Developing Cost-Effective Blockchain-Powered Applications: A Case Study of the Gas Usage of Smart Contract Transactions in the Ethereum Blockchain Platform

ABDULLAH A. ZARIR, EconTech team at Amazon, Canada
GUSTAVO A. OLIVA, Software Analysis and Intelligence Lab (SAIL) at Queen's University, Canada
ZHEN M. (JACK) JIANG, Dept. of Electrical Engineering & Computer Science at York University, Canada
AHMED E. HASSAN, Software Analysis and Intelligence Lab (SAIL) at Queen's University, Canada

Ethereum is a blockchain platform that hosts and executes smart contracts. Executing a function of a smart contract burns a certain amount of gas units (a.k.a., gas usage). The total gas usage depends on how much computing power is necessary to carry out the execution of the function. Ethereum follows a free-market policy for deciding the transaction fee for executing a transaction. More specifically, transaction issuers choose how much they are willing to pay for each unit of gas (a.k.a., gas price). The final transaction fee corresponds to the gas price times the gas usage. Miners process transactions to gain mining rewards, which come directly from these transaction fees. The flexibility and the inherent complexity of the gas system pose challenges to the development of blockchain-powered applications. Developers of blockchain-powered applications need to translate requests received in the frontend of their application into one or more smart contract transactions. Yet, it is unclear how developers should set the gas parameters of these transactions given that (i) miners are free to prioritize transactions whichever way they wish and (ii) the gas usage of a contract transaction is only known after the transaction is processed and included in a new block. In this paper, we analyze the gas usage of Ethereum transactions that were processed between Oct. 2017 and Feb. 2019 (the Byzantium era). We discover that most miners prioritize transactions based on their gas price only, (ii) 25% of the functions that received at least 10 transactions have an unstable gas usage (coefficient of variation = 19%), and (iii) a simple prediction model that operates on the recent gas usage of a function achieves an R-Squared of 0.76 and a median absolute percentage error of 3.3%. We conclude that (i) blockchain-powered application developers should be aware that transaction prioritization in Ethereum is frequently done based solely on the gas price of transactions (e.g., a higher transaction fee does not necessarily imply a higher transaction priority) and act accordingly and (ii) blockchain-powered application developers can leverage gas usage prediction models similar to ours to make more informed decisions to set the gas price of their transactions. Lastly, based on our findings, we list and discuss promising avenues for future research.

CCS Concepts: • **General and reference → Empirical studies**; **Estimation**; • **Computer systems organization → Distributed architectures**.

Additional Key Words and Phrases: Gas Usage, Smart Contracts, Ethereum, Blockchain

Authors' addresses: Abdullah A. Zarir, azzarir@amazon.com, EconTech team at Amazon, Vancouver, Canada; Gustavo A. Oliva, gustavo@cs.queensu.ca, Software Analysis and Intelligence Lab (SAIL) at Queen's University, Kingston, Canada; Zhen M. (Jack) Jiang, zmjiang@cse.yorku.ca, Dept. of Electrical Engineering & Computer Science at York University, Toronto, Canada; Ahmed E. Hassan, ahmed@cs.queensu.ca, Software Analysis and Intelligence Lab (SAIL) at Queen's University, Kingston, Canada.

1049-331X/2020/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Abdullah A. Zarir, Gustavo A. Oliva, Zhen M. (Jack) Jiang, and Ahmed E. Hassan. 2020. 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.* 1, 1 (November 2020), 39 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Ethereum is a popular blockchain platform. Different from Bitcoin, Ethereum supports smart contracts. Smart contracts are general-purpose computer programs that are both hosted and executed by Ethereum. In practical terms, a smart contract can be thought of as a class (as in object-oriented programming) that is instantiated once the contract is deployed in Ethereum. Smart contracts can be written in several languages, including Solidity (the most popular one) and Vyper. Users interact with Ethereum by sending transactions to other users (to transfer cryptocurrency) and to smart contracts (to trigger the execution of a function of a contract). In this study we focus on the latter.

Gas is a measurement unit for computational work in the Ethereum Blockchain. Every computation in Ethereum has an associated cost. The cost is specified in units of *gas*. The gas that is burnt as a result of the execution of a transaction is called the *gas usage* of a transaction. In the case of smart contract transactions, the gas usage depends on the number and type of instructions that are executed during runtime, as well as the pieces of information that need to be stored in the blockchain.

Furthermore, every transaction has to be setup with two parameters before it can be triggered. The first parameter is called *gas limit* and it corresponds to the maximum amount of gas units that are allowed to be burnt by the transaction. An out-of-gas error occurs when the execution of a transaction surpasses the gas limit (i.e., *gas limit* < *gas usage*). The second parameter is called *gas price* and corresponds to the per-unit price of gas, which is given in the Ether (ETH) cryptocurrency. The *transaction fee* paid by a transaction issuer in order to process a transaction corresponds to *gas usage* × *gas price*. Hence, as opposed to typical cloud services in which users choose a compute capacity and pay by the hour (*pay-as-you-go*), transaction fees in Ethereum do not have a fixed value. Gas prices and transaction fees are usually reported in terms of GWEI, where 1 GWEI = 1e-9 ETH.

Mining is a process carried out by a *node* (a.k.a miner) in a blockchain network to provide computational resources to process transactions and append new blocks to the blockchain. The work done by a miner for each transaction is rewarded with an incentive in the form of a cryptocurrency. In Ethereum, the reward for mining a block is equivalent to the *transaction fees* paid by the issuers of the transactions that were included in the block. The transaction fee corresponds to the amount of gas burnt by that transaction (*gas usage)* multiplied by the per-unit price of gas (*gas price*) chosen by the issuer of the transaction. Miners are free to select and prioritize transactions whichever way they wish. More specifically, each miner has a *transaction pool*, where transactions waiting to be processed (a.k.a., *pending transactions*) are kept and ranked according to some prioritization criterion.

A key characteristic of blockchain technology is that it offers trust and security in maintaining the growing list of blocks by leveraging a distributed and decentralized network. These benefits, along with the programmability brought by smart contracts, enabled the development of blockchain-powered applications. These applications use a blockchain platform (e.g., Ethereum) as their backend. For instance, a blockchain-powered bank application could implement financial transactions (e.g., money transfer between accounts) using smart contract transactions.

However, the flexibility of how Ethereum operates poses challenges to blockchain-powered application developers, particularly with regards to the aforementioned gas system. Assume that an end-user wants to transfer some money from his bank account to some other account holder using a blockchain-powered bank application. Once the end-user triggers the money transfer function on the frontend of the application, such a request needs to be translated into one or more smart contract transactions to be processed in the blockchain. The question then becomes: *how should developers set the gas parameters (i.e., gas limit and gas price) of these smart contract transactions?*

In order to answer such a question, we focus on two key requirements:

i) Blockchain-powered application developers need to know how miners prioritize transactions. As we mentioned before, pending transactions can be prioritized by miners in whichever way they wish. Nevertheless, miners commonly employ Ethereum clients (a.k.a., mining tools) such as `geth`[1] and `parity`[2] to mine blocks. These tools provide transaction prioritization strategies out-of-the-box, which often rely on either *gas price*, *gas limit*, or both. Therefore, in RQ1, we empirically investigate how miners prioritize the mining of pending transactions.

ii) Blockchain-powered application developers need to be able to accurately estimate the gas usage of transactions. The reason is twofold. First, accurately estimating the gas usage of a transaction enables developers to set a suitable gas limit for such a transaction (e.g., to avoid running into out-of-gas errors [22]). Second, accurately estimating the gas usage of a transaction enables developers to make a more informed gas price choice (since the transaction fee is a function of both gas usage and gas price). Gas estimation is particularly important in the context of code reuse. More specifically, given the possibly disastrous consequences of having bugs in smart contracts [28], it is common for blockchain-powered application developers to reuse smart contracts from reputable third-parties (e.g., OpenZeppelin[3] instead of writing them from scratch. Nevertheless, the gas usage behavior (e.g., a min-max range) of the functions exposed in the API of reusable contracts is rarely documented. In fact, determining the gas usage of certain smart contract transactions can be remarkably challenging. For instance, a function might burn different amounts of gas depending on the current state of its governing contract. Alternatively, a function might burn different amounts of gas depending on how much data is provided to it via input parameters during a call (e.g., the number of elements in an array) [22]. Given these challenges, gas usage prediction models need to be developed to support the development of *cost-effective* blockchain-powered applications. In RQ2, we study the stability of the gas usage of smart contract functions (e.g., if most smart contract functions have a somewhat constant gas usage, then it becomes trivial to predict their gas usage). In RQ3, we build models to study our ability to accurately predict the gas usage of smart contract transactions.

In summary, in this study we leverage the ledger nature of Ethereum to empirically determine (a) how miners tend to select transactions, (b) the stability of the gas usage of smart contract functions, and, more practically, (c) whether gas usage can be accurately predicted. More generally, our paper stands in the realm of *blockchain-oriented software engineering* (BOSE) [38], which focuses on the application and definition of software engineering principles and practices that are specific for blockchain platforms and their supporting technologies (e.g., smart contracts).

Our research questions, along with the key results that we obtained, are listed below.

RQ1) How do miners prioritize pending transactions? We analyzed the ordering of transactions in each block to understand the common strategies behind block creation. We observed that: *Miners tend to prioritize transactions based solely on gas price, which is the default prioritization*

[1]https://geth.ethereum.org
[2]https://www.parity.io/ethereum
[3]https://github.com/OpenZeppelin/openzeppelin-contracts

strategy of the two most popular mining tools (geth and parity). Hence, two transactions with the same exact transaction fee can have different priorities. For example, if transaction t_1 has a gas usage of 21,000 units and is set with a gas price of 2 GWEI and t_2 has a gas usage of 42,000 units and is set with a gas price of 1 GWEI, then t_1 is more likely to have higher priority than t_2, even though the transaction fee is identical in both cases (i.e., 42,000 GWEI).

RQ2) How stable is the gas usage of smart contract functions? We grouped transactions based on the smart contract functions that they target. Next, we studied functions that received at least a certain minimum number of transactions and analyzed the stability of their gas usage. We observed that: *25% of the studied functions have an unstable gas usage (coefficient of variation higher than 19%). In particular, these unstable functions received together approximately half of all transactions sent to these studied functions.*

RQ3) How accurately can we predict the gas usage of smart contract transactions? We built a simple model that leverages the *recent* historical gas usage of functions in order to produce predictions. More specifically, we predict the gas usage of a given contract transaction as the average gas usage computed over the prior 10 transactions sent to the targeted function. Our hypothesis is that the recent historical gas usage of a function might be relatively stable (e.g., due to seasonality in the contract's state or in the values of input parameters sent to the function). We observed that: *Our simple model can predict the gas usage of contract transactions with an RSquared of 0.76 and a median absolute percentage error of 3.3%.*

A supplementary package with our preprocessed data is available online[4] in an effort to inspire others to further study this crucial aspect of the development of blockchain-powered software applications.

Paper organization. The remainder of this paper is organized as follows. Section 3 explains the data collection procedures that we employed to answer our research questions. Section 4 presents the motivation, approach, and findings of our research questions. Section 5 discusses the implications of our findings. Section 6 provides an overview of prior research on the gas mechanism of Ethereum. Section 7 discusses the threats to the validity of our findings. Finally, Section 8 concludes the paper by summarizing our key observations.

2 BACKGROUND

This section introduces key concepts that are employed throughout this paper. We note that this section builds on portions of an appendix published alongside one of our prior studies [33]. Readers might consider skipping this section (or portions of it) in case they are familiar with smart contracts and the transaction processing mechanism of Ethereum. A summary is provided at the end of the section.

2.1 Blockchain

A blockchain is a distributed, chronological database of transactions that is shared and maintained across nodes that participate in a peer-to-peer network. The name blockchain comes from the manner in which the data is stored. Roughly speaking, pieces of information are packaged in blocks, which are linked to one another as a chain. Blocks store a set of uniquely identifiable transactions (e.g., denoting money transfers). Once a block is appended to the blockchain, its contents cannot be altered without changing every other block that came after it. In Ethereum, a transaction t belonging to block b is deemed final and irreversible after n new blocks have been appended after b. There is no consensus on what the exact value of n should be. For instance, the Ethereum whitepaper

[4]https://github.com/SAILResearch/suppmaterial-18-zarir-ethereum_gas_usage

suggests n = 7 [10], which translates to approximately 01m 45s (since blocks are appended every 15s in average).

The Ethereum platform supports two types of accounts: *user accounts* (a.k.a., externally-owned accounts) and *smart contract accounts*. A user account is very simple in structure. A user account has an address (40-digit hexadecimal ID), a transaction counter, and the ETH balance (ETH is the official Ethereum cryptocurrency). A smart contract account, in turn, holds the bytecode of a smart contract in addition to the previously mentioned fields. Blockchain platforms that support smart contracts accounts are known as *programmable blockchains* (e.g., Bitcoin is *not* a programmable blockchain).

2.2 Smart Contracts

A smart contract is a general-purpose computer program. In practical terms, a smart contract can be thought of as a class (as in object-oriented programming) that is instantiated once the contract is deployed in Ethereum. Just like an instantiated object, a deployed smart contract also has a state.

Smart contracts in Ethereum are frequently written in the Solidity[5] language. Less popular languages include Serpent and Vyper. Only the bytecode of a smart contract is stored in the blockchain (during deployment). The bytecode is executed by the Ethereum Virtual Machine[6] (EVM), which runs on the computers of miners. Miners are the entities that effectively process Ethereum transactions.

2.2.1 Source Code. The syntax of the Solidity language resembles that of Java. A smart contract is similar to a class (as in object-oriented programming). As such, a smart contract contains state variables (a.k.a., attributes) and functions (a.k.a., methods). An illustrative example is shown in Listing 1, which depicts a minimalistic implementation of a coin. Users of this contract can send coins to one another. This illustrative contract was adapted from a slightly more complex example[7] provided as part of the Solidity official documentation.

