# Exploring Blockchain-based Decentralized Exchanges

## Bachelor Thesis

Benjamin Helmy
from
Solothurn, Switzerland

Faculty of Science, University of Bern

10. Oktober 2021

Prof. Christian Cachin
Jovana Micic
Cryptology and Data Security Group
Institute of Computer Science
University of Bern, Switzerland

# Abstract

Frontrunning has become a frequently discussed subject in the world of cryptocurrencies and decentralized exchanges. This is an act where a malicious attacker reacts to given information by trying to get his transactions executed before another transaction. This thesis covers some base concepts from transactions on the Ethereum blockchain, giving an in depth understanding of gas prices, miners and transaction ordering and how these topics are related to frontrunning. The architecture of the most popular decentralized exchange called Uniswap will be explained and its vulnerabilities to frontrunning. A literature research of blockchain concepts shows that each transaction on the Ethereum blockchain is in need of the native cryptocurrency Ether, which defines how fast a transaction will be executed. The participants who add transactions to the blockchain are called miners and are being incentivised for their services by a fee paid in Ether. Because miners are profit-oriented stakeholders in the Ethereum blockchain ecosystem, the possibility of frontrunning transactions by paying higher gas prices to animate a miner to execute a transaction before another is possible. The fact that miners are profit-oriented also leads to a trade-off between fast and cheap transactions. The Uniswap protocol, which is a system of non-upgradeable smart contracts on the Ethereum blockchain, is powering an approach of an automated market maker and is mostly used for swaps between two ERC-20 tokens. By looking at the architecture of Uniswap there are obviously lucrative possibilities to frontrun various transactions. A final approach tries to put together all previous gathered information by developing an algorithm which automatically tries to frontrun Uniswap transactions.

# Contents


**1 Introduction** **1**

**2 Definitions** **3**
2.1 Ethereum Blockchain . . . 3
2.1.1 Cryptocurrencies . . . 4
2.1.2 Transaction Pool . . . 4
2.1.3 Blockchain Transactions . . . 4
2.1.4 Overview . . . 5

**3 Frontrunning** **7**
3.1 Traditional Frontrunning . . . 7
3.2 Frontrunning on the Blockchain . . . 7

**4 Uniswap** **9**
4.1 Description and Usage . . . 9
4.1.1 Automated Market Maker . . . 9
4.1.2 Permissionless System . . . 9
4.1.3 Liquidity Pools . . . 10
4.1.4 Pair Prices . . . 10

**5 Frontrunning Uniswap** **13**
5.1 Introduction . . . 13
5.2 Structure . . . 13
5.2.1 Pending Transactions . . . 14
5.2.2 Information of Pending Transactions . . . 16
5.2.3 Uniswap Prices . . . 16
5.2.4 Arbitrage . . . 17
5.2.5 Frontrunning and Afterrunning . . . 17
5.2.6 Slippage Tolerance . . . 17
5.3 Potential Extractable Amount . . . 18

**6 Conclusion** **19**

# Chapter 1

# Introduction

Since the adventurous rise of the cryptocurrency Bitcoin, almost everybody has heard from this new technology called *blockchain*. Many new startups and platforms have risen with a blockchain based business model. Moreover, the sector of cryptocurrencies has increased massively. Up until now, there are more than six thousand cryptocurrencies listed on CoinMarketCap, the most popular platform for crypotcurrencies [10].
One of the most famous places to trade such cryptocurrencies is *Uniswap* [24]. Uniswap is a decentralized open source exchange platform to swap ERC-20 tokens with an average daily trading volume of approximately one billion USD [26].
A pivotal difference to other exchange platforms is that they use the approach of an *automated market maker* instead of the traditional order book system. An automated market maker system is not in need of a buyer and a seller for a trade, but holds a liquidity pools of two assets where the prices of each asset gets calculated by a constant product formula. Traders can input a given amount of one asset into the pool and receive an output according to a constant product formula.
One of the biggest security risks with Uniswap is *frontrunning*. With frontrunning a malicious attacker can observe transactions on the blockchain, before they are confirmed, and then react accordingly. This means that they attempt to have their own transaction finalised before or instead of the observed transaction [3].
Frontrunning is possible because Uniswap transactions occur on the Ethereum blockchain, where transactions get executed according to their gas prices [12]. The gas price is the amount of money an individual needs to pay so that their transaction gets executed. The participants who are in charge of executing transactions and including them into the blockchain are called *miners*. They bundle multiple transactions together into a block which they then include into the blockchain being incentivised by the sum of all fees of transactions they include into their block. So to frontrun a transaction one needs to gather knowledge about the gas price of a specific transaction and then send a transaction with an adjusted higher gas price, so that it gets included first.
To protect Uniswap transactions against frontrunning, a slippage parameter got included which checks if there is any significant price change which is higher than the defined slippage, since the time the transaction got placed, before the transaction gets finalised. If somebody would have frontrun a transaction changing the price of an asset significantly (more than the set slippage value), the transaction would be aborted.
There is already some related work about frontrunning attacks on blockchains like the paper from Eskandari, Moosavi, and Clark, where frontrunning attacks on decentralized exchanges are analyzed as well as abnormal miner behavior during an inital coin offering of a company [3].
A paper written by Sobol where the potential of frontrunning attacks on the decentralized exchange *Quipuswap* gets analysed gives some additional information [5]. Another engaging paper was written by Daian et al. where strategies of trading bots get analysed as well as how bots take part in so called priority gas auctions [2].
The goal of this thesis is to give an overview of different related topics concerning frontrunning

and decentralized exchanges and in a second step to develop an algorithm which tries to automatically frontrun specific transaction which happen on the decentralized exchange Uniswap, with the goal to make arbitrage.
In chapter two the manner of transactions which happen on the blockchain will be explained as well as a quick analysis of the data structure of transactions. Additionally, key concepts like mining and gas prices will be discussed in connection to a in depth analysis of frontrunning will be made in chapter three. Furthermore in chapter four an overview of the core concepts and architecture of the Uniswap protocol will be given. Finally, there will be an approach of developing an algorithm which frontruns Uniswap transactions, combing the established knowledge and concepts from the first part of the thesis.

