Permuting Transactions in Ethereum Blocks: An Empirical Study

Jan Droll[0009−0003−5003−3898]

Karlsruhe Institute of Technology
Karlsruhe, Germany

Abstract. Several recent proposals implicitly or explicitly suggest making use of randomized transaction ordering within a block to mitigate centralization effects and to improve fairness in the Ethereum ecosystem. However, transactions and blocks are subject to gas limits and protocol rules. In a randomized transaction order, the behavior of transactions may change depending on other transactions in the same block, leading to invalid blocks and varying gas consumptions. In this paper, we quantify and characterize protocol violations, execution errors and deviations in gas consumption of blocks and transactions to examine technical deployability. For that, we permute and execute the transactions of over 335,000 Ethereum Mainnet blocks multiple times. About 22% of block permutations are invalid due to protocol violations caused by privately mined transactions or blocks close to their gas limit. Also, almost all transactions which show execution errors under permutation but not in the original order are privately mined transactions. Only 6% of transactions show deviations in gas consumption and 98% of block permutations deviate at most 10% from their original gas consumption. From a technical perspective, these results suggest that randomized transaction ordering may be feasible if transaction selection is handled carefully.

Keywords: Ethereum · Empirical study · Permutation · Errors · Gas.

1 Introduction

Under Ethereum's current block construction mechanism, block producers may select and arrange transactions at their discretion, as long as the constructed block conforms to the protocol specifications. While the selection of transactions is designed to be driven by economic profits, the freedom to arrange transactions is nowadays used to gain additional profits, summarized under the term MEV [8]. These additional profits have led to centralization in block production, facilitated censorship, and might endanger the decentralization of block validation in the long term. A randomized transaction order might counter such centralization forces and enhance fairness in Ethereum. In [10] for example, the ordering of transactions within a block is unknown until their inclusion is guaranteed to protect users from MEV. However, the effects of randomized transaction ordering on gas consumption and on block validity according to protocol rules

remain unclear. Other approaches, e.g. [19,5], propose to encrypt at least some information of transactions. As a result, block producers would be unable to determine the gas consumption of individual transactions and the block under construction, or check blocks for protocol violations. The concept BRAID [20] introduces multiple entities simultaneously proposing transactions that must be included in the next block. An execution order must then be agreed upon, and dependencies between transactions may affect both gas consumption and the risk of protocol violations. Lastly, inclusion lists (ILs), e.g. [18], would require block producers to include specific, e.g. censored, transactions in a block. Again, dependencies between transactions, especially when executed at the bottom of a block as proposed in [24], may affect gas consumption and lead to protocol violations. Hence, we find that both the gas consumption and possible Ethereum protocol violations in unknown execution orders need investigation.

We report on the results of an experiment that permutes and executes transactions of Ethereum Mainnet blocks to analyze protocol violations and the gas consumption of blocks and transactions. In particular, we answer the following questions with an empirical study of more than 335,000 Ethereum Mainnet blocks:

- How many block permutations violate current protocol rules, rendering the block permutation invalid, and for what reason?
- How many blocks and transactions deviate from their original gas consumption when permuted?
- How large is the gas deviation of blocks and transactions when permuted?

Permuting transactions in blocks is close to an unknown execution order and activates dependencies between transactions. The presented approach to permute transactions in Ethereum blocks avoids protocol violation at the cost of only minor, backward-compatible modifications of block validation rules. We also analyze execution errors that, unlike protocol violations, affect only individual transactions and user experience.

The remainder of the paper is structured as follows: Section 2 briefly introduces concepts of Ethereum required to understand this paper and differentiates this work from existing work before presenting our methodology in Section 3. After presenting the result of the experiment in Section 4, we discuss the findings in Section 5 and finally draw our conclusion in Section 6.

2 Fundamentals and Related Work

The Ethereum [6] network operates a replicated state machine everyone is able to interact with. Interactions are expressed in form of transactions, i.e., signed statements that transition the state of the replicated state machine during their execution. Transactions are executed in the order they are arranged in a block. Executing a block is equivalent to consecutively executing all contained transactions on the state that results from processing previous blocks in the blockchain.

State transitions depend on a transactions' data and follow deterministic rules specified as instructions. Instructions are in particular capable to read from and write to the state, support logical loops, and conditional execution branching.

For each executed instruction, a specific amount of gas, an abstract unit for computation and storage costs, is due. Transactions specify a limit for the cumulative amount of gas that can be consumed during their execution together with a conversion rate between gas and Ether, Ethereum's native currency. Based on the actually consumed gas g and specified conversion rate p, the transaction fee $f = g \cdot p$ is deducted from the account of the transaction sender.

Accounts are objects, identified by addresses, that define the state of the replicated state machine and are used to keep track of individual Ether balances in Ethereum. In addition, accounts record the number of executed transactions with the account's address specified as sender up to the current state, referred to as *nonce*. Transactions also specify a number called *nonce* that must match the nonce recorded in the transaction sender's account on execution. So-called smart contract accounts contain instructions organized into functions, which are executed when triggered by transactions.

