Sluggish Mining: Profiting from the Verifier's Dilemma

Beltrán Borja Fiz Pontiveros[1], Christof Ferreira Torres[1], and Radu State[1]

Center for Security, Reliability & Trust
University of Luxembourg
{beltran.fiz,christof.torres,radu.state}@uni.lu

Abstract. Miners in Ethereum need to make a choice when they receive a block: they can fully validate the block by executing every transaction in order to validate the new state, but this consumes precious time that could be used on mining the next block. Alternatively, miners could skip some of the verification stages and proceed with the mining, taking the risk of building on top of a potentially invalid block. This is referred to as the verifier's dilemma.
Although the gas limit imposed on Ethereum blocks mitigates this attack by forcing an upper bound on the time spent during verification, the slowdown that can be achieved within a block can still be enough to have an impact on profitability.
In this paper we present a mining strategy based around sluggish contracts; these computationally intensive contracts are purposely designed to have a slow execution time in the Ethereum Virtual Machine to provide an advantage over other miners by slowing their contract verification time.
We validate our proposed mining strategy by designing and evaluating a set of candidate sluggish smart contracts. Furthermore, we provide a detailed analysis that shows under which conditions our strategy becomes profitable alongside a series of suggestions to detect this type of strategy in the future.

Keywords: Ethereum · smart contracts · mining strategy · security · cryptocurrencies

1 Introduction

Ethereum is a blockchain protocol which keeps a record of state transition transactions and the state of the Ethereum Virtual Machine (EVM): a stack-based run-time environment designed for smart contracts, programs that are executed in a distributed and decentralised fashion across the Ethereum network [23]. This turned the Ethereum blockchain into a global decentralised computing platform. As of December 2018 its market capitalisation is evaluated at over $10 Billion, making it third in volume after Bitcoin and Ripple.

The EVM currently supports over 150 instructions, commonly referred to as opcodes. Each of these opcodes performs a different operation on the stack of the

EVM. Smart contracts are usually developed using a dedicated high-level programming language such as Solidity [19], which afterwards gets translated into a sequence of opcodes. Each opcode has an assigned gas cost, which represents the amount of resources required to use this operation. Although ideally each operation should take a similar amount of time per gas consumed, due to the need of having cheap, yet CPU intensive operations such as hash calculations, means some opcodes are considered to be under-priced.

At the time of writing, Ethereum achieves consensus through a proof-of-work algorithm similar to Bitcoin. However, one of the reasons why Ethereum became so popular among casual miners, was thanks to their choice of a memory-hard proof-of-work algorithm: Ethash [10]. The algorithm was designed to be ASIC-resistant and thus to only be run using commodity hardware such as CPUs or graphics cards. During block validation, miners are expected to execute each transaction to calculate the new state and validate the block. This however poses a problem: if they choose to validate a block upon receipt, they need to spend time running the transactions that could be used to create the next block; however if they do not validate the block and proceed mining on top of it, they have a risk of building on top of an incorrect block. This is known as the verifier's dilemma [14]. If however a miner were to gain an edge by being able to execute the validation faster than other miners, they would be able to begin work ahead of other miners and thus gain a competitive advantage. In this paper we present a mining strategy for the Ethereum network, built around this verification process of blocks by miners.

We propose a strategy that deploys a *sluggish* smart contract: a purposely designed slow execution contract for which we have a simplified, faster version available. We then create a new block with a transaction executing this sluggish contract in order to slow down other miners during block validation. The difference here is that we run the optimised version to create our block, while other miners will run the under-optimised version. This will gain us time and increase our chances to build the next block by having additional time over other miners in the network.The paper is structured as follows: In section II, we present a summary of some of the related work currently available. In Section III, we present the methodology of our proposed mining strategy, with a step-by-step description. In Section IV, we evaluate our mining strategy as a Markov Decision Process and extract the viability conditions. In Section V, we proceed to evaluate our strategy by extracting and evaluating past and current Ethereum settings and show that our strategy is viable under certain conditions. Finally in section VI, we present some conclusions of our work with some suggestions on potential future work.

2 Related Work

Since its release in 2015 [23], a lot of research has been produced around Ethereum, spanning across multiple rapidly developing areas, as shown in the work performed in [21].

The evaluation of different mining strategies and the game theory behind it is a very active domain within the Bitcoin research community [12]. Modelling the mining process as a Markov Decision Process was performed in [9], where the authors simulated a selfish mining strategy and assessed its efficiency with respect to the standard behaviour. Additional work continued along these lines, modelling additional strategies and assessing their viability [16].

