Performance Benchmarking of Smart Contracts to Assess Miner Incentives in Ethereum

Workshop on Blockchain Dependability

Amjad Aldweesh, Maher Alharby, Ellis Solaiman, Aad van Moorsel
School of Computing, Newcastle University, Newcastle upon Tyne, UK
Email: {a.y.a.aldweesh2, m.w.r.alharby2, ellis.solaiman, aad.vanmoorsel}@ncl.ac.uk

***Abstract*—A defining feature of the Ethereum blockchain is its ability to execute smart contracts, providing a Turing complete programming model for distributed applications in non-trusted environments. The successful operation of the Ethereum blockchain depends on whether the miners' incentives (in the form of fees) to execute contracts is proportional to the miners' cost (in terms of energy usage, and thus CPU usage). In general, if the received fee is not proportional to the computational cost, miners would prefer some tasks over others, thus potentially adversely affecting the continuing dependable operation of the blockchain. In this paper we design a benchmark to compare smart contract execution time with the award a miner would receive, to determine if incentives align. We present the design of the benchmarking approach and provide initial results for the Python Ethereum client running on a Mac. The results indicate that for functions in Ethereum's most popular contracts the difference of reward per CPU second can be up to a factor of almost 50. In addition, contract creation, which is done once for each new contract, can be up to 6 times more lucrative than the regular execution of contract functions. Potentially, these discrepancies result in misaligned incentives that impact the dependable operation of the blockchain.**

***Index Terms*—Blockchain, Smart Contracts, Ethereum, benchmarking.**

I. Introduction

Permissionless blockchains, such as Bitcoin [1] and Ethereum [2], rely on miners for their successful operation. Miners invest computational resources and the energy to run them, and in return receive a fee, expressed in the unit of crypto-currency belonging with the blockchain. By establishing the right incentives, the miners are sufficiently motivated to keep operating the blockchain in a dependable manner. Recent literature investigates the alignment of incentives, for instance in relation to the long-term development of block and transaction rewards [7], in the context of miner pool strategies [9] and associated with denial of service attacks [8].

In this paper we consider incentives for miners when executing smart contracts in Ethereum. Ethereum contains a crypto-currency, called Ether, but the defining feature of Ethereum is the ability to execute smart contracts, providing a Turing complete programming environment for distributed applications. Smart contracts are executed by the Ethereum Virtual Machines (EVM) of miners, who earn Ether depending on the executed machine language operations (called opcodes). In particular, when executing a contract, the EVM keeps track of the amount of 'used gas', based on the amounts of gas specified per opcode by the Ethereum foundation [4]. To determine the fee, the used gas is multiplied by the 'gas price' offered by the submitter of a transaction.

The fee structure for contract execution is important for a number of reasons, including assuring termination of execution and avoiding denial of service attacks. We discuss this in more detail in Section II-C. In this paper, we are particularly interested in a less commonly researched reason why the fee structure is important, namely that the fee structure determines the incentives required to assure the dependable long-term operation of the blockchain. Specifically, if there is no clear relationship between the computational effort needed and the fee awarded, miners cannot rely on a reasonable award for their energy investment. In [6] we demonstrated that under such uncertainty miners cannot optimize their profits. Moreover, if certain smart contracts are known not to be attractive, transactions using that smart contract would not be executed by miners. Therefore, to assure the dependable operation of blockchains with smart contracts, miners and submitters of transactions need to be confident that the fee structure correctly incentivizes miners.

The aim of this paper is to describe a benchmark for Ethereum smart contracts that evaluates whether the fee awarded for the execution of smart contracts is proportional to the computational effort required. We envisage that such a benchmark could be run periodically, on a variety of software and hardware platforms, to demonstrate to the community if and how well costs and benefits are aligned within Ethereum. Users as well as miners would adjust their confidence in the dependable operation of Ethereum based on such a benchmark. We note that the performance of smart contracts execution becomes even more important once Proof of Work will be abandoned by Ethereum [15]. Currently, the hashing for Proof of Work dominates the computational effort of miners, but in the future effort to compute smart contracts will be increasingly important, especially if applications rely on increasingly complex contracts such as in [11], [12].