# Chapter 2

# Definitions

## 2.1 Ethereum Blockchain

In order to understand how transactions can be frontrun and what that means, one first needs to take a deeper look into how the Ethereum blockchain works.
A blockchain can in simplified terms be seen as a huge online database where people can store and fetch data. While in a traditional databases entries are made and stored into different tables, on the blockchain information gets stored in the manner of a distributed ledger. Different information gets collected into a block, which then, when the block has fulfilled his storage capacity gets added to the the blockchain. Every block has an unique hash from which it can be identified. Additionally, each block holds the hash value of the block in front, forming a chronological chain. In **Figure 2.1** a blockchain containing 3 blocks can be seen. The first two blocks have already fulfilled their storage of possible transactions and are therefore already mined. It can also be seen that the second block holds the hash of the previous block. Because the third block has not fulfilled its storage capacity, it is still pending.

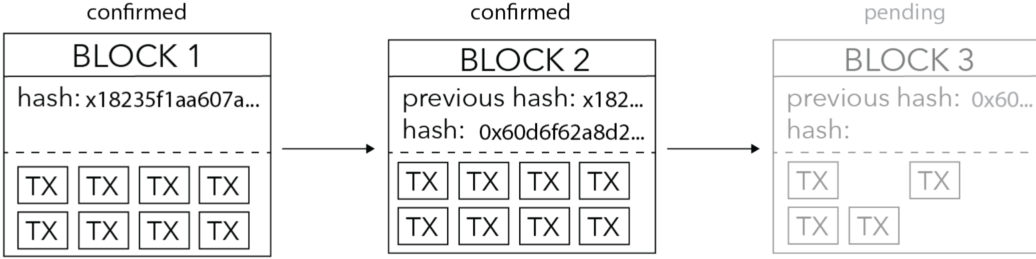


**Figure 2.1.** Two mined blocks and one pending block

Each piece of information added to the blockchain, also know as transactions, can also be identified by its unique hash. Furthermore, each transaction always contains the address of its *sender* and *receiver* as well as a *gas price*. The gas price is the fee the sender pays so that their transaction gets included in the blockchain. The higher the fee, the faster the transaction gets executed. If the fee is too low, the transaction may be pending forever and therefore will never get executed. As already mentioned, each block can be identified with its unique hash value and also has a link to its previous block. As all transactions are signed and hashed via cryptographic hash functions, it generates an unforgeable log containing all transactions ever made [4]. The process of linking a finished block with multiple transactions and then incorporating them into the blockchain is called *mining*. While this process is in need of computational power, the miner gets rewarded with money for his services which contains the sum of the gas money from all transactions the miner included in his block [12].
This fee is paid in Ether (ETH) the native currency of Ethereum. Nowadays (Sep. 2021), 1 ETH

is worth approximately 3000 USD [30].

### 2.1.1 Cryptocurrencies

Blockchains are most popular for transactions and mining of cryptocurrencies. As previously mentioned, the native currency of the Ethereum blockchain is Ether and miners get rewarded with it for offering their computational power to expand the blockchain.
While Ether can only be received when one takes part in the mining process or bought with fiat money[1], it is possible to create own tokens on the Ethereum Blockchain, so called ERC-20 tokens. These tokens are similar to other cryptocurrencies with the difference that they do not have an own underlying blockchain but are built on top of the Ethereum blockchain [21].

### 2.1.2 Transaction Pool

When a transaction gets made on the blockchain it first gets stored into a *transaction pool*, often also referred to as the *mempool*, waiting to get mined. Miners are profit-oriented actors earning money from every transaction they include into their blocks. So from a full transaction pool miners will always choose those transactions with the highest gas price. This leads to a trade-off for the users between cheap and fast transactions. If the gas price is too low there is a risk that the transaction will stay eternally in the transaction pool with a pending status.

### 2.1.3 Blockchain Transactions

**Listing 2.1** shows the transactions structure of a pending ethereum blockchain transaction.

```
{
  hash: '0xb0fe9a02ed7e2ac38c8f3688443267766bc6fe98dc5e5a8edf5934b23778f683',
  type: 0,
  accessList: null,
  blockHash: null,
  blockNumber: null,
  transactionIndex: null,
  confirmations: 0,
  from: '0x47135FC56cF8E979c2eA2CAcbaFF3461Ba2e3eD6',
  gasPrice: BigNumber { _hex: '0x195af22100', _isBigNumber: true },
  gasLimit: BigNumber { _hex: '0x03641a', _isBigNumber: true },
  to: '0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D',
  value: BigNumber { _hex: '0x00', _isBigNumber: true },
  nonce: 77,
  data: '0x791ac94700000000000000000000000000000000000000000850d73dbab067eb2e79
  8e2800000000000000000000000000000000000000000000000000551913039b73f2000000000
  00000000000000000000000000000000000000000000000000000a00000000000000000000000
  007135fc56cf8e979c2ea2cacbaff3461ba2e3ed6000000000000000000000000000000000000
  0000000000000000000061292087000000000000000000000000000000000000000000000000
  000000000000002000000000000000000000000e1030b48b2033314979143766d7dc1f40ef8ce
  11000zzz00000000000000000000c02aaa39b223fe8d0a0e5c4f27ead9083c756cc2',
  r: '0x7c4ca0231ea48023f9a6400bc54c7ade629f30577955b0b361a125a9888ddc02',
  s: '0x0c58171802b99321129cf8c403b5c6e8a216fd55a6559ac71e0be041aa998ab1',
  v: 38,
  creates: null,
  chainId: 1,
  wait: [Function (anonymous)]
}
```

**Listing 2.1.** Ethereum blockchain transaction structure

[1] Normal money that is used to pay for goods.