The verifier's dilemma was first introduced in [14]; the authors suggest a 'divide and conquer' solution by dividing computationally intensive transactions into multiple smaller transactions, spread across multiple blocks. In [20] the authors proposed a system named TrueBit. It provides an alternative to Ethereum's need to replicate calculations in every node by reducing it to a small set of entities. This would prevent computationally intensive contracts to slow down the network. In [7] the authors suggest moving some of the work off-chain while providing guarantees under a threat model that includes selfish nodes.

In addition to computationally intensive transactions that may cause denial-of-service attacks (DoS), there are also numerous vulnerabilities that are caused due to developers programming bad code. These vulnerabilities range from smart contract specific ones such as reentrancy and transaction order dependence (TOD) [13] to classic vulnerabilities such as integer overflows [22]. Finally, several attacks on the peer-to-peer network of Bitcoin and Ethereum have been proposed in order to reduce the computational power required to reach the majority vote in the consensus algorithm [11].

3 Methodology

In this section, we describe how our mining strategy works, the preparation steps required to deploy it and finally an evaluation of when this strategy becomes more profitable than the default mining strategy.

3.1 Sluggish Mining Strategy

In Ethereum the final block reward obtained by miners can be divided into three sub rewards:

- Block creation reward (R_{blck}): A static reward initially set to 5 Ether, which has been lowered in October 2017 [1] to 3 Ether and is expected to be lowered once again to 2 Ether with the release of [17].
- Gas reward (R_{gas}): This includes the transaction fees, consisting of all the gas that has been spent during execution of the transactions in the block. This is to outset the time and effort spent on validating the block, as defined in the consensus protocol. The overall reward obtained depends on the gas price offered by the transactions, and the transaction load of the network. Like in the Bitcoin protocol, it is expected to have this reward eventually replace the block creation reward entirely.

– Uncle reward (R_{uncle}): If any uncles (stale blocks) are included in the created block, providing an additional reward of $\frac{1}{32}$ of the current block creation reward. A maximum of 2 uncles are allowed per block.

The main idea behind the sluggish mining strategy is to forfeit the gas reward obtained from executing and including other users transactions (R_{gas}), and instead create a transaction designed to use all the gas available in a block and send it to a smart contract purposefully designed to be as slow as possible to execute. In addition to being slow, the state change caused to the Ethereum blockchain must be predictable or basically unchanged, such that the attacker does not require to run the contract himself. In order to ensure we do not waste time validating our transaction, we simply replace the pointer in our database from our sluggish smart contract (SSC) to an identical optimised smart contract (OSC) with an identical state transition function while being optimised for a negligible execution time. This can be seen in Figure 1.

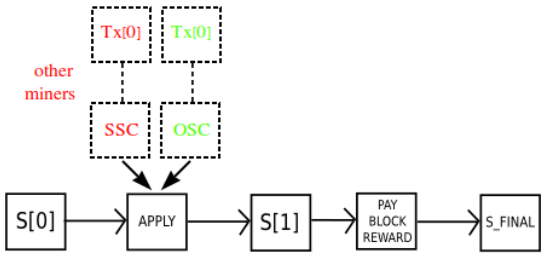


Fig. 1. Using OSC allows for a faster state transition than using SSC.

The purpose of this sluggish contract is to gain an advantage over other miners by having them execute this slow contract during the block validation phase. This will allow the attacker to gain an advantage in terms of time as compared to other miners on the next block, at the expense of the fee rewards not taken to accommodate for our transaction. This strategy makes the assumption that other nodes validate a received block before starting work on the next block although it is possible that miners simply skip the validation stage and proceed to the block creation stage of the next block without wasting time on our contract. As discussed before, this is part of the verifier's dilemma.

The following assumptions are made in our mining strategy:

– Executions times are similar between different architectures. With the rise of specialised hardware such as FPGAs and ASICs purposely built for specific tasks, this condition might not hold in the future.
– We assume that miners will validate a block before beginning the mining process on top of it. However, in practice the validation could be simplified by verifying soley the previous hash without executing the transactions.

This would mean that no slow down effect would be caused by our sluggish contract.
- We treat miners as purely rational agents wanting to maximise their revenue given the current mining conditions. We do not investigate on any effects such a mining strategy could have on the value of Ethereum. This could be seen as an attack (and thus trust in the system lowered).
- We do not take into consideration the network delays in block propagation. However, it should be noted that our strategy could yield blocks with only one single transaction, hence producing small size blocks and therefore susceptible for faster propagation.