In the design of the smart contract benchmark there are a number of issues we consider. The granularity of measurement can be of two types, either based on the overall contract or based on individual functions, and we propose that a benchmark should do both. (Note that this is different from benchmarking opcode execution time, as proposed in [10], which is meant to assess the correctness of the gas per opcode in [4].) Within current implementations of EVM, contracts are also executed once before they are added to the blockchain. Since the fee as well as the functions executed differs the benchmark should measure both such contract creation and normal contract execution. In terms of the contracts to be considered in a benchmark, we propose to use the most commonly used ones on current-day Ethereum and we describe a procedure to obtain these. We use CPU usage as our main measure, as a first approximation of energy consumption and thus cost to miners.

Our experiments in this paper concern a single hardware platform and a single EVM, the Python Ethereum virtual machine (PyEthApp) [5]. Although preliminary in terms of coverage of software and hardware platforms, the results are quite striking. We conclude that used gas is not always well aligned with the computational effort for either creating or executing smart contracts. There is a factor of almost 50 difference in gas reward per CPU second between varying contract functions. In addition, the amount of used gas per CPU consumption for creating contracts is significantly more than that for contract execution (on average 6 times more).

The structure of the paper is as follows. In section II we introduce essential background for Ethereum concepts such as smart contracts and EVM. SectionIII describes in detail the process of collecting the data needed for the experiment. The experiment design and procedure are presented in sectionIV. SectionV shows our benchmarking approach results and observations. Finally, sectionVI concludes the paper.

II. Background

Ethereum is perhaps the most commonly used platform that supports creating and executing smart contracts. In Ethereum, the currency is known as ether (ETH). The execution fee for every operation made on Ethereum is known as Gas. The price of Gas is expressed in Ether and is used to reward miners for donating computing resources. Smart contracts can be developed in various programming languages such as Solidity (the most common), Serpent and LLL. Regardless of which language is used the smart contract source code will be compiled into bytecode that can be interpreted by the Ethereum Virtual Machine (EVM).

Ethereum has two types of users, also known as accounts: *externally owned accounts* and *contract accounts*. Each account has a balance and if it is a *contract account* it also has associated code and storage. The associated code is stored on the blockchain in a bytecode form and interpreted by the EVM. To execute a contract, a transaction, which has different fields such as *Receiver, Data* and *GasPrice*, is signed and sent to the blockchain. Miners, who are responsible for maintaining the blockchain, collect these transactions and execute the code using their EVM. The execution, if successful, results in a change in the state of the smart contact variables, storage and balance, otherwise nothing changes.

Ethereum has two types of transactions: *Financial* and *Contract-creation/Execution*. In a *Financial transaction* the amount of currency transferred is given in the *Value* field. The *Data* field in the *Financial transaction* is optional and can contain an arbitrary data to be stored permanently in the blockchain. Including the *Data* field in the transaction, costs fees that are proportional to the size of the data. The *Contract-creation/Execution Transaction* is either to deploy a new smart contract to the blockchain (Contract-creation) or to trigger an existing contract (Contract-execution). The *GasLimit* field is the maximum amount of gas that a user can pay for the transaction. Each machine instruction (or opcode) consumes an amount of gas paid to the miners (see section II-C).

A. Smart contracts overview

A smart contract is a computer program that is stored and executed on a blockchain such as Ethereum, with execution correctness enforced by the consensus protocol. It can contain any sets of agreements represented in any of its high-level languages. Smart contracts can be used to implement a variety of applications, including but certainly not limited to financial applications (e.g. saving wallets, wills). For example, a set of smart contracts for cloud services [12]. At the time of writing this paper, the number of smart contracts that have been deployed in the Ethereum network is growing to about 28,000 smart contracts.

Smart contracts live on the blockchain as bytecode and can be triggered by its identifier address (160-bit). Smart contracts are deployed to the blockchain by submitting a *Contract-creation* Transaction. Once this transaction is accepted and included in the blockchain, any Ethereum account can invoke this contract and its functions. Invocation of the code is done by sending transactions to the address of the contract. More precisely, when a new transaction with a contract address as recipient is accepted by the blockchain, then the smart contract is executed by all miners, with the blockchain current state and the transaction payloads as input. The results of transaction executions are permanently recorded in the blockchain.

