# Uniswap Whitepaper

⚠⚠⚠ **WORK IN PROGRESS** ⚠⚠⚠

[color=#F578FF] [TOC]

# Introduction

Uniswap is a protocol for automated token exchange on Ethereum. It is designed around ease-of-use, gas efficiency, censorship resistance, and zero rent extraction. It is useful for traders and functions particularily well as a component of other smart contracts which require guaranteed on-chain liquidity.

Most exchanges maintain an order book and facilitate matches between buyers and sellers. Uniswap smart contracts hold liquidity reserves of various tokens, and trades are executed directly against these reserves. Prices are set automatically using the constant product (x*y=k) market maker mechanism, which keeps overall reserves in relative equilibrium. Reserves are pooled between a network of liquidity providers who supply the system with tokens in exchange for a proportional share of transaction fees.

An important feature of Uniswap is the utilization of a factory/registry contract that deploys a separate exchange contract for each ERC20 token. These exchange contracts each hold a reserve of ETH and their associated ERC20. This allows trades between the two based on relative supply. Exchange contracts are linked through the registry, allowing for direct ERC20 to ERC20 trades between any tokens using ETH as an itermediary.

This document outlines the core mechanics and technical details for Uniswap. Some code is simplified for readability. Safety features such as overflow checks and purchase minimums are ommited. The full source code is availible on GitHub.

**Protocol Website:**

uniswap.io

**Documentation:**

docs.uniswap.io

**Code:**

github.com/Uniswap

**Formalized Model:**

https://github.com/runtimeverification/verified-smart-contracts/blob/uniswap/uniswap/x-y-k.pdf)

# Gas Benchmarks

Uniswap is very gas efficient due to its minimalistic design. For ETH to ERC20 trades it uses almost 10x less gas than Bancor. It can perform ERC20 to ERC20 trades more efficiently than 0x, and has significant gas reductions when compared to on-chain order book exchanges, such as EtherDelta and IDEX.

| Exchange | Uniswap | EtherDelta | Bancor | Radar Relay (0x) | IDEX | Airswap |
|----------------------|---------|-------------|---------|------------------|---------------------|---------------------|
| ETH to ERC20 | 46,000 | 108,000 | 440,000 | 113,000* | 143,000 | 90,000 |
| ERC20 to ETH | 60,000 | 93,000 | 403,000 | 113,000 | *143,000* | *120,000* |
| ERC20 to ERC20 | 88,000 | no | 538,000 | 113,000 | no | no |

*wrapped ETH

The cost of a direct ERC20 token transfer is 36,000 gas - approximately 20% less than an ETH to ERC20 trade on Uniswap.

# Creating Exchanges

uniswap_factory.vy is a smart contract that serves as both a factory and registry for Uniswap exchanges. The public function createExchange() allows any Ethereum user to deploy an exchange contract for any ERC20 that does not already have one.

```java
exchangeTemplate: public(address)
token_to_exchange: address[address]
exchange_to_token: address[address]
@public
def __init__(template: address):
self.exchangeTemplate = template
@public
def createExchange(token: address) -> address:
assert self.token_to_exchange[token] == ZERO_ADDRESS
new_exchange: address = create_with_code_of(self.exchangeTemplate)
self.token_to_exchange[token] = new_exchange
self.exchange_to_token[new_exchange] = token
return new_exchange
```

A record of all tokens and their assoicated exchanges is stored in the factory. With either a token or exchange address, the functions getExchange() and getToken() can be used to look up the other.

```java
@public
@constant
def getExchange(token: address) -> address:
return self.token_to_exchange[token]
@public
@constant
def getToken(exchange: address) -> address:
return self.exchange_to_token[exchange]
```

The factory does not perform any checks on a token when launching an exchange contract, other than enforcing the one-exchange-per-token limit. Users and frontends should only interact with exchanges associated with tokens they trust.

# ETH ⇄ ERC20 Trades

Each exchange contract ([uniswap_exchange.vy](uniswap_exchange.vy)) is associated with a single ERC20 token and holds a liquidity pool of both ETH and that token. The exchange rate between ETH and an ERC20 is based on the relative sizes of their liquidity pools within the contract. This is done by maintaining the relationship eth_pool * token_pool = invariant . This invariant is held constant during trades and only changes when liquidity is added or removed from the market.