The steps required to deploy this strategy are as follows:

1. The design of a sluggish smart contract i.e. a contract that shows a dreadful execution time.
2. The deployment of the sluggish smart contract and modification to the client in order to avoid or replace the execution of our sluggish smart contract.
3. The mining and broadcast of a block which includes a transaction executing our sluggish smart contract.

In the following subsections we will explain the design and deployment of sluggish smart contracts in greater detail.

Sluggish Contract Design. The first step is to design a sluggish smart contract. The purpose of this contract is to maximise the time taken to execute, given a fixed amount of gas. Although the Ethereum community goes to great lengths to ensure that the gas cost of each opcode reflects its computational time expense, there are always exceptions or opcodes that fail to reflect their real cost, leading to attacks such as the one that took place in mid 2017, leading to a hard fork [3], which caused the following opcodes to change their gasprice: $EXTCODESIZE$ and $EXTCODECOPY$ from 20 to 700, $BALANCE$ from 20 to 400, $SLOAD$ from 50 to 200, $CALL$, $DELEGATECALL$, $CALLCODE$ from 40 to 700 and finally $SEFLDESTRUCT$ from 0 to 5000. Despite these readjustments to the opcode pricing we show in this work that our strategy remains viable. In summary, we want our sluggish contract to satisfy the following criteria:

1. We need to be able to precompute the state change that our contract has on the EVM world state, without actually executing it.
2. The contract needs to be as slow as possible given the maximum amount of gas that can be consumed inside a block, known as the *gaslimit*. This limit is currently set to about 8 million.

Algorithm 1 provides a very simple and generic design, where *OPCODE* may be replaced by any particular opcode that has a major impact on the execution time. The algorithm takes as input an opcode and the opcode's input

Algorithm 1 Sluggish Smart Contract

Input: (OPCODE, INPUT)
Output: 0
1: Push INPUT onto the stack
2: Execute OPCODE
3: **if** gas left > 0 **then**
4: Jump to line 1
5: **end if**
6: **return** 0

values. As described in Section 2, there have already been numerous studies and benchmarks regarding the under-pricing of several EVM opcodes [5], [6] and based on the literature we selected the following opcodes in order to affect either the CPU or the access to I/O:

1. **CPU Bound:** The goal of CPU bound sluggish contracts is to ensure that the CPU remains busy as possible during the contract execution. Two opcodes were selected:
   - *SHA3* (0x20): This opcode computes the Keccak-256 hash [23] and has a gas cost of 30 + 6 * (size of input in words). It pops two values from the stack, the memory offset p and the size n, and finally pushes the result onto the stack.
   - *EXP* (0x0a): This opcode computes the exponential on two values popped from the stack, the base b and the exponent e. If e is 0 then the gas used is 0. If e is greater than 0, then the gas used is 10 + 50 * a factor related to the size of the log of the exponent. It should be noted that the factor gas cost of EXP has already been increased from 10 to 50 [2].
2. **I/O Bound**: As described in the previous section, operations dealing with peristent storage have always been tricky to price. We test two of the most common opcodes:
   - *SLOAD* (0x54): This opcode loads a value from storage. Storage in Ethereum is implemented as a key-value store. The opcode takes as input the storage location (key) and returns the value associated to this location. It has a cost of 200 gas.
   - *SSTORE* (0x55): This opcode stores a value into storage. The gas cost is 20,000, if the storage value is set from a zero value to a non-zero value. Otherwise, the gas cost amounts to 5,000.

The sluggish smart contracts used during our experiments were formed using the layout defined in Algorithm 1 and the opcodes listed above. Given that some EVM client implementations have optimisations for simple input values such as 0 and 1, we decided to use input values of 15 instead (0xe); these contracts can be seen in Figures 2-5, alongside their corresponding sequence of opcodes and bytecode. The evaluation of the total delay obtained can be seen in Section 5.