B. Ethereum EVM overview

EVM is a stack-based virtual machine where each operational code is pushed/popped onto the stack and executed. Each EVM operational code (opcodes) consumes gas and the EVM keeps a tally of the amount of gas used. For instance, addition (ADD) opcode consumes 3 units of gas and division (DIV) consumes 5 units of gas [4]. Gas costs of EVM opcodes are predefined and cannot be changed by anyone but the Ethereum foundation. The gas costs of EVM opcodes are set according to three factors: *Storage*, *Network* and *Computational effort*. The first two factors are based on formulas (e.g., exponential (EXP) opcode costs 10 units of gas if the exponential is 0 otherwise 10 plus 10 multiply by the exponential size in byte). The third factor is determined according to estimated computational cost in terms of CPU usage.

C. Ethereum gas mechanism

As mentioned earlier, there are two types of transactions: *Financial* and *Contract-creation/Execution*. *Financial* transactions, which are used to transfer Ether between accounts, cost

21000 units of gas. On the other hand, the cost of *Contract-creation/Execution* transactions depends on the used gas as determined by the EVM, plus the transaction cost of 21000 units of gas. The EVM takes into account the *Operations performed by the bytecode* and *Data stored on the blockchain*. The former is based on the operational codes executed by the transaction where each opcode has associated a predefined cost. The latter charges 20000 units of gas for setting storage from zero to non-zero, otherwise 5000 units of gas.

Transactions in Ethereum also include *GasPrice* and *GasLimit* [2]. Executing transactions consumes gas that can be converted into Ether using the *GasPrice*, and the Ether is charged to the transaction submitters. Moreover, the higher the *GasPrice*, the more chance to execute the transaction faster. *GasLimit* is the maximum amount of gas that transaction submitters commit to the transactions. If the *GasLimit* is less than the execution needs, the transaction will fail with an exception out-of-gas, and the full amount of gas will be paid to the miners. Note that all unused gas is returned to the transaction submitters. All submitted transactions are located in the memory pool of each node, miners collect these transactions and execute them to include them in their block. Miners tend to select transactions that look more profitable to them based on the *GasPrice*. In addition to the *GasPrice*, miners prefer transactions that involve deploying new contracts to gain more gas.

As mentioned in the introduction, the fee structure is important for a number of reasons, which we discuss here for completeness. The limit to the amount of gas by the submitter will ensure the execution terminates, since for every opcode executed a strictly positive amount of gas is added to the used gas tally. This also provides a resolution for the halting problem of Turing complete programmes, since it assures termination. A second important effect of the fee structure is that it avoids denial of service attacks [8] [3], which would arise if miners use all their resources to execute useless contracts at no cost, thus not allowing the miners to work on tasks important for the dependable operation of the blockchain. In fact, the issue here is slightly more subtle, it is not only important that a fee is associated with the smart contract, it also needs to be sufficiently high. In particular, the attacks on Ethereum (EXTCODESIZE attack on 2016) [13] were possible because certain opcodes would demand considerable computation, but would not cost the submitter a large fee. In response to this attack, Ethereum changed the fee for the EXTCODESIZE opcode to mitigate against the attack.

III. Smart Contracts Selection

As discussed in [14], among other properties, a well-designed benchmark needs to be representative for the system at hand. Our approach is to select the most commonly used smart contracts available on Ethereum, by reducing the set of all existing smart contracts based on exclusion criteria. To start, we use EtherScan to identify all the available verified Ethereum smart contracts. EtherScan provides for all contracts in Ethereum the following: contract address, contract name, compiler, balance, transaction count, settings and verified date. At the time of collecting this data (10 May 2018), there were over 28000 verified contracts. Since EtherScan contains a large number of verified contracts distributed into many HTML pages, we had to write JavaScript code that fetches the contents from all those HTML pages. We were then able to produce a single HTML page that contains a single table with all verified contracts. To establish a reasonable set of contracts for our study, we decided to select the first 100 most commonly used smart contracts. These most commonly used contracts were ranked by the number of transactions each contract experiences (transaction count). For instance, EtherDelta was the most commonly used contract in Ethereum as it was invoked by about 9.6 million transactions. For each contract, we manually collected the bytecode for the contract creation as well as analysed the source code. The analysis process is important in order to know what functions each contract is using and what possible inputs (if any) each function required. We also went through the transactions that were sent to invoke each contract in order to collect a transaction example for each function. This is to allow us to test and benchmark real Ethereum transactions instead of creating our own transactions. Finally, we excluded all contracts that are without either source code or bytecode available. Thus, we ended up with 80 different contracts. For the functions in each contract, we excluded some functions that cannot be obviously benchmarked without further assumptions about input parameters, such as transfer, transferFrom, withdraw, depositToken and some other token related functions. We ended up with 77 different functions to be considered in our benchmark.

