s_1 is the total quantity of outstanding shares at time t_1:

$$\frac{s_m}{s_m + s_1} = \phi \cdot f_{1,2} \tag{5}$$

After some manipulation, including substituting $1 - \frac{\sqrt{k_1}}{\sqrt{k_2}}$ for $f_{1,2}$ and solving for s_m, we can rewrite this as:

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

[5] Because Uniswap charges fees on input amounts, the fee relative to the *borrowed* amount is actually slightly higher: $\frac{1}{1-0.03} - 1 \approx 0.3092\%$.

[6] We can use this invariant, which does not account for liquidity tokens that were minted or burned, because we know that fees are collected every time liquidity is deposited or withdrawn.

Setting ϕ to $\frac{1}{6}$ gives us the following formula:

$$s_m = \frac{\sqrt{k_2} - \sqrt{k_1}}{5 \cdot \sqrt{k_2} + \sqrt{k_1}} \cdot s_1 \tag{7}$$

Suppose the initial depositor puts 100 DAI and 1 ETH into an exchange, receiving 10 shares. Some time later (without any other depositor having participated in that exchange), they attempt to withdraw it, at a time when the exchange has 96 DAI and 1.5 ETH. Plugging those values into the above formula gives us the following:

$$s_m = \frac{\sqrt{1.5 \cdot 96} - \sqrt{1 \cdot 100}}{5 \cdot \sqrt{1.5 \cdot 96} + \sqrt{1 \cdot 100}} \cdot 10 \approx 0.0286 \tag{8}$$

2.5 Meta transactions for pool shares

Pool shares minted by Uniswap v2 exchanges natively support meta transactions. This means users can authorize a transfer of their pool shares with a signature[7], rather than an on-chain transaction from their address. Anyone can submit this signature on the user's behalf by calling the *permit* function, paying gas fees and possibly performing other actions in the same transaction.

3 Other changes

3.1 Solidity

Uniswap v1 is implemented in Vyper, a Python-like smart contract language. Uniswap v2 is implemented in the more widely-used Solidity, since it requires some capabilities that were not yet available in Vyper (such as the ability to interpret the return values of non-standard ERC-20 tokens, as well as access to new opcodes such as `chainid` via inline assembly) at the time it was being developed.

3.2 Contract re-architecture

One design priority for Uniswap v2 is to minimize the surface area and complexity of the core exchange contract—the contract that stores liquidity providers' assets. Any bugs in this contract could be disastrous, since millions of dollars of liquidity might be stolen or frozen.

When evaluating the security of this core contract, the most important question is whether it protects *liquidity providers* from having their assets stolen or locked. Any feature that is meant to support or protect *traders*—other than the basic functionality of allowing one asset in the pool to be swapped for another—can be handled in a "helper" contract.

In fact, even part of the swap functionality can be pulled out into the helper contract. As mentioned above, Uniswap v2 stores the last recorded balance of each asset (in order to prevent a particular manipulative exploit of the oracle mechanism). The new architecture takes advantage of this to further simplify the Uniswap v1 contract.

In Uniswap v2, the seller sends the asset to the core contract *before* calling the swap function. Then, the contract measures how much of the asset it has received, by comparing

[7]The signed message conforms to the EIP-712 standard, the same one used by meta transactions for tokens like CHAI and DAI.

the last recorded balance to its current balance. This means the core contract is agnostic to the way in which the trader transfers the asset. Instead of transferFrom, it could be a meta transaction, or any other future mechanism for authorizing the transfer of ERC-20s.

3.2.1 Adjustment for fee

Uniswap v1's trading fee is applied by reducing the amount paid into the contract by 0.3% before enforcing the constant-product invariant. The contract implicitly enforces the following formula:

$$(x_1 - 0.003 \cdot x_{in})) \cdot y_1 >= x_0 \cdot y_0 \tag{9}$$

With flash swaps, Uniswap v2 introduces the possibility that x_{in} and y_{in} might both be non-zero (when a user wants to pay the exchange back using the same asset, rather than swapping). To handle such cases while properly applying fees, the contract is written to enforce the following invariant:[8]

$$(x_1 - 0.003 \cdot x_{in}) \cdot (y_1 - 0.003 \cdot y_{in}) >= x_0 \cdot y_0 \tag{10}$$

To simplify this calculation on-chain, we can multiply each side of the inequality by 1,000,000:

$$(1000 \cdot x_1 - 3 \cdot x_{in}) \cdot (1000 \cdot y_1 - 3 \cdot y_{in}) >= 1000000 \cdot x_0 \cdot y_0 \tag{11}$$

3.2.2 sync() and skim()

To protect against bespoke token implementations that can update the exchange contract's balance, and to more gracefully handle tokens whose total supply can be greater than 2^{112}, Uniswap v2 has two bail-out functions: `sync()`and `skim()`.

