Selfish Mining in Ethereum

Erik Finnesand and Kristian Horve Tjessem

1 Abstract

Is it possible to earn a higher reward from mining Ethereum compared to your relative hashing power? This project report will answer that question. We start off by a short introduction of the paper before explaining some general background information needed to understand the rest of the report. After this a more in depth explanation of the paper is written. This in depth explanation contains the mining strategy, setup and simulation method used in the paper. We then explain our simulation setup before showing the simulation results and comparing them to the results in the paper.

2 Introduction

The paper chosen for the project in DAT650 is "Selfish Mining in Ethereum" by Jianyu Niu and Chen Feng [1]. The paper explores an algorithm for selfish mining in Ethereum and looks at the performance of this algorithm compared to the normal Ethereum protocol in a simulated environment. The authors of the paper argues that selfish mining in Ethereum has not received as much attention as selfish mining in Bitcoin, as they want to explore if selfish mining in Ethereum is more profitable than Bitcoin thanks to the introduction of uncle(Ommer) and nephew blocks.

3 Background

Some background information is needed before going further.

3.1 Ethereum

Ethereum was released in July 2015 and is the second largest cryptocurrency by market capitalization just behind bitcoin . It is also the biggest decentralized platform that runs smart contracts. Smart contracts are simple programs stored on the blockchain that run when a simple condition is met. An example of this is a crowdfunding campaign that has to reach a specific goal within a time limit before the funded money is paid out to the creator of the campaign. If the goal is reached the money is paid out to the campaign owner, if not it is paid back to the crowdfunders.

Ethereum and Bitcoin are both cryptocurrencies with some key differences. They both use proof of work, but Ethereum is moving to proof of stake in the near future. Bitcoin and Ethereum are both alternatives to traditional fiat currency. Ethereum is also programmable, which means that it is possible to build and deploy decentralized applications on its network. Bitcoin block mining time is 10 minutes average compared to Ethereums 15 second avarage. Ethereum can also process 30 transactions per second compared to bitcoins 7 transaction per second. The last major difference is that Ethereum has an infinite supply while bitcoin has a finite supply capped at 21 million bitcoins.

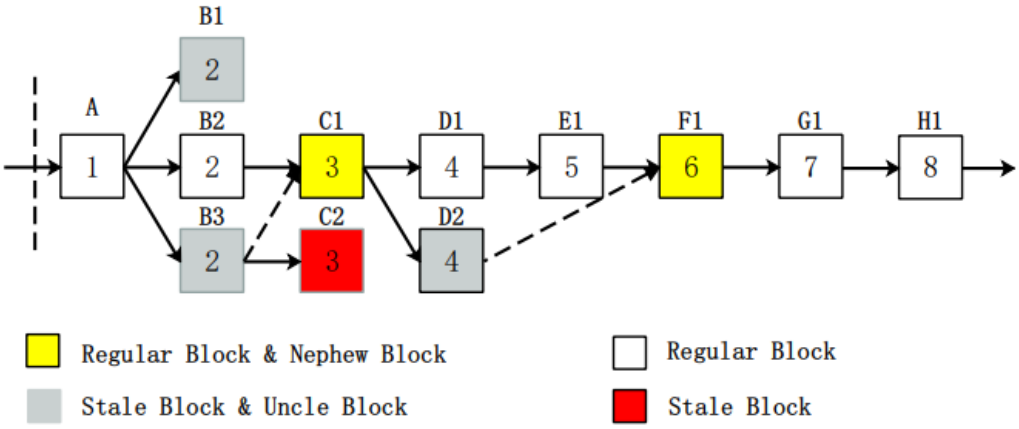

Figure 1: An example of the Ethereum blockchain

Ethereum also use different block types for its blockchain as seen in figure 1. The different types of blocks are regular blocks, stale blocks, uncle blocks and nephew blocks. A regular block is a block that follows the systems main chain. These blocks give a static reward to the miner when mined of 2 Ethereum and transaction fees of the transactions in the block. Stale blocks are blocks not included in the main chain and not referenced later by the main chain. These blocks give no reward for the miner and are simply dropped by the system. Uncle and nephew blocks are unique in Ethereum. An uncle block is a stale block that is the direct child of a block in the main chain. If the main chain is forked the first block in the fork not included in the main chain will be an uncle block. The creator of an uncle block receives a reward if the uncle block is referenced by some future regular block. If a regular block reference an uncle block the regular block is also a nephew block. The reward of the uncle block is based on the distance between the uncle block and the nephew block and is calculated with the formula

$$UncleBlockReward = \frac{8-l}{8} \cdot StaticMiningReward$$

Where "l" is the distance between the uncle and nephew block. If the distance is greater than 6 the uncle and nephew block will receive no reward additional reward.