IV. Design of Benchmark Measurement System

It is possible to measure the computational efforts of both creation and execution of transactions in the Ethereum testnet or in a private network, but it is not straightforward to control all aspects that may impact measuring CPU usage. In this section we describe our system design to measure CPU usage for smart contracts.

Specifically, smart contracts are 'created' by deploying them on the blockchain through submitting a *Contract Creation Transaction* that contains the correct bytecode (Creation Code) for a smart contract. After creation of the contract, this contract has an address at which it can be called. Once deployed, contracts can be executed by submitting transactions that refer to the contract's address. However, there is potential interference of a number of factors, including transaction validation overhead, signature validation overhead and proof or work computation. Therefore, we propose an alternative approach that isolates the execution of the smart contract from other computation. We measure the execution time in a single client EVM and simulate any of the other aspects as needed, for instance the submission of transactions.

For the smart contracts selected as explained in Section III, we investigate in detail the code of each collected smart contract at a function level, to identify how many functions each contract has, what inputs these need, and how to call the function. We initialised a set of Ethereum accounts with available balances to launch the transactions as well as to prepare the inputs for each function. For all functions we use real transaction inputs submitted to the Ethereum as mentioned in the Section III.

The next phase is to execute transactions. To that end, we create transactions, submit them using the accounts we set up and then execute each transaction in the local machine. For instance, to measure CPU use for a function within some contract, we first launch a transaction to deploy the smart contract containing this function and then launch another transaction with all associated inputs to trigger that function. We record the timer for each of the described steps and store the results in a separate file. To compare different contracts, we calculate used gas per second of CPU use, so that one knows how much reward one can gain from per unit of CPU time invested. For some contracts and functions, we were unable to successfully execute transactions. For instance if a contract calls other unverified contracts, or because of source code issues. Eventually, we managed to execute 76 contracts (out of 80 contracts) and 21 functions (out of 77 functions) in total. All results and observations are presented in section V.

V. Results and Discussion

This section presents and discusses a first set of results for benchmarking both contract creation and function execution transactions. We conduct the experiment on a single machine that submits and mines all experiment transactions. The machine is a MacBook Pro and equipped with a 2.8 GHz Intel i5 CPU and 8 GB RAM. All transactions are executed 100 times and the average time is calculated as well as the confidence interval. We use the Python PyEthApp [5] client. For each contract creation and function execution we read the data from the first phase and run the experiment 100 times, which provides 95% confidence intervals width a half-width of less than 0.2 times the average.

A. *Contract creation transactions*

First we measure the CPU usage for contract creation, for all 76 contracts. Figure 1 shows the amount of used gas that can be collected by miners per microsecond of CPU usage for contract creation transactions. The main curve presents the amount of used gas for creating each contract, while the straight line presents the average used gas for all the tested contracts. There is quite some difference between the return on investment for miners when creating contracts, roughly up to a factor of 5. The amount of used gas for creating contracts varies between 51 and 269 units of gas with an average of 182. Contracts such as Nagacoin, StatusContribution, MatchingMarket and Dragon are the most profitable ones with over 260 units of used gas awarded per microsecond of CPU usage. However, contracts such as TronToken, Controller, EKT, CybereitsToken and GnosisToken are more costly, with an amount of used gas awarded less than 100 units of gas. From these early experiments we see that the profit (amount of used gas) miners can get from contract creation transactions is not particularly well aligned with their computational effort.

Contract Name	CPU Usage (μs)	Used Gas	Gas/CPU
MatchingMarket	13193	3544767	269
StatusContribution	7918	2120935	268
NAGACoin	8491	2231105	263
Dragon	6226	1634521	263
EKT	7316	712297	97
TronToken	12476	891859	71
Controller	20160	1094303	54
GnosisToken	20419	1036626	51

TABLE I: The most profitable and expensive contract creation transactions.