The *hash* value is an unique key generated for every transaction. With the hash value it is possible to get all information for the underlying transaction. The *blockNumber* indicates into which block the transaction got mined. When *blockNumber* is **null**, the transaction is still pending. The *from* and *to* values indicate from which address the transactions has come and where it is directed.
The *gasPrice* is in regards to this work, the most important attribute. It defines how much money the sender needs to pay for the transaction to get executed.
The *data field* is an encoded piece of information about which contracts and functions got called.

### 2.1.4 Overview

In **Figure 2.2** the complete blockchain system can be seen. It consists of *pending transactions* in the *transaction pool*, which get grouped into a *block*, which then gets added to the *blockchain* in a chained manner. A block holds a specific amount of transactions which can either be *confirmed* if the block has already been *mined* or still *pending* if the block has not yet been mined or the transaction has not yet be choosen from the transaction pool by a miner. Because miners are profit-oriented, they include the transactions with the highest *gas prices*.

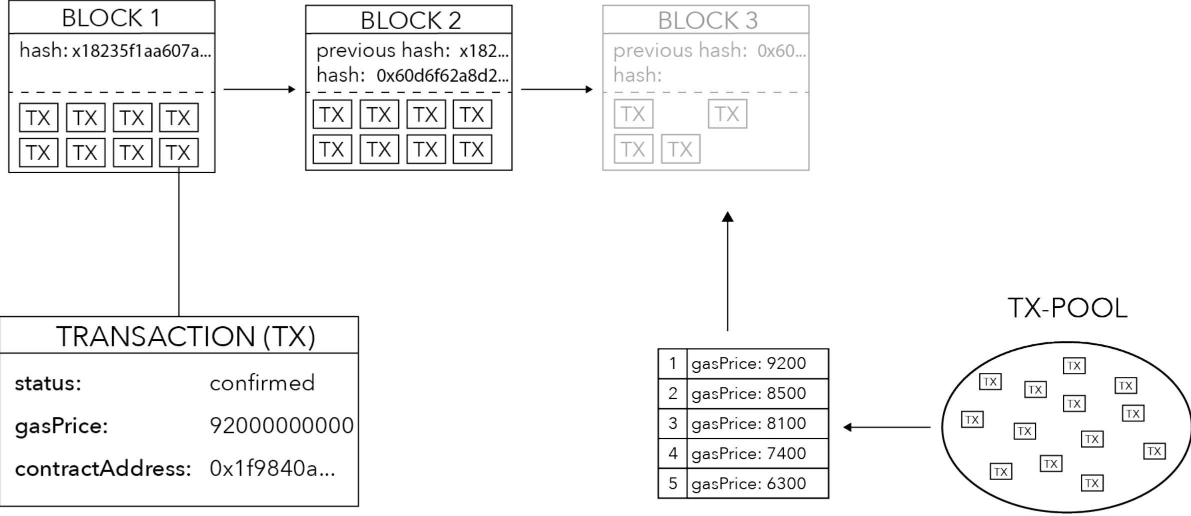


**Figure 2.2.** Blockchain ecosystem overview

# Chapter 3

# Frontrunning

## 3.1 Traditional Frontrunning

Frontrunning is a long-standing phenomenon. It is defined as a the reaction of someone to prior access to market information about upcoming events.
In 1977 the US. Securities Exchange Commission (SEC) defined frontrunning as follows: "The practice of effecting an options transaction based upon non-public information regarding an impending block transaction in the underlying stock, in order to obtain a profit when the options market adjusts to the price at which the block trades."[1]

**Example.** Imagine a client giving his stock broker the order to buy 100'000 shares of the company XY. This enormous buy action leads to a rise of the stock price by 10%. A malicious broker, would now buy shares of this stock for themselves before executing the order of their client, benefiting from the 10% increase of the stock price.

## 3.2 Frontrunning on the Blockchain

The concept of frontrunning works slightly different on the blockchain. As already mentioned in chapter 2.1, transactions which happen on the blockchain get bundled sequentially together and then get mined as a block into the blockchain. The miner then gets rewarded with a fee made up from all the gas prices of each individual transaction. Obviously, miners strive to get the highest possible reward. Therefore, they always choose those transactions with the highest available gas prices from the *transaction pool*. The current average paid *gasPrice* on the Ethereum blockchain is 123 GWEI, which matches an execution time of approximately three minutes (Oct. 2021) [13]. Because miners have the freedom to choose the transactions they want to include into their block, it makes sense that a miner will only choose those transactions with the highest gas price. To speed up an own transaction the logical action would be to increase the gas price.
This concept suggests why frontrunning on the Ethereum blockchain is possible. A transaction could get sent after another transaction and still get executed before it. The only thing one has to do, is to pay a higher gas price. Timing here is a crucial element. The average time a transaction is pending before it gets included into a block is about 180 seconds (Sep. 2021). Therefore, the transaction that should frontrun the first executed transaction, should be sent in the range of 180 seconds after the first transaction so that it will still be included in the same block. The more time passed after the first transaction the less likely it is that it can successfully frontrun the first transaction [13].
The action to pay a higher gas price than any other practitioner is not per se called frontrunning. The malicious action where a user sees or receives confidential information and then reacts accordingly to them is frontrunning.
It is also important to differentiate from insider trading and arbitrage. With insider trading, a

party has access to privileged information that **might** predict future actions, while frontrunning especially on the blockchain is a specific **observation** of pending transactions and a **reaction** to them. On the other hand it is difficult to distinguish between arbitrage and frontrunning. Because arbitrage is defined as the benefit of public information where the fastest to react profits, it could be argued that the access to the transaction pool is public as well. This would make sense in some cases, but with arbitrage nobody gets harmed, meanwhile the person who gets frontrun, mostly suffers from damages like worse stock prices or missed opportunities [3].
A common method where advantage of frontrunning is taken, is *sandwich trading*. This is an action where one spots a transaction in the transaction pool which will have an affect of a specific asset, for example a buy order which changes the price of a token by 10%. One will then frontrun this transaction with an own buy order as well executing a sell order with a lower gas price than the spotted transaction, "sandwiching" the transaction and exploiting the slippage the spotted transaction induces.

