DollaramaBot: Optimizing An Arbitrage Bot

Fabian Krause
dept. of Computer Science
University of Freiburg
fabian.krause@mail.utoronto.ca
UTORid: krausefa

Maksym Muzychenko
dept. of Computer Science
University of Toronto
maksym.muzychenko@mail.utoronto.ca
UTORid: muzychen

Vladyslav Nekriach
dept. of Theoretical Cybernetics
Taras Shevchenko National University of Kyiv
vladyslav.nekriach@mail.utoronto.ca
UTORid: nekriach

Yuntao Wu
dept. Electrical and Computer Engineering
University of Toronto
winstonyt.wu@mail.utoronto.ca
UTORid: wuyuntao

***Abstract*—This paper presents a decentralized arbitrage bot implemented in Solidity using the Uniswap V2 protocol. The bot comprises four components: a Mempool API, a TypeScript server, an Arbitrage Smart Contract, and an on-chain data provider (Web3 connection). The Mempool API fetches transactions from the mempool, simulates their execution, and calculates the net changes in token balances of monitored pools. The TypeScript server listens to the Mempool API and checks for profitable arbitrage opportunities, taking into account the total gas price of the arbitrage transaction. If a profitable opportunity is detected, the TypeScript server calls the Arbitrage Smart Contract, which atomically performs the arbitrage transaction. The bot's gas usage was optimized in three iterations, with the final implementation minimizing the amount of code in the smart contract and using off-chain calculations in TypeScript. Overall, this decentralized arbitrage bot optimizes current approach to profiting from price discrepancies in Uniswap V2 pools.**

***Index Terms*—Arbitrage bot, Blockchain, Gas optimization, Mempool, Solidity, Uniswap V2**

I. Introduction

Blockchain technology has revolutionized the way we think about trust and decentralized systems. One of the most popular applications of this technology is the creation of digital currencies such as Bitcoin and Ethereum. However, blockchain technology has also enabled the creation of other innovative applications, including smart contracts and decentralized applications (DApps). Blockchain technology enables decentralized exchanges, which makes it easier to do arbitrage and make a profit than traditional centralized markets.

In recent years, the use of arbitrage bots has become increasingly popular in the cryptocurrency market. An arbitrage bot is a software program that automatically buys and sells digital assets across different exchanges to take advantage of price discrepancies. However, these bots often face challenges such as high transaction fees (known as gas fees) and the unpredictable nature of transaction confirmation times, which can lead to missed opportunities and losses.

In this project, we implement an arbitrage bot using Solidity and Uniswap V2 protocol and investigate ways to improve the arbitrage outcome. Our approach involves reducing gas usage and incorporating mempool information to improve the bot's efficiency and reliability. By leveraging blockchain technology, we demonstrate how our optimized arbitrage bot can provide significant benefits to traders and investors in the cryptocurrency market.

This paper is organized as follows. Section 2 provides a background on blockchain technology and arbitrage bots. Section 3 describes our methodology for optimizing the arbitrage bot. Section 4 presents our experimental results and evaluation. Finally, Section 5 concludes the paper with a discussion of the contributions and future directions.

II. Background

A. Memory Pool

The memory pool is a data structure used by nodes in a blockchain network to store unconfirmed transactions that are waiting to be included in the next block. Transactions are first broadcast to the network and are then validated by nodes before being added to the mempool. Once a transaction is in the mempool, miners can choose to include it in the next block they mine, depending on factors such as the transaction fee and size.

One of the main benefits of using mempool data is that it provides more up-to-date information about transactions that will be included in the future block. Offchain calculations, which rely on historical data or external sources, may not accurately reflect the current state of the network and can lead to suboptimal transaction parameters. By contrast, mempool data provides real-time information on the current state of the network and can be used to adjust transaction parameters accordingly.

Moreover, using mempool data can help reduce the risk of missed opportunities in arbitrage trading. Because the cryptocurrency market is highly volatile and prices can fluctuate rapidly, even small delays in transaction confirmation times can result in missed opportunities for profitable trades. By incorporating mempool data into the optimization of an arbitrage bot, traders can increase the likelihood of successful trades and reduce the risk of missed opportunities.