For example if the distance is 1 as seen in figure 1 at block B3 and C1 the reward to the miner of B3 will be 7/8 of the static block reward. The nephew reward is always 1/32 of the static reward.

The introduction of uncle and nephew blocks in Ethereum is to give miners not in a mining pool the incentive to mine Ethereum.

3.2 Selfish mining

When mining a new block a miner will collect transactions into a block, run a proof of work (pow) algorithm and then publish the block to the network if the pow is valid. Miners will get a reward from publishing a valid block. This is because they will add a transaction at the start of the block

that is their reward. Other miners will validate the block that has been published, if it is valid it will be appended to the block chain, else it will be discarded.

In order to maximize ones profit, a miner can employ a strategy called selfish mining. The goal differs from that of honest miners by keeping blocks secret before strategically publishing them to the main chain. The paper introduce an algorithm that is explained in section 4.1.

Depending on how much hashing power(α) and how much network power(γ) the revenue of the selfish miner can change drastically. A selfish miner with a lot of mining power will gain a higher revenue than the miner would if using the normal Ethereum protocol. This will also cause the honest miners to gain less revenue because the selfish miner will force forks and make the honest miners waste hashing power on blocks that wont be a part of the main chain.

3.3 Mining pools

In Ethereum individual miners can team up to create mining pools. These miners mine together and split the reward based on the individuals hashing power. These mining pools are normally honest but could be used for selfish mining.

4 The paper

The paper was published in July 2019. Therefore some parts of the paper is outdated. An example of this is that the paper specifies that the reward of mining a regular block is 3 ethereum compared to todays reward of 2 ethereum.

4.1 Strategy

The strategy the authors of the paper use is to divide the mining nodes into two pools. A pool of selfish miners and a pool of honest miners.

The honest miners follow the normal Ethereum protocol where each miners observe the blockchain and mines new blocks on the main chain. Once a new block is produced the honest miners broadcast the block to everyone in the system. They also include as many references as possible to connect uncle blocks to nephew blocks.

The selfish miners use an algorithm specified in the paper instead of the normal ethereum protocol. This algorithm can withhold its newly mined block and publish them strategically to maximize the selfish pools revenue. The main idea behind this algorithm is to increase the pools share of static rewards and at the same time gain as many uncle and nephew rewards as possible.

When an honest miner create a block the length of the public chain will increase by 1. The selfish mining algorithm contains 4 cases for what to do when this happens. These are:

Case 1: If the new public chain is longer than the private chain, the selfish pool will discard its private chain and start mining on the public chain.

Case 2: If the new public chain has the same length as the private chain, the pool will immediately publish its private block, create a fork in the chain and hope that most miners will choose this chain to mine on.

Case 3: If the new public chain shorter than the private chain by just 1 block, the pool will publish its private chain so that all honest miners will start working on this chain since it is longer than the public chain.

Case 4: If the new public chain is shorter than the private chain by at least 2 blocks, the pool will publish the first few unpublished block in the private chain until it catches up to the public chain and force a fork. This is because the pool still has an advantage.

4.2 Setup

For the selfish mining simulation the authors of the paper simulate a system with 1000 miners. The selfish mining pool will at most control 450 of these miners, giving an $\alpha <= 0.45$. As mentioned before the honest blocks will use the normal ethereum protocol while the selfish pool will use the algorithm described in chapter 4.1. The simulation results are based on an avarage of 10 runs, where each run generates 100,000 blocks. The simulation uses a network power of 0.5 for the selfish miners and a static uncle reward of $4/8$ the mining reward. That authors of the papers decided to ignore the transaction fee reward for the blocks.

4.3 Simulation

The paper uses a 2-dimensional Markov process to model the behaviour of selfish mining in Ethereum. A markov process is a stochastic model describing a sequence of possible events where the probability of each event depends of the state of the previous event. This solution is pure statistical and may differ from a practical simulation where programming is used.

5 Simulation Setup

5.1 Model in go

We have since the initial creation of the software stumbled upon limitations and problems that are discussed in 6.3. Even still we are happy with that which we have achieved in the limited time of this project.

5.1.1 Simplified blockchain

Our blockchain is really simple. Each block has a hash which is just a randomly generated 10 character string. It has a reference to the previous block and a data unit. The unit contains information about the blocks value and if it was created by the selfish miner or the honest miner.

Doing it this way made it simple to tally up the final statistics. All that is needed is to loop through the chain and sum the values in the blocks data unit.