2.2.2 Interaction Model. Transactions are at the heart of the Ethereum platform. They are the means through which one *interacts* with the blockchain. Similarly to Bitcoin, user accounts can send transactions to other user accounts to transfer cryptocurrency (Ether). However, since Ethereum is a programmable blockchain, user accounts can also send transactions to *deploy* contracts and *interact* with them.

Deployment. The deployment is done by means of a transaction sent to the blockchain. This transaction packs the bytecode that describes the contract (a.k.a., the *runtime bytecode* portion) and initializes it (a.k.a., the *creation bytecode* portion). This transaction is commonly referred to as the *contract creation transaction*. A smart contract receives its address as a result of the execution of the contract creation transaction.

Interaction. Once a contract is deployed, users accounts (e.g., developers of blockchain-powered applications) can interact with it by sending transactions to it. These transactions are known as *contract transactions*. Contract transactions always invoke a function from a smart contract. More specifically, a contract transaction always specifies the *address of the targeted contract*, the *id of the targeted function* (i.e., the function that should be executed), and the *values of the input parameters* (if any) to this function. Since contracts are stateful, a contract transaction may modify the state of a contract.

[5]https://solidity.readthedocs.io/en/v0.6.11/

[6]https://github.com/ethereum/wiki/wiki/Ethereum-Virtual-Machine-(EVM)-Awesome-List

[7]https://solidity.readthedocs.io/en/v0.6.11/introduction-to-smart-contracts.html#subcurrency-example

```
1  // SPDX-License-Identifier: GPL-3.0
2
3  //Compiler version specification
4  pragma solidity >=0.5.0 <0.7.0;
5
6  contract Coin {
7    // State variables
8    address private minter;
9    mapping (address => uint) private balances;
10
11   // Constructor code is only run when the contract
12   // is created
13   constructor() public {
14   minter = msg.sender;
15   }
16
17   // Sends an amount of newly created coins to an address (receiver)
18   // Can only be called by the contract creator
19   function mint(address receiver, uint amount) public {
20     require(msg.sender == minter);
21     require(amount < 1e60);
22     balances[receiver] += amount;
23   }
24
25   // Sends an amount of existing coins (amount)
26   // from any caller to an address (receiver)
27   function send(address receiver, uint amount) public {
28     require(amount <= balances[msg.sender], "Insufficient balance.");
29     balances[msg.sender] -= amount;
30     balances[receiver] += amount;
31   }
32 }
```

Listing 1. A minimalistic implementation of a coin in Ethereum.

The details of a real contract transaction are shown in Figure 1. In the transaction's *input data* field, we can see that the targeted function was `claimFor(address _user)` from the *Digix: Token Sale* contract. In Listing 2, we show the source code of the `claimFor(address _user)` function.

Finally, we note that only functions defined with the *public* or *external* visibility can be the target of a contract transaction. When no visibility is explicitly specified (as in the case of the `claimFor(address _user)` function), the compiler defaults it to *public*. More details about function visibility can be seen in the Solidity's official documentation[10].

Interaction between contracts. Contract transactions are *always* initiated by a user account (i.e., the transaction sender is always a user account). Nonetheless, smart contracts themselves can also (i) deploy other contracts and (ii) invoke functions defined in other smart contracts. These processes occur through a mechanism popularly known as internal transactions. Internal transactions are not real transactions, as they are not kept on the blockchain.

In line 20 of the source code shown in Listing 2, the TokenSales smart contract is invoking two functions (`mint(address, uint256)` and `mintBadge(address, uint256)` that are defined in some other contract. These invocations happen through *internal transactions*.

2.2.3 Gas usage of contract transactions. Transactions in Ethereum need to be paid for. Ethereum uses the *gas system* to charge transaction fees. Cryptocurrency transactions (i.e., transfers of Ether from one user account to another) have a fixed cost of 21,000 gas units. For smart contract transactions, it depends on the number and type of bytecode operations that are executed during runtime (i.e., all bytecode operations associated with the execution of the targeted function). The

[8]https://etherscan.io/tx/0x7d111d21579c484cef5b768a553afb6cc70b448c7de6abcf07172f8d16929e81.
[9]https://etherscan.io/address/0xf0160428a8552ac9bb7e050d90eeade4ddd52843#code.
[10]https://solidity.readthedocs.io/en/v0.6.11/contracts.html#visibility-and-getters

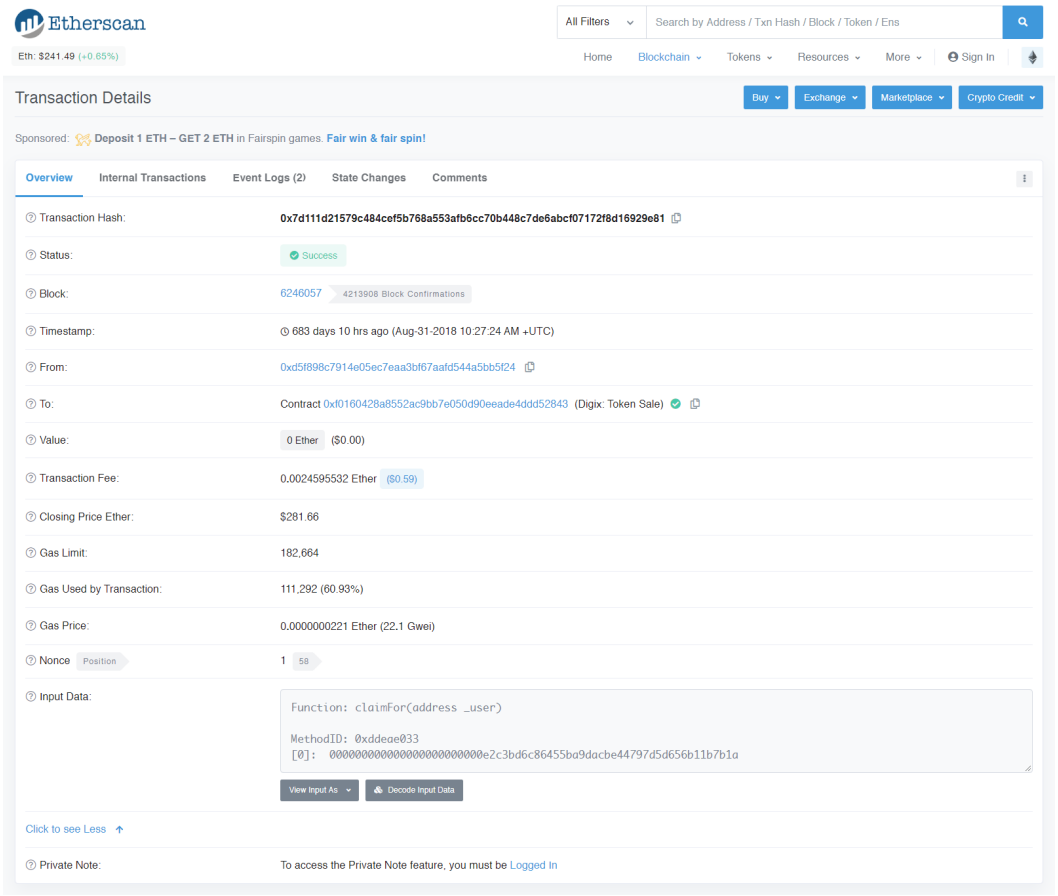


Fig. 1. An example of a smart contract transaction. Image extracted from Etherscan[8].

cost of all bytecode operations are described in the Appendix G of the Ethereum yellow paper [45]. For illustrative purposes, we show such an appendix in Figure 2. For instance, a SHA3 bytecode operation (a hash function) burns 30 gas units (G_{sha3}). Popular operations such as addition (`ADD`), subtraction (`SUB`), number comparison (e.g., `LT`, `GT`), and internal stack operations (e.g., `PUSH*`) are included in the $W_{verylow}$ set of operations, which burn 3 gas units ($G_{verylow}$). In summary, the gas usage of a contract transaction is the sum of the gas cost associated with every bytecode operation that was executed by a miner node in order to run the transaction's targeted function.

As we described in Section 2.2.2, contracts might call other contracts via internal transactions. For instance, if line 20 from Listing 2 happens to be executed as the result of a contract transaction t targeting function `claimFor(address _user)`, then the total gas usage of t will also encompass the gas units burnt to execute the functions `mint(address, uint256)` and `mintBadge(address, uint256)` defined in the external contract. In fact, the transaction shown in Figure 1 does execute line 20. Traces of the internal transactions can be seen by clicking the *Internal Transactions* tab in Etherscan (upper-left portion Figure 1, right beside the *Overview* tab).

```
1  contract TokenSales is TokenSalesInterface {
2    ...
3    // Allows user to claim the DGD tokens and badges if the goal is reached or refunds the
4    // ETH contributed if goal is not reached at the end of the crowdsale
5    function claimFor(address _user) returns (bool success) {
6      if ( (now < saleConfig.endDate) || (buyers[_user].claimed == true) ) {
7        return true;
8      }
9
10     if (!goalReached()) {
11       if (!address(_user).send(buyers[_user].weiTotal)) throw;
12       buyers[_user].claimed = true;
13       return true;
14     }
15
16     if (goalReached()) {
17       address _tokenc = ConfigInterface(config).getConfigAddress("ledger");
18       uint256 _tokens = calcShare(buyers[_user].centsTotal, saleInfo.totalCents);
19       uint256 _badges = buyers[_user].centsTotal / saleConfig.badgeCost;
20       if ((TokenInterface(_tokenc).mint(msg.sender, _tokens)) && (TokenInterface(_tokenc).mintBadge(
             _user, _badges))) {
21         saleStatus.releasedTokens += _tokens;
22         saleStatus.releasedBadges += _badges;
23         saleStatus.claimers += 1;
24         buyers[_user].claimed = true;
25         Claim(_user, _tokens, _badges);
26         return true;
27       } else {
28         return false;
29       }
30     }
31   }
32 }
```

Listing 2. Source code of the `claimFor(address _user)` function. Source code extracted from Etherscan[9].

2.2.4 Gas price, gas limit, and transaction fees. Every transaction has a specified amount of gas that can be consumed for its execution. The amount is set in the *gas limit* transaction argument. The per unit price of gas, or simply *gas price*, is also specified for each transaction. Gas price is set in the Ether (ETH) cryptocurrency. Both gas limit and gas price are set by the issuer of a transaction. If the execution of a transaction requires the burning of more gas than that specified by the gas limit parameter, such a transaction fails with an *out-of-gas* error and gets rolled back. The transaction fee paid by the issuer of the transaction corresponds to *gas usage* $\times$ *gas price*. Fees are also paid for failed transactions, including those that fail with an out-of-gas error. Fees from all the transactions in a block go to the miner who successfully mined the block. Hence, miners' reward depends on *both* the gas price and the gas usage of each transaction (however, only the former is known before executing a transaction).

Appendix G. Fee Schedule

The fee schedule G is a tuple of 31 scalar values corresponding to the relative costs, in gas, of a number of abstract operations that a transaction may effect.

Name	Value	Description*
G_{zero}	0	Nothing paid for operations of the set W_{zero}.
G_{base}	2	Amount of gas to pay for operations of the set W_{base}.
$G_{verylow}$	3	Amount of gas to pay for operations of the set $W_{verylow}$.
G_{low}	5	Amount of gas to pay for operations of the set W_{low}.
G_{mid}	8	Amount of gas to pay for operations of the set W_{mid}.
G_{high}	10	Amount of gas to pay for operations of the set W_{high}.
$G_{extcode}$	700	Amount of gas to pay for operations of the set $W_{extcode}$.
$G_{balance}$	400	Amount of gas to pay for a BALANCE operation.
G_{sload}	200	Paid for a SLOAD operation.
$G_{jumpdest}$	1	Paid for a JUMPDEST operation.
G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
G_{sreset}	5000	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
$R_{selfdestruct}$	24000	Refund given (added into refund counter) for self-destructing an account.
$G_{selfdestruct}$	5000	Amount of gas to pay for a SELFDESTRUCT operation.
G_{create}	32000	Paid for a CREATE operation.
$G_{codedeposit}$	200	Paid per byte for a CREATE operation to succeed in placing code into state.
G_{call}	700	Paid for a CALL operation.
$G_{callvalue}$	9000	Paid for a non-zero value transfer as part of the CALL operation.
$G_{callstipend}$	2300	A stipend for the called contract subtracted from $G_{callvalue}$ for a non-zero value transfer.
$G_{newaccount}$	25000	Paid for a CALL or SELFDESTRUCT operation which creates an account.
G_{exp}	10	Partial payment for an EXP operation.
$G_{expbyte}$	50	Partial payment when multiplied by $\lceil \log_{256}(exponent) \rceil$ for the EXP operation.
G_{memory}	3	Paid for every additional word when expanding memory.
G_{txcreate}	32000	Paid by all contract-creating transactions after the *Homestead* transition.
$G_{txdatazero}$	4	Paid for every zero byte of data or code for a transaction.
$G_{txdatanonzero}$	68	Paid for every non-zero byte of data or code for a transaction.
$G_{transaction}$	21000	Paid for every transaction.
G_{log}	375	Partial payment for a LOG operation.
G_{logdata}	8	Paid for each byte in a LOG operation's data.
G_{logtopic}	375	Paid for each topic of a LOG operation.
G_{sha3}	30	Paid for each SHA3 operation.
$G_{sha3word}$	6	Paid for each word (rounded up) for input data to a SHA3 operation.
G_{copy}	3	Partial payment for *COPY operations, multiplied by words copied, rounded up.
$G_{blockhash}$	20	Payment for BLOCKHASH operation.
$G_{quaddivisor}$	20	The quadratic coefficient of the input sizes of the exponentiation-over-modulo precompiled contract.

Fig. 2. Appendix G of the Ethereum Yellow Paper showing the gas cost (usage) associated with abstract bytecode operations.