Table I shows the CPU usage (execution time in microseconds) and the amount of used gas for the most profitable and expensive contract creation transactions. The profitability is based on the amount of used gas offered to miners per time unit of CPU execution. It is clear that the CPU usage is not proportional to the amount of used gas offered by contract creation transactions. For example, both EKT and NAGACoin contracts consume roughly the same amount of CPU time, while the latter offers over 3 times as much used gas, and thus, offers more profit to the miners. As already mentioned, the profitability to miners can differ more than a factor of 5, for the contracts MatchingMarket and GnosisToken, respectively.

B. *Function execution transactions*

When executing a smart contract, often only a single function is called, and therefore we compare in for the execution phase based on functions, as opposed to full contracts. Figure 2 shows the amount of used gas collected by miners per microsecond of CPU usage for function execution transactions. For each function, we provide the contract they belong with, labeled C1 to C17. The main line presents the amount of used gas for executing each function, while the straight line presents the average used gas for all the tested functions. Some contracts have multiple functions that we call (e.g., contract C10 has two functions, namely, makeWallet and logSweep). In addition, some contracts share the same function name (e.g., both C1 and C2 have a function called deposit). However, the source code for those functions that have the same name is not necessarily identical, and thus, the benchmarking results may differ accordingly (see later in this section for a more detailed comparison for functions with identical names).

The amount of used gas collected from executing functions varies between 1 and 48 units of gas per CPU microsecond with an average of 27. This presents a considerable difference in terms of profitability to the miners. Functions such as makeWallet (in both C3 and C10) and mine (in C17) are the most profitable functions with 40 units of gas or more awarded for each microsecond of CPU usage. However, functions such as deposit (in C1), deposit (in C9), released (in C12) are more costly since the amount of used gas awarded is less than 15 units of gas. From these early experiments we see that the profit (amount of used gas) miners can collect from function execution transactions is not well aligned with their

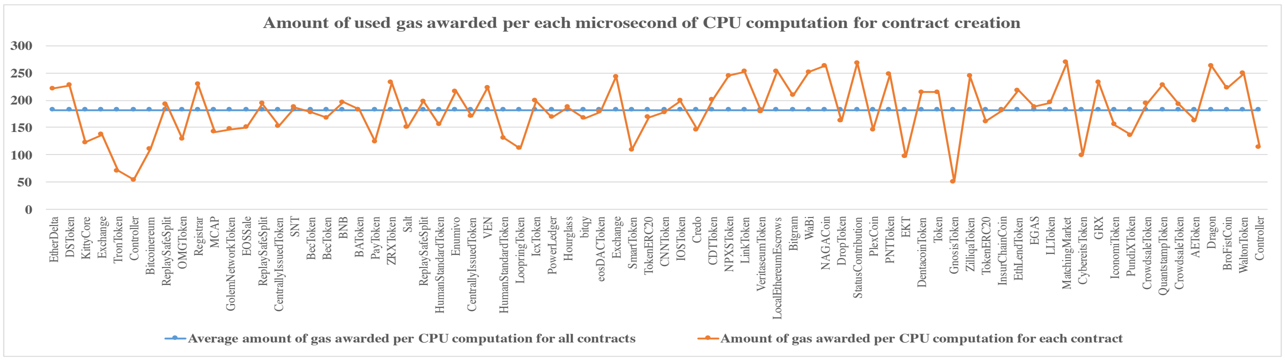

Fig. 1: Amount of used gas awarded per each microsecond of CPU usage for contract creation