5.1.2 Concurrency

Go is a great language for concurrency, although not necessarily a requirement, we choose to attempt it such that our model is closer to reality. Though our model may be closer than a Markov calculation, it is still far away from what actually happens due to a simpler block chain model.

We have structured the program such that it has two goroutines (threads). The two routines are the miners. One is running the honest algorithm and the other runs the selfish mining algorithm. There is a third goroutine that is a controller of sorts. In our implementation it is the one that has the actual chain, more on the controller in section 5.1.3. So when a miner wants to publish a block, it sends it to the controller through a go channel.

5.1.3 Controller

Our controller plays an important part in our implementation. It is the sole authority on what the true state of the blockchain is. It has a simple loop that checks if there are any blocks in the channels from the miners, if so, add it to the chain. To prevent the process from maxing a CPU constantly checking, it has a sleep time of about 50ms when nothing is found before a new check.

Forks are a possibility and necessity for selfish mining in Ethereum. Our implementation struggled a lot with forks. The main thing we wanted to do was to make it so each miner had their own copy of the chain, but by the time we realised this, we had to little time to refactor the code. So the controller is the sole authority due to it being the only holder of the actual chain. This made it so we have a lot of bad coupling in the code that could have been avoided if each had their own chain. It should also give and easier time handling the logic of forks which quickly grew out of hand. Still we managed to get it to work good enough.

5.1.4 Miners

As mentioned section 5.1.2, there are two miners, each running their own algorithm. In our simple model the miners roll a random number between zero and one thousand, if the number is smaller or equal to its mining power, then it has successfully created a new block. Its power is percentage based, 100% means it has a power of 1000. 90% means 900, 45% means 450 etc. For each attempt it has to wait for 200ms before it can attempt again. We would want to go lower, but doing so created problems due to concurrency. A solution to this we wanted to implement was a concept of tickrate explored in 6.3.1.

That is all there is to it for the honest miner, it rolls a number, if number meets criteria, create and publish new block, then wait 200 ms and repeat. The selfish miner does the same rolling to check if it has successfully created a new block, but instead of immediately publishing the block, it runs the algorithm explained in 4.1.

5.1.5 Saving results

Storing the results for use later is easily done by saving it to a file. We chose to save it in a .csv format where semicolons are the separator. This makes is easy to import/open in a spreadsheet application to create nice figures from the data collected.

5.2 Running the simulation

We set a goal of creating a chain with 100 000 blocks, as in the paper. Due to the sleep time of the goroutines, this was going to take some time to complete. $0.2s * 100000/3600 \approx 5.55$ hours,

and that is if there is no failed block creations. Actual run time ended up being just shy of 11 hours.

To lessen the blow of the run time we have two branches on our GitHub project. One that uses a BASH script to run multiple in parallel. The second one uses go to start a bunch in parallel. When a chain is completed it runs through and calculate the rewards and prints this out to a file under the data folder.

Due to the simple chain, this was not resource intensive to run on our machines and leaving it on over night was not a hassle to do either. Like the paper we wanted to try 10 runs. During the operation of the software and unknown error caused 3 of them to crash leaving us with 7 runs of 100k blocks each. Each run has multiple parallel tests running. Each run has selfish miners with powers of $5\%, 10\%, 15\%...45\%$. The first set of runs used a static uncle block reward of $4/8$ of the static mining reward, the second set of runs used a dynamic uncle block reward explained in chapter 3.1. Giving total block count of about 12.6 million over all runs.

6 Simulation Result and Discussion

6.1 Results

As mentioned in 5.2, 2 sets of simulations was used in this paper. A run with a static uncle block reward of $4/8$ the static mining reward and a run with dynamic uncle block rewards.

6.1.1 Static uncle block reward

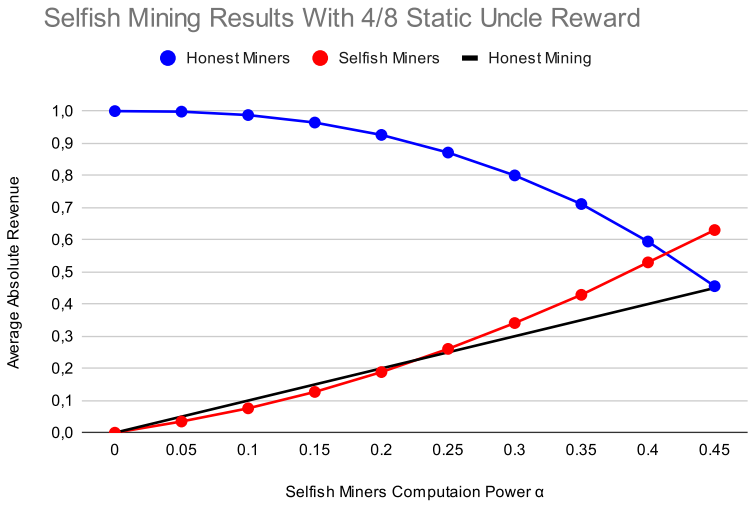

Figure 2: Revenue rate for the selfish and honest pool when Ku = 4/8Ks and α changes from 0 to 0.45

.