Summary

- Ethereum is a programmable blockchain (i.e., it supports smart contracts).
- User accounts interact with a smart contract by sending transactions to it. These transactions are known as *contract transactions*.
- A contract transaction always targets a public/external function from a smart contract. The function id and parameter values are encoded in the transaction input data field.
- The sender of a contract transaction is always a user account.
- The gas usage of a contract transaction t corresponds to the sum of the gas units (cost) associated with each individual bytecode operation that was executed in order to fulfil t.
- Transactions in Ethereum need to be paid for. The transaction fee corresponds to *gas price* × *gas usage* and only the former is known prior to the execution of the transaction.

3 DATA COLLECTION

In this section, we describe the data source that we used (Section 3.1), the rationale behind our choice for the analysis period (Section 3.2), and the data collection steps that we followed in order to answer our research questions (Section 3.3).

3.1 Data Source

Ethereum dataset on Google BigQuery. Google BigQuery is an online platform for the analysis of large datasets. The public Ethereum dataset on Google BigQuery[11] contains several tables that store key blockchain-related data, including metadata of transactions, blocks, and contracts. The dataset is synced daily with nodes in the network that run the `parity` client software.

3.2 Analysis Period

Ethereum has gone through several hard-forks. A hard-fork can be thought of as a new major release of Ethereum, which contains radical changes to the protocol of the blockchain. As shown in Figure 3, the number of received transactions varies significantly per hard-fork. The number of transactions per hard-fork is as follows (excluding Constantinople, which is not finalized yet): Frontier (2,317,095, 0.6%), Homestead (10,672,107, 2.6%), Spurious Dragon (54,967,950, 13.8%), and Byzantium (330,781,145, 83%).

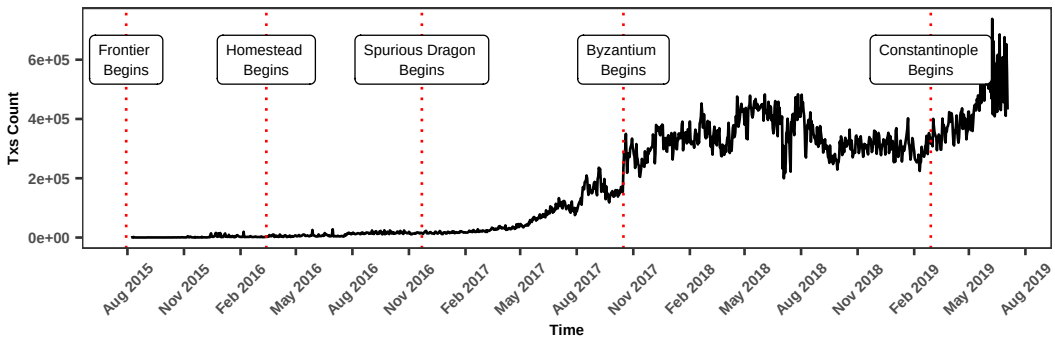


Fig. 3. Daily transaction count of Ethereum Blockchain.

In this study, we analyze transactions from the Byzantium hard-fork only. The Byzantium hard-fork took place on 16 Oct 2017 (Block No 4,370,000) and lasted until 28 Feb 2019 (Block No 7,280,000). The span of the Byzantium era is exactly 500 days. Our justification for analyzing the Byzantium era is threefold. First, this period received the largest amount of transactions. Second, it the most recent finalized hard-fork. And finally, the release of each hard-fork brings new features and architectural changes to the platform that are likely to affect the behavior of users and miners. For instance, the newest hard-fork Constantinople provides native bitwise shifting operations at a low gas cost. Hence, contracts created during the Constantinople era can leverage this function to implement more gas-efficient functions. Hence, by focusing our analysis on a single hard-fork, we ensure that all gas rules remain the same.

3.3 Approach

In this section, we describe our data collection approach. An overview is shown in Figure 4. In the following, we describe each data collection step in more detail:

[11] https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics

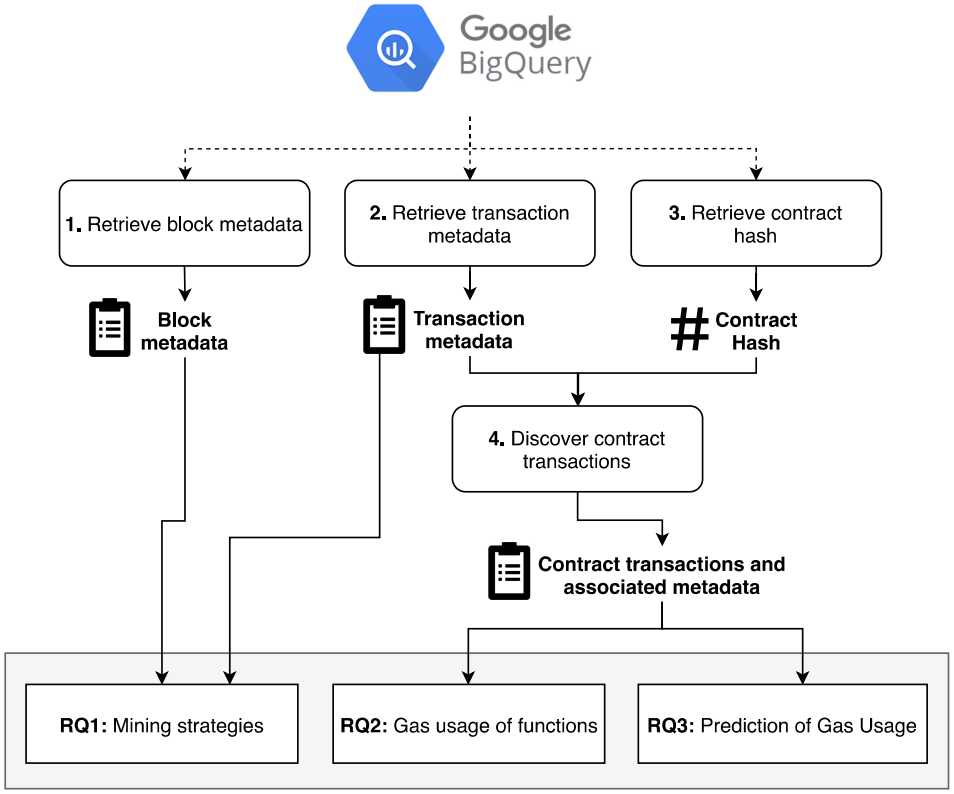


Fig. 4. Flowchart summarizing our data collection approach.

Step 1: Retrieve block metadata. We extract block metadata from the `blocks` table. More specifically, we retrieve block numbers, the address of the miner who mined each block, and the timestamp at which each block was mined. These pieces of information support the investigation block mining strategies (RQ1).

Step 2: Retrieve transaction metadata. We extract transaction metadata from the `transactions` table. More specifically, we retrieve gas-related information (e.g., gas usage, gas limit, and gas price), the transaction nonce (a transaction counter for each transaction issuer), the destination address (which can be either a smart contract or a user account), the input data (which encodes a function call in the case of a contract transaction), the timestamp at which the transaction was mined, and the index (order) of the transaction within the block. Similarly to the block metadata, the transaction metadata also supports answering RQ1. For instance, we investigate the transaction index to understand how miners prioritize pending transactions.

Step 3: Retrieve contract hash. From the `contracts` table, we extract the contract hash (address) of all deployed contracts in Ethereum.

Step 4: Discover contract transactions. We match the contract hashes (retrieved in step 3) with the destination address of transactions (retrieved in step 2) in order to discover contract transactions. We rely on these contract transactions and their associated metadata (e.g., gas usage, gas limit, and

transaction timestamp) to study the stability of the gas usage of contract functions (RQ2) and to build a gas usage prediction model (RQ3).

Summary

- Data collection day: March 1st, 2019.
- Data source: Ethereum dataset on Google BigQuery.
- Data period: Ethereum Byzantium
- Pieces of collected data: Metadata for transactions, blocks, and smart contracts.

4 FINDINGS

In the following, for each research question, we describe our motivation for studying it, the approach that we employed to answer it, and the findings that we observed. In Section 5, we discuss the implications of our findings.

4.1 RQ1: How do miners prioritize pending transactions?

Motivation. Blockchain-powered application developers need to translate requests captured in the frontend of their applications into or more smart contract transactions. In turn, these transactions are processed by miners, who are free to prioritize pending transactions whichever way they wish. Hence, it is imperative that developers know the typical prioritization criterion in use (if any). In particular, developers might want to offer contractual *Quality of Service* (QoS) for their applications. For instance, in the bank example discussed in Section 1, a developer might want to ensure that their blockchain-powered bank application processes 95% of the money transfers in at most 1 minute.

Approach. To determine how miners prioritize transactions, we first started with simple online searches on the Stack Exchange platform. From a post on the Ethereum Exchange, we observed anecdotal evidence that miners prioritize transactions based solely on the gas price of transactions[12]. We then decided to explore the codebase of the two most popular[13] Ethereum clients, namely `geth` (from the Ethereum Foundation) and `parity` (third party). We observed that the default setting, in both tools, is to indeed prioritize transactions solely based on gas price. We refer to this strategy as the *gas price prioritization strategy*.

We decided to investigate more closely how `geth` (the official reference client) implements the gas price prioritization strategy. We noticed that, over the course of the Byzantium era, two versions of the strategy were implemented. These versions differ in how they deal with the *nonce* transaction parameter. The nonce transaction parameter records the number of previously sent transactions by the transaction issuer (i.e., every time someone sends a new transaction, their nonce is increased by 1). This parameter exists to preserve transaction ordering. For instance, a transaction with nonce 3 cannot be mined before a transaction with nonce 2. The two versions of the price ranking strategy differ in how they deal with the nonce parameter. A summary is shown in Figure 5.

Let $T = \langle sender, nonce, price \rangle$ be a tuple that describes a transaction, where *sender* is the address of the transaction issuer, *nonce* is the transaction nonce, and *price* is the transaction gas price. Assume that the list of transactions that need to be ranked is the one shown in step 1 of both versions of the algorithm. In version 1 of the algorithm, step 2 consists of selecting the transactions

[12] https://ethereum.stackexchange.com/questions/1113

[13] The market share of Ethereum clients can be seen in the Etherscan Node Tracker webpage at https://etherscan.io/nodetracker. As of September 30th 2019, `geth` holds 42% of the market share, while `parity` holds 38%.

with the lowest nonce from each sender (a.k.a., the *head* transactions). In step 3 these heads are sorted by price (descending order) and processed. Step 4 consists of repeating steps 2 and 3 until no more transactions are available. In version 2 of the algorithm, steps 1 and 2 are identical to those of version 1. However, in step 3, instead of sorting and processing all head transactions, only the top one is processed. In step 4, the processed transaction is replaced with the subsequent transaction from the same sender (if it exists) and transactions are sorted again by gas price. Steps 3 and 4 are repeated until all transactions are processed.

Fig. 5. The two versions of the gas price ranking method implemented in Geth during the Byzantium period. They differ in how the *nonce* parameter is dealt with.

Therefore, to answer this RQ, we investigate historical transaction data and determine how frequently miners have used the gas price prioritization strategy. We proceed as follows. First, we extract the actual index of each transaction in a given block b as recorded in the Ethereum blockchain. Next, we apply the gas price prioritization strategy of `geth` using the two aforementioned versions of the algorithm and record the transaction indexes for each version. As a result, we obtain three ranked lists of transactions for each block: one with the actual ordering (as recorded in the blockchain), another with the first version of the algorithm, and the last one with the second version of the algorithm. Finally, we say that a block b was sorted based on gas price only if the actual ordering of transactions matches that produced by either version 1 or version 2 of the algorithm. Given the large number of blocks in the Byzantium hard-fork, we draw a statistically representative sample of blocks (99% confidence level, confidence interval of 1) and analyze this sample.

After determining how frequently miners employ the gas price prioritization strategy, we investigate how concentrated block mining is across all miners. We simply calculate the percentage of mined blocks per miner, sort the distribution in descending order, and plot the relationship between the percentage of miners and the percentage of mined blocks.

Findings. Observation 1) ***Two-thirds of the blocks were mined based on the gas price prioritization strategy.*** We observe that 66.5% of the blocks mined during the Byzantium hard-fork

contain a set of transactions whose ordering matches that of the gas price prioritization strategy. We find this result rather surprising, since miners profit from transaction fees and, in turn, the fee of a transaction is calculated as a function of *both* gas price and gas usage. That is, if a transaction has a high gas price, it does not necessarily mean that the transaction fee will be high.

In order to better understand the prevalence of the gas price prioritization strategy, we studied its usage over time. The results are depicted in Figure 6. The red dashed line denotes the aforementioned 66.5% value. Although there seems to be a small downward trend in the usage of the gas price prioritization strategy, we observe that the actual percentage of blocks that are sorted according to this strategy was never lower than 57.4%.