B. AMM and Arbitrage

An automated market maker (AMM) is an autonomous trading mechanism that eliminates the need of a centralized exchanges for users to trade and builds the foundation of decentralized exchanges (DEX) [1].

Uniswap is the first platform to use AMM [2]. Many of the current AMMs, such as Sushi Swap and PancakeSwap (BSC) are based on Uniswap [3], [4]. Our project focuses on these Uniswap based AMMs.

We can do arbitrage between these AMMs once the prices of same token pair diverges on different AMMs. We can encapsulate arbitrage transactions in one EVM transaction so that we can guarantee that the price won't move during the execution of the arbitrage.

Suppose we want to do arbitrage on token pair ETH/USDC. The ETH/USDC pair must exists on multiple AMMs on chain. Assume USDC is the token with actual value we want to get profit in. We will keep it reserved after arbitrage, while the ETH tokens will not be reserved after the arbitrage. If both of the tokens have actual values, we can reserve either one, but we need to be consistent.

The arbitrage can be done using Flashswap [10] of Uniswap V2. Suppose *pool1* and *pool2* are two pools that have the same two tokens on different AMMs. Once the price diverges, we can do arbitrage using smart contract. The contract calculates the price denominated in ETH. Suppose the price of ETH in *pool1* is lower:

1. Borrow x ETH from *pool1*. The contract need to repay the debt to *pool1* no matter the outcome of the arbitrage succeeds or not. The debt can be denominated in USDC. The flash loan must be repaid in the same transaction, which is enforced by the smart contract. If the we cannot pay back the money to the lender, the network will reject the transaction and the lender will always get the fund back.
2. Sell all the borrowed ETH on *pool2*. We get y_2 USDC.
3. Repay the debt to *pool1* using y_1 USDC.
4. As a result, we get a profit $y_2 - y_1$ USDC.

Our goal is to find x such that $y_2 - y_1$ is maximized. In the next section, we show how this x can be calculated.

C. Mathematical Formulation

Suppose initially, *pool1* has a_1 USDC and b_1 ETH, *pool2* has a_2 USDC and b_2 ETH. Then the following 2 equations must be satisfied in Uniswap,

$$a_1 b_1 = k_1, a_2 b_2 = k_2, \tag{1}$$

where k_1 and k_2 are constants depending on the reserves in the pool.

In pool1, we use USDC to borrow ETH, so the amount of USDC increases by Δa_1 and the amount of ETH decreases by Δb_1 in pool1. The equation must still be satisfied, so we have

$$\begin{aligned}(a_1 + \Delta a_1)(b_1 - \Delta b_1) &= k_1,\\ (a_1 + \Delta a_1) &= \frac{k_1}{b_1 - \Delta b_1},\\ \Delta a_1 &= \frac{k_1}{b_1 - \Delta b_1} - a_1,\\ \Delta a_1 &= \frac{k_1 - a_1 b_1 + a_1 \Delta b_1}{b_1 - \Delta b_1},\\ \Delta a_1 &= \frac{a_1 \Delta b_1}{b_1 - \Delta b_1}.\end{aligned} \tag{2}$$

Similarly, in *pool2*, the amount of USDC will decrease and the amount of ETH will increase. And we have

$$\Delta a_2 = \frac{a_2 \Delta b_2}{b_2 + \Delta b_2}. \tag{3}$$

Since we use all the ETH we get from *pool1* to trade for USDC in *pool2*, we have $\Delta b_1 = \Delta b_2$. This is equivalent to x in the previous section AMM and Arbitrage. Also, Δa_1 is equivalent to y_1, Δa_2 is equivalent to y_2.