**Example.** Imagine person A scans the transaction pool and sees a transaction, where person B buys 1000 units of a specific token, with a gas price of 120 GWEI. This buy order would lead to a 15% price increase. Person A would now send an own transaction, with a gas price that is higher than 120 GWEI, where he also buys tokens of this token, as well as a second transaction, with a gas price smaller than 120 GWEI, selling the tokens again.
The order the transactions will be executed is Buy A - Buy B - Sell A. User A therefore has sandwiched the transaction of user B, taking advantage of the 15% price increase from the transaction of user B.

# Chapter 4

# Uniswap

## 4.1 Description and Usage

Uniswap is a peer-to-peer decentralized exchange implemented as a system of non-upgradeable smart contracts on the Ethereum blockchain, where ERC-20 Tokens can be exchanged and traded. Because of its blockchain basis, matters of decentralization and security get prioritized. The whole code is open-source and licensed under the GPL [22] [14]. Other than typical exchange platforms, Uniswap uses a automated market maker system. To understand the differences and advantages Uniswap offers against traditional exchanges, one first needs to look at the concept of automated market makers and how they differ from central limit order book based exchanges, as well as how permissionless systems work.
Until today, Uniswap released 3 versions of their protocol. Version 1 was released back in 2018 as a proof of concept for automated market makers. It was limited only to swaps between Ether and ERC20 tokens. With the upgrade to Version 2, token swaps between two ERC20 tokens got possible. Other additional features got introduced as well as some technical improvements were made[11]. With the release of Version 3 in March 2021, Uniswap tried to improve the flexibility and efficiency of their automated market maker system and as well improve capital efficiency for liquidity providers [20] [27].

### 4.1.1 Automated Market Maker

Currently, most public exchange platforms and markets use a classic order book system where buyers and sellers meet at a current price level. Uniswap on the other hand uses the approach of a so called *Automated Market Maker* AMM or also known as Function Market Maker.
The main idea of an automated market maker is to enable trades at any time. This works with a system of liquidity reserves of two assets where users can trade them at a price which gets determined by a fixed formula based on the amount of liquidity of both assets in the pool. After every trade a new ratio of the two assets is the pool where accordingly the new market price get calculated. The advantage this system should provide is that there is no need to always have a buyer and seller, instead it can directly be traded with the pool [25]. To provide enough liquidity, users are requested to add their own capital to these liquidity pools, earning incentives according to how much they contribute [21].

### 4.1.2 Permissionless System

The second thing where Uniswap differentiates itself from traditional exchanges is the permissionless design of the Uniswap protocol. Meaning that nobody can get restricted from using its services. In traditional financial markets, some services get restricted based on wealth or for example geography, while with the Uniswap protocol anyone can swap tokens at any time.

### 4.1.3 Liquidity Pools

The base concept of an automated market system is that users can trade a any time they want and are not restricted to a order book system where buyer and seller need to meet. This gets realised by the implementation of the already mentioned *liquidity pools*. Such a pool is only capable of holding two assets which then can be traded for each other. Such two assets in a pool are called *pair*. For every new liquidity pool that is, made a new smart contract on the blockchain gets created [23]. The initial amount of tokens in the pool is zero and someone needs to make an initial deposit at a ratio that reflects the current market situation, providing otherwise opportunities for arbitrage.
To incentivise people to deposit tokens into a liquidity pool, user receive specific *liquidity tokens* when they deposit tokens. These tokens should display a liquidity providers part of the total volume they provided for the pool. If a new liquidity pool gets created the initial liquidity provider receives liquidity tokens equal to the following equation

let x = amount token A
let y = amount token B

$$\sqrt{x * y} = t \tag{4.1}$$

**Example.** Imagine one deposits 20 units of token X and 5 units of token Y into a new created liquidity pool he will receive $\sqrt{20 * 5} = 10$ liquidity tokens. Now another user deposits as well 20 units of token X and 5 units of token Y he will as well receive 10 liquidity tokens because he now provided as much liquidity as user 1, providing now 50% of the whole pool.

Liquidity providers are incentivised for providing liquidity. For every transaction that is made with the liquidity pool the trader has to pay a 0.03% fee on his trade, which then gets dispensed to every provider according to their proportional contribution of the whole pool [23].

### 4.1.4 Pair Prices

The most frequent way users interact with the Uniswap protocol is the "Swap". This is a process where an individual exchanges ERC-20 tokens for each other [24]. Because of its automated market maker system, a trader trades against a liquidity pool. Inputting a specific amount of tokens as well as an additional 0.03% fee for the liquidity providers, then receiving the output of his desired token.
Because the price of a pair is dependent on the amount of liquidity of both tokens in the pool, every transaction has an influence on the price, which gets determined by the **constant product formula** below.

$$x * y = k \tag{4.2}$$

Trades that occur must not change the product $k$ of a pairs token reserve. Because $k$ stays constant it is often referred to as the *invariant*. The structure of this constant product formula leads to the result that bigger trades lead to a higher slippage in price.