A block violates the Ethereum protocol rules if during the block's execution the nonce of a transaction does not equal the nonce of the senders account, the transaction sender owns less Ether than required to settle maximum transaction fees plus the amount of ether to be transferred, or if the cumulative gas used by transactions exceeds the block gas limit. Note that the maximum transaction fee $f^{\max} = l \cdot p$ is calculated from the gas price p and gas limit l as specified by the transaction instead of the actually consumed gas g.

The gas consumption of Ethereum transactions was investigated in [26] with the goal to determine the stability of the gas consumption of smart contract functions. The authors analyzed the on-chain recorded execution results of transactions between October 2017 and February 2019 and found that 25% of functions are unstable, i.e., show significantly varying gas consumptions among invocations by transactions. In addition, 50% of all transactions invoke one of these unstable functions. We analyze the gas consumption along with execution and protocol errors primarily based on permuted execution results, not on on-chain executions.

A related field of research is to estimate the actual gas consumption of single individual transactions. This estimation is required during transaction creation to delimit the amount of spendable gas while avoiding 'out-of-gas' errors during transaction execution. Common strategies involve compiler outputs, e.g. as provided by solc[1], and binary searches, e.g. when calling 'eth_estimateGas' on go-ethereum[2]. The authors of [22,17] deal with optimizing such estimations. However, our work does not focus on estimating gas for new transactions, instead we analyze if limits are well chosen.

[1] https://docs.soliditylang.org/en/latest/using-the-compiler.html (2025-05-20)
[2] https://github.com/ethereum/go-ethereum (2025-05-20)

Permutations have been used for analysis purposes in [1,2]. The papers aim to quantify MEV by comparing the cost for users across block permutations. While [2] is limited to a formal definition based on permutations, [1] also conduct an experiment. They analyzed up to 16 permutations, but only of specific sets of MEV related transactions, to estimate a value for their 'reordering slippage' metric. In contrast, we quantify the technical sensitivity of Ethereum regarding protocol violations, execution errors and gas consumption when entire blocks are permuted - based on thousand permutations per block.

Some non-Ethereum systems, e.g. Solana [25], optimize transaction ordering for parallel execution, taking state-dependencies into account. Rather than optimizing transaction ordering, we empirically analyze the effects of state and ordering dependencies in Ethereum.

3 Methodology

We conducted an experiment to analyze the effects of permuted transaction execution on block validity and on the gas consumption of both blocks and transactions. In the experiment, the transactions of Ethereum Mainnet blocks are permuted and the results of the executions are recorded for analysis.

3.1 Setup and Workflow

The technical setup, illustrated in Fig. 1, is made up of an Ethereum node, consisting of a standard consensus client and a modified execution client, a permutation software, and a database. The consensus client ensures that the execution client follows the canonical blockchain of the Ethereum Mainnet. Regular executions clients are responsible to maintain the state of the replicated state machine based on the canonical blockchain. The modified execution client used in our experiment provides additional API functionality to execute custom blocks. Custom blocks consists of a list of transactions together with a state number[3] and other essential information. Transactions in the custom block are consecutively executed on the specified state in the given order, but the performed state transitions are ephemeral for the time of the execution of the custom block. The consecutive execution is crucial for our experiment to guarantee interdependent execution of transactions. However, for the experiment, we used only functionality of the standardized 'eth' API provided by regular execution clients, except for the execution of custom blocks. The permutation software creates custom blocks by permuting transactions of blocks received from the Ethereum node, instructs the modified execution client to execute these custom blocks, and records the results in a database used for further analysis.

[3] The state with number n is the resulting state after processing the block with number n in the chain.

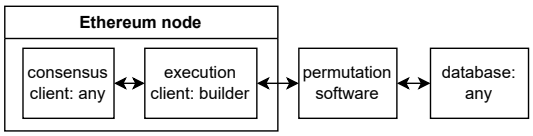

Fig. 1. Components required for the experiment. Arrows represent communication between components. The execution client 'builder' and our own permutation software are the experiment's core components. Any standard consensus client is suitable for the experiment and the permutation software is database agnostic.

For the experiment, we used 'builder'[4], a modified version of 'go-ethereum'[5], as execution client. The permutation software is implemented by ourself and public available [9].

The workflow of the permutation software (in the following simply called 'software') is as follows: After successfully connecting to the Ethereum node, the software subscribes to new block heads. On receiving a new head, the block processing starts by requesting the full block referred to in the header together with receipts of each transaction included in the block from the Ethereum node.

Re-execution Then, the software requests the execution client to re-execute the block as a custom block. This custom block is identical to the received full block, which means that it consists of all transactions of the previously received full block in the identical order. The purpose of the re-execution is to verify the correct behavior of the execution client with the modifications in place. The execution client either responds with a so called *core error* or the result of the re-execution. A core error means that the block violates a rule specified in the Ethereum specification and the custom block execution aborted because the block would be invalid. In case of an core error, the software extracts the concrete type of the core error and during which transaction's execution the error occurred, and records both the type and the transaction in the database. In addition, any further processing of the current block is aborted. Otherwise, the software extracts the amount of gas consumed by each transaction during the re-execution together with possible execution errors from the result and stores both with the gas consumed in the original block, extracted from the receipts, in the database. In addition, some information regarding the block, including the gas consumed as declared in the header is stored in the database.