The final profit we get is

$$f(x) = \Delta a_2 - \Delta a_1 = \frac{a_2 x}{b_2 + x} - \frac{a_1 x}{b_1 - x}. \tag{4}$$

We want to find x that maximizes the profit $f(x)$. This can be done by finding the solution of $f'(x) = 0$ and

$$\begin{aligned}f'(x) &= \frac{a_2(b_2 + x) - a_2 x}{(b_2 + x)^2} - \frac{a_1(b_1 - x) + a_1 x}{(b_1 - x)^2}\\ &= \frac{a_2 b_2}{(b_2 + x)^2} - \frac{a_1 b_1}{(b_1 - x)^2}.\end{aligned} \tag{5}$$

Setting Eq.5 to equal to 0 and ignoring the possibility of denominators being 0, we get

$$\begin{aligned}&\frac{a_2 b_2}{(b_2 + x)^2} - \frac{a_1 b_1}{(b_1 - x)^2} = 0\\ \Rightarrow\ &(a_2 b_2 - a_1 b_1)x^2 - 2b_1 b_2 (a_1 + a_2)x\\ &\qquad + b_1 b_2 (a_2 b_1 - a_1 b_2) = 0.\end{aligned} \tag{6}$$

This is a quadratic equation and the solution can be easily found:

$$\begin{aligned}x &= \frac{-b \pm \sqrt{b^2 - 4ac}}{2a},\\ \text{where } a &= a_2 b_2 - a_1 b_1,\\ b &= -2b_1 b_2 (a_1 + a_2),\\ c &= b_1 b_2 (a_2 b_1 - a_1 b_2),\\ &\text{with the constraints } 0 < x < b_1.\end{aligned} \tag{7}$$

And the solution x is the value we need to borrow from *pool1*.

III. Methodology

We implemented the bot in Solidity using the Uniswap V2 protocol following the idea of amm-arbitrageur [8].

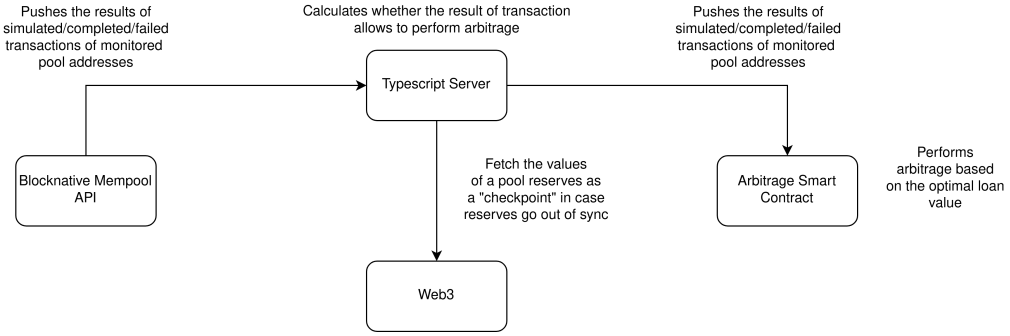

Fig. 1. System design

A. System architecture

The system comprises four components: Mempool API (3rd party service from Blocknative [9]), TypeScript server, Arbitrage Smart Contract, and on-chain data provider (Web3 connection). The overall system design can be found in Fig. 1.

Mempool API fetches transactions from mempool that affect monitored pools and simulates the execution of them. It calculates the net changes in token balances of these pools after executing mempool transactions. The TypeScript server is listening to the Mempool API, and on each balance change event, it calculates whether this transaction creates an arbitrage opportunity, taking into account the total price of gas the arbitrage transaction will burn. If current reserves allow for profitable arbitrage, the TypeScript server calls the Arbitrage Smart Contract, which atomically performs the arbitrage transaction.

In order to understand whether a balance change event is creating a profitable transaction, the TypeScript server has to maintain a local balance state of the monitored pools. However, we cannot use the balance changes from the simulated transactions as the only means to update the reserve states. The problem lies in the fact the Blocknative Mempool API executes simulations of transactions against the state of previous block. This limitation forces the system to update the local reserves state using on-chain data every block. That's when the Web3 connection comes into play, serving as a secure data source of the balance data. The problem is further discussed in section Blocknative Mempool limitations.

B. Blocknative Mempool limitations

The Blocknative Mempool API has a limitation: it does not take into account the possibility that some transactions that initially fail against the state of the previous block may actually succeed if another transaction is executed before the failing transaction. For example, if an arbitrage bot sends a transaction $T_{arbitrage}$ based on a mempool transaction $T_{mempool}$, even though $T_{arbitrage}$ will fail against the state of the previous block, it will be successful after $T_{mempool}$ is executed.

This limitation causes the Mempool API to report incorrect net balance changes to the Typescript server, which can result in the bot sending failing transactions. Malicious actors can exploit this limitation, which reduces the usability of the bot to zero.