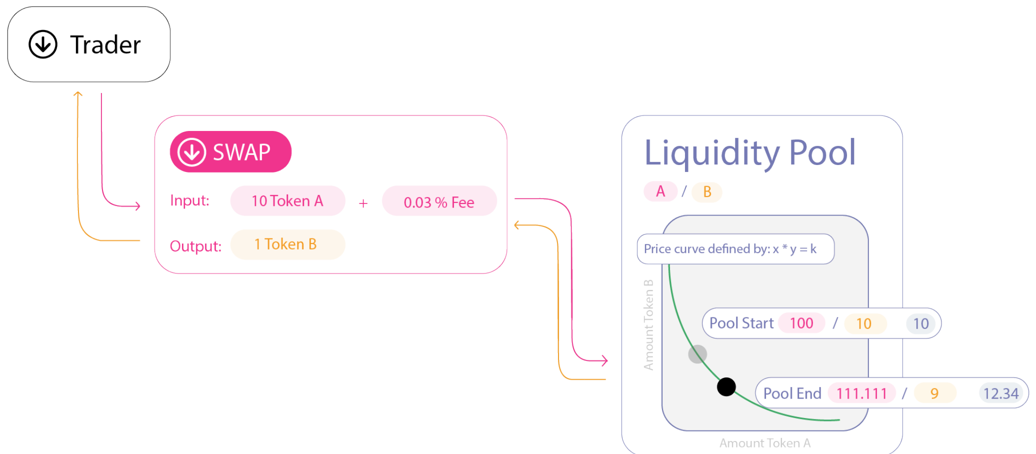


**Figure 4.1.** Effect of swap on token ratio [22]

In **Figure 4.3** it can be seen that the current liquidity pool holds 100 units of token A and 10 tokens of unit B. From the constant product formula, the invariant $k = 100 * 10 = 1000$.

**Example .** If a trader wants to receive one unit of Token B, changing the amount of token B to 9, he needs to input 11.111 units of Token A, as well as a 0.03% fee, so that the equation $111.111 * 9 = 1000$ still holds.
Imagining a trader wants to receive seven units of token B, he would need 200 units of token A, increasing the price from 11.111 units of token A for one unit of Token B, to 40 units of Token B for one unit of Token A.

This example shows very clearly that the liquidity pool system works very well for large pools and small trading volumes. But when the trades increase in volume, the prices may react very sensitively to it.
The structure of constant product formula leads to a plot similar to a $\frac{1}{x}$ curve, where it can be seen that as long as the liquidity pool is more or less evenly filled with both tokens, trades can be executed without any problem at a relatively normal price.

# Chapter 5

# Frontrunning Uniswap

## 5.1 Introduction

A real danger decentralized exchanges face are frontrunning attacks. For educational purposes it is analysed if and how it would be possible to make a frontrunning attack on transactions made on *Uniswap*, changing the transaction order by bribing the miners with a higher gas price and using a sandwich technique to make some profits [7].

## 5.2 Structure

Such a frontrunning algorithm consists of multiple steps and pieces of information one needs to put together. Since it is known that frontrunning a transaction from a non-miner position can only be made by adjusting gas prices, this will be our main focus. A coarse approach is described below:

- Watch the Ethereum blockchain for pending transactions.
- Filter for *Uniswap Router V2* contract calls.
- Extract information from pending transactions (pair, value, gas price).
- Get current trading-price of the pair.
- Get new price of the pair after observed pending transaction gets executed.
- Calculate the price slippage of the transaction.
- Calculate potential arbitrage from slippage.
- *If* lucrative: frontrun first transaction with higher gas fee with buy/sell order.
- Let observed transaction be executed.
- Run second transaction with lower gas fee with sell/buy order.

In **Algorithm 1** below, a pseudocode of how such an algorithm could look is being displayed. The *while true* statement in line 1 states an infinite loop, which lets the algorithm run forever. Line 2 and 3 then scan all pending transactions and filter for *Uniswap Router V2* contracts. Lines 4 and 5 extract the current price the pair tokens from the spotted Uniswap transactions are currently trading at and then on line 7 and 8 the new prices the tokens will have after the transaction will be finalised get extracted. Line 10 and 11 then would call another function which calculates the slippage the tokens would experience after the transaction would be finalised. Now

another function can be called that calculates the potential arbitrage that could be made from the token slippages with a specific order value [L. 13]. The consideration that two transactions with each a higher and lower gas price needs to be made, leads to the calculation of the expenses on line 14, where simply the gas price of the filtered transaction gets doubled. From there, if the potential arbitrage is higher than the expenses [L.16], one would buy tokens at the calculated order value with a higher gas price to frontrun the filtered transaction [L.17] and then sell the tokens again with a lower gas price [L.18], sandwiching the the filtered transaction.

**Algorithm 1** Pseudocode; Frontrunning Algorithm

```
1:  while True do
2:      for i in pending_Transactions do
3:          if i.contract = "Uniswap Router V2" then
4:              pair0_price = get_current_price(i.pair0);
5:              pair1_price = get_current_price(i.pair1);
6:
7:              pair0_new_price = get_next_price(i.pair0, i.value);
8:              pair1_new_price = get_next_price(i.pair1,i.value);
9:
10:             slippage_pair0 = get_slippage(pair0_price, pair0_new_price)
11:             slippage_pair1 = get_slippage(pair1_price, pair1_new_price)
12:
13:             arbitrage = get_arbitrage(slippage_pair0, slippage_pair1, order_value)
14:             expenses = (i.gas_price) * 2
15:
16:             if arbitrage > expenses then
17:                 buy_pair1(i.gas_price + 1)
18:                 sell_pair1(i.gas_price - 1)
19:
20:             end
21:         end
22:     end
23: end
```