A simplified version of ethToTokenSwap(), the function for converting ETH to ERC20 tokens, is shown below:

```java
eth_pool: uint256
token_pool: uint256
token: address(ERC20)
@public
@payable
def ethToTokenSwap():
fee: uint256 = msg.value / 500
invariant: uint256 = self.eth_pool * self.token_pool
new_eth_pool: uint256 = self.eth_pool + msg.value
new_token_pool: uint256 = invariant / (new_eth_pool - fee)
tokens_out: uint256 = self.token_pool - new_token_pool
self.eth_pool = new_eth_pool
self.token_pool = new_token_pool
self.token.transfer(msg.sender, tokens_out)
```

:::info

**Note:** For gas efficiency eth_pool and token_pool are not stored variables. They are found using self.balance and through an external call to self.token.balanceOf(self)

:::

When ETH is sent to the function eth_pool increases. In order to maintain the relationship eth_pool * token_pool = invariant , token pool is decreased by a proporitonal amount. The amount by which token_pool is decreased is the amount of tokens purchased. This change in reserve ratio shifts the ETH to ERC20 exchange rate, incentivizing trades in the opposite direction.

Exchanging tokens for ETH is done with the function tokenToEthSwap():

```java
@public
def tokenToEthSwap(tokens_in: uint256):
fee: uint256 = tokens_in / 500
invariant: uint256 = self.eth_pool * self.token_pool
new_token_pool: uint256 = self.token_pool + tokens_in
new_eth_pool: uint256 = self.invariant / (new_token_pool - fee)
eth_out: uint256 = self.eth_pool - new_eth_pool
self.eth_pool = new_eth_pool
self.token_pool = new_token_pool
self.token.transferFrom(msg.sender, self, tokens_out)
send(msg.sender, eth_out)
```

This increases token_pool and decreases eth_pool, shifting the price in the opposite direction. An example ETH to OMG purchase is shown below.

---

## Example: ETH → OMG

:::info

**Note:** This example uses a fee of 0.25%. The real Uniswap contract has a fee of 0.3%.

:::

10 ETH and 500 OMG (ERC20) are deposited into a smart contract by liquidity providers. An invariant is automatically set such that ETH_pool * OMG_pool = invariant .

> [color=#F578FF] ETH_pool = 10
>
> OMG_pool = 500
>
> invariant = 10 * 500 = 5000

An OMG buyer sends 1 ETH to the contract. A 0.25% fee is taken out for the liquidity providers, and the remaining 0.9975 ETH is added to ETH_pool. Next, the invariant is divided by the new amount of ETH in the liquidity pool to determine the new size of OMG_pool. The remaining OMG is sent to the buyer.

> [color=#F578FF] Buyer sends: 1 ETH
>
> Fee = 1 ETH / 500 = 0.0025 ETH
>
> ETH_pool = 10 + 1 - 0.0025 = 10.9975
>
> OMG_pool = 5000/10.9975 = 454.65
>
> Buyer receieves: 500 - 454.65 = 45.35 OMG

The fee is now added back into the liquidity pool, which acts as a payout to liquidity providers that is collected when liquidity is removed from the market. Since the fee is added after price calculation, the invariant increases slightly with every trade, making the system profitable for liquidity providers. In fact, what the invariant really represents is ETH_pool * OMG_pool at the end of the previous trade.