Permuted execution Now, depending on the content of the block, the software either randomly permutes all transactions in the block with the algorithm described in Sec. 3.2 to create custom blocks or the software computes a custom block for each possible permutation. In either case, the software requests the execution client to execute the now permuted block as custom block. In case of an error response, the type of the core error and the transaction that caused the core

[4] https://github.com/flashbots/builder (2025-05-20)
[5] https://github.com/ethereum/go-ethereum (2025-05-20)

error is extracted and recorded in the database. Further processing of the block continues, which is a different error handling compared to the re-execution in which block processing aborts. Otherwise, the amount of gas consumed by each transaction is recorded in the database and, if the execution of a transaction resulted in an so called *execution error*, the type of the execution error is also extracted and recoded in the database. Execution errors do not result in invalid blocks, which is the difference to core errors. We randomly permute blocks 1000 times except for blocks that contain less than seven distinct transaction senders.

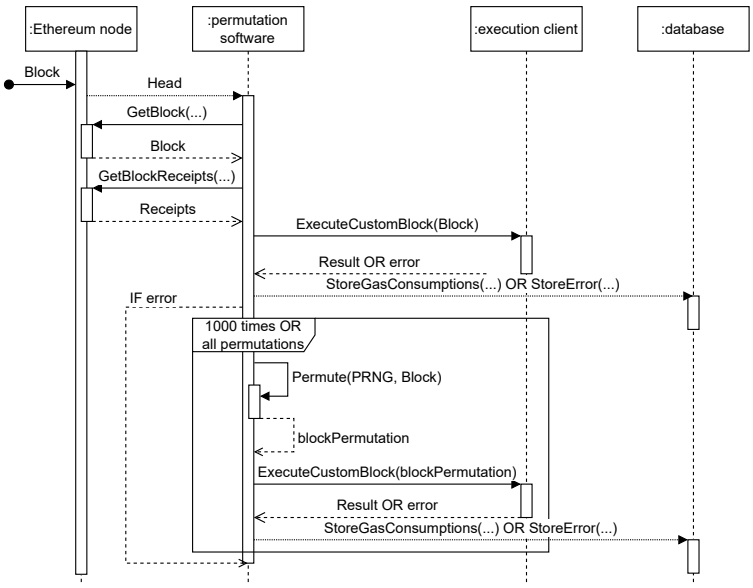

Fig. 2. Block processing workflow. Interaction between permutation software and Ethereum node use the standardized 'eth' API. Interaction between the permutation software and the execution client utilize the additional custom block functionality. In practice, the execution client is part of the Ethereum node as depicted in Fig. 1.

In pre-production runs, we discovered two incorrect behaviors of builder and corrected them before conducting the experiment: First, for 0.2% of analyzed transactions (13% of blocks) the state transitions was incorrect because the API did not provide an option to set the header field 'mixHash' in the header, even transactions can access the field's value during execution. See EIP-4399 [12] for more details. Second, the builder did not implement the EIP-4788 [23] oracle that provides the root of the parent beacon block to the EVM. Some transactions, likely all related to EigenLayer, show a different gas consumption if the oracle is not implemented.

3.2 Permutation

We designed the permutation algorithm to keep the relative order of transactions by a single sender unchanged. This design avoids protocol violations (see Sec. 2) due to a mismatch of the nonce specified by the transaction and in the current state[6], and the implementation is straightforward.

The resulting algorithm is provided in Algorithm 1 and works as follows: First, the sender of each transaction is recovered from the transactions' signature in the order of the original block. The sender address is appended to a list if not already present to avoid duplicates. Then, the list of addresses is shuffled using the Fisher-Yates shuffle algorithm as implemented in the Golang standard library 'math/rand'[11]. If the block contains less than seven distinct senders, the algorithm deterministically computes each possible permutation of the list of addresses (which are at most 720 permutations) instead of shuffling the list.

Algorithm 1 Permutation algorithm

```
1: procedure PermuteTransactions
2:     input: txs // list of transactions
3:     input: seed // number
4:     var distinctSenders // emtpy list
5:     for i ← 0; i < len(txs); i++ do
6:         sender ← ecrecover(txs[i]) // Ethereum precompile
7:         if sender ∉ distinctSenders then
8:             append(distinctSenders, sender)
9:     permutedSenders ← shuffle(seed, distinctSenders) // Fischer-Yates
10:    var permutedTxs // emtpy list
11:    for j ← 0; j < len(permutedSenders); j++ do
12:        permutedSender ← permutedSenders[j]
13:        for k ← 0; k < len(txs); k++ do
14:            originalSender ← ecrecover(txs[k])
15:            if permutedSender = originalSender then
16:                append(permutedTxs, txs[k])
17:    return permutedTxs.
```

The Fischer-Yates algorithm guarantees that every permutation of the list is equally likely [15]. The final list of permuted transactions is then incrementally constructed by iterating over the now shuffled list of distinct sender addresses and thereby appending each transaction of the sender from the original list of transactions.

The Fisher-Yates algorithm requires a source of randomness to randomly shuffle the list of transactions. In the experiment we used seed-based pseudo-random number generators (PRNG) so that the experiment is reproducible. We instantiated an independent PRNG for each block (block-level PRNG) with the

[6] A transaction execution increases the nonce within the account.

hash of the respective block as seed. This block-level PRNG is then used to seed an independent PRNG for each permutation (permutation-specific PRNG) with a freshly drawn pseudo-random value. The permutation-specific PRNG is then used as source of randomness by the Fisher-Yates algorithm.