### 5.2.1 Pending Transactions

The first step in realising the trading algorithm is to get all pending transactions on the Ethereum Blockchain. In **Listing 5.1** on Line 1, it can be seen that this is done by using the *ethers.js* library, which has the goal to interact with the Ethereum blockchain and its ecosystem [19]. Instead of running a full node on ones own, a Infura Node and it's API is used to access the Ethereum blockchain over a web-socket [L.2][17]. To fetch all pending transactions a custom web-socket provider is created, searching all transactions with the keyword "pending" [L.5].
To filter only transactions which call the "Uniswap Router V2" contract, a simple *if* clause is entered checking whether the *to* address is the address of the Uniswap contract [L.8] [29]. The V2 router is used although there was a release of a V3 with improved price prediction, but on starting this thesis V3 was not released yet.

```
1 var ethers = require("ethers");
```

```
2  var url = "wss://mainnet.infura.io/ws/v3/373902b40f7245d8af5da39f89ae4349";
3
4
5  customWsProvider.on("pending", (tx) => {
6    customWsProvider.getTransaction(tx).then(function (transaction) {
7      try{
8        if(transaction.to == "0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D"){
9        console.log(transaction);
10       }
11           else
12           {console.log("Transaction not relevant")}
13     } catch{
14       console.log("error")
15     }
16   });
17 });
18
```

**Listing 5.1.** Get pending Uniswap transactions

### Transaction Structure

The output one gets is a constant stream of information about pending transactions. A discussion of the specific values has already been made in chapter 2.2. An important thing to mention is that the values *blockHash*, *blockNumber*, *blockNumber* in **Listing 5.2** which are **null**, identify that it is a pending transaction, otherwise it would have some information.

```
1  {
2    hash: '0xb0fe9a02ed7e2ac38c8f3688443267766bc6fe98dc5e5a8edf5934b23778f683',
3    type: 0,
4    accessList: null,
5    blockHash: null,
6    blockNumber: null,
7    transactionIndex: null,
8    confirmations: 0,
9    from: '0x47135FC56cF8E979c2eA2CAcbaFF3461Ba2e3eD6',
10   gasPrice: BigNumber { _hex: '0x195af22100', _isBigNumber: true },
11   gasLimit: BigNumber { _hex: '0x03641a', _isBigNumber: true },
12   to: '0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D',
13   value: BigNumber { _hex: '0x00', _isBigNumber: true },
14   nonce: 77,
15   data: '0x791ac94700000000000000000000000000000000000000000850d73dbab067eb2e79
16   8e2800000000000000000000000000000000000000000000000000551913039b73f2000000000
17   00000000000000000000000000000000000000000000000000000a00000000000000000000000
18   007135fc56cf8e979c2ea2cacbaff3461ba2e3ed6000000000000000000000000000000000000
19   0000000000000000000061292087000000000000000000000000000000000000000000000000
20   000000000000002000000000000000000000000e1030b48b2033314979143766d7dc1f40ef8ce
21   11000zzz00000000000000000000c02aaa39b223fe8d0a0e5c4f27ead9083c756cc2',
22   r: '0x7c4ca0231ea48023f9a6400bc54c7ade629f30577955b0b361a125a9888ddc02',
23   s: '0x0c58171802b99321129cf8c403b5c6e8a216fd55a6559ac71e0be041aa998ab1',
24   v: 38,
25   creates: null,
26   chainId: 1,
27   wait: [Function (anonymous)]
28 }
```

**Listing 5.2.** Example of an Ethereum blockchain transaction structure

### 5.2.2 Information of Pending Transactions

When one takes a deeper look into the output, one can see that the *gas price* has to be converted as well as the *value* from a hexadecimal value to a decimal.
The last information which cannot be analysed yet is the *data* section. The cryptic string has to be decoded using a custom ABI for the Uniswap V2 Contract. This is needed to be done to see which functions and input values this transaction has. One has to watch out for transactions which call a Uniswap swap function and extract all information about the token pair, trading volume and gas price.

### 5.2.3 Uniswap Prices

Since it should be know which pairs got included in the filtered transaction, one can now query the current prices of the pair and the new prices the tokens should have after the transaction gets executed, because every buy or sell action changes the price of a token.
For this example one will look at the Pair **Ether - DAI**.
DAI is an Ethereum based stable coin which is soft-pegged at the U.S Dollar and some other cryptocurrencies. One unit of Dai currently trades at a price of 1.00 USD [8].
To retrieve exact prices of the Uniswap network, the *Uniswap SDK* which helps developers to build on top of Uniswap is beeing used [28] [15].
The SDK can be installed with:

```
1 npm install @uniswap/sdk
```