`sync()` functions as a recovery mechanism in the case that a token asynchronously deflates the balance of an exchange. In this case, trades will receive sub-optimal rates, and if no liquidity provider is willing to rectify the situation, the exchange is stuck. `sync()` exists to set the reserves of the contract to the current balances, providing a somewhat graceful recovery from this situation.

`skim()` functions as a recovery mechanism in case enough tokens are sent to an exchange to overflow the two uint112 storage slots for reserves, which could otherwise cause trades to fail. `skim()` allows a user to withdraw the difference between the current balance of the exchange and $2^{112} - 1$ to the caller, if that difference is greater than 0.

3.3 Handling non-standard and unusual tokens

The ERC-20 standard requires that transfer() and transferFrom() return a boolean indicating the success or failure of the call [4]. The implementations of one or both of these functions on some tokens—including popular ones like Tether (USDT) and Binance Coin (BNB)—instead have no return value. Uniswap v1 interprets the missing return value of

[8]Note that using the new architecture, x_{in} is not provided by the user; instead, it is calculated by measuring the contract's balance after the callback, x_1, and subtracting (x_0 - x_{out}) from it. This logic does not distinguish between assets sent into the contract before it is called and assets sent into the contract during the callback. y_{in} is computed in the same way, based on y_0, y_1, and y_{out}.

these improperly defined functions as false—that is, as an indication that the transfer was not successful—and reverts the transaction, causing the attempted transfer to fail.

Uniswap v2 handles non-standard implementations differently. Specifically, if a transfer() call[9] has no return value, Uniswap v2 interprets it as a success rather than as a failure. This change should not affect any ERC-20 tokens that conform to the standard (because in those tokens, transfer() always has a return value).

Uniswap v1 also makes the assumption that calls to transfer() and transferFrom() cannot trigger a reentrant call to the Uniswap exchange. This assumption is violated by certain ERC-20 tokens, including ones that support ERC-777's "hooks" [5]. To fully support such tokens, Uniswap v2 includes a "lock" that directly prevents reentrancy to all public state-changing functions. This also protects against reentrancy from the user-specified callback in a flash swap, as described in section 2.3.

3.4 Initialization of liquidity token supply

When a new liquidity provider deposits tokens into an existing Uniswap exchange, the number of liquidity tokens minted is computed based on the existing quantity of tokens:

$$s_{minted} = \frac{x_{deposited}}{x_{starting}} \cdot s_{starting} \tag{12}$$

But what if they are the first depositor? In that case, $x_{starting}$ is 0, so this formula will not work.

Uniswap v1 sets the initial share supply to be equal to the amount of ETH deposited (in wei). This was a somewhat reasonable value, because if the initial liquidity was deposited at the correct price, then 1 liquidity pool share (which, like ETH, is an 18-decimal token) would be worth approximately 2 ETH.

However, this meant that the value of a liquidity pool share was dependent on the ratio at which liquidity was initially deposited, which was fairly arbitrary, especially since there was no guarantee that that ratio reflected the true price. Additionally, Uniswap v2 supports arbitrary pairs, so many pairs will not include ETH at all.

Instead, Uniswap v2 initially mints shares equal to the *geometric mean* of the amounts deposited:

$$s_{minted} = \sqrt{x_{deposited} \cdot y_{deposited}} \tag{13}$$

This formula ensures that the value of a liquidity pool share at any time is essentially independent of the ratio at which liquidity was initially deposited. For example, suppose that the price of 1 ABC is currently 100 XYZ. If the initial deposit had been 2 ABC and 200 XYZ (a ratio of 1:100), the depositor would have received $\sqrt{2 \cdot 200} = 20$ shares. Those shares should now still be worth 2 ABC and 200 XYZ, plus accumulated fees.

If the initial deposit had been 2 ABC and 800 XYZ (a ratio of 1:400), the depositor would have received $\sqrt{2 \cdot 800} = 40$ pool shares.[10]

[9] As described above in section 3.2, Uniswap v2 core does not use transferFrom().