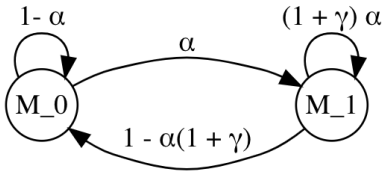
SHA3	**EXP**	**SLOAD**	**SSTORE**
JUMPDEST 1 gas	JUMPDEST 1 gas	PUSH 0 3 gas	JUMPDEST 1 gas
PUSH 0 3 gas	PUSH 0 3 gas	JUMPDEST 1 gas	PUSH 1 3 gas
PUSH 0 3 gas	PUSH 0 3 gas	SLOAD 200 gas	PUSH 0 3 gas
SHA3 30 gas	EXP 60 gas	PUSH 222 3 gas	SSTORE 5000 gas
POP 2 gas	POP 2 gas	GAS 2 gas	PUSH 20028 3 gas
PUSH 60 3 gas	PUSH 60 3 gas	GT 3 gas	GAS 2 gas
GAS 2 gas	GAS 2 gas	PUSH 2 3 gas	GT 3 gas
GT 3 gas	GT 3 gas	JUMPI 10 gas	PUSH 0 3 gas
PUSH 0 3 gas	PUSH 0 3 gas	STOP 0 gas	JUMPI 10 gas
JUMPI 10 gas	JUMPI 10 gas		STOP 0 gas
STOP 0 gas	STOP 0 gas		
Bytecode:	**Bytecode:**	**Bytecode:**	**Bytecode:**
5b6000600020506	5b600060000a506	60005b5460de	5b6001600055614
03c5a1160005700	03c5a1160005700	5a1160025700	e3c5a1160005700

Fig. 2. SHA3 (CPU) **Fig. 3.** EXP (CPU) **Fig. 4.** SLOAD (I/O) **Fig. 5.** SSTORE (I/O)

Contract Deployment. Smart contracts are deployed on the blockchain via transactions. A transaction has a base fee of 21,000 gas. The cost of deploying our sluggish smart contract has a minimum fee of 32,000 gas for the *CREATE* opcode. This is meant to cover the cost of performing an elliptic curve operation to recover the sender address from the signature in addition to the cost of the disk and bandwidth space of storing the transaction. In addition to the contract creation, we also must pay another 200 gas per byte of the contract's bytecode [23]. The largest contract in our test set had a size of 14 bytes, meaning that the cost of deploying our largest contract would haven been in total 54,800 gas. Considering an average gas limit of 8 million per blocks, the deployment costs solely represent 0.68% of the total amount of gas that can be used by a block.

4 Mining Strategy Evaluation

In order to evaluate our proposed mining strategy, we use a Markov Decision Process to model the mining process using a similar notation to the models proposed in [9] and [15]. As in the other models, we do not take the block propagation time into consideration. Figure 6 shows that once we have created a block (state M_1), we gain a slight advantage of $\gamma \cdot \alpha$ over the other miners. This is due to the slow execution time of the sluggish smart contract that get executed by the other miners during validation.



M_0: State when failed to create previous block.
M_1: State when created previous block.
α: Hashing power ratio of the miner.
γ: Ratio of the delay with respect to the block time.

Fig. 6. Markov Decision Process for Sluggish Mining

We can now calculate the steady-state distribution of our Markov Process to determine the overall gain in computational power obtained in the long run by our strategy. The steady-state distribution can be calculated using:

$$\pi \cdot \mathbf{P} = \pi \tag{1}$$

And substituting the values from our model we have that:

$$\begin{bmatrix} \pi_0 \ \pi_1 \end{bmatrix} \begin{bmatrix} 1-\alpha & \alpha \\ 1-\alpha\cdot(\gamma+1) & \alpha\cdot(\gamma+1) \end{bmatrix} = \begin{bmatrix} \pi_0 \ \pi_1 \end{bmatrix} \tag{2}$$

$$\begin{cases} \pi_0 = \pi_0 \cdot (1-\alpha) + \pi_1(1-\alpha\cdot(\gamma+1)) \\ \pi_1 = \pi_0 \cdot \alpha + \pi_1 \cdot \alpha \cdot (\gamma+1) \\ \pi_0 + \pi_1 = 1 \end{cases} \tag{3}$$

$$\pi = \begin{bmatrix} 1 - \frac{\alpha}{1-\alpha\cdot\gamma} & \frac{\alpha}{1-\alpha\cdot\gamma} \end{bmatrix} \tag{4}$$

As a result of our sluggish contract, our overall ratio of the total hashing power of the system has increased from α to $\frac{\alpha}{1-\alpha\cdot\gamma}$. This gain in hashing power however comes at the cost of forfeiting the reward obtained through transaction fees. Every time a block is successfully mined, a transition into state M_1 occurs. In order to determine when our sluggish mining strategy is more lucrative than honest mining, we solve the inequality shown below:

$$\frac{\alpha}{1-\alpha\gamma} \cdot \frac{R_{blck}}{t_{blck}} \ \geq \ \frac{\alpha}{t_{blck}} \cdot (R_{blck} + R_{fees}) \tag{5}$$

Where R_{blck} is the block reward, R_{fees} is the transaction fees (average) and $\gamma = \frac{timedelay}{blocktime}$. Once the inequality is solved we can isolate the variables such that we can determine the requirements needed for our strategy to become profitable:

$$transactionfees \leq \frac{R_b \cdot \alpha \cdot \gamma}{1-(\alpha\cdot\gamma)} \tag{6}$$

So given a miner with 15% of the total hashing power, a sluggish contract execution of about 3 seconds ($\gamma \approx 0.2$) and using the current static block reward of 3 Ether, we find that if the average transaction reward fees obtained from a block are below 0.0963 Ether, our strategy becomes viable.