Before starting to program some libraries have to be imported. Furthermore in **Listing 5.3** on Line 3, the blockchain, the code should interact with, gets defined, in this case the mainnet. Also the token address of DAI needs to be added [L.4].
Now the chosen tokens are getting created. Because of the specifications of the Uniswap V2 contracts, one cannot use Ether directly but need to use wrapped Ether, the ERC-20 compatible version of Ether [6]. One can now create the desired token-pair [L.10].
A *route* object can then be created, to then query the current prices of the tokens [L.12]. From this point on, one can get the current prices or tokens. Because WETH is set as the input amount of the route object it will per default query the price of how many DAI one WETH would cost. To get the price of one DAI in WETH, the query can simply be inverted [L.15, L.16].
Now a new trade is being specified where a specified amount of WETH is getting swapped for DAI. Furthermore, one needs to specify the amount one wants to trade in Gwei, a small unit form of Ether [L.13]. Later, when all parts are getting put together, this value will be the value extracted from the spotted pending transactions.
Now query the new prices of the token pair [L.17]. It is important to keep in mind, that no actual trade is being made, but just a potential action has been defined where potential changes in price would follow.
A crucial thing which needs to be mentioned is the difference between the execution price and the midprice. The midprice is an aggregated price where past values play a important role, while the execution price is the price where it will actually be executed [18].

```
const {ChainId, Fetcher, WETH, Route, Trade, TokenAmount, TradeType} = require('
    @uniswap/sdk');

const chainId = ChainId.MAINNET;
const tokenAddress = '0x6B175474E89094C44Da98b954EedeAC495271d0F'; //DAI Address

const init = async () => {
const dai = await Fetcher.fetchTokenData(chainId, tokenAddress);
const weth = WETH[chainId];

const pair = await Fetcher.fetchPairData(dai, weth);

const route = new Route([pair], weth);
const trade = new Trade(route, new TokenAmount(weth, '100000000000000000'),
    TradeType.EXACT_INPUT); //100 Ether are beeing swapped

var first_midprice = route.midPrice.toSignificant(6);
var first_midprice_inverted = route.midPrice.invert().toSignificant(6);
var second_midprice = trade.nextMidPrice.toSignificant(6);
var slippage = (second_midprice/first_midprice)-1;
var difference = first_midprice - second_midprice
```

**Listing 5.3.** Get Uniswap prices [18]

### 5.2.4 Arbitrage

Every transaction which is made, has an impact on the price of the pair. The impact a transaction has on the price, is dependent on the size of the liquidity pool. A small swap in a big liquidity pool is likely to have no price impact, while a big transaction in a smaller liquidity pool can have a big impact.
With the knowledge gathered earlier, the potential arbitrage can now be calculated.

**Example.**
let $x$ = old price of token 1
let $y$ = new price of token 1
slippage = $(x/y) - 1$
arbitrage = $(input * slippage) - (gas_price * 2)$

### 5.2.5 Frontrunning and Afterrunning

When deciding that it would make sense to frontrun this transaction, one can simply send a transaction with a higher gas price to place it in front of the spotted transaction and benefit from the price increase of the token price. Then one would send a second transaction with an inverted sell/buy order to sandwich the spotted transaction to get back to the starting point.
The current documentation of Uniswap is very incomprehensible and not very developed yet. It is not clear how an implementation of a swap, where the gas price can be manipulated could be implemented, therefore, it would go beyond the scope of this thesis leaving the whole approach of this bot on a theoretical base.

### 5.2.6 Slippage Tolerance

The slippage tolerance is the percentage of how much the price of a pair may change between submission and execution. This is per default set on Uniswap to either 0.1% or 0.5%.
Slippage may happen, because between submission and execution, depending on the heights of

the gas price, it may take up to 20 second until the transaction gets executed. In this time other transactions may occur. With a pair with a high volume there is a high probability that there will happen transactions from other people and therefore changing the price.
To protect the swaper from unpleasant surprises, one can set a slippage tolerance, where one can set how much the price may change before the transaction gets cancelled.
Because of this slippage, it is important, that the frontrunning transaction one tries to execute is not too high, such that the defined slippage of the spotted transaction later aborts it.
An interesting thing to mention, is that if one sets the slippage tolerance bigger than 1% one gets a warning from Uniswap, saying that there is a risk that the transaction could get frontrunned.

## 5.3 Potential Extractable Amount

To check whether it is worth to frontrun a transaction, one needs to keep in mind that the arbitrage which can be made from frontrunning a transaction, needs to be bigger than the gas prices one needs to pay, while the input should not change the price of the pair more than the set slippage tolerance. If this is the case, it would be lucrative to frontrun the transaction.
An example of what could be possible with a functioning frontrunning bot shows the following example. In March 2021, a frontrunning bot made a profit of 0.056 Ether which was at that time approximately 84 USD from one sandwiching attack [9].
**Figure 5.1** shows three transactions where the first and the last transaction came from a frontrunning bot. All transactions were mined in the same block. It can be clearly seen that the frontrunning bot spotted the order for 11'496 MANA and then successfully sandwiched it [16].

| Date | Type | Price USD | Price ETH | Amount MANA | Total ETH |
|---|---|---|---|---|---|
| 2021-03-14 11:16:31 | sell | $1.1469328 | 0.00060687 | 22,816.616 | 13.846757 |
| 2021-03-14 11:16:31 | buy | $1.1595113 | 0.00061353 | 11,496.844 | 7.0536272 |
| 2021-03-14 11:16:31 | buy | $1.1422316 | 0.00060438 | 22,816.616 | 13.79 |

**Figure 5.1.** Example frontrunning bot [16]

# Chapter 6

# Conclusion

As the whole field of blockchain based technologies is relatively new, there are still many opportunities and technologies that are emerging at the moment. At the same time and also because there is no central control or surveillance, there are some wicked threats to keep an eye on. Miners can be bribed to execute some transactions before others and thereby provide the possibility of frontrunning. It is also posssible to get a glimpse into the future, with just some lines of code, to observe the transaction pool about the upcoming transactions.
Furthermore, frontrunning transactions is not that difficult. Pending transactions only need to be analysed how much gas they are paying and then an own transaction with a higher gas price can be sent.
Since it was not possible to decode the data which has been retrieved from pending transactions, the part of calculating the potential arbitrage belongs to a theoretical area.
While it is possible to scan pending transactions and then frontrun them, the question remains how much potential arbitrage could be extracted. The arbitrage needs to be large enough to compensate for the gas prices of both the "sandwich" transactions, while one needs to adjust their input value, such that the price change the first transaction leads to, does not exceed the set slippage tolerance canceling the chosen pending transaction.
However the goal of this thesis was to implement a working frontrunning algorithm, it was not possible to fully implement this because of a lack of knowledge and time. It was possible to retreive pending transactions and filter for those which interact with the Uniswap smart contracts, but decoding the content of the transactions was not possible.
While one cannot be punished for frontrunning other transactions, because there is no central surveillance or governmental institution, it still harms the individual who gets frontrun. In the case of sandwich trading, the frontrunner takes advantage of the change in price the frontrunned transaction is responsible for, but because every transaction changes the price of the token pair it will leave the original transaction with a worse price than if it did not frontrun it.
Because frontrunning is always an existing potential risk that could threaten a transaction and additionally, argued in the Flash Boys 2.0 paper from Daian et al., that the miner extractable value resulting from order optimization is a threat to the blockchain consensus stability threatening the whole Ethereum ecosystem, there needs to be thought of ways to impede or even prevent frontrunning in the future.