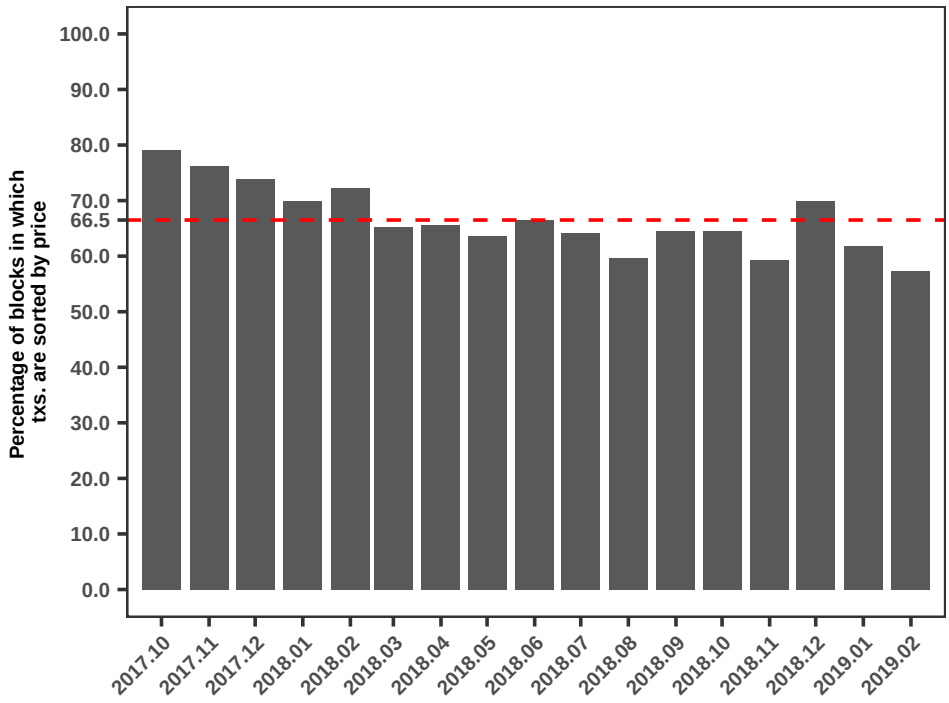


Fig. 6. Usage of the gas price prioritization strategy over time.

Observation 2) ***The manner in which transactions are prioritized is mostly in the hands of only 13 miners.*** From Figure 7, we observe that the ratio of blocks per miner is heavily skewed. In particular, 90% of the blocks were mined by only 1% (13) of the miners. This disproportionate distribution of blocks per miner indicates that a very small group of miners dictated the manner in which transactions were prioritized during the Byzantium era (i.e., the gas price prioritization strategy).

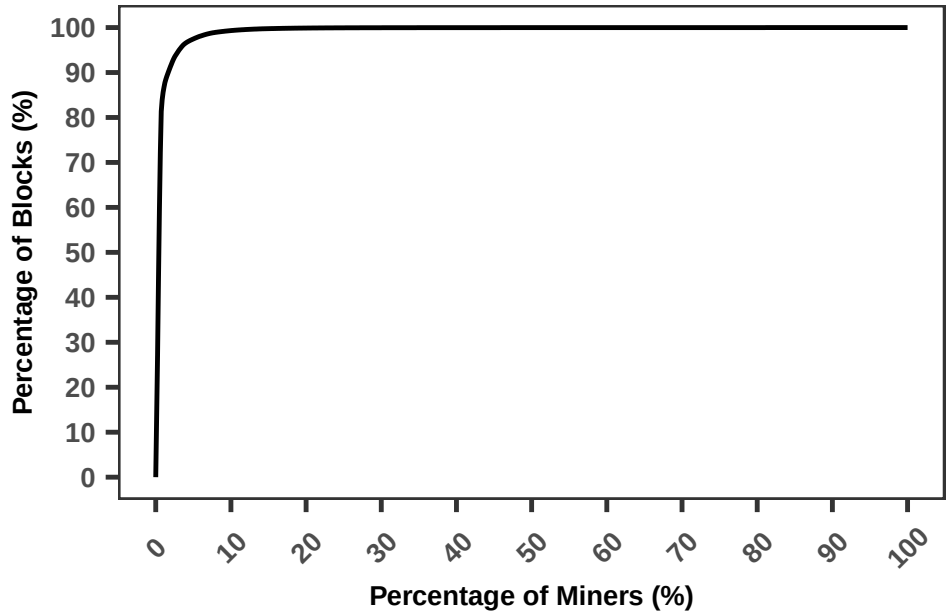


Fig. 7. The percentage of miners corresponding to the ratio of blocks that they mined.

RQ1: How do miners prioritize pending transactions?

Miners prioritize transactions based **exclusively** on their gas price in two-thirds of the cases. In particular:

- The gas price prioritization strategy is the default transaction prioritization strategy for both `geth` and `parity`.
- A very small group of miners (13) mined 90% of the blocks. Consequently, this small group dictated how transactions were prioritized during the Byzantium era.

4.2 RQ2: How stable is the gas usage of smart contract functions?

Motivation. From RQ1, we observed that miners frequently prioritize transactions based on their gas prices. Therefore, setting a proper gas price is crucial for the development of cost-effective blockchain-powered applications. That is, gas prices should be set in such a way that the trade-off between transaction processing times (which influences the end-user final experience) and transaction cost (which influences the cost of offering a blockchain-powered application) is optimized.

The fee of a transaction corresponds to *gas price* $\times$ *gas usage*. Therefore, one has to reason about gas usage in order to set the gas price parameter. Nevertheless, the actual gas usage of a transaction can only be known after such a transaction is processed. This cyclic dependency makes it challenging for blockchain-powered application developers to set optimal gas prices. As we mentioned in the introduction (Section 1), this challenge is particularly relevant when blockchain-powered application developers wish to integrate third-party smart contracts into their application.

To overcome the aforementioned challenge, gas usage prediction models are needed. As a first step, in this research question we investigate the stability of the gas usage of smart contract functions. If the gas usage is stable for most functions, then devising a prediction model becomes trivial. On the other hand, if gas usage tends to be unstable for most functions, then devising a prediction model becomes more complex.

Approach. First, we determine the stability of the gas usage of each function. We operationalize the notion of stability using the *coefficient of variation* (CoV) metric. The CoV is defined as the ratio of the standard deviation to the mean. The CoV can be thought of as a normalized standard deviation. Such a property allows us to compare the stability of gas usage across different functions. We study functions that received at least 10 transactions.

Next, we analyze the distribution of gas usage CoV per function. After reasoning about gas usage stability, we investigate how it relates to the popularity of a function. We define popularity as the number of received transactions for a function.

Finally, we perform statistical tests to compare distributions. If we observe a statistically significant difference, we compute the Cliff's Delta (δ) effect-size score to better understand the practical significance of this difference. We assess Cliff's Delta using the following thresholds [39]: *negligible* for $|\delta| \leq 0.147$, *small* for $0.147 < |\delta| \leq 0.33$, *medium* for $0.33 < |\delta| \leq 0.474$, and *large* otherwise.

Findings. Observation 3) ***50% of the functions have a stable gas usage, 25% have an unstable gas usage, and the remaining 25% are inconclusive.*** Figure 8 shows the distribution of gas usage CoV per function using a violin plot.

The blob in the left-most part of the violin plot indicates that several functions have a gas usage CoV close to zero (i.e., their gas usage is almost constant). More specifically, 50% of the functions have a gas usage CoV that is less than or equal to 0.8%. As an illustrative example, the gas usage distribution shown below belongs to a studied function whose gas usage CoV is 0.8% (note how values are almost constant):

- Function 0xa9059cbb from contract 0x88f70a4aadfe8898d7942944c2d5263f771edc1f
GasUsage(0xa9059cbb) = {52682,52554,52810,53649,53649,52682,52682,52682,52682,52682}

Around the third quartile (19%), we notice the existence of another blob. At this point, there is already a more perceivable variation in the gas usage. As another illustrative example, the gas usage distribution shown below belongs to a studied function whose gas usage CoV is 19% (note the two spikes in gas usage):

- Function 0x23b872dd from contract 0x2d36e20eae9182f1853788bfc341ad433a311dea
GasUsage(0x23b872dd) = {30199,30199,**45199**,30199,30199,**45135**,30199,30199,30135,30199}

Indeed, the distribution shown in Figure 8 has a large amplitude, containing also very large numbers (maximum = 635.5%). These large numbers denote functions with extremely unstable gas usage. Hence, we conclude that 25% of the functions have an unstable gas usage (i.e., those above Q3).

We consider the interval from 0.8% (median) until 19% (Q3) to be a grey area. Functions with a CoV close to 0.8% are very stable, while functions with a CoV close to 19% already show perceivable variation. In addition, due to the very definition of the *coefficient of variation*, it is not possible to derive one simple characterization for the distributions that lie in the aforementioned grey area. For instance, the following three studied functions received exactly 10 transactions and have a gas usage CoV of 11% ± 0.3%. Yet, their gas usage distributions have different characteristics:

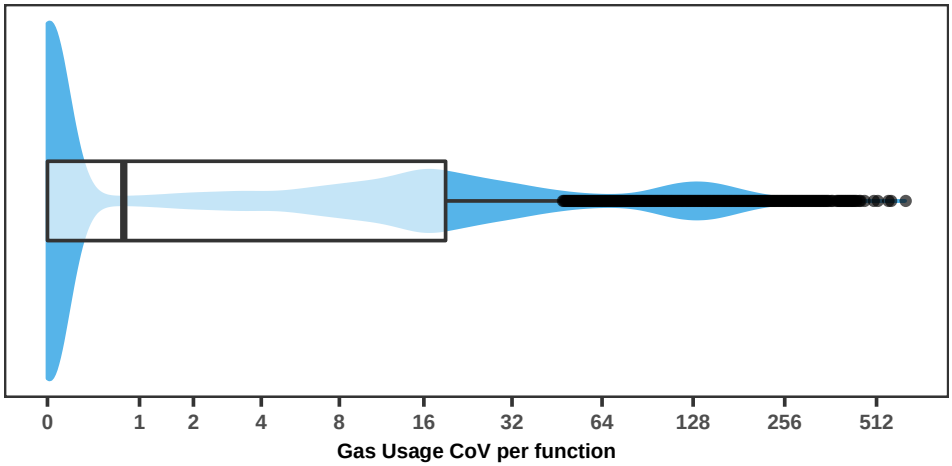


Fig. 8. Gas usage CoV of all studied functions (i.e., those that received at least 10 transactions). Horizontal axis uses a log(1+x) scale.

• Function 0x9dabff25 from contract 0x3d1f1573e757c66f2c3517d1b1713c5ed3b06fcc. This function has a smooth initial decline in gas usage and then becomes constant:
GasUsage(0x9dabff25) = {**28637**,**25131**,**23148**,**18055**,25131,25131,25131,25131,25131,25131}

• Function 0x59d667a5 from contract 0x634b07a0959599a81ce33a04800f81a341bd70a1. This function has *fluctuating* gas usage around the median (104556):
GasUsage(0x59d667a5) = {132169,104159,119492,91482,117233,104556,117233,104556,102169,101275}

• Function 0x6ea056a9 from contract 0x60cd2a6e8547b4ef92e22136ced200dcd1ab99ee. The gas usage distribution of this function has repeated values:
GasUsage(0x6ea056a9) = {**49993**,**49993**,**49993**,**49993**,37837,50057,46569,**41159**,**41159**,**41159**}

Observation 4) ***Unstable functions received approximately half of all transactions sent to the studied functions.*** Given the wide range of gas usage CoV, we wanted to evaluate how such variable relates to function popularity. This relation is explored in Figure 9 by means of a heatmap. We choose a heatmap in order to handle the excessive overplotting (i.e., several observations with the same (x,y) coordinate). The dashed black lines serve as reference points that separate functions into groups that have unstable, inconclusive, and stable gas usage CoV.

A visual inspection of Figure 9 reveals that the gas usage CoV of a function is not associated with popularity. We conduct a two-tailed Mann-Whitney test ($\alpha = 0.05$) to determine whether unstable functions (i.e., those above the upper dashed line) have different popularity compared to stable functions (i.e., those below the lower dashed line). The result indicates that there is a statistically significant difference (p-value < 2.2e-16). Yet, a calculation of Cliff's Delta reveals that the magnitude of the difference is negligible (0.11). Therefore, we conclude that stable and unstable functions have similar popularity. That is, unstable functions are not being particularly ignored or abandoned. Indeed, despite the general concentration of functions in the left-most part of the map, there are unstable functions with very high popularity (e.g., more than 100k received transactions).

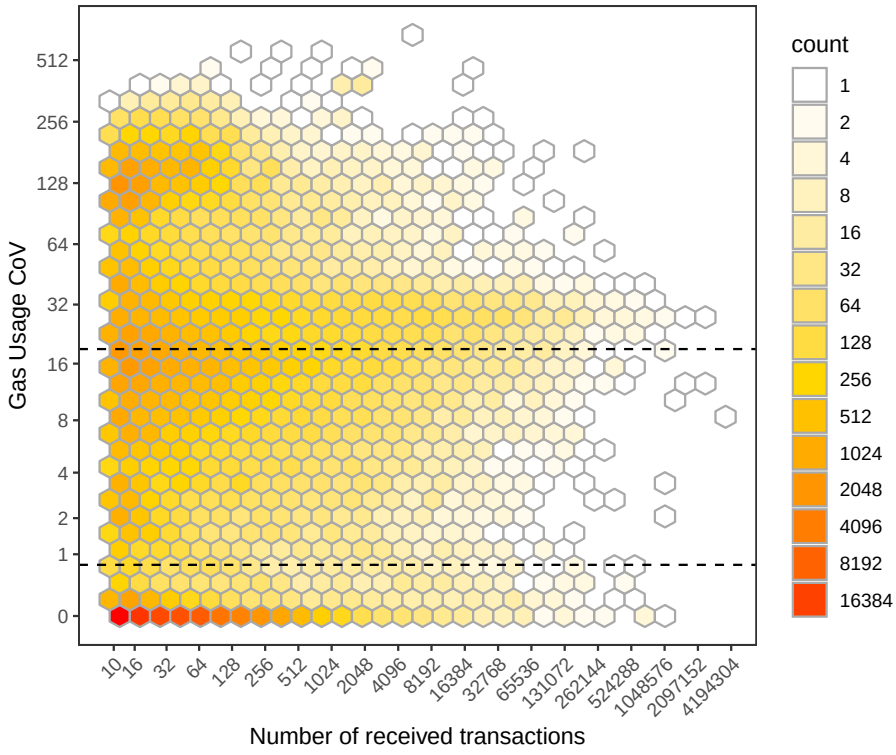


Fig. 9. Number of received transactions (popularity) vs gas usage CoV. Both axes use a log(1+x) scale.

RQ2: How stable is the gas usage of smart contract functions?