5 Experimental Results

In this section we describe the experiments that we performed and the evaluation of the results. The experiments were conducted using a MacBook Pro (13-inch, 2016) with a 2.9 GHz Intel Core i5 Processor and 8 GB 2133 MHz LPDDR3 of Memory. Although the results might vary slightly between different machines, as mentioned in Section 3, we expect similar slowdowns on other machines as well. The average block time in Ethereum is 14.5 seconds.

5.1 Ethereum Clients

In order to ensure that our slow down is as effective as possible, we executed our contracts using the EVM implementation of the two most popular Ethereum clients at the time of writing: *Geth* and *Parity*. In particular we tested our contracts on the following two client versions: `geth v1.8.17` and `parity v2.0.9`.

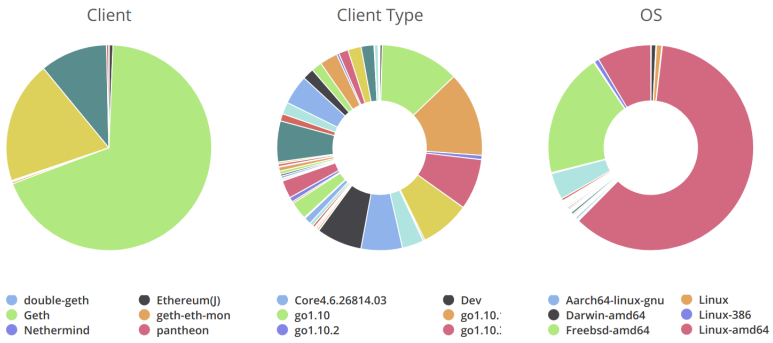


Fig. 7. Distribution of Ethereum clients, versions and OS.

While there are other clients and versions in use, as can be seen in Figure 7, without selection we cover over 88% of the clients based on the data collected in [8].

5.2 Transaction Fees

The average transaction fees per block can vary quite drastically from day to day based on the current transaction backlog in the system, as can be seen in

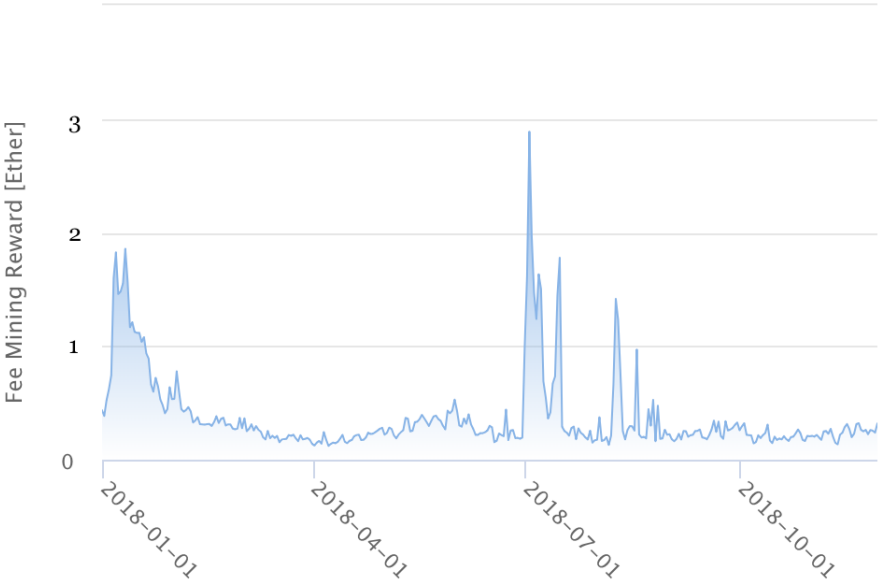


Fig. 8. Average transaction fees based on Etherscan [8]

Figure 8. For the purposes of our work, we calculated the average transaction fee for the blocks published during the month of January and November:

- January 2018: $\mu = 0.15$ *Ether/block*
- November 2018: $\mu = 0.08$ *Ether/block*

5.3 Block Reward

Initially the Ethereum block reward was 5 Ether. With the release of the Metropolis hard fork [1] this reward was lowered to 3 Ether, where it currently stands. This value is expected to be further reduced to 2 Ether with the release of the planned Constantinople hard fork in 2019 [18]. Although the trend seems to be to have mining rewards be lowered, there are some EIP proposals advocating a raise in incentives for miners, such as the EIP1227 which seeks to restore the reward to 5 Ether. Therefore for the purposes of our tests we will take these three possible block reward values into consideration: 2, 3 and 5 Ether.