> [color=#F578FF] ETH_pool = 10.9975 + 0.0025 = 11
>
> OMG_pool = 454.65

new invariant = 11 * 454.65 = 5,001.15

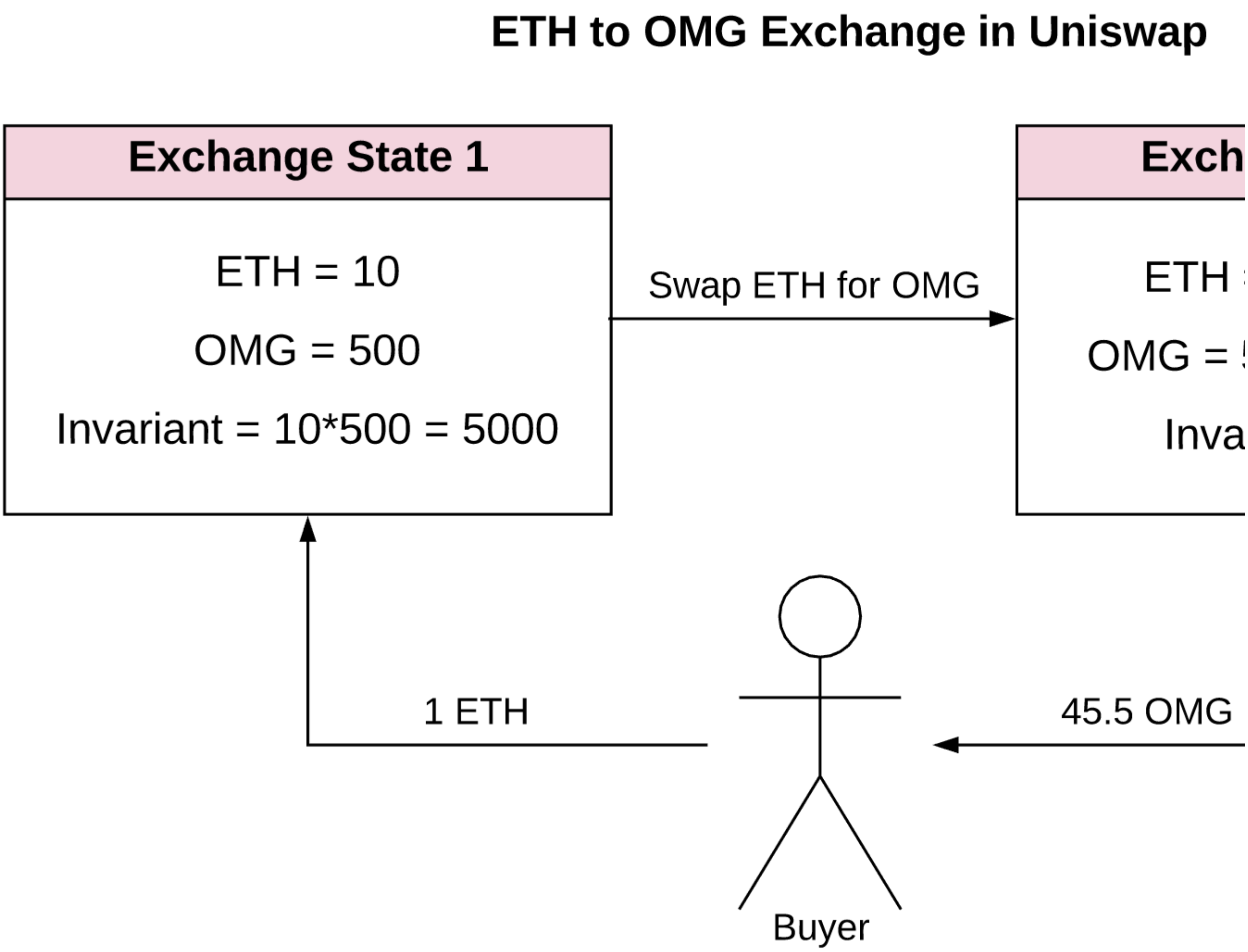
In this case the buyer received a rate of 45.35 OMG/ETH. However the price has shifted. If another buyer makes a trade in the same direction, they will get a slightly worse rate of OMG/ETH. However, if a buyer makes a trade in the opposite direction they will get a slightly better ETH/OMG rate.

[color=#F578FF] 1 ETH in

44.5 OMG out

Rate = 45.35 OMG/ETH

Purchases that are large relative to the total size of the liquidity pools will cause price slippage. In an active market, aribitrage will ensure that the price will not shift too far from that of other exchanges.




## ERC20 ⇄ ERC20 Trades

Since ETH is used as a common pair for all ERC20 tokens, it can be used as an intermediary for direct ERC20 to ERC20 swaps. For example, it is possible to convert from OMG to ETH on one exchange and then from ETH to KNC on another within a single transaction.