3.3 Ethical considerations

The experiment uses only public data, namely some blocks of the Ethereum Mainnet. This public data consists of pseudonymous data in form of cryptographic addresses, and further, potentially personal, data supplied by transactions senders or block builders. The experiment's database contains only data that is stored and processed by every Ethereum node together with permutation results. Analyzing the permutation results involves only data processed and stored by every Ethereum node and a list of transactions from analysis by [16]. No attempts are made to deanonymize data and no additional new data links are created beyond the one to [16]. Therefore, we do not see ethical aspects that originate from performing the experiment.

4 Results

With the experiment described in the previous section, we collected data for about six weeks during March and April 2025 (blocks 22039048 to 22374561). During that period, we observed 335,886 blocks[7] from which 335,646 blocks contained transactions. We permuted only the latter blocks, so we refer to them as 'permuted blocks', and those blocks contained more than 56 million transactions. About 64% of transactions interact with smart contracts while the remaining transactions are simple Ether transfers. In 1.8 million cases, the same smart contract was invoked by at least two transactions within a single block. However, we expect even higher interdependencies, as smart contracts can directly interact with one another. In this section, with 'original', as in 'original block', we refer to data exactly as agreed on by the Ethereum network and persisted in the blockchain. In the experiment, we recorded the gas consumptions for more than 42 billion permuted transaction executions, and for more than 72 million block permutations, we recorded the reasons for the their invalidity. Note that a permuted block has typically up to 1000 block permutations from which some executed successfully and some were invalid due to protocol violations. For 628 blocks we computed all possible permutations, because they contained less than 7 distinct transaction senders. In the following, we characterize blocks and transactions based on the results of the experiment, starting with protocol violations (Sec. 4.1), then analyze execution errors (Sec. 4.2) and finally covering the gas consumption of blocks and transactions (Sec.4.3).

[7] We observed not only canonical blocks, but also blocks that got not finalized.

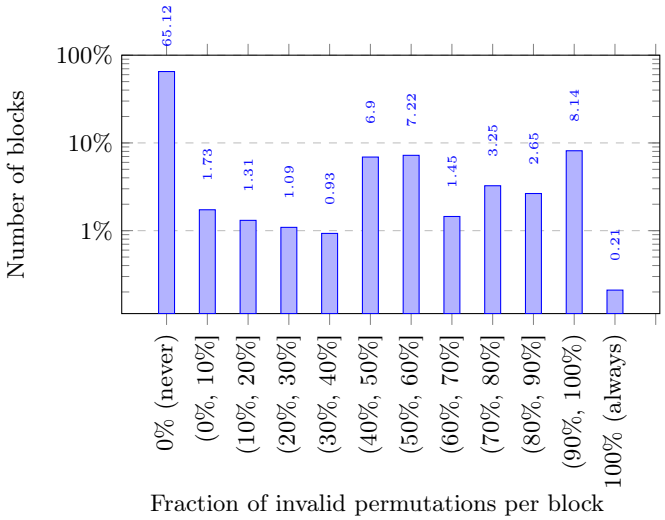

Fig. 3. Distribution of blocks according to the fraction of core errors observed in their permutations. Note the log-scaled y-axis and x-axis interval ranges.

4.1 Protocol Violations

While we designed the permutation algorithm for the experiment in a way that eliminates not matching nonces in block permutations, we observed two other protocol violations in form of core errors which invalidated 22% of all block permutations, affecting 35% of blocks. In Fig. 3, the fraction of invalid block permutations per permuted block is shown as a distribution across all permuted blocks. For 65% of permuted blocks, all block permutations were valid while for 0.21% of permuted blocks, all block permutations were invalid. With 71%, most invalid block permutations observe an *insufficient funds* error which can be related to privately mined transactions. The remaining 29% invalid block permutations run into an *gas limit reached* error that could be avoided by applying a gas margin. Only 4.6% of blocks have both gas limit reached and insufficient funds errors in permutations. Invalid block permutations could be avoided by restraining to mine transactions privately and respecting a gas margin.

Insufficient funds errors occur if the sending account of a transaction owns not enough Ether to settle the maximum transaction fee $f^{\max}$ together with the amount of Ether to be transferred. Two scenarios can cause this kind of core error in block permutations: Either the sending account received some Ether[8] by an earlier transaction in the original block or a transaction α caused a higher gas consumption of a transaction β sent by account σ so that σ has no more sufficient funds for a subsequent transaction γ. Future work can analyze the frequency of such scenarios by analyzing intermediate states of executions. However,

[8] From itself, another external owned account or a smart contract.

99.93% of transactions with insufficient funds in permutations can be attributed to privately mined transactions according to data from [16]. Privately mined transactions are first seen in a block and not on the network.