Of all studied functions, we conclude that 50% of them have a stable gas usage, 25% have an unstable gas usage, and the remaining 25% are somewhere in between. Hence, it is not trivial to predict the gas usage of all studied functions. We also note that:

- Functions that we classified as stable have a gas usage CoV lower than or equal to 0.8%
- Functions that we classified as unstable have a gas usage CoV higher than or equal to 19%.
- Unstable functions are as popular as stable functions.

4.3 RQ3: How accurately can we predict the gas usage of smart contract transactions?

Motivation. Observations from RQ2 indicate that several functions are popular yet unstable in terms of gas usage. Such instability makes it harder for blockchain-powered application developers to predict the gas usage of contract transactions sent to those functions.

Approach. The gas usage of a transaction depends on which bytecode instructions are executed during runtime, which in turn depends essentially on the values of the input parameters of the function and the state of the contract (e.g., attribute values). However, decoding input parameters requires access to the smart contract ABI, which is not always publicly available on platforms such as Etherscan. The ABI (application binary interface) defines how functions should be called, i.e., how the transaction input data (a hex string) translates into a function call with specific parameter

```
pragma solidity ^0.4.21;

contract AuctusWhitelist {

  ...
  mapping(address => WhitelistInfo) public whitelist;

  ...
  struct WhitelistInfo {
    bool _whitelisted;
  bool _unlimited;
    bool _doubleValue;
  bool _shouldWaitGuaranteedPeriod;
  }

  ...
  function listAddresses(bool whitelisted, bool unlimited, bool doubleValue, bool shouldWait, address[]
        _addresses) public {
    require(canListAddress[msg.sender]);
    for (uint256 i = 0; i < _addresses.length; i++) {
      whitelist[_addresses[i]] = WhitelistInfo(whitelisted, unlimited, doubleValue, shouldWait);
    }
  }
  ...
}
```

Listing 3. Excerpt of the AuctusWhitelist smart contract.

values and types. In addition, reading the state of a given contract and tracing its changes over time becomes particularly challenging when the source of the contract is not available. Hence, in this RQ, we explore a lightweight prediction model that does not rely on the contract code.

We hypothesize that *there might be historical patterns in how functions burn gas*. For example, assume that a function adds an element to a list and then performs an operation on each element of this list. Thus, each new execution of this function burns more gas than the previous execution. If we analyze the whole history of this function, we might obtain a high gas usage CoV. If we instead analyze only the recent history of this functions (e.g., the last 10 executions), we might obtain a substantially smaller CoV. For example, the CoV of a hypothetical distribution $D = \{1, 2, ..., 100\}$ is 57.4%. In contrast, the CoV of a hypothetical distribution $D_{recent} = \{90, 91, ..., 100\}$ is 3.5%. As a consequence, predicting the gas usage of a function based on its *recent* gas usage history might yield accurate results.

Historical patterns in how functions burn gas might arise due to how transaction issuers interact with a function (and not because of the function implementation itself). As a concrete real-world example, consider the AuctusWhitelist[14] smart contract shown in Listing 3. The function `listAddresses` is simple. For each address in the array of addresses (`_addresses` parameter), a `WhitelistInfo` object is instantiated and then stored in a key-value map (key is the address and value is the `WhitelistInfo` object). Hence, the higher the number of addresses passed to the function, the larger the map gets. In other words, the higher the number of addresses passed to the function, the higher the gas usage of the transaction.

Figure 10 depicts the historical gas usage of the `listAddresses` function. Analysis of the curve indicates that the first transactions sent to the function had a remarkably high gas usage (frequently in the range of 2.8M gas units). Starting from the 80th transaction, the vast majority of transactions had a gas usage of only 60.8k gas units. Upon closer manual inspection, we observed that the older transactions were providing a large array of addresses to the function, whereas the more recent transactions commonly included only one address in the array. It is thus clear that there is a historical pattern in how the `listAddresses` function has been used and, consequently, in how it has burnt gas. In particular, a gas usage CoV computed over a few prior transactions is going

[14]https://etherscan.io/address/0xa6e728e524c1d7a65fe5193ca1636265de9bc982#code

to be much lower than a gas usage CoV computed over all past transactions sent to this function. In other words, a gas usage prediction model that operates on *recent* gas usages (e.g., those from a few prior transactions) would likely provide accurate estimates for the `listAddresses` function.

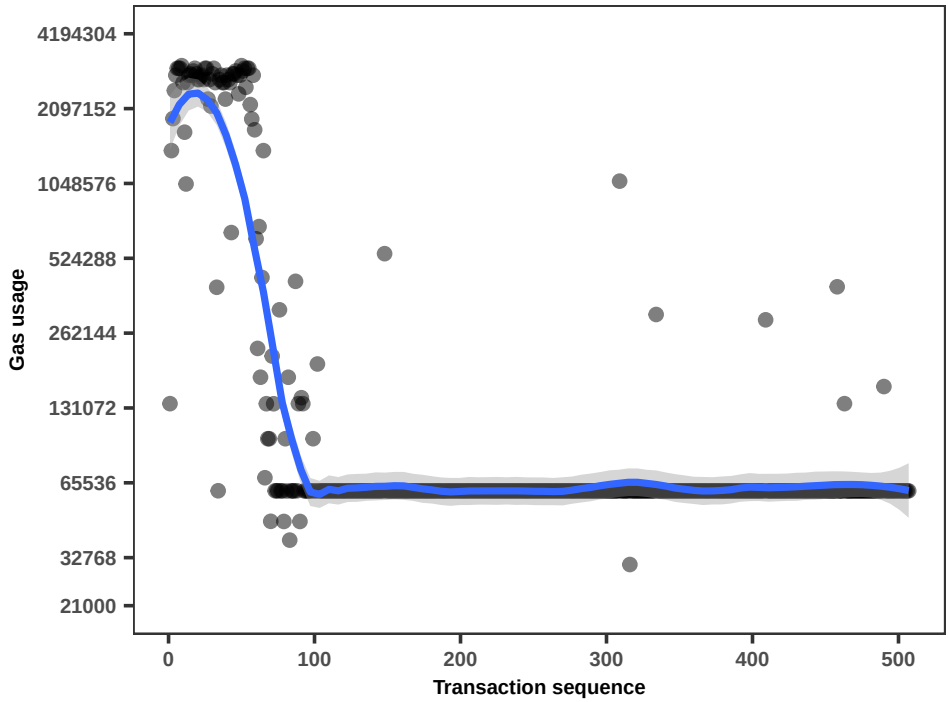


Fig. 10. Line plot depicting the historical gas usage of the function (smoothing applied using loess). The y-axis is on a log10 scale.

In light of the aforementioned discussion, in this research question we investigate whether the *recent gas usage* of a function f can be used to accurately predict the gas usage of an upcoming transaction sent to f. We thus build a simple model that predicts the gas usage of a contract transaction as the mean gas usage computed over the prior 10 transactions sent to the targeted function. Formally, for a targeted function f at a transaction t_j, our predicted gas usage is:

$$\widehat{gas\_usage}(f, t_j) = \frac{\sum_{i=j-10}^{j-1} gas\_usage(f, t_i)}{10} \quad (1)$$

We emphasize that such a prediction model also leverages the fact that some functions simply have a very stable gas usage. Indeed, in RQ2 we observed that approximately half of the functions that received at least 10 transactions have an almost constant gas usage. From a computational perspective, however, there are corner cases that we need to deal with. First, we only take into account the prior 10 transactions that have already been processed (i.e., that have already been included in blocks), otherwise our model would be unrealistic. If the history of already processed transactions of a given function includes k transactions and $k < 10$, then we compute the average over these k transactions. For the very first transaction sent to a function (i.e., when no transactional history exists), we output the gas limit of the transaction as the predicted gas usage.

Since our simple model solely depends on historical gas usage, it can be applied to any smart contract function (regardless of whether its source is available or not). Hence, to evaluate our

approach, we consider all contract transactions of the Byzantium period. The only exception is failed transactions. A failed transaction results in premature termination of the function execution, commonly resulting in much smaller gas usage. We interpret these failed transactions as a confounding factor and we thus preemptively remove them before conducting our study. The final dataset resulted in 161.6M transactions.

We evaluate the accuracy of our model as follows. For each transaction, we calculate the *absolute percentage error* (APE). This metric corresponds to $\frac{|actual-predicted|}{actual} * 100$ and indicates how off a prediction is compared to the actual value. In our context, actual refers to the blockchain-recorded gas usage and predicted refers to our prediction of gas usage (Equation 1). Next, we perform two analyses. In our first analysis, we simply investigate the APE distribution, which indicates how accurate the model is from a *global perspective*. We also estimate its goodness of fit using the *RSquared* measure. In our second analysis, we group transactions based on their target function. Afterwards, for each function, we calculate its median APE. Subsequently, we analyze the distribution of median APEs. This latter analysis indicates how accurate the model is for each function in our studied sample.

Lastly, in order to better understand the benefit of using *recent* gas usage to predict current gas usage, we compare our results to those produced by a baseline model. We define this baseline model as one that predicts gas usage as the mean gas usage computed over the entire history of transactions (i.e., all prior already-processed transactions in lieu of the prior 10 only). Similarly to RQ2, when convenient, we perform statistical tests and compute effect sizes.

Findings. Observation 5) ***From a global perspective, recent gas usage is a remarkably good predictor of gas usage (RSquared = 0.76, median APE = 3.3%).*** The APE distribution of the recent gas usage model is illustrated by the left-most violin plot in Figure 11. We note that 50% of the predictions are at most 3.3% off. As another point of reference, 75% of the predictions are at most 16.9% off. Nevertheless, we observe outliers in the distribution, indicating that the model performs poorly for a few transactions.

Visual analysis of Figure 11 indicates that the recent gas usage model performs better than the all-history gas usage model. A two-tailed Wilcoxon test ($alpha = 0.05$) indicates that there is a statistically significant difference between the distributions (p-value < 2.2e-16), which is not surprising given the very large size of our distributions. A Cliff's Delta effect size test indicates that the magnitude of the difference is small (-0.21), albeit non-negligible.

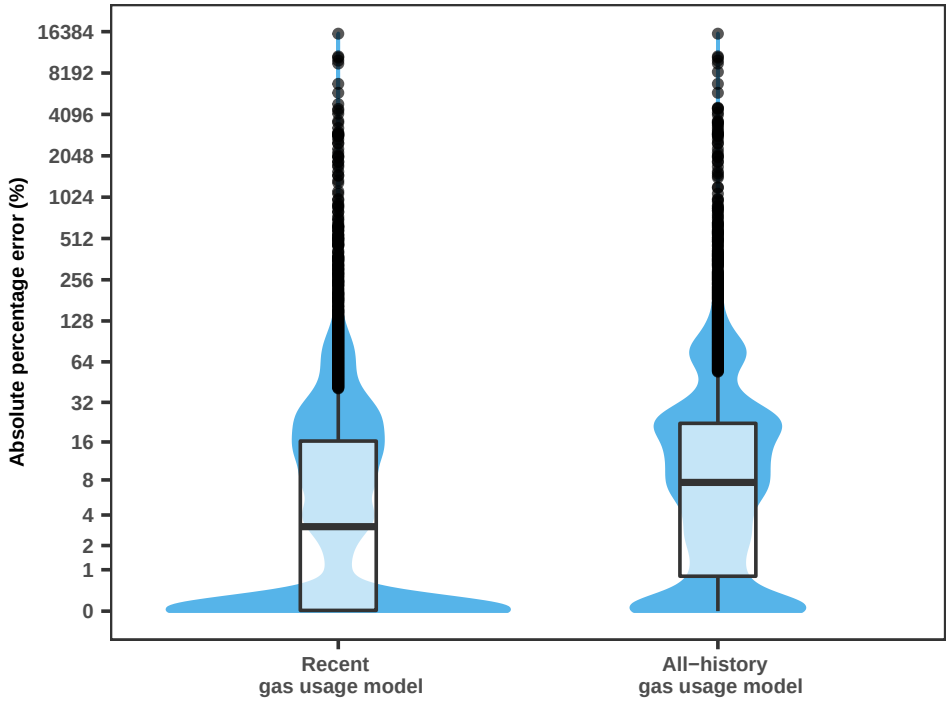


Fig. 11. Violin plot depicting the APE distribution.

Observation 6) ***The recent gas usage model performs well for functions that received at least 3 transactions (median APE is at most 11.6% for 75% of these functions).*** The prediction accuracy per function is depicted in Figure 12. Analysis of the figure reveals that the two models perform poorly.

In order to better understand why the models perform poorly, we control for function popularity (i.e., their number of received transactions). The results are depicted in Figure 13. We reach the following conclusions from analyzing the figure: (a) the two models perform similarly, (b) the performance of the models increase drastically when functions with less than 3 transactions are discarded, and (c) the performance stabilizes when functions with at least 3 transactions are considered. In particular, conclusion (b) implies that the model performs poorly for functions with less than 3 transactions and that many of the functions have less than 3 transactions. Indeed, further investigation revealed that 55.1% of the transactions received only one transaction and 17.3% of the functions received 2 transactions (i.e., 72.4% of the transactions received less than 3 transactions). In both models, we use the gas limit as a prediction for the very first transaction received by a function. Our results thus suggest that gas limit is a poor predictor for gas usage (more details in Section 4.3.1).