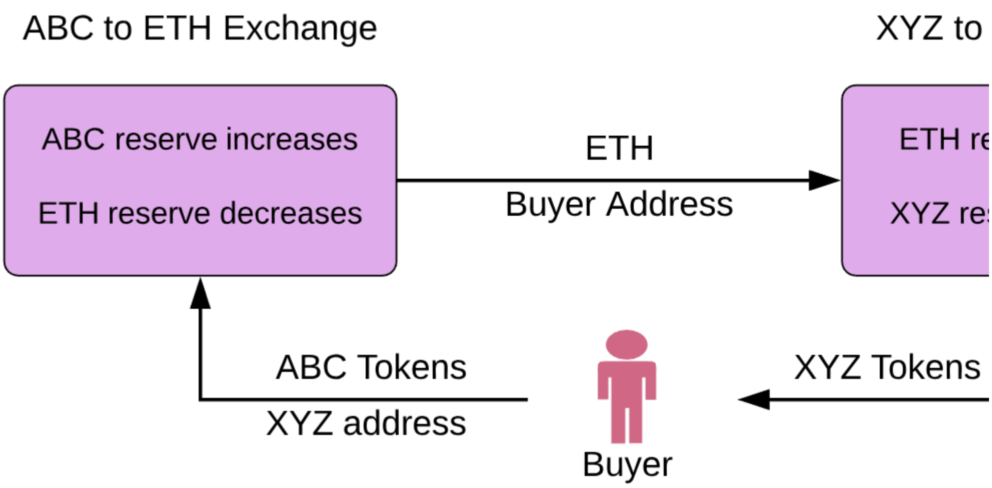
To convert from OMG to KNC (for example), a buyer calls the function tokenToTokenSwap() on the OMG exchange contract:

```java
contract Factory():
def getExchange(token_addr: address) -> address: constant
contract Exchange():
def ethToTokenTransfer(recipent: address) -> bool: modifying
factory: Factory
@public
def tokenToTokenSwap(token_addr: address, tokens_sold: uint256):
exchange: address = self.factory.getExchange(token_addr)
fee: uint256 = tokens_sold / 500
invariant: uint256 = self.eth_pool * self.token_pool
new_token_pool: uint256 = self.token_pool + tokens_sold
new_eth_pool: uint256 = invariant / (new_token_pool - fee)
eth_out: uint256 = self.eth_pool - new_eth_pool
self.eth_pool = new_eth_pool
self.token_pool = new_token_pool
Exchange(exchange).ethToTokenTransfer(msg.sender, value=eth_out)
```

where token_addr is the address of KNC token and tokens_sold is the amount of OMG being sold. This function first checks the factory to retreive the KNC exchange address. Next, the exchange converts the input OMG to ETH. However instead of returning the purchased ETH to the buyer, the function instead calls the payable function ethToTokenTransfer() on the KNC exchange:

```java
@public
@payable
def ethToTokenTransfer(recipent: address):
fee: uint256 = msg.value / 500
invariant: uint256 = self.eth_pool * self.token_pool
new_eth_pool: uint256 = self.eth_pool + msg.value
new_token_pool: uint256 = invariant / (new_eth_pool - fee)
tokens_out: uint256 = self.token_pool - new_token_pool
self.eth_pool = new_eth_pool
self.token_pool = new_token_pool
self.invariant = new_eth_pool * new_token_pool
self.token.transfer(recipent, tokens_out)
```

ethToTokenTransfer() receives the ETH and buyer address, verifies that the call is made from an exchange in the registry, converts the ETH to KNC, and forwards the KNC to the original buyer. ethToTokenTransfer() functions indentically to ethToTokenSwap() but has the additional input parameter recipient: address. This is used to forward purchased tokens to the original buyer instead of msg.sender, which in this case would be the OMG exchange.




## Swaps vs Transfers

The functions ethToTokenSwap(), tokenToEthSwap() , and tokenToTokenSwap() return purchased tokens to the buyers address.

The functions ethToTokenTransfer(), tokenToEthTransfer(), and tokenToTokenTransfer() allow buyers to make a trade and then immediately transfer purchased tokens to a recipient address.

# Providing Liquidity

## Adding Liquidity

Adding liquidity requires depositing an equivalent value of ETH and ERC20 tokens into the ERC20 token's associated exchange contract.

The first liquidity provider to join a pool sets the initial exchange rate by depositing what they believe to be an equivalent value of ETH and ERC20 tokens. If this ratio is off, arbitrage traders will bring the prices to equilibrium at the expense of the initial liquidity provider.

All future liquidity providers deposit ETH and ERC20's using the exchange rate at the moment of their deposit. If the exchange rate is bad there is a profitable arbitrage opportunity that will correct the price.

#### Liquidity Tokens

Liquidity tokens are minted to track the relative proportion of total reserves that each liquidity provider has contributed. They are highly divisible and can be burned at any time to return a proporitonal share of the markets liquidity to the provider.