Gas limit reached errors occur if the execution of a transaction exceeds the gas limit of the block. Section 4.3 demonstrates that some transactions require more gas in certain block permutations than in the original block, thereby potentially causing this error. There is a moderate to strong positive correlation (0.58) between the extend to which the block gas limit is used in the original block and the ratio of gas limit reached errors in permutations of a block. For example, 70% of the permuted blocks with gas limit reached errors consumed at least 91% of their gas limit in the original block. Also, less than 0.3% of permuted blocks with gas limit reached errors that utilize at most half of the gas limit (The so called *target* since EIP-1559) in the original block observe an invalidation due to reaching the block gas limit in our experiment. According to these findings, the closer blocks are to their block gas limit, the more likely they will be invalid when permuted due to gas limit reached core errors.

4.2 Execution Errors

We tracked execution errors[9] that affect only the transaction causing the error, and not the validity of block permutations. We expected more failing transactions due to state dependencies and found that those contributing to the increased failure rate tend to send Ether directly to the block assembler and are privately mined. Also supported by the observation that these transactions fail due to a revert instruction, we conjecture they are MEV-related, and believe regular users' experience is not negatively affected by random transaction ordering.

Execution errors occur in 81% of original blocks, increasing to 94% under permutation. This trend is even more prevalent in transactions where 2.7% of transactions fail due to an execution error in original blocks and 4.7% in block permutations. The increase is reflected by new failing transactions (transactions which do not fail in the original block, but in a block permutation), only a small number of failing transactions in the original block remain non-failing across all permutations. One might argue that with a randomized transaction order, execution errors occur more frequently due to new failing transactions, while transactions that failed in the original block will continue to fail in block permutations. Such a statement is only partially supported because the latter transactions fail in 72% of their executions in block permutations on average and new failing transactions in 42% on average.

More interestingly, all new failing transactions share a common property. They either directly affect the coinbase address of the block in a positive way (send Ether directly to the block producer) or they do not pay fees beyond the base fee.[10] Only 2% of transactions with this property are new failing transac-

[9] `https://github.com/ethereum/execution-specs/blob/master/src/ethereum/cancun/vm/exceptions.py` (2025-05-20)

[10] Fees beyond the base fees might be ignored as more than 99.99% of new failing transactions share the first property.

tions. While not paying fees beyond the base fee is unremarkable, transferring Ether to the block producer draws attention. The effect of a direct payment to block producer is that the sender of the transactions can decide that only specific block producers profit from their transaction. We found that 99% of the transactions that send Ether directly to the block producer are privately mined according to [16]. Several institutions offer services that result in privately mined transactions, for example to protect from MEV extraction but also to protect MEV extraction itself, and such services guarantee a specific order of transactions. We conjecture that new failing transactions are related to so called MEV extraction bundles without providing further evidence or elaborating details.

In both original blocks and block permutations, we observed seven different execution error types. 74% of execution errors in block permutations (66% in original blocks) are revert errors, meaning that the logic of the application invoked by a transaction executed a special instruction that causes all state modifications made by applications to be rolled back. In 23% of execution errors in block permutations (31% in original blocks), the gas limit of the transaction was reached. In 2.5% of execution errors in block permutations (3.4% in original blocks), an invalid instruction was attempted to be executed. The remaining 0.04% of execution errors in block permutations (0.03% in original blocks) correspond to scenarios in which the stack of the EVM underflowed, a variable to track the gas consumption overflowed, the maximum code size was exceeded or an application tried to jump to an invalid destination. We highlight that the execution error type of new failing transactions is a revert error in 99.7% of execution errors and the gas limit of such transactions was reached in 0.02%.

4.3 Gas Deviations

The purpose of the experiment described in Sec. 3 was also to quantify the variations in gas consumption of blocks and transactions. We break down the results by blocks and transactions at first and then across all recorded executions, without associating them to individual blocks and transactions.

Block and Transaction Associated Results In this section, we evaluate data per block and per transaction: We calculate our metrics for each block and transaction with the executions of the respective block or transaction. The visualizations, however, plot the metric for each individual blocks and transactions.

Fraction of Deviations The first metric f^{D} measures the fraction of executions with gas deviations relative to the total number of successful executions N of a block B or transaction T according expressions (1) and (2). Note that successful executions are block permutations without protocol violations.

$$f_B^{\mathrm{D}} = \frac{n_B^{\mathrm{D}}}{N_B} \qquad (1) \qquad\qquad f_T^{\mathrm{D}} = \frac{n_T^{\mathrm{D}}}{N_T} \qquad (2)$$

Values for both f_B^{D} and f_T^{D} must be in the interval $[0, 1]$ since between zero and N executions can deviate. The metric is calculated for each permuted block

and transaction in our database and Fig. 4 plots f_B^{D} for all 335,646 permuted blocks as well as f_T^{D} for all 56 million transactions.

Over 45% of permuted blocks deviate in all permutations from their original gas consumption in our experiment, while about 11% of permuted blocks never deviate. However, permuted blocks clearly tend to deviate from their original gas consumption. In contrast, in our experiment, more than 94% transactions never diverge from their original gas consumption. 6% of transactions are deviating, a large fraction ($> 2.8\%$) deviates in 40%-60% of their executions and three out of four deviating transactions are privately mined. The data are coherent when arguing that a block with 168 transactions on average has 4 transactions with at least one deviating execution on average. Interestingly, according to our data, blocks contain on average 10 deviating transactions.