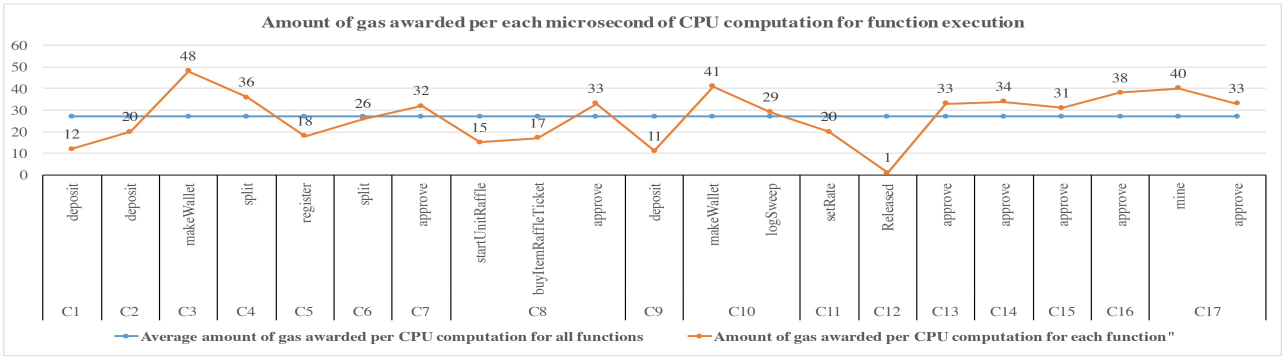

Fig. 2: Amount of used gas awarded per each microsecond of CPU usage for function execution

computational effort. Miners could gain a factor of up to 50 times more profit if they selected profitable function execution transactions such as makeWallet compared to the costly ones such as released.

Contract	Function Name	CPU Usage (μs)	Used Gas
C1	deposit	2365	29223
C2	deposit	2472	49265
C3	makeWallet	3861	184563
C4	split	2097	75753
C5	register	5072	89734
C6	split	1895	49664
C7	approve	1420	45677
C8	startUnitRaffle	1492	22899
	buyItemRaffleTicket	1297	22643
	approve	1387	45423
C9	deposit	2594	29411
C10	makeWallet	3484	142501
	logSweep	925	26509
C11	setRate	1406	28109
C12	Released	97669	116182
C13	approve	1451	47174
C14	approve	1345	45167
C15	approve	1487	45384
C16	approve	1190	45677
C17	mine	2131	84605
	approve	1378	45381

TABLE II: The average execution time (in microsecond) and the amount of used gas for all function execution transactions.

Table II shows the CPU usage (the average execution time in microseconds) and the amount of used gas for all function execution transactions. It is clear that the CPU usage is not proportional to the amount of used gas offered by function execution transactions to the miner. For instance, released function (in C12) is the most expensive function as it consumes much more CPU time compared to the used gas offered. Functions such as makeWallet (in both C3 and C10) are more profitable than the released function since they consume by far less CPU time while they offer more used gas.

We looked in more detail at the benchmarking results for functions with identical names (such as deposit, approve, split and makeWallet) used by different contracts. For the deposit function which is used by three contracts, namely, C1, C2 and C9, we found the benchmarking results for this function in C2 is significantly different from that for other functions. Deposit function in C2 offers about 70% more used gas per each microsecond of CPU usage compared to deposit function in both C1 and C9. We inspected the source code for all deposit functions and found that the deposit function in C2 has slightly different code with an extra line of computation, which takes the current block number and store it inside a specific mapping to keep tracking of the last active transaction. This extra line of computation resulted in a significant increase in the amount of used gas, while the computational efforts was increased by less than 5%. Therefore, miners could gain more profit be selecting deposit function in C2 instead of other deposit functions.

Similarly for the split function which is used by both C4 and C6. Split function in C4 offers 38% more used gas compared to split function in C6. We inspected the source code for both functions and found that the latter has two extra IF checks. These checks resulted in a significant increase in the amount of used gas, while the computational efforts was only increased by about 10%. Therefore, miners could gain more profit by selecting the split function in C4 instead of C6. We finally note that, on the contrary, the results for makeWallet functions in both C3 and C10 are almost consistent. Similarly for approve functions in C7, C8, C13, C14, C15, C16 and C17.

C. Comparison between contract creation and function execution transactions

Finally, it is important to point out the significant difference in terms of the amount of used gas that can be collected by miners per each microsecond of CPU usage between contract creation and function execution transactions. The average amount of used gas awarded for contract creation transactions is nearly six times more than that for function execution transactions. That means miners could collect more profit by selecting and including contract creation transactions.

In conclusion, we have identified two type of discrepancies in terms of the reward for computational effort. First, both for contract creation and for function execution, the amount of used gas awarded is not consistently proportional to the CPU usage, there can be a factor of 5 difference in creation of different contracts and a factor of almost 50 difference in execution of different functions. Secondly, between contract creation and function execution there is considerable difference in reward for CPU usage, roughly a factor 6. Miners gain more profit if they select transactions based on these two observations. We believe that this implies a threat to the dependable operation of the blockchain, for which it is important to ensure that the used gas awarded is proportional to the CPU usage for all types of operations that are required. The results in this paper indicate that it would require modifications for Ethereum to establish a fair incentive model for miners as well as a fair cost model for the submitter of the transactions.