5.4 Sluggish Contract Execution Times

In order to calculate the slow down caused by our contracts we used the EVM stand-alone implementations: `evm` available in Geth and `parity-evm` available in Parity. This allows us to test and measure the execution time of our contracts without the need to deploy them on a test network. The commands used in our tests are:

```
$ time evm --gas 8000000 --code 5b60ff60ff2050603c5a1160005700 run
$ time evm --gas 8000000 --code 5b60ff60ff0a50603c5a1160005700 run
$ time evm --gas 8000000 --code 60005b5460de5a1160025700 run
$ time evm --gas 8000000 --code 5b6001600055614e3c5a1160005700 run

$ time parity-evm --gas 8000000 --code 5b60ff60ff2050603c5a1160005700
$ time parity-evm --gas 8000000 --code 5b60fe60fe0a50603c5a1160005700
$ time parity-evm --gas 8000000 --code 60005b5460de5a1160025700
$ time parity-evm --gas 8000000 --code 5b6001600055614e3c5a1160005700
```

Listing 1.1. CMDs used for experiments

We ran each command 100 times and then averaged the results, obtaining the following average slow downs in seconds:

	Geth	**Parity**
SHA3	0.15s	2.52s
EXP	1.21s	3.42s
SLOAD	0.03s	1.13s
SSTORE	0.02s	0.29s

The slowest contract, given a maximum of 8 million gas to run, appears to be the contract based on the EXP opcode, by quite a large margin for both Geth and Parity. This seems to suggest that the opcode is under-priced, as already explored in [4]. The difference between the execution times in Geth and Parity might indicate some issue in the implementation of the exponentiation function in Parity. An inspection of their code shows that both clients include shortcuts for specific values of the base and exponent ($x^0 = 1$, for example). Interestingly enough, both appear to implement the same technique, namely exponentiation by squaring, by using a left to right binary representation.

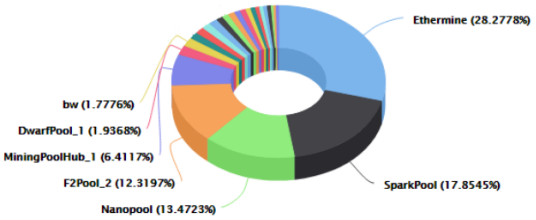
5.5 Evaluation

We would need to know the distribution of computational power of every single Ethereum client, in order to determine the overall slow down on the entire Ethereum network caused by our sluggish smart contract execution. Since we lack this information, we can only provide the lower and upper bounds of our attack. In the best case scenario for our strategy, all our nodes will be using Parity, while in the worst case, all will be using Geth. This can be seen in Figure 10 and Figure 11, by the vertical delimiters.

If we use the number of blocks mined to estimate the total hashing power of a mining pool and their names as given in Etherscan [8], we can extract the values shown in Figure 9 for the first week of December 2018.

Given the Ethereum settings described above, we can now define the conditions under which our strategy becomes viable using the equations shown in Section 4.

With an average transaction fee of 0.08 Ether we find that our strategy becomes viable under the conditions shown in Figure 10. Each of the vertical lines represent the boundaries for our strategy to become viable (as described



Rank	Name	Blocks
1	Ethermine	28.2778%
2	SparkPool	17.8545%
3	Nanopool	13.4723%
4	F2Pool	12.3197%
5	MiningPoolHub	6.4117%

Fig. 9. Mining pools hashing power distribution [8] as of December 2018.