For functions that received at least 3 transactions, we observe that the two models perform equally well. Their median APE is at most 12% for 75% of these functions. The reason why the two models perform identically for these functions is twofold. First, several of these functions have a very stable gas usage (as we observed in RQ2). Second, the distribution of received transactions per function is extremely skewed (Figure 14). That is, the vast majority of functions received very few transactions. By construction, the models perform identically for functions that received at most ten transactions. Therefore, in order to better understand how the models compare to each

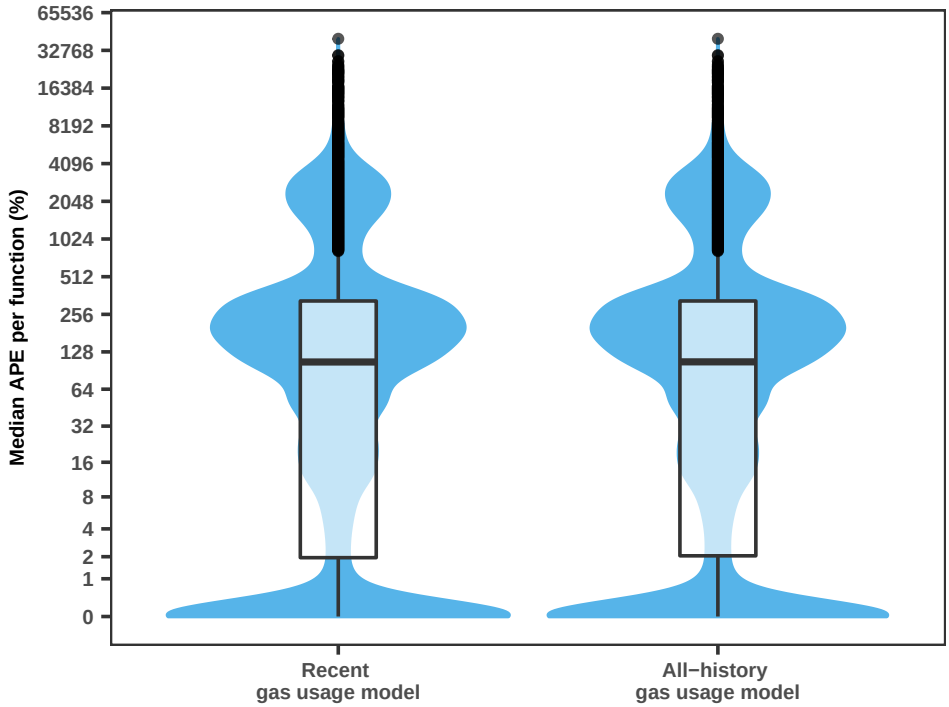


Fig. 12. Violin plot depicting the median APE per function.

other, we filtered in only unstable functions (i.e., those with gas usage CoV higher than 19%) and compared the performance of the two models across several degrees of function popularity. The results are shown in Table 1.

Table 1. Comparison of the two models for unstable functions and different degrees of function popularity.

Number of transactions $\geq t$	**Number of functions**	**Median model performance** *Recent history*	*All history*	**Statistically significant perf. difference** ($\alpha < 0.05$)	**Effect Size (Cliff's Delta)**
t = 3	162983	67.6	60.7	Yes	negligible (0.02)
t = 4	125761	58.4	53.8	Yes	negligible (0.01)
t = 8	62407	36.3	39.4	Yes	negligible (-0.05)
t = 16	35805	23.3	28.5	Yes	negligible (-0.11)
t = 32	22683	18.5	24.6	Yes	small (-0.17)
t = 64	14798	15.9	21.2	Yes	small (-0.24)
t = 128	9874	14.2	19.4	Yes	small (-0.30)
t = 256	7201	13.0	18.9	Yes	medium (-0.35)
t = 512	5236	12.2	18.4	Yes	medium (-0.40)
t = 1024	3860	12.0	18.3	Yes	medium (-0.42)
t = 2048	2781	11.9	18.0	Yes	medium (-0.44)
t = 4096	1915	11.9	18.0	Yes	medium (-0.45)
t = 8192	1235	12.0	18.4	Yes	medium (-0.46)
t = 16384	754	12.1	18.6	Yes	medium (-0.45)
t = 32768	406	12.6	18.7	Yes	medium (-0.44)
t = 65536	181	14.2	18.3	Yes	medium (-0.37)
t = 131072	90	15.7	18.5	Yes	medium (-0.37)
t = 262144	32	14.4	17.5	Yes	medium (-0.38)

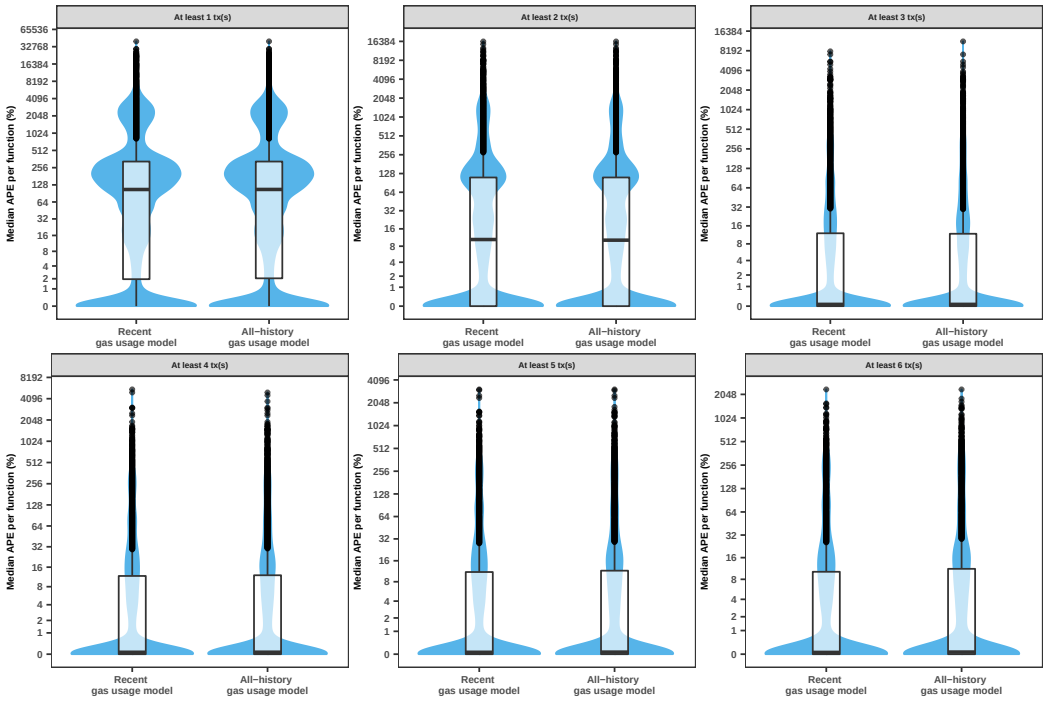


Fig. 13. Violin plot depicting the median APE per function (controlled by function popularity).

Table 1 indicates that the recent history model outperforms the all-history model for unstable functions that received at least 32 transactions (small effect size). The difference in performance becomes larger for functions that received at least 256 transactions (medium effect size). This result provides evidence that indeed there are patterns in how certain functions burn gas.

4.3.1 On the predictive power of gas limit. As described in Section 2, gas limit is a parameter that needs to be set by transaction issuers for every transaction. Gas limit corresponds to the maximum amount of gas units that are allowed to be burnt by a transaction.

We used the gas limit to predict the gas usage of the first transaction sent to a function. Yet, the left-most violin plots shown in Figure 13 suggests that gas limit is a poor predictor of gas usage (despite the intrinsic relationship between the two).

We built a trivial model that takes the gas limit of a transaction as input and outputs this very same limit as the predicted gas usage. In other words: $\widehat{gas\_usage} = gas\_limit$.

Observation 7) ***Gas limit is a poor predictor of gas usage (RSquared = -4.72, median APE = 96.62%).*** The APE distribution is shown in Figure 15. Analysis of the figure indicates that 75% of the predictions are at least 39% off. That is, the gas limit parameter set by transaction issuers is commonly not very "tight". As a consequence, gas limit cannot by itself predict gas usage accurately. Indeed, the RSquared of the model is -4.72.

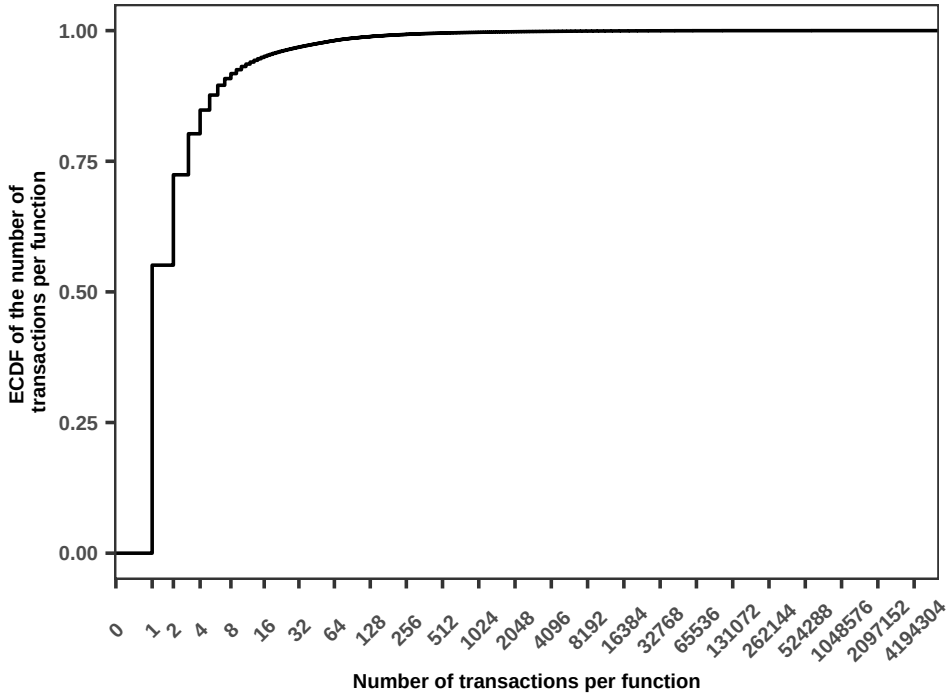


Fig. 14. Empirical cumulative distribution (ECDF) of the number of received transactions per function.

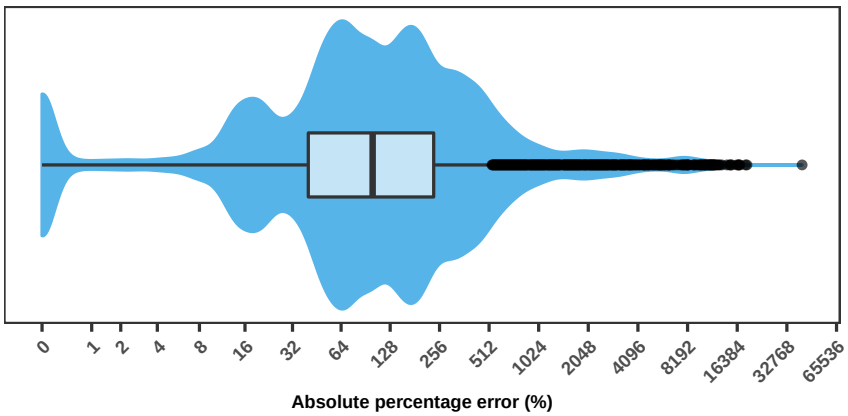


Fig. 15. Violin plot depicting the APE distribution of the gas limit model.

RQ3: How accurately can we predict the gas usage of smart contract transactions?

We can predict the gas usage of contract transactions with an RSquared of 0.76 and a median absolute percentage error (APE) of 3.3% by means of a very simple model that relies on the recent gas usage of functions. We also observe that:

- Our recent history model performs well for functions that received at least 3 transactions (median APE is at most 11.6% for 75% of these functions).
- Our recent history model outperforms the all-history model (baseline) for unstable functions that received at least 32 transactions (small effect size). The difference in performance becomes larger for functions that received at least 256 transactions (medium effect size).
- The gas limit is a poor predictor of gas usage (RSquared = -4.72, median APE = 96.2%)

5 DISCUSSION

5.1 Implications

In the following, we discuss the implications of our findings. We focus on developers of blockchain-powered applications (Section 5.1.1), but also briefly discuss implications to other stakeholders as a side product of this study (Section 5.1.2).

5.1.1 Implications to Developers of Blockchain-Powered Applications. **Implication 1) *Blockchain-powered application developers should be aware that transactions tend to be prioritized solely based on their gas prices.*** Observation 2 notes that the majority of blocks are mined based on the default strategy offered out-of-the-box by `geth` and `parity`, which consists of prioritizing transactions solely based on gas price. Such an observation has strong practical consequences for blockchain-powered application developers. Assume that Charlie developed a blockchain-powered bank application. Also assume that a money transfer operation from one bank account to another is done in the blockchain, which involves submitting a transaction to a smart contract function that always burns a large amount of gas units. In this scenario, Charlie cannot engineer or configure the application to always issue that transaction with a very high gas price, otherwise the transaction could end up costing too much and rendering the bank application economically unviable. Given that miners frequently rank transactions based on gas price only, Charlie's cheaper-priced transaction is likely going to sit in pending pools for a certain time (despite the large revenue that a miner would get from processing such a heavyweight transaction).

To circumvent the problem, blockchain-powered application developers should thus *avoid developing (or reusing) functions that burn high amounts of gas* (e.g., those that perform too much computation or store too many values), as sending transactions with high gas price to these functions might be economically unfeasible in practice. Instead, blockchain-powered application developers should consider engineering applications in such a way that an application request can be converted into multiple smart contract transactions that (i) are set with high gas prices and (b) target one or more low-gas-usage functions.

Implication 2) *Blockchain-powered application developers can leverage a simple gas usage prediction model to facilitate integration with third-party smart contracts.* In observation 3, we note that 50% of the studied functions (i.e., those that received at least 10 transactions) have a very stable gas usage (CoV $\leq$ 0.8%). Therefore, one can get a very precise estimate of the gas usage for many functions by simply checking their last gas usage. Yet, we also note that 25% of the studied functions have an unstable gas usage (CoV $\geq$ 19%). Most importantly, we note in observation 4 that these unstable functions account for approximately half of all contract transactions that were sent to the studied functions. Hence, if our blockchain-powered application developer, Charlie, wants to integrate any of these smart contract functions into an application, then Charlie will have a hard time determining the optimal gas price to be used in transactions towards these functions (since the gas usage of these functions is not only unknown *a priori* but also unstable).