As seen in figure 2 when α is above around 0.23 the selfish pool gains a higher revenue from the selfish mining algorithm rather than following the Ethereum protocol. When the α value is increasing the selfish mining revenue seems to grow exponentially while the honest mining revenue seems to decrease exponentially. The total revenue split between the two pools also increases as the α value gets higher. This is because the selfish mining algorithm forces forks which causes it to produce additional uncle and nephew rewards for the whole system. The higher the α value, the more forks will be created in favour of the selfish miner.

The graph also shows that when α is below 0.23 the selfish mining pool doesn't loose a lot of revenue compared to selfish mining in Bitcoin. This is because of the additional uncle block rewards the selfish pool gains.

6.1.2 Dynamic uncle block reward

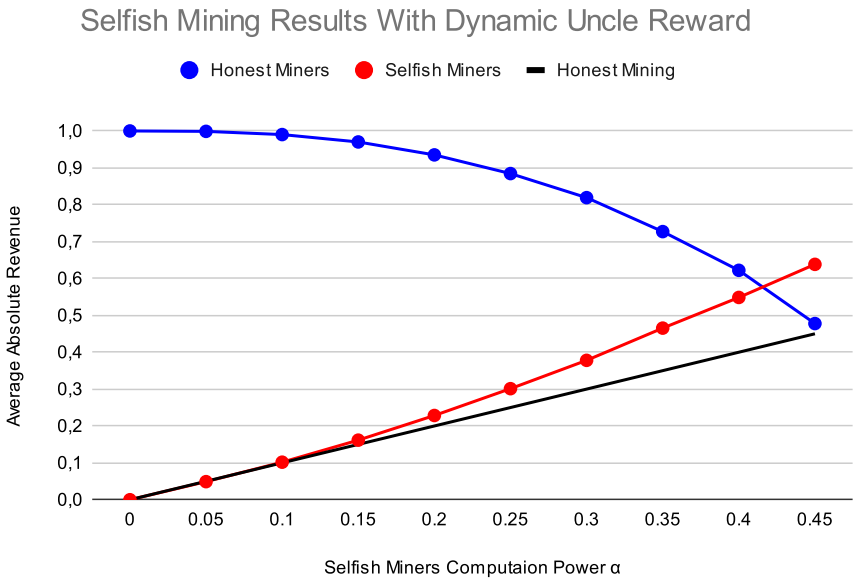

Figure 3: Revenue rate for the selfish and honest pool when Ku is dynamic and α changes from 0 to 0.45

.

For this set of runs the uncle block reward ranges from 2/8 to 7/8 of the static mining reward. As seen if figure 3 the selfish pool already gains a higher revenue from using the selfish mining algorithm compared to the Ethereum protocol at an α value of around 0.1. This is because when the hashing power of the selfish pool is low it will loose most of the forks almost immediately. When the fork is lost the regular block in the main chain will reference the selfish block with a low distance giving the selfish miner a high uncle block reward.

This also happens when the uncle block reward is static in chapter 6.1.1, but since the distance doesn't affect the uncle block reward the selfish pool needs a higher α value to gain the same

amount of revenue. However when the α value reaches around 0.43 the dynamic and static results are approximately equal to each other. This is because a lot of long forks will occur which causes the dynamic uncle block rewards to average at around 4/8 the static mining reward.

6.2 Validity of results

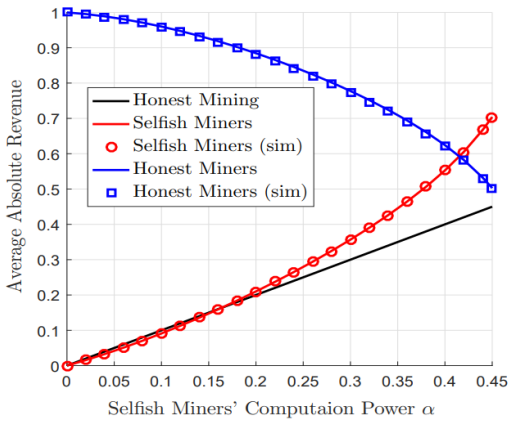

Figure 8. Revenue rate for the selfish pool and honest miners when $\gamma = 0.5$, $K_u = 4/8K_s$ and α changes from 0 to 0.45.