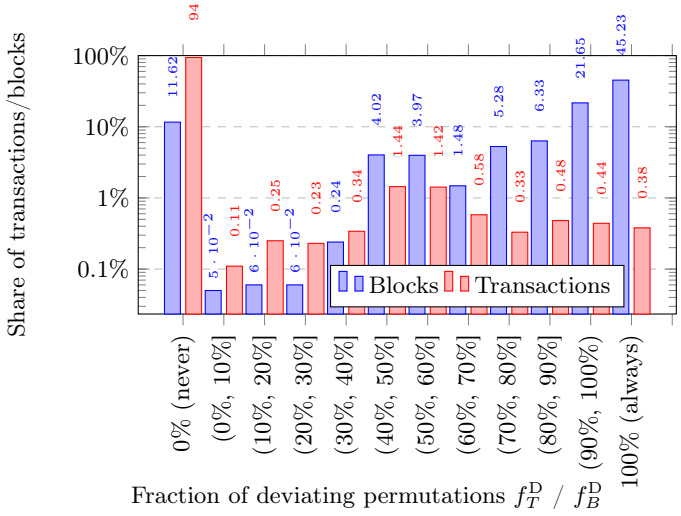

Fig. 4. Distribution of blocks (blue) and transactions (red) according to the fraction of deviating block / transaction permutations (f_B^{D} or f_T^{D}) observed in their permutations. Note the log-scaled y-axis and x-axis interval ranges.

We conjecture that most transactions have a constant gas consumption because Ethereum provides no inclusion or ordering guarantees for transactions and transaction senders typically only sign transactions that will succeed regardless of block inclusion and ordering. Hence, most transactions are not ordering dependent and consume the same amount of gas in every permuted execution.

Execution-focused Results In this section, we evaluate data per execution: We calculate metrics for each execution **without** grouping them according to blocks or transactions. Consequently, the visualizations plot the metrics over all recorded executions.

Intensity of Deviation Our second metric i^{R}_{E} measures the intensity of the gas deviation of block executions E_{B} and transaction executions E_{T} relative to their original gas consumption g according to expressions 3 and 4.

$$i^{\mathrm{R}}_{E_{\mathrm{B}}} = \frac{g_{E_{\mathrm{B}}}}{g_{\mathrm{B}}} \qquad (3) \qquad\qquad i^{\mathrm{R}}_{E_{\mathrm{T}}} = \frac{g_{E_{\mathrm{T}}}}{g_{\mathrm{T}}} \qquad (4)$$

Values for $i^{\mathrm{R}}_{E_{\mathrm{B}}}$ must be in the interval $[0.12\%, 85714.29\%]$ and values for $i^{\mathrm{R}}_{E_{\mathrm{T}}}$ must be in the interval $[0.06\%, 171,428.57\%]$.[11]

Fig. 5 shows the empirical distribution of $i^{\mathrm{R}}_{E_{\mathrm{B}}}$ over all 262 million block permutations in the database. Values are in the range $[19.78\%, 363.88\%]$ with an average of 99.23% and standard deviation of 2.88%. With more than 98%, almost all block permutations deviate at most 10% from their original gas consumption. Note that invalid executions (from block permutations with protocol violations) are not included because we do not have their actual gas consumption.

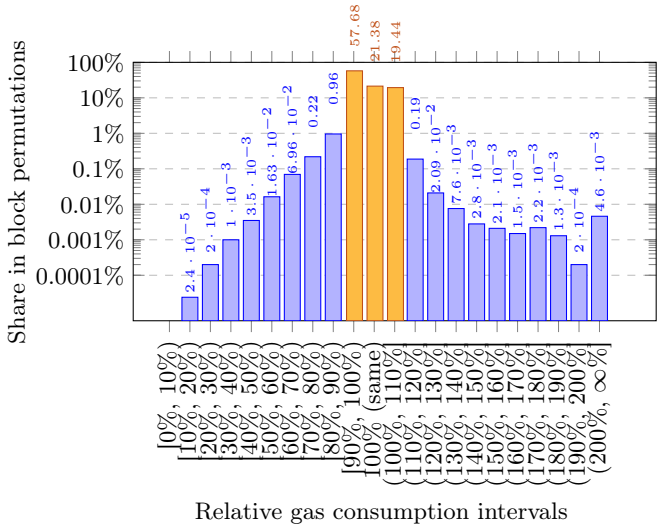

Fig. 5. Distribution of block permutations according to the gas deviation intensity $i^{\mathrm{R}}_{E_{\mathrm{B}}}$. Note the log-scaled y-axis and x-axis interval ranges. Orange-colored bars represent more than 98% together.

Fig. 6 shows the distribution of $i^{\mathrm{R}}_{E_{\mathrm{T}}}$ over all 42 billion permuted transaction executions in the database. Values are in the range $[0.31\%, 31,856.31\%]$ with an average of 100.04% and standard deviation of 14.66%. With 96%, most permuted transaction executions do not deviate from their original gas consumption.

According to these results, only very few block permutations result in significant deviations in gas consumption. However, even less intense gas deviations

[11] Based on the minimum of 21,000 units of gas for transactions and the new block gas limit of 36,000,000 for blocks which we use throughout the paper.

can result in gas limit reached errors and hence invalid executions as shown in Sec. 4.1. In addition, the previously stated conjecture is also supported by the results.