[10] This also reduces the likelihood of rounding errors, since the number of bits in the quantity of shares will be approximately the mean of the number of bits in the quantity of asset X in the reserves, and the number of bits in the quantity of asset Y in the reserves:

$$\log_2 \sqrt{x \cdot y} = \frac{\log_2 x + \log_2 y}{2} \tag{14}$$

The above formula ensures that a liquidity pool share will never be worth less than the geometric mean of the reserves in that pool. However, it is possible for the value of a liquidity pool share to grow over time, either by accumulating trading fees or through "donations" to the liquidity pool. In theory, this could result in a situation where the value of the minimum quantity of liquidity pool shares (1e-18 pool shares) is worth so much that it becomes infeasible for small liquidity providers to provide any liquidity.

To mitigate this, Uniswap v2 burns the first 1e-15 (0.000000000000001) pool shares that are minted (1000 times the minimum quantity of pool shares), sending them to the zero address instead of to the minter. This should be a negligible cost for almost any token pair.[11] But it dramatically increases the cost of the above attack. In order to raise the value of a liquidity pool share to $100, the attacker would need to donate $100,000 to the pool, which would be permanently locked up as liquidity.

3.5 Wrapping ETH

The interface for transacting with Ethereum's native asset, ETH, is different from the standard interface for interacting with ERC-20 tokens. As a result, several other decentralized exchanges do not support ETH, instead using a canonical "wrapped ETH" token, WETH [6].

Uniswap v1 is an exception. Since every Uniswap v1 pair included ETH as one asset, it made sense to handle ETH directly, which was slightly more gas-efficient.

Since Uniswap v2 supports arbitrary ERC-20 pairs, it now no longer makes sense to support unwrapped ETH. Adding such support would double the size of the core codebase, and risks fragmentation of liquidity between ETH and WETH pairs[12]. Native ETH needs to be wrapped into WETH before it can be traded on Uniswap v2.

3.6 Deterministic exchange addresses

As in Uniswap v1, all Uniswap v2 exchange contracts are instantiated by a single factory contract. In Uniswap v1, these exchange contracts were created using the CREATE opcode, which meant that the address of such a contract depended on the order in which that exchange was created. Uniswap v2 uses Ethereum's new CREATE2 opcode [8] to generate an exchange with a deterministic address. This means that it is possible to calculate an exchange's address (if it exists) off-chain, without having to look at the chain state.

3.7 Maximum token balance

In order to efficiently implement the oracle mechanism, Uniswap v2 only support reserve balances of up to $2^{112} - 1$. This number is high enough to support 18-decimal-place tokens with a totalSupply over 1 quadrillion.

If either reserve balance does go above $2^{112} - 1$, any call to the `swap` function will begin to fail (due to a check in the `_update()` function). To recover from this situation, any user can call the `skim()` function to remove excess assets from the liquidity pool.

[11] In theory, there are some cases where this burn could be non-negligible, such as pairs between high-value zero-decimal tokens. However, these pairs are a poor fit for Uniswap anyway, since rounding errors would make trading infeasible.

[12] As of this writing, one of the highest-liquidity pairs on Uniswap v1 is the pair between ETH and WETH [7].

References

[1] Hayden Adams. 2018. URL: https://hackmd.io/@477aQ9OrQTCbVR3fq1Qzxg/HJ9jLsfTz?type=view.

[2] Guillermo Angeris et al. *An analysis of Uniswap markets*. 2019. arXiv: 1911.03380 [q-fin.TR].

[3] samczsun. *Taking undercollateralized loans for fun and for profit*. Sept. 2019. URL: https://samczsun.com/taking-undercollateralized-loans-for-fun-and-for-profit/.

[4] Fabian Vogelsteller and Vitalik Buterin. Nov. 2015. URL: https://eips.ethereum.org/EIPS/eip-20.

[5] Jordi Baylina Jacques Dafflon and Thomas Shababi. *EIP 777: ERC777 Token Standard*. Nov. 2017. URL: https://eips.ethereum.org/EIPS/eip-777.

[6] Radar. *WTF is WETH?* URL: https://weth.io/.

[7] Uniswap.info. *Wrapped Ether (WETH)*. URL: https://uniswap.info/token/0xc02aaa39b223fe8d0a0e5c4f27

[8] Vitalik Buterin. *EIP 1014: Skinny CREATE2*. Apr. 2018. URL: https://eips.ethereum.org/EIPS/eip-1014.

4 Disclaimer

This paper is for general information purposes only. It does not constitute investment advice or a recommendation or solicitation to buy or sell any investment and should not be used in the evaluation of the merits of making any investment decision. It should not be relied upon for accounting, legal or tax advice or investment recommendations. This paper reflects current opinions of the authors and is not made on behalf of Paradigm or its affiliates and does not necessarily reflect the opinions of Paradigm, its affiliates or individuals associated with Paradigm. The opinions reflected herein are subject to change without being updated.