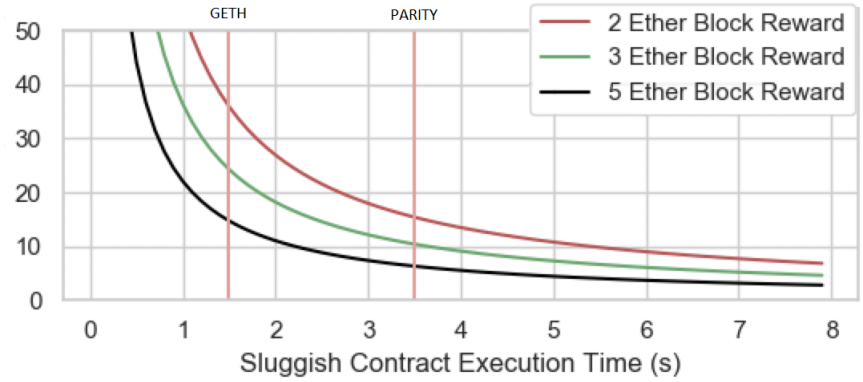


Fig. 10. Sluggish Mining inequality for a transaction fee of 0.08 *Ether/block*.

before, depending on the composition of mining clients), and any value above the block reward curve. This means that with a block reward of 3 Ether, any computational power above 10% and 24% would benefit from a switch to sluggish mining. These margins rise to 15%-35% for a block reward of 2 Ether, and become as low as 8% to 15% in the case of a 5 Ether block reward. Given the data shown in Figure 9, this suggests that the top 4 mining pools could currently benefit from our approach.

However, if we assume a higher average transaction fee of 0.15 Ether, as seen in Figure 11, then the strategy only becomes viable for mining pools with a hashing power between 20% and 45%, and hence only becoming viable for the Ethermine mining pool.

Moreover, once Constantinople is released and assuming the block reward does indeed drop, it would further reduce the viability of our strategy for mining pools with a hashing power within the range of 15% to 35%. However, it serves to highlight how a change in incentives for miners at the reward level can have such a drastic effect on their optimal strategy.

Although this strategy could be interpreted as a type of resource exhaustion attack, it is not significant enough to become apparent by miners. In order

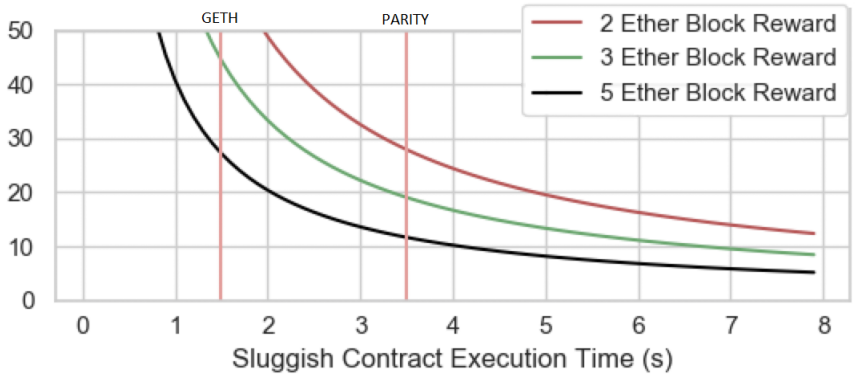


Fig. 11. Sluggish Mining inequality for a transaction fee of 0.15 *Ether/block*.

to ensure this strategy is not implemented to the potential detriment of the Ethereum community the best way is to ensure that the economic incentive is never there. We propose four ways to ensure this:

1. Reduce the block reward and therefore force mining pools to include transactions if they wish to continue make profit.
2. Increase the transaction fees by including a fee that compensates the mining pool.
3. A safeguard for this strategy would be to simply add these conditions to the benchmarking process of EVM opcodes, to ensure it never becomes a viable mining strategy for any mining pool by adequately pricing their opcodes.
4. Shift to another consensus mechanism such as proof-of-stake (PoS), in which having a delay in the execution time does not have such a tremendous impact as in PoW based mining. This is because there is no arms race with regard to who gets to mine the next block.

A scan of all transactions for each block would have to be done in order to determine if this type of mining strategy is already being used by mining pools. If at every block there is a transaction sent to a contract which fills up the remainder of that blocks remaining gas (and this same address is used across multiple blocks) then this could be an indicator for the deployment of our strategy.

6 Conclusions

In this work we have shown that sluggish smart contracts that aim at gaining an advantage over other miners who are validating received blocks before starting to create a new block, is a viable mining strategy provided certain conditions are met. We evaluated our strategy given current Ethereum conditions and shown that it would be a viable strategy for the top mining pools. Given that no

user transactions are added to these 'sluggish blocks', the overall usefulness of the Ethereum network lowers, meaning that it might not be in a miners best interest to implement this strategy.