To overcome this limitation, a custom implementation of a Mempool API is needed, one that simulates transactions based on the state obtained after previous mempool transactions have been executed. However, implementing such an API is outside the scope of this project.

C. Gas Optimization

We have done three iterations while developing the bot, trying to optimize its gas usage for sending transactions and for deploying it to the chain in the later iterations.

1) First Pass: The initial idea was to include all the calculation (e.g. getProfit which requires the code to solve a quadratic equation) in the smart contract and have minimum code in Typescript. The following issues are encountered.

1. There is no native support for floating point numbers in solidity. We need to use uint256 type to store it with multiples of 10s and still need to be careful about overflow. The accuracy may still be low.
2. There is no built-in function for calculating square roots, so we need to use Newton's method to approximate the solution, which could increase the gas usage due to large number of iterations.

Before any calculation, we need to divide all the a_1, a_2, b_1, b_2 by a large number 10^i so that the calculation will not overflow the 256 bit integer, and we need to multiply this number back in the solution to get an approximated integer solution that is close enough to the exact solution.

After the reduction by 10^i to avoid overflow issue, there are several ways we explore to solve the quadratic equation $g(x) = ax^2 + bx + c$, with a, b, c defined in E.q. 7:

1. Apply Newton's method to $g(x)$, and iteratively solve $x_{t+1} = x_t - \frac{g(x_t)}{g'(x_t)} = x_t - \frac{a(x_t)^2+bx_t+c}{2ax_t+b}$ until $x_{t+1} = x_t$ (convergence), with initial condition $x_0 = \pm c$. Since we don't know which root will satisfy the condition, we need to calculate both from 2 distinct initial conditions.
2. Use the quadratic formula, and calculate the square root function using Newton's method, $x_{t+1} = \frac{1}{2}(x_t + \frac{x_0}{x_t})$. We iteratively solve this until $x_{t+1} = x_t$ (convergence).