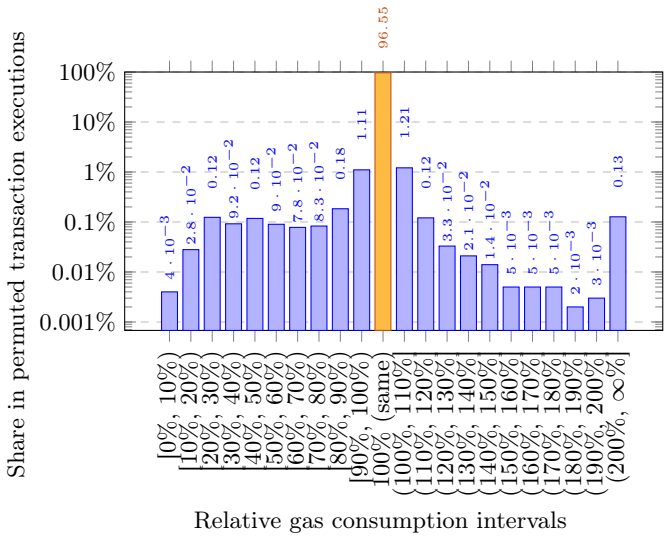

Fig. 6. Distribution of permuted transaction executions according to the gas deviation intensity $i^{\mathrm{R}}_{E_{\mathrm{T}}}$. Note the log-scaled y-axis and x-axis interval ranges. The orange-colored bar represents more than 96%.

5 Discussion and Future Work

The data we used for the experiment (i.e. Ethereum Mainnet blocks, transactions and states) reflect the current usage of Ethereum. The results of the experiment are only valid for the observed usage which can change in the future for various reasons. Also, we executed at most 1000 permutations per block because computing all permutations is infeasible for most blocks. In addition to computational complexity, functionality to execute custom blocks is challenging too. Common client software is shipped without the capability to consecutively execute multiple transactions on a specific state so that incremental state changes are taken into account. Available developer tools provide functionality to trace the execution of only a single transaction on a specific state with custom overrides and to simulate a block on a given state. The issue with the latter is the missing ability to export a specific state from the Mainnet without huge storage requirements. However, the community realized there is a need to simulate custom blocks on specific states and is actively working on a standardized functionality to simulate even multiple blocks with the 'simulateV1' API [13,14].

The experiment could be enhanced by capturing, augmenting, and analyzing execution traces (sequences of the performed steps during the state transition) to develop a deep understanding of the reasons for gas deviations and protocol violations in block permutations by identify resposible applications and usage patterns. We used Tenderly[12] to inspect execution traces of two transactions for which we recorded the most significant gas deviation. The first transactions consumed just 0.3% (44k gas) of its original gas consumption (14M gas), in which the transaction executed a set of methods of the UniswapV3 application many times until reaching its gas limit. The second transaction represents the other end of the scale and through the traces, we see that the same contracts as with the first transaction are involved but the transaction consumed 318 times (14M gas) of its original gas consumption (44k gas) in which the transaction reverted. An augmentation of execution traces, however, requires consideration of ethical aspects that go beyond the experiment described in this paper.

Another way of using execution traces is to determine conflicts between transactions [4,3,21]. The purpose of such analysis is to quantify the potential for parallel transaction execution and to find a scheduling well suited for parallel execution. Our experiment shows that the resource consumptions in terms of gas varies depending on the scheduling and must be considered by scheduling algorithms to achieve the best performance. We consider this line of research as future work, which should also incorporate methods from program analysis to better capture the dynamic execution behavior of transactions.

Introducing randomized transaction orderings in practice requires the verifiable and unpredictable generation of a seed value to permute transactions, an approach is presented in [10]. Also, based on our results, further aspects need to be handled appropriately. Reaching the block gas limit, as in 6% of block permutations in the experiment, might be acceptable because block assemblers can manage the risk of block invalidation by putting less transactions in a block, as our data shows. To avoid invalid blocks due to insufficient funds, the block assembler can include at most one transaction per sender, who must have sufficient funds. However, the recently introduced account abstraction [7] requires modifications to this method to guarantee that sending accounts do not lose funds due to other transactions in the block. Economic effects on stakeholders should also be evaluated.

6 Conclusion

In this empirical study, we permuted transactions of Ethereum blocks to quantify the effects of intra-block dependencies regarding protocol violations, execution errors and gas consumption. Invalid block permutations due to protocol violations can be attributed to exceeded gas limits and insufficient funds to settle fees. Permuting transactions does not affect the observed execution error types but they occur slightly more often than in original blocks. The gas consumption

[12] https://tenderly.co/transaction-simulator (2025-05-20)

of most transactions remains constant under permutation, and the gas consumption of blocks is mostly stable as long as the block limit is not exceeded. Privately mined transactions stand out in all analyzed categories. They lead to insufficient fund errors, account for new failing transactions and are responsible for a large share of gas deviating transactions. Based on these findings, we suppose that senders of privately mined transactions trust on a specific order or position of their transactions in blocks while senders of other transactions predominantly do not. Our findings support the technical practicability of randomized transaction ordering. Block producers can manage to not exceed the block gas limit with gas margins and by avoiding privately mining transactions, and eliminate insufficient funds with a content-aware transaction selection strategy. Nonce-related errors are prevented with the present permutation algorithm. However, avoiding privately mined transactions would constitute a strong and disruptive step — one likely to face resistance and require further justification.