We have also shown that given the expected Ethereum changes in the pipeline, this type of strategy will be less useful in the future. In addition, we also provided a series of mechanisms for detecting this behaviour and how to ensure that this strategy does not become a profitable strategy in the future.

In this work we have shown that our mining strategy is viable by creating a block that is composed of one transaction sent to the sluggish contract. However a mining pool could instead include a series of normal pending transactions as the standard behaviour and make use of the remaining block gas by injecting a transaction that invokes a sluggish contract.

In future work it would be interesting to see if this type of strategy could be applied with different consensus mechanisms. For example the delay caused by computationally intensive contracts could be used in a proof-of-stake mechanism as an attack: given that miners have a certain time frame to create and broadcast a block, a long enough delay caused by the validation of the previous block could cause them to miss the window and therefore effectively to lose money.

References

1. Afri Schoedon (@5chdn), V.B.v.: Metropolis Difficulty Bomb Delay and Block Reward Reduction. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-649.md (2017), [Online; accessed December 2018]
2. Buterin, V.: EXP cost increase. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-160.md (2016), [Online; accessed November 2018]
3. Buterin, V.: Ethereum Improvement Proposal: Gas cost changes for IO-heavy operations . https://github.com/ethereum/EIPs/blob/master/EIPS/eip-150.md (2017), [Online; accessed December 2018]
4. Buterin, V.: Blockchain resource pricing. . (2018)
5. Chen, T., Li, X., Luo, X., Zhang, X.: Under-optimized smart contracts devour your money. In: Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on. pp. 442–446. IEEE (2017)
6. Chen, T., Li, X., Wang, Y., Chen, J., Li, Z., Luo, X., Au, M.H., Zhang, X.: An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. In: International Conference on Information Security Practice and Experience. pp. 3–24. Springer (2017)
7. Das, S., Ribeiro, V.J., Anand, A.: Yoda: enabling computationally intensive contracts on blockchains with byzantine and selfish nodes. arXiv preprint arXiv:1811.03265 (2018)
8. Etherscan: Etherscan, the Ethereum Block Explorer. https://etherscan.io/ (2018), [Online; accessed December 2018]
9. Eyal, I., Sirer, E.G.: Majority is not enough: Bitcoin mining is vulnerable. Communications of the ACM **61**(7), 95–102 (2018)
10. Foundation, E.: Ethash (dec 2018), https://github.com/ethereum/wiki/wiki/Ethash
11. Heilman, E., Kendler, A., Zohar, A., Goldberg, S.: Eclipse attacks on bitcoin's peer-to-peer network. In: USENIX Security Symposium. pp. 129–144 (2015)

12. Kiayias, A., Koutsoupias, E., Kyropoulou, M., Tselekounis, Y.: Blockchain mining games. In: Proceedings of the 2016 ACM Conference on Economics and Computation. pp. 365–382. ACM (2016)
13. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269. ACM (2016)
14. Luu, L., Teutsch, J., Kulkarni, R., Saxena, P.: Demystifying incentives in the consensus computer. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 706–719. ACM (2015)
15. Nayak, K., Kumar, S., Miller, A., Shi, E.: Stubborn mining: Generalizing selfish mining and combining with an eclipse attack. In: Security and Privacy (EuroS&P), 2016 IEEE European Symposium on. pp. 305–320. IEEE (2016)
16. Sapirshtein, A., Sompolinsky, Y., Zohar, A.: Optimal selfish mining strategies in bitcoin. In: International Conference on Financial Cryptography and Data Security. pp. 515–532. Springer (2016)
17. Savers, N.: Hardfork Meta: Constantinople . https://eips.ethereum.org/EIPS/eip-1013 (2018), [Online; accessed December 2018]
18. Schoedon, A.: Constantinople Difficulty Bomb Delay and Block Reward Adjustment. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1234.md (2017), [Online; accessed December 2018]
19. Solidity: Solidity 0.5.1 documentation (dec 2018), https://solidity.readthedocs.io/en/v0.5.1/
20. Teutsch, J., Reitwießner, C.: A scalable verification solution for blockchains. . (2017)
21. Tikhomirov, S.: Ethereum: state of knowledge and research perspectives. In: International Symposium on Foundations and Practice of Security. pp. 206–221. Springer (2017)
22. Torres, C.F., Schütte, J., State, R.: Osiris: Hunting for integer bugs in ethereum smart contracts. In: Proceedings of the 34th Annual Computer Security Applications Conference. pp. 664–676. ACSAC '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3274694.3274737, http://doi.acm.org/10.1145/3274694.3274737
23. Wood, G.: Ethereum yellow paper (2014)