(a) Static Uncle Block Reward

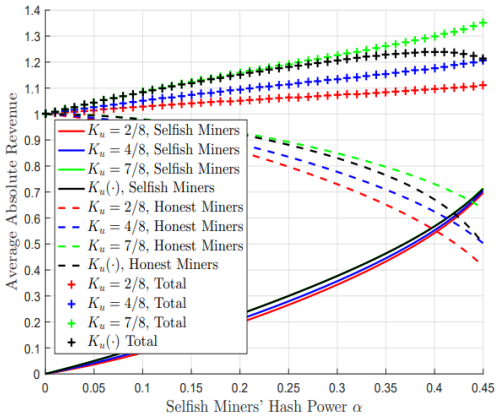

Figure 9. Revenue rate for the selfish pool, honest miners under different uncle block reward K_u.

(b) Black Line = Dynamic Uncle Block Reward

Figure 4: Results From Paper [1]

As seen from the results in the paper in figure 4 there are some deviations between the results from the paper and our results. These deviations can be seen in both the static and dynamic uncle block reward simulations.

For the static uncle reward simulation we concluded in our results that an α value of 0.23 was needed to make the selfish mining profitable while the paper conclude that the α value has to be 0.163 to be profitable. This is not the case in the dynamic uncle reward simulation where we and the authors of the paper conclude that an α value of around 0.1 is needed to make the selfish mining profitable. Another deviation is that in the papers static and dynamic uncle block reward simulations the total absolute revenue seems to be higher than in our simulations. Both of these deviations indicate that more uncle blocks has been generated in the simulations from the paper compared to our simulations.

The main reason for this is while the paper generates 1000 mining nodes divided into two pools we generate two mining nodes representing each pool. More of this is explained in chapter 6.3. Another reason could be that the paper uses Markov processes to generate the results, while we use a more practical solution to generate our results.

Other than that our results seems to closely match the results from the paper. The trend of the lines are very similar and the point of intersection between the selfish and honest miners average

absolute revenue seems to happen at the same α value in both the static and dynamic uncle block reward simulations.

6.3 Shortcomings

As mentioned throughout the sections of 5.1, we had difficulty with some parts of the code later on. This was mostly due to bad handling of fork logic. There are some unresolved issues, 10 runs where started but only 7 finished because 3 crashed due to an unknown reason. Sometimes on rare occasions a block will be added to the chain with the wrong parent, this problem has been narrowed down to bad handling of fork and uncle logic. Much of these problems we speculate would be fixed if we did as explained in section 5.1.3, having each party have their own chain.

Due to the complex and convoluted logic in forks and uncle block handling, we know there are some problems with uncle blocks in some edge cases, but it still works good enough for our needs. One goal that we could not test due to this harsh logic was to have multiple nodes. We think this has affected the results and slightly lowered the over all gain of the selfish miner. This is because only having two nodes, selfish and honest, limits the amount of forks that can be generated in parallel. When looking at our chain, it was very rare that there was more than one uncle block referenced in a regular block. Having more honest nodes could have led to more uncle blocks at the same depth, leading to the selfish miner being able to include more uncles in its blocks, giving a slightly higher reward for both pools.

Another part of our fork logic that suffers, is the part that decides which fork is the correct one. Our current implementation does not use the GHOST protocol as Ethereum does, instead we basically use the longest chain rule. We suspect this may have some impact, but it should not be a large impact. For instance we speculate that the lack of potential for more than one uncle block may have a bigger effect.

6.3.1 Tickrate

The idea of a tick rate was thought of early but for whatever reason we just forgot about it and continued without it. The thinking was that we could set it up such that 1 tick would equal some quantum of time. In this time frame the same amount of action is performed in each tick. Essentially, if 1 tick was 1 second and we wanted to generate about 1 block every 15 seconds, we need to generate 1 block every 15 ticks on average.

Doing it this way would fix the problem of having a minimum wait time of 200ms between attempts at blocks by instead waiting for a tick. We are confident that this would allow for higher speed of ticks compared to the minimum 200ms wait time. This, if correct, could have helped shorten the run time of the final simulation.

7 Conclusion

Our results seems to closely match the paper, leading us to the same conclusion that selfish mining in Ethereum is more profitable than in bitcoin mining. It is also more profitable with less mining power compared to in bitcoin.

References

[1] C. Feng and J. Niu, "Selfish mining in ethereum," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1306–1316, IEEE, 2019.