# Bibliography


[1] H. O. S. et al., “Report of the special study of the options markets to the securities and exchange commission,” 1979.

[2] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, “Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges,” Apr. 2019.

[3] S. Eskandari, M. Moosavi, and J. Clark, “Sok: Transparent dishonesty: Front-running attacks on blockchain,” pp. 170–189, Feb. 2019. DOI: `10.1007/978-3-030-43725-1_13`.

[4] S. Ferretti and G. D’Angelo, “On the ethereum blockchain structure: A complex networks theory perspective,” Aug. 2019.

[5] A. Sobol, “Frontrunning on automated decentralized exchange in proof of stake environment,” 2020. [Online]. Available: `https://eprint.iacr.org/2020/1206`.

[6] ADolmatov, “Wrapped ether (weth),”

[7] *CoinGecko, Top Decentralized Exchanges (DEX) Ranking by Trading Volume - Spot*, `https://www.coingecko.com/en/exchanges/decentralized`, Accessed: 2021-08-21.

[8] *Coinmarketcap, Dai*, `https://coinmarketcap.com/currencies/multi-collateral-dai/`, Accessed: 2021-08-23.

[9] *Coinmarketcap, Ethereum*, `https://coinmarketcap.com/currencies/ethereum/`, Accessed: 2021-08-21.

[10] *CoinMarketCap, Today’s Cryptocurrency Prices by Market Cap*, `https://coinmarketcap.com`, Accessed: 2021-08-20.

[11] R. Das, *Medium, Should you use Uniswap v1 or v2?* `https://medium.com/coinmonks/should-you-use-uniswap-v1-or-v2-70f8e6cb3c2c`, Accessed: 2021-10-09.

[12] *Ethereum, Ethereum official website*, `https://ethereum.org/en/eth/`, Accessed: 2021-08-26.

[13] *Etherscan, Ethereum Gas Tracker*, `https://etherscan.io/gastracker`, Accessed: 2021-08-29.

[14] *Github, Uniswap repository*, `https://github.com/Uniswap`, Accessed: 2021-09-24.

[15] *Github, Uniswap V2 - SDK*, `https://docs.uniswap.org/sdk/2.0.0/introduction`, Accessed: 2021-09-24.

[16] R. Huber, *Arbitrage und Frontrunning in DeFi*, `https://www.bitcoinsuisse.com/de/research/decrypt/arbitrage-und-frontrunning-in-defi`, Accessed: 2021-09-30.

[17] I. Inc, *Infura, The Infura Ethereum API*, `https://infura.io/product/ethereum`, Accessed: 2021-08-21.

[18] J. Klepatch, *EatTheBlocks YouTube Channel, Uniswap Tutorial for Developers (Solidity Javascript)*, `https://www.youtube.com/watch?v=0Im5iaYoz1Y&t=918s`, Accessed: 2021-09-01.

[19] R. Moore, *Ethers.js, Ethers Developer Documentation*, https://docs.ethers.io/v4/, Accessed: 2021-08-21.

[20] V. Systems, *Medium, V Swap v.s. Uniswap V3: A Brief Comparison*, https://medium.com/vsystems/v-swap-v-s-uniswap-v3-a-brief-comparison-d229657f1f6c, Accessed: 2021-10-09.

[21] Uniswap, *Uniswap Docs, Glossary*, https://docs.uniswap.org/protocol/concepts/V3-overview/glossary, Accessed: 2021-08-27.

[22] ——, *Uniswap Docs, How Uniswap works*, https://docs.uniswap.org/protocol/V2/concepts/protocol-overview/how-uniswap-works, Accessed: 2021-09-23.

[23] ——, *Uniswap Docs, Pools*, https://docs.uniswap.org/protocol/V2/concepts/core-concepts/pools, Accessed: 2021-09-24.

[24] ——, *Uniswap Docs, Swaps*, https://docs.uniswap.org/protocol/concepts/V3-overview/swaps, Accessed: 2021-08-21.

[25] ——, *Uniswap Docs, What Is Uniswap?* https://docs.uniswap.org/protocol/introduction, Accessed: 2021-08-29.

[26] *Uniswap Info, Overview*, https://info.uniswap.org/#/, Accessed: 2021-09-24.

[27] *Uniswap, Introduction to Version 3*, https://uniswap.org/blog/uniswap-v3/, Accessed: 2021-10-09.

[28] *Uniswap, SDK V2*, https://github.com/Uniswap/v2-sdk, Accessed: 2021-09-24.

[29] *Uniswap, Swap Router Contracts*, https://github.com/Uniswap/swap-router-contracts/blob/main/contracts/V2SwapRouter.sol, Accessed: 2021-09-24.

[30] *Webull, Price ETHUSD*, https://www.webull.com/quote/ccc-ethusd, Accessed: 2021-08-27.

# Erklärung

*Erklärung gemäss Art. 30 RSL Phil.-nat. 18*

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

12. November 2021 / Bern

Ort/Datum

Unterschrift