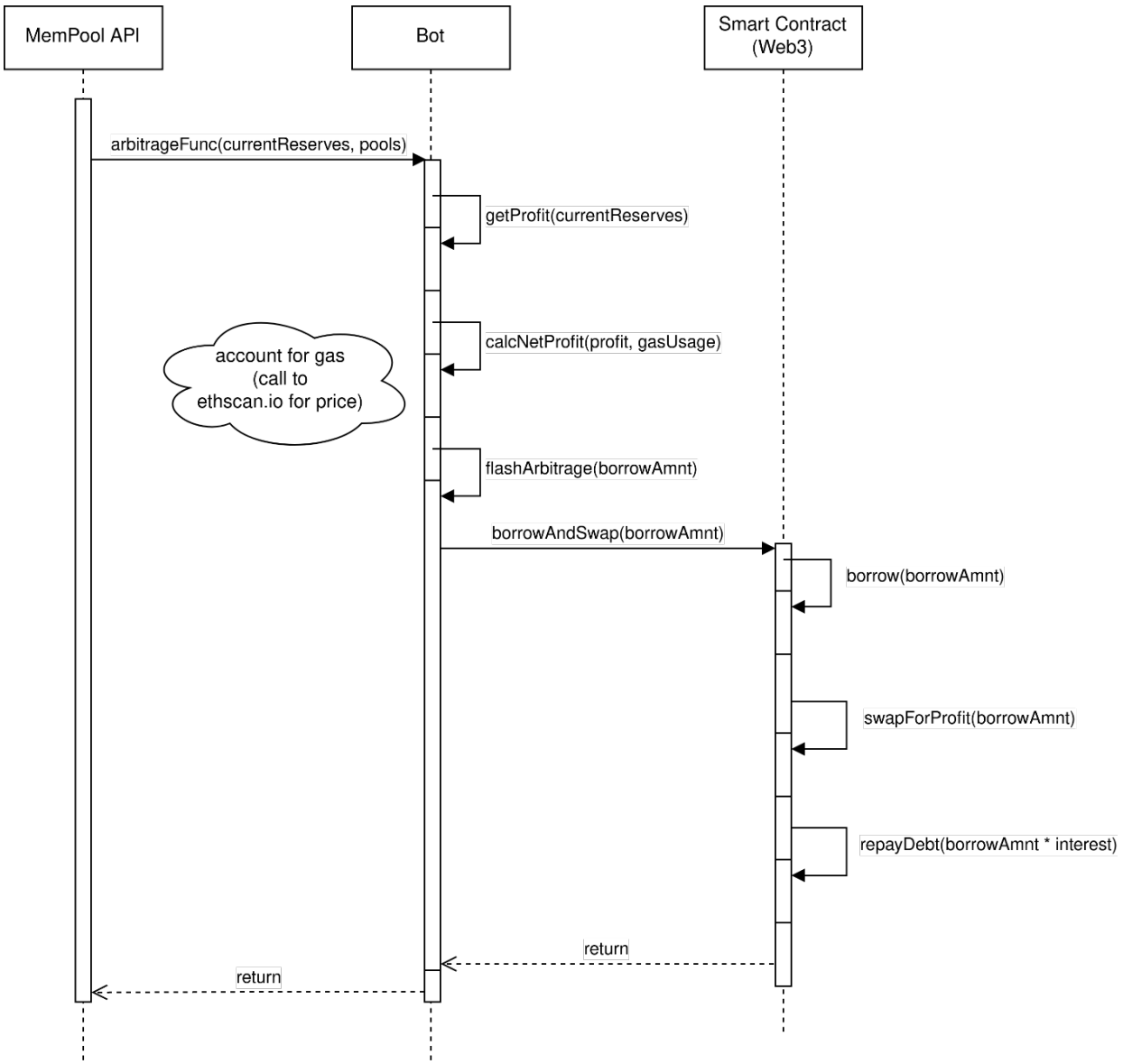

Fig. 2. Sequence diagram for performing arbitrage

3. A slight improvement to 2. Instead of non-deterministic amount of iterations, we exploit the property of 256 bit integers. To get the square root of 256 bit integers, we require at most 7 iterations of bit shift (division by 2) to reduce at most 128 bits for the final solution as is done in ABDK and PRB math [5], [6].

The results of total gas usages and accuracy can be find in Table I.

implementation	accuracy	gas usage of the full arbitrage
1. Naive Newton on $g(x)$	off by 1	X (cannot be deployed, gas over used)
2. Quadratic formula naive square root	closest integer	284,867
3. Quadratic formula efficient square root	closest integer	210,478

TABLE I
GAS USAGE AND ACCURACY OF THE 3 IMPLEMENTATIONS

2) Second Pass: We are using Solidity complier 0.8.7, and SafeMath library for Uniswap V2 together. Since Solidity version newer than 0.8.0 handles the overflow statements [7], we can also remove one of them. We find that using the language built-in overflow check is slightly more expensive, so we decided to go with the SafeMath library and reduced the gas usage to 206,970.

3) Third Pass: When we go through the Typescript code that deploy and trigger the smart contract functions, we find that several calculations, including the quadratic equation solver and UniswapV2 functions such as getAmountIn and getAmountOut, doesn't have to be in the smart contract. The

system after the modification can be seen in Fig. 2. With this modification, we reduced the gas usage to 177,700 without smart contract optimizer, and 173,394 with smart contract optimizer. We also reduced the gas used for deployment from 4,944,282 to 1,293,338. This also increases the accuracy of the solutions calculated, since Typescript supports floating points and can store and apply operations on any arbitrarily large numbers using the BigNumber library.

IV. Evaluation

We tested our bot on Ethereum Goerli testnet. Bot was able to capture arbitrage opportunities that were artificially created (example arbitrage transaction [12]). The bot was able to capture the transactions while they were in the mempool, allowing to capture arbitrage opportunities before they were recorded on-chain.

We also evaluated the advantage of using the Blocknative Mempool API by watching 100 UniswapV2 pools on the Ethereum main net for transactions, limited by the maximum daily simulated transactions in the free tier of Blocknative. Of the 152 total registered transactions, 57% were first discovered in the Mempool before successfully being confirmed later on, 37% were first seen on-chain, and the remaining transactions failed. Elapsed time between simulation and confirmation can be seen in Fig. 3. Two transactions with a high delay of 32s and 152s are not shown for better visibility. The median elapsed time is 8.4 seconds, the average elapsed time 10.6 seconds. The average time per block is 12.2 seconds as of April 8, 2023 [11].