Future work might identify applications and usage patterns responsible for gas deviations since they are also relevant for parallel execution.

References

1. Adams, A., Chan, B.Y., Markovich, S., Wan, X.: The Costs of Swapping on Decentralized Exchanges. Financial Crypto (preprint) (2024), https://fc24.ifca.ai/preproceedings/206.pdf
2. Angeris, G., Chitra, T., Diamandis, T., Kulkarni, K.: The Specter (and Spectra) of Miner Extractable Value. arXiv preprint (2023), https://arxiv.org/abs/2310.07865v2
3. Anjana, P.S., Ravi, S.: Empirical analysis of transaction conflicts in ethereum and solana for parallel execution (2025), https://arxiv.org/abs/2505.05358v1
4. Biton, D.D., Friedman, R., Hay, Y.: Ethereum conflicts graphed. IEEE ICBC (2025)
5. Bormet, J., Faust, S., Othman, H., Qu, Z.: BEAT-MEV: Epochless Approach to Batched Threshold Encryption for MEV Prevention. Cryptology ePrint Archive (2024), https://eprint.iacr.org/2024/1533
6. Buterin, V.: A Next-Generation Smart Contract and Decentralized Application Platform. Tech. rep. (2014), https://ethereum.org/en/whitepaper/
7. Buterin, V., Wilson, S., Dietrichs, A., @lighclient: Eip-7702: Set code for eoas (5 2024), https://eips.ethereum.org/EIPS/eip-7702
8. Daian, P., Goldfeder, S., Kell, T., Li, Y., Zhao, X., Bentov, I., Breidenbach, L., Juels, A.: Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. IEEE Symp. S&P pp. 1106–1120 (5 2020). https://doi.org/10.1109/SP40000.2020.00040
9. Droll, J.: Transaction permutation software. Zenodo (2025). https://doi.org/10.5281/zenodo.17228426
10. Droll, J., Stengele, O., Hartenstein, H.: Short Paper: Unpredictable Transaction Arrangement for MEV Mitigation in Ethereum. IEEE ICBC pp. 625–629 (2024). https://doi.org/10.1109/ICBC59979.2024.10634470
11. Google: Shuffle - Golang rand source code, https://cs.opensource.google/go/go/+/refs/tags/go1.22.5:src/math/rand/rand.go;l=247

12. Kalinin, M., Ryan, D.: EIP-4399: Supplant DIFFICULTY opcode with PREVRANDAO (2021), https://eips.ethereum.org/EIPS/eip-4399
13. Killari: eth_multicallV1 (2023), https://mirror.xyz/killaridev.eth/NXR9v4r8b4SWl-hbqJqRQyIAcE4GACobzMObREvL0fo
14. @KillariDev: add 'eth_simulateV1' #484 (2023), https://github.com/ethereum/execution-apis/pull/484
15. Knuth, Donald Ervin: The art of computer programming. Vol. 2: Seminumerical algorithm. Addison-Wesley, 3. ed. edn. (1998), https://zbmath.org/?q=an:0895.65001
16. Lab, S.D.S.: Mempool inslight | privately mined transactions, https://mempool.guru/pmt
17. Li, C.: Gas Estimation and Optimization for Smart Contracts on Ethereum. IEEE/ACM International Conference on Automated Software Engineering, ASE pp. 1082–1086 (2021). https://doi.org/10.1109/ASE51524.2021.9678932
18. Neuder, M., Wahrstätter, T.: Unconditional inclusion lists (2024), https://ethresear.ch/t/unconditional-inclusion-lists/18500
19. Piet, J., Nair, V., Subramanian, S.: MEVade: An MEV-Resistant Blockchain Design. IEEE ICBC (2023). https://doi.org/10.1109/ICBC56567.2023.10174966
20. Resnick, M.: BRAID (2024), https://www.youtube.com/watch?v=mJLERWmQ2uw
21. Sei: 64.85% of ethereum transactions can be parallelized (2024), https://blog.sei.io/research-64-85-of-ethereum-transactions-can-be-parallelized/
22. Soto, D., Bergel, A., Hevia, A.: Fuzzing to Estimate Gas Costs of Ethereum Contracts. IEEE ICSME (2020). https://doi.org/10.1109/ICSME46990.2020.00073
23. Stokes, A., Dietrichs, A., Ryan, D., Swende, M.H., @lightclient: EIP-4788: Beacon block root in the EVM (2022), https://eips.ethereum.org/EIPS/eip-4788
24. @terence: Inclusion List Timing Constraints (2024), https://ethresear.ch/t/inclusion-list-timing-constraints/20198
25. Yakovenko, A.: Solana: A new architecture for a high performance blockchain | Enhanced Reader, https://solana.com/solana-whitepaper.pdf
26. Zarir, A.A., Oliva, G.A., Jiang, Z.M.J., Hassan, A.E.: Developing cost-effective blockchain-powered applications: A case study of the gas usage of smart contract transactions in the ethereum blockchain platform. ACM Trans. Softw. Eng. Methodol. **30**(3) (2021). https://doi.org/10.1145/3431726