In observation 6, we note that a simple prediction model based on *recent* gas usage can already achieve a median APE per function that is at most 11.6% for 75% of all functions that received more than 2 transactions. We highlight that our model does not rely on source code features, thus it can be built even for non-verified smart contracts. Our model is also conceptually straightforward, as checking the recent gas usage of a function is something that practitioners might consider doing in practice when gas consumption is undocumented or when the source code of the governing contract is unavailable. Therefore, Charlie can leverage models similar to ours to predict the gas usage of the transactions that need to be submitted to third-party contract functions (even if the

source code of these contracts is not available). Having good gas usage estimates will help Charlie determine an optimal gas price.

Implication 3) ***Blockchain-powered application developers should be aware that Ethereum is not as decentralized as they might think.*** Observation 1 indicates that there is a heavy centralization of the mining activity. Despite claims regarding the decentralized nature of Ethereum and its peer-to-peer network, the decisions dictating which transactions should be processed first are in the hands of a few mining clusters. Therefore, in practice, Ethereum is less decentralized than most people might think. Consequently, blockchain-powered application developers should be aware that the prioritization of transactions and, more generally, the dynamics of transaction processing in Ethereum are mostly governed by only 13 miners.

5.1.2 Implications to other stakeholders. In this paper, we focus on blockchain-powered application developers. Yet, as a side-result of our study, we also derive implications to other stakeholders, namely miners and end-users. These implications are discussed below.

Implication 4) ***Miner's revenue is suboptimal.*** From the miner perspective, relying solely on gas price yields suboptimal rewards. This result follows by construction, since the mining reward is a function of both gas price and gas usage. Therefore, miners could also leverage our gas usage prediction model discussed in RQ3 as a cheap way to estimate the gas usage of a contract transaction, ultimately optimizing their transaction ranking strategy and increasing their revenue. We also note in observation 7 that gas limit is a poor predictor of gas price. We conjecture that this is the reason why miners rank transactions mostly based on the gas price only (i.e., in lieu of employing the other out-of-the-box transaction ranking strategies that rely on gas limit).

Implication 5) ***End-users of simple blockchain-powered applications should also be aware of how the gas system operates and how transactions are prioritized.*** Simple blockchain-powered applications expose the complex blockchain interaction model to the end-users. End-users of games like CryptoKitties[15] have to install a wallet such as Metamask and submit blockchain transactions themselves. Most importantly from the perspective of our study here, users also have to fill in the gas parameters. Hence, implications 2 and 3 discussed in the previous section also apply to these end-users. Advanced end-users can even leverage the model discussed in RQ3 to make more informed decisions about gas price when issuing transactions to smart contract functions.

5.2 Actionable Items for Smart Contract Developers and Tool Developers

Nowadays, blockchain-powered application developers are left in the dark regarding the gas usage of third-party smart contract functions. In the following, we propose a list of actionable items for smart contract developers and tool developers that would assist blockchain-powered application developers in setting the gas price of transactions.

Actionable Item 1) ***Smart contract developers should provide gas usage information as part of the API documentation.*** If a smart contract is developed with reuse in mind (e.g., those distributed by the OpenZeppelin project[16]), then such a smart contract should include gas usage information in their API documentation. Providing a min-max range of gas usage (e.g., determined based on testing the smart contract) would already help blockchain-powered application developers to integrate these third-party contracts into their application. Such documentation is particularly important when the source code of a smart contract is not available, since its absence leaves

[15] https://www.cryptokitties.co

[16] https://github.com/OpenZeppelin/openzeppelin-contracts

blockchain-powered application developers wondering which and how many instructions are typically processed by a given function.

Actionable Item 2) ***Etherscan and Metamask should consider providing historical gas usage information.*** For every smart contract, Etherscan now has an *Analytics* tab in which they show historical information of ether balance, transactions, and token transfers. Etherscan should considering adding a gas usage item in which they could simply show the historical gas usage of the functions of a given contract. Indeed, we noted in observation 3 that 50% of the studied functions (i.e., those that received at least 10 transactions) have a very stable gas usage. Hence, just being able to quickly check the historical gas usage information of a function would already be beneficial in several cases. Etherscan could even categorize the gas usage of functions according to their stability (e.g., by means of the coefficient of variation, as we did in RQ2). Wallets such as Metamask could also provide historical gas usage information (e.g., as a line graph) to help end-users make more informed decisions regarding gas prices.

5.3 Avenues for Future Research

In the following, we discuss avenues for future research that derive from our study.

5.3.1 Why do certain functions have an unstable gas usage? In RQ2, we observed that certain functions have a remarkably unstable gas usage. Future research should investigate the reasons behind such instability. Nonetheless, given that (i) contracts are stateful, (ii) can call each other, and (iii) their source code is not always available, determining the rationale behind gas usage instability is far from trivial. Our hypotheses for gas usage instability include:

- The function has one or more gas leaks, such as *an unbounded mass operation* [22]. An example of an unbounded mass operation is a *for loop* whose number of iterations depend on the number of elements provided as part of the transaction data (i.e., the input parameters to the triggered function). The function shown in Listing 3 contains an unbounded mass operation (lines 19 to 21).
- The function has high cyclomatic complexity. The higher the cyclomatic complexity of a function, the higher the number of possible execution paths. Individual execution paths might burn considerably different amounts of gas units.
- The function calls functions from other contracts, which in turn might call functions from other contracts. Since each contract has a state, these chains of calls might lead to an unstable gas usage.

We believe that examining the stack traces of function executions would enable an initial investigation of the aforementioned hypotheses. Therefore, we also encourage future research to leverage and improve upon existing smart contract profiling tools [37].

Preliminary study. One of our hypotheses is that the gas usage stability of a function is associated with the cyclomatic complexity of that function. Since we cannot extract the function bytecode from the contract bytecode without having access to the contract's ABI, we investigate a slightly different hypothesis in this preliminary study. We study whether the gas usage stability of a function is associated with the cyclomatic complexity of the contract holding this function. While cyclomatic complexity cannot be computed from the bytecode of a contract, we observed in our prior work that cyclomatic complexity is often correlated with size [33]. Therefore, we use size in place of cyclomatic complexity.

Figure 16 shows a heatmap that relates the *gas usage CoV* to *contract bytecode size* for functions that received at least 10 transactions. We do not see any clear pattern in the relationship between these variables. We calculated the Spearman correlation (ρ) between the two variables and obtained a score of $\rho = 0.38$. According to the criteria listed in Evans [19], this score refers to a *weak* correlation ($0.20 \leq |\rho| \leq 0.39$). Therefore, we conclude that the contract bytecode size, by itself,

cannot accurately determine the stability of the gas usage of a function. Nevertheless, we highlight that this preliminary study only investigated the bytecode size of the contract holding the function that is being targeted in a transaction. Given that functions can call other functions (including those defined in other contracts), future correlation studies should account for chains of function calls (c.f., the hypotheses defined at the beginning of this section).

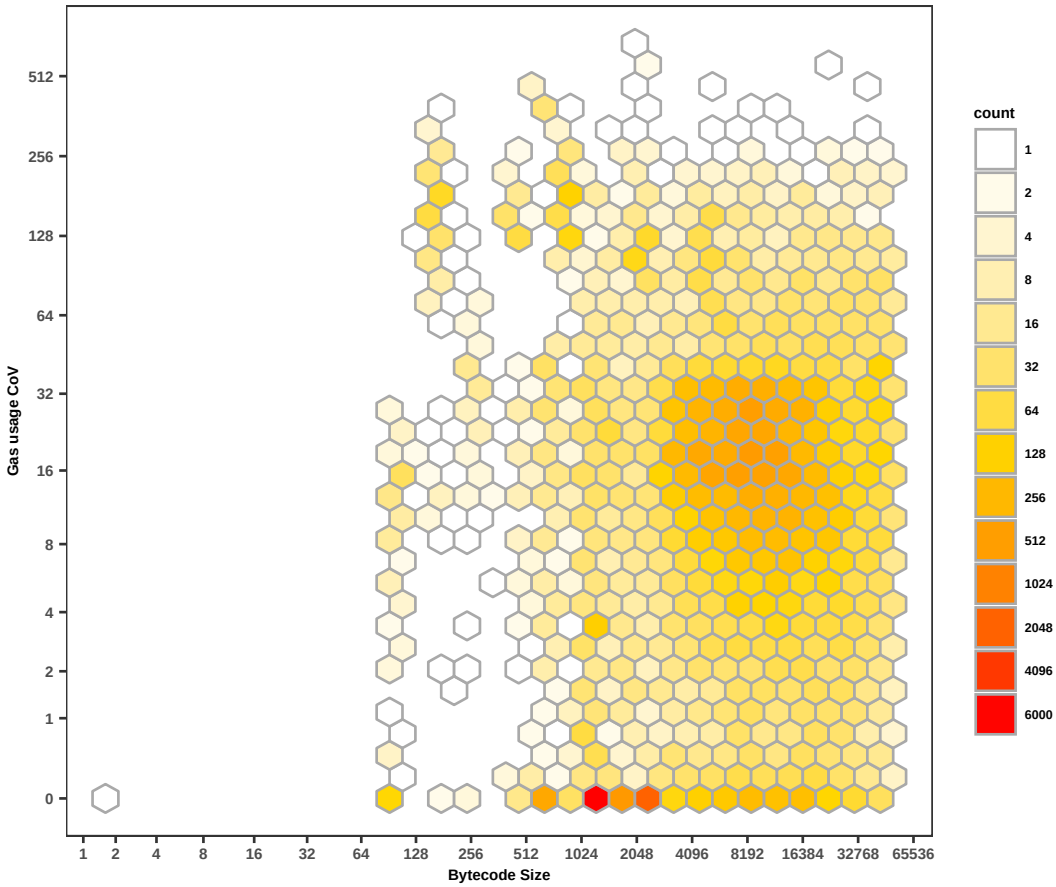


Fig. 16. Heatmap showing the relationship between a function's gas usage CoV and the size of its parent contract.

5.3.2 Devising more accurate gas usage prediction models. In RQ3, we evaluate the performance of our model for gas usage prediction. Figure 13 indicates that a massive number of functions in Ethereum received only either 1 or 2 transactions. Our model performs poorly for these functions. However, the practical relevance of these functions is unclear. Since Ethereum is an open platform, anyone can deploy and trigger contract transactions. Hence, we conjecture that a significant proportion of these rarely used functions refer to toy examples and tests of the platform. Indeed, Oliva et al. [33] also observe that 94.7% of the contracts in Ethereum received less than 10 transactions (analysis period: July 30th 2015 until September 15th 2018). Therefore, further research is necessary to determine the relevance of very low activity contracts and functions.

The Ethereum's official JSON-RPC API[17] contains a function `eth_estimateGas` that returns a gas usage estimate for a given transaction. The function operates by running the provided transaction on the blockchain. Yet, the transaction is never actually added to the blockchain. Despite being convenient and popular, the API documentation acknowledges that the `eth_estimateGas` function may provide estimates that are "significantly more than the amount of gas actually used by the transaction." (Figure 17). Such an acknowledgement of overestimation was one of the reasons behind the proposal of our own prediction model in RQ3. In addition, our model is more lightweight (e.g., does not require a blockchain infrastructure) and does not require the source code or ABI of a smart contract. Nevertheless, our model is only a very first attempt of approaching the problem of gas usage prediction from a different angle. We expect that future research will build on our results and improve upon them. More specifically, promising research avenues include: (a) empirically investigating the accuracy of the `eth_estimateGas` function, (b) comparing the accuracy of the `eth_estimateGas` function to that of other estimators (e.g., our historical model proposed in RQ3), and (c) combining existing estimators into a single one (e.g., a machine learning model that uses the predictions of each estimator as a feature).

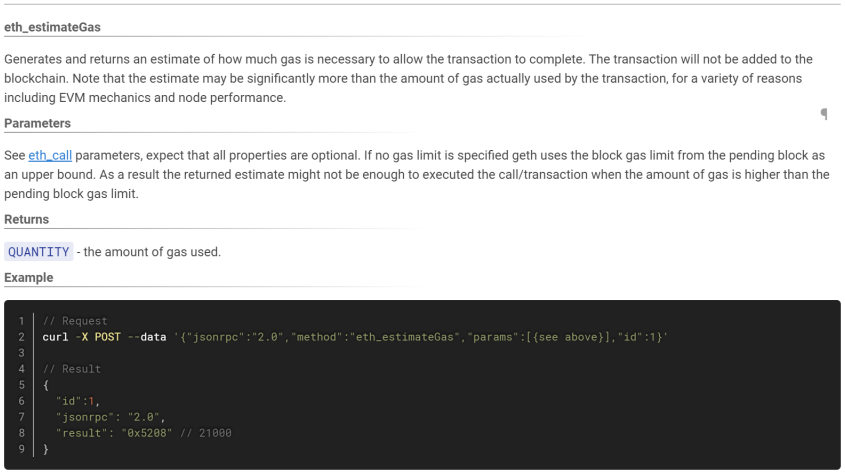


Fig. 17. Screenshot from the Ethereum's official JSON-RPC API.

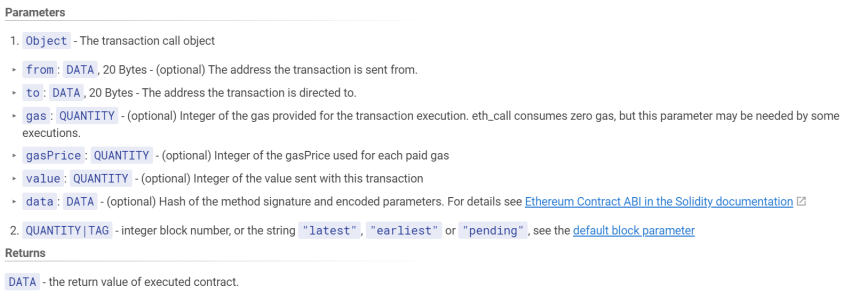
Preliminary study 1. We conduct a preliminary study in which we compare the accuracy of the `eth_estimateGas` function with that of the prediction model that we propose in RQ3. In the following, we summarize the main steps that of our comparison approach:

1) Obtaining the set of transactions to be tested. We start with the set of functions that received at least 100 transactions. Next, we randomly draw one transaction per function. As a result, we obtain a set T of 37,728 transactions.