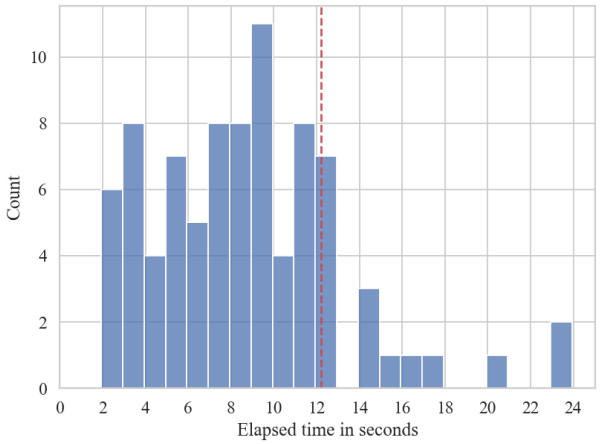

Fig. 3. Elapsed time between simulation of transaction and its inclusion in a block. Dashed red line shows average block generation time.

V. Discussion

In this paper, we presented an implementation of an arbitrage bot on the Uniswap V2 protocol using Solidity. Our system comprises four components: Mempool API, TypeScript server, Arbitrage Smart Contract, and on-chain data provider. The bot listens to balance change events, calculates whether an arbitrage opportunity exists, and atomically executes the arbitrage transaction if profitable.

Our evaluation of the Mempool API has shown that we can get a significant time advantage for the majority of transactions. One of it's main limitations is that the API executes simulations of transactions against the state of the previous block, which can lead to wrong balance changes being returned and failing transactions being sent. We identified the need for a custom implementation of a Mempool API that simulates transactions based on the state obtained after previous Mempool transactions were executed. However, this implementation was out of the scope of our project.

We also encountered issues with gas optimization during the development of our bot. Our initial idea was to include all the necessary calculations in the smart contract, but this approach had drawbacks due to the lack of native support for floating point numbers and built-in functions for calculating square roots in Solidity. We iterated on our design to minimize gas usage for sending transactions and deploying the bot to the chain.

Future work includes exploring alternative Mempool APIs and optimizing gas usage further. We also plan to evaluate the profitability of our bot in real-world scenarios and compare it to existing arbitrage bots to assess its effectiveness. Finally, we plan to set up a node in one of the blockchains to optimize bot's performance. Using our own node would allow us to have direct access to the blockchain data, bypassing the need for an external data provider. This could potentially reduce the latency and improve the reliability of the data, leading to more accurate identification of arbitrage opportunities and faster execution times.

References

[1] What Is an Automated Market Maker (AMM)? — Gemini. (n.d.). Gemini. https://www.gemini.com/cryptopedia/amm-what-are-automated-market-makers

[2] Uniswap V2 Overview. (2020, March 23). Uniswap Protocol. https://blog.uniswap.org/uniswap-v2

[3] Sushi. (n.d.). Sushi. https://www.sushi.com/

[4] Home — PancakeSwap. (n.d.). https://pancakeswap.finance/

[5] ABDK library. https://github.com/abdk-consulting/abdk-libraries-solidity

[6] PRB math. https://github.com/PaulRBerg/prb-math

[7] Solidity 0.8.0 changes https://docs.soliditylang.org/en/latest/080-breaking-changes.html

[8] Arbitrageur https://github.com/paco0x/amm-arbitrageur

[9] Blocknative Mempool Explorer https://explorer.blocknative.com

[10] Wang, D., Wu, S., Lin, Z., Wu, L., Yuan, X., Zhou, Y., Wang, H. and Ren, K., 2021, May. Towards a first step to understand flash loan and its applications in defi ecosystem. In Proceedings of the Ninth International Workshop on Security in Blockchain and Cloud Computing (pp. 23-28).

[11] Etherscan: Ethereum Average Block Time. https://etherscan.io/chart/blocktime

[12] Example of successful transaction performed by created arbitrage bot. shorturl.at/dACI4, transaction 0x64d8c25945b618ef3cf38ff617ada11a69f871df94febbbd9019fb5df9336172