VI. Conclusion

This paper proposes a benchmarking approach to assess whether the fees miners receive from creating and executing smart contracts is proportional to the cost as expressed in terms of CPU usage. To illustrate our approach, we conducted the first benchmarking study to investigate whether the used gas for creating smart contracts and executing contract functions is proportional to the computational effort. We created a benchmark of the 100 most commonly used smart contracts in Ethereum. Due to some technical limitations, we managed to benchmark the creation of 76 contracts and the execution of 21 functions.

Our results show that the used gas is not always proportional to the computational effort for both creating and executing smart contracts. More specifically, contract creation is about a factor 6 more profitable to miners than execution of contract functions. In addition, some functions can be up to almost 50 times more profitable than others. This indicates that miners could gain more profit when creating and deploying new contracts instead of executing existing ones, and by being selective about which contract functions to execute. Potentially, this forms a threat to the dependability of the overall blockchain, in that some important computational tasks may be not sufficiently attractive for miners to dedicate their resources to.

Vice versa, transaction submitters may offer different gas prices to compensate for the observed discrepancies, thus incentivising miners more fairly again. How to achieve this in practice is a topic for further research. With respect to our proposed benchmarking approach, additional refinement of the proposed methods can be envisaged, for instance considering computing effort beyond CPU usage (e.g. storage), and relating computational effort more directly to actual energy costs. In addition, we thus far only conducted experiments on a single platform, and additional experiments are in order to cover more contracts, functions, types of client codes, operating systems and hardware.

References

[1] Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system.

[2] Wood, G. (2014). Ethereum: *A secure decentralised generalised transaction ledger*. Ethereum Project Yellow Paper, 151, 1-32.

[3] Chen, T., Li, X., Luo, X., & Zhang, X. (2017, February). *Under-optimized smart contracts devour your money*. In Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on (pp. 442-446). IEEE.

[4] Ethereum Foundation. (2013). *EVM Operation Codes Gas Cost*. `https://docs.google.com/spreadsheets/d/`

[5] Ethereum foundation. (2013). *Pyethereum client*. `https://github.com/ethereum/pyethereum`

[6] Alharby, A. and Van Moorsel, A. (2018). *The Impact of Profit Uncertainty on Miner Decisions in Blockchain Systems*. In UK Performance Engineering Workshop, 2018.

[7] Carlsten, M., Kalodner, H., Weinberg, A. and Narayanan, A. (2016). *On the Instability of Bitcoin Without the Block Reward*, ACM SIGSAC Conference on Computer and Communications.

[8] Gervais, A., Karame, G., Wuest, K., Glykantzis, V., Ritzdorf, H. and Capkun S. (2016). *On the Security and Performance of Proof of Work Blockchains*, ACM SIGSAC Conference on Computer and Communications Security.

[9] Zamyatin, A., Wolter, K., Werner, S., Mulligan, C., Harrison, P., and Knottenbelt, W. (2017). *Swimming with Fishes and Sharks: Beneath the Surface of Queue-based Ethereum Mining Pools*, IEEE Mascots

[10] Aldweesh, A., Alharby, M. and Van Moorsel, A. (2018). *Performance Benchmarking for Ethereum Opcodes (Short Paper)*, submitted for publication.

[11] Ezhilchelvan, P., Aldweesh, A. and Van Moorsel, A. (2018). *Non-blocking two phase commit using blockchain*, MobiSys CryBlock.

[12] Dong, C., Wang, Y., Aldweesh, A., McCorry, P. and Van Moorsel, A. (2017). *Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing*, ACM SIGSAC Conference on Computer and Communications Security.

[13] Buterin, V. (2016) *Transaction spam attack: next steps* (2016). https://goo.gl/uKi9Ug

[14] Vieira, M., Madeira, H., Sachs, K. and Kounev, S. (2012). *Resilience Benchmarking*, in *Resilience Assessment and Evaluation of Computing Systems, Wolter* et al. (Editors), Springer.

[15] Hertig,A. (2017). *Ethereum's Big Switch: The New Roadmap to Proof-of-Stake* (2017). https://bit.ly/2xRmPjB