Liquidity providers call the addLiquidity() function to deposit into the reserves and mint new liquidity tokens:

```python
@public
@payable
def addLiquidity():
token_amount: uint256 = msg.value * token_pool / eth_pool
liquidity_minted: uint256 = msg.value * total_liquidity / eth_pool


eth_added: uint256 = msg.value
shares_minted: uint256 = (eth_added * self.total_shares) / self.eth_pool
tokens_added: uint256 = (shares_minted * self.token_pool) / self.total_shares)
self.shares[msg.sender] = self.shares[msg.sender] + shares_minted
```

```
self.total_shares = self.total_shares + shares_minted

self.eth_pool = self.eth_pool + eth_added

self.token_pool = self.token_pool + tokens_added

self.token.transferFrom(msg.sender, self, tokens_added)
```

The number of liquidity tokens minted is determined by the amount of ETH sent to the function. It can be calulcated using the equation:

$amountMinted=totalAmount*\frac{ethDeposited}{ethPool}$

Depositing ETH into reserves requires depositing an equivalent value of ERC20 tokens as well. This is calculated with the equation:

$tokensDeposited=tokenPool*\frac{ethDeposited}{ethPool}$

### Removing Liquidity

Providers can burn their liquidity tokens at any time to withdraw their proportional share of ETH and ERC20 tokens from the pools.

$ethWithdrawn=ethPool*\frac{amountBurned}{totalAmount}$

$tokensWithdrawn=tokenPool*\frac{amountBurned}{totalAmount}$

ETH and ERC20 tokens are withdrawn at the current exchange rate (reserve ratio), not the ratio of their originial investment. This means some value can be lost from market fluctuations and arbitrage.

Fees taken during trades are added to total liquidity pools without minting new liquidity tokens. Because of this `ethWithdrawn` and `tokensWithdrawn` include a proportional share of all fees collected since the liquidity was first added.

### Liquidity Tokens

Uniswap liquidity tokens represent a liquidity providers contribution to an ETH-ERC20 pair. They are ERC20 tokens themselves and include a full implementation of EIP-20.

This allows liquidity providers to sell their liquidity tokens or transfer them between accounts without removing liquidity from the pools. Liquidity tokens are specific to a single ETH⇄ERC20 exchange. There is no single, unifying ERC20 token for this project.

## Fee Structure

- ETH to ERC20 trades
  - 0.3% fee paid in ETH
- ERC20 to ETH trades
  - 0.3% fee paid in ERC20 tokens
- ERC20 to ERC20 trades
  - 0.3% fee paid in ERC20 tokens for ERC20 to ETH swap on input exchange
  - 0.3% fee paid in ETH for ETH to ERC20 swap on output exchange
  - Effectively 0.5991% fee on input ERC20

There is a 0.3% fee for swapping between ETH and ERC20 tokens. This fee is split by liquidity providers proportional to their contribution to liquidity reserves. Since ERC20 to ERC20 trades include both an ERC20 to ETH swap and an ETH to ERC20 swap, the fee is paid on both exchanges. There are no platform fees.

Swapping fees are immediately deposited into liquidity reserves. Since total reserves are increased without adding any additional share tokens, this increases that value of all share tokens equally. This functions as a payout to liquidity providers that can be collected by burning shares.

Since fees are added to liquidity pools, the invariant increases at the end of every trade. Within a single transaction, the `invariant` represents `eth_pool * token_pool` at the end of the previous transaction.

## Custom Pools

### ERC20 to Exchange

The additional functions `tokenToExchangeSwap()` and `tokenToExchangeTransfer()` add to Uniswap's flexibility. These functions convert ERC20 tokens to ETH and attempts an `ethToTokenTransfer()` at a user input address. This allows ERC20 to ERC20 trades against custom Uniswap exchanges that do not come from the same factory, as long as they implement the proper interface. Custom exchanges can have different curves, managers, private liquidity pools, FOMO-based ponzi schemes, or anything else you can think of.

### Opt-in Upgrades

Upgrading censorship resistant, decentralized smart contracts is hard. Hopefully **Uniswap 1.0** is perfect but it probably is not. If an improved **Uniswap 2.0** design is created, a new factory contract can be deployed. Liquidity providers can choose to move to the new system or stay in the old one.

The `tokenToExchange` functions enable trades with exchanges launched from different factories. This can be used for **backwards compatibility**. ERC20 to ERC20 trades will be possible within versions using both `tokenToToken` and `tokenToExchange` functions. However, across versions only `tokenToExchange` will work. All upgrades are opt-in and backwards compatible.

## Frontrunning

Uniswap can be frontrun to some extent. This is bounded by user set minimum/maximum values and transaction deadlines.

## DEX inside a Whitepaper