2) Obtaining the gas usage prediction from eth_estimateGas. As we described above, the `eth_estimateGas` function operates by replaying transactions. The set of input parameters to this function thus correspond to the set of parameters needed to issue a transaction in Ethereum (Figure 18). Therefore, we call the `eth_estimateGas` function for each transaction t in T using the same original parameters of t. However, in order to call such a function, we need an easy to use interface to Ethereum. The two most popular interfaces to Ethereum that implement the JSON-RPC API are

[17] https://eth.wiki/json-rpc/API

Etherscan[18] and Infura[19]. We choose the latter for this study, as it provides a more generous quota of requests per second. However, there is one caveat. Neither of the two interfaces support the specification of the block number in the call to `eth_estimateGas` (see parameter 2 in Figure 18). This implies that the gas usage estimate that we receive from the `eth_estimateGas` is calculated based on the *current state* of the contract in the blockchain (i.e., the latest mined block as of the time of the call to the function). Hence, the results that we show for this preliminary study should be taken with a grain of salt. In addition, due to the different state of contracts, 33.9% of the calls to eth_estimateGas failed with the message "gas required exceeds allowance or always failing transaction". When either Etherscan or Infura implements the block parameter in the future, this study can be replicated and the estimates obtained from `eth_estimateGas` will likely be more accurate.


Parameters

1. `Object` - The transaction call object
   - `from`: `DATA`, 20 Bytes - (optional) The address the transaction is sent from.
   - `to`: `DATA`, 20 Bytes - The address the transaction is directed to.
   - `gas`: `QUANTITY` - (optional) Integer of the gas provided for the transaction execution. eth_call consumes zero gas, but this parameter may be needed by some executions.
   - `gasPrice`: `QUANTITY` - (optional) Integer of the gasPrice used for each paid gas
   - `value`: `QUANTITY` - (optional) Integer of the value sent with this transaction
   - `data`: `DATA` - (optional) Hash of the method signature and encoded parameters. For details see Ethereum Contract ABI in the Solidity documentation
2. `QUANTITY|TAG` - integer block number, or the string `"latest"`, `"earliest"` or `"pending"`, see the default block parameter

Returns

`DATA` - the return value of executed contract.

Fig. 18. Parameters to the `eth_estimateGas` function.

3) Obtaining the gas usage prediction from our model. We obtain the gas usage prediction from our model for each transaction t in T that also got a successful estimate from `eth_estimateGas`.

The results that we obtained are shown in Figure 19. As the figure indicates, our model clearly outperforms the `eth_estimateGas` function. To better understand the difference in performance, we run a Wilcoxon signed-rank test ($\alpha = 0.05$) and compute the Cliff's Delta effect size measure. Results indicate that the difference is statistically significant (p-value < 2.2e-16) and that the effect size is medium 0.44 [39].

Preliminary study 2. As described in Section 2, the gas usage depends on the number and type of bytecode operations that are executed during runtime. It is thus reasonable to hypothesize that the gas usage of a function might be correlated with the bytecode size of the contract holding such a function. Figure 20 shows a heatmap that relates the *median gas usage per function* to *contract bytecode size*. We do not see any discernible pattern in the relationship between the two variables. We calculated the Spearman correlation (ρ) between the two variables and obtained a score of $\rho = 0.11$. According to the criteria listed in Evans [19], this score refers to a *very weak* correlation ($\rho \leq 0.19$). Therefore, we conclude that the contract bytecode size, by itself, cannot be used to accurately predict the gas usage of a function. Nonetheless, we emphasize that this preliminary study only investigated the bytecode size of the contract holding the function that is being targeted in a transaction. As we mentioned in Section 5.3.1, functions defined in a smart contract can call other functions during runtime. Hence, future correlation studies should account for chains of function calls.

[18]https://etherscan.io/apis#proxy
[19]https://infura.io/docs/ethereum/json-rpc

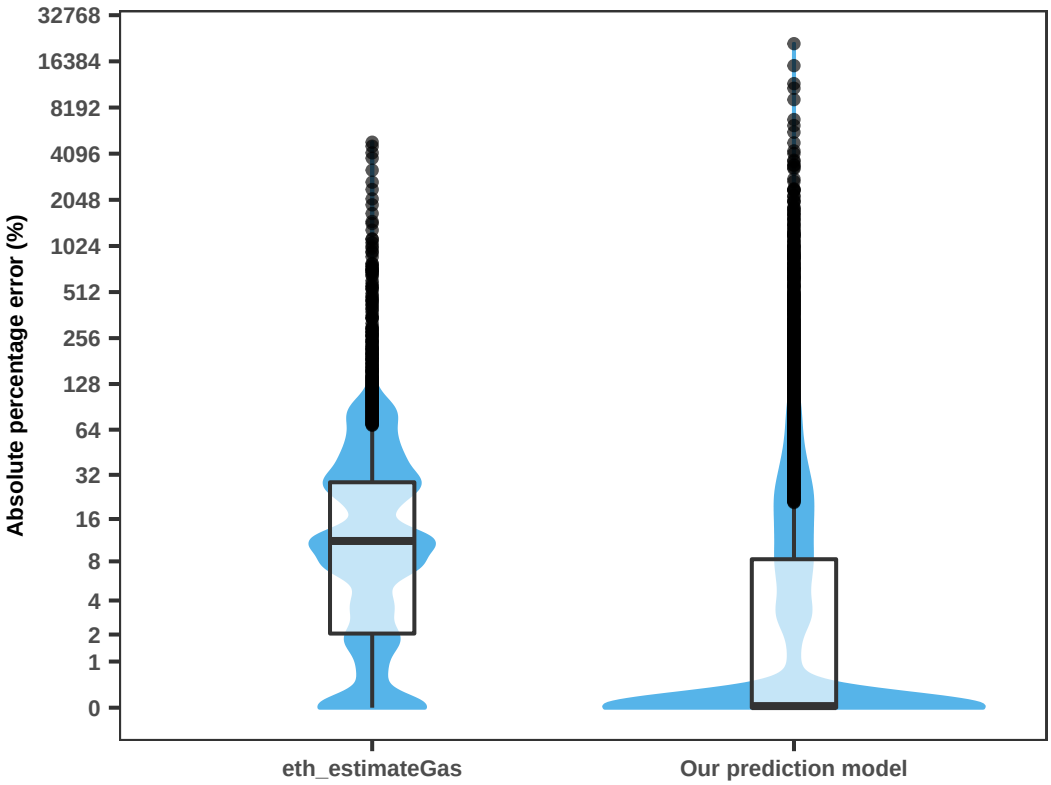


Fig. 19. Absolute percentage error (APE) of the `eth_estimateGas` function and our prediction model.

5.3.3 Leveraging gas usage history to understand the suitability of gas limit choices. Gas usage and gas limit are two interrelated concepts. As we discuss in Section 2.2.4, the gas limit is a parameter that transaction issuers have to specify for all transactions in Ethereum. The gas limit corresponds to the maximum number of units of gas that the transaction issuer is willing to pay for. If the transaction requires more gas to run than the specified gas limit, the transaction fails with an out-of-gas error (and the transaction issuer still pays for the transaction, since miners spent resources to execute it).

Understanding how transaction issuers choose gas limits and assisting them in setting up adequate values is a challenging task. In particular, determining what an *adequate* value actually means is far from trivial. A transaction can run out of gas either because (i) the transaction issuer inadvertently set a gas limit that is too low (i.e., lower than what the function normally burns) or because (ii) the function has a gas leak [22] that makes the transaction burn an abnormally high amount of gas units. In the first case, the transaction should not fail. In the second case, however, it is a positive outcome that the transaction failed – the gas limit safeguarded the transaction issuer from spending more money than he/she wanted (which is the very reason behind the existence of the gas limit concept). In summary, failures due to gas exhaustion should not necessarily be avoided.

We thus consider that a deeper understanding of the interplay between gas usage and gas limit is a fruitful research avenue. For instance, the existence of *spikes* in the gas usage history of a function might point to the existence of gas leaks. Moreover, functions with an unstable gas usage might

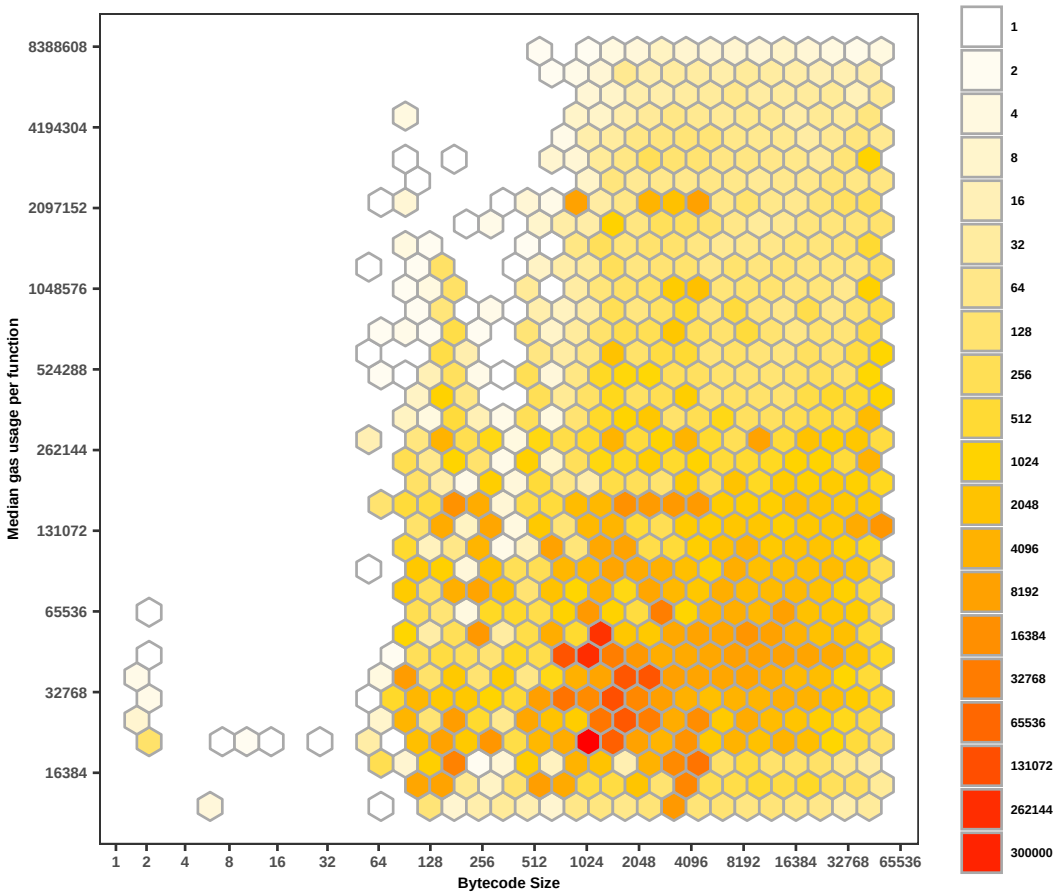


Fig. 20. Correlation between the median gas usage per function and contract size. Log1p scale applied to both axes.

entice transaction issuers to use abnormally high gas limit values, ultimately making transaction issuers more susceptible to paying higher transaction fees in the case of gas leaks.

6 RELATED WORK

Due to the financial aspect of blockchain platforms, a substantial portion of research has been devoted to the discovery and prevention of vulnerabilities in smart contracts [9, 15, 17, 20, 21, 23, 25, 26, 28, 30–32, 34, 35, 41, 42, 46]. However, only few studies have explored the gas system of Ethereum and its impact on the dynamics of the platform. In the following, we discuss prior work regarding the gas system of Ethereum.

Estimation of gas usage. Zou et al. [47] conducted semi-structured interviews and an online survey with smart contract developers to uncover the key challenges and opportunities in the development of contracts. According to the authors, the majority of interviewees mentioned that gas usage deserves special attention. Moreover, 86.2% of the survey respondents also declared that they frequently take gas usage into consideration when developing smart contracts. The reason is twofold: “gas is money” and “transaction failure due to insufficient amount of gas”. In terms of

challenges, the authors highlight that "there is a need for source-code-level gas-estimation and optimization tools that consider code readability.", since most gas-optimization tools operate at the bytecode level (e.g., Remix[20]). Although the goal of our paper is *not* to derive source-code-level gas estimations and optimizations, our model from RQ3 indicates that gas usage tends to be stable when inspecting recent transactional history. Developers can leverage this fact to design tests that can show the typical range of gas usage of a function. This range can then serve as a baseline when experimenting with different possibilities of gas optimizations to the source code.

Prior studies have also tried to estimate the gas usage of smart contract functions. We distinguish between two types of studies. The first type of study focuses on estimating upper bounds for the gas usage of a given function (i.e., worst-case gas consumption), such that out-of-gas errors can be prevented. For instance, Marescotti et al. [29] conduct a feasibility study in which they aim to uncover the worst-case gas usage of a given contract transaction. The authors introduce the concept of *gas consumption paths* (GCPs), which maps gas consumptions to execution paths of a function. The authors examine the GCPs of a function using two symbolic methods in order to derive exact worst-case gas estimations. The two symbolic methods build on the theory of *symbolic bounded model checking* [8] and use efficient SMT solvers. Despite theoretically solid, the proposed estimation model is only evaluated on a single example contract. In the same vein, Albert et al. [4] propose a tool called *Gasol* that takes as input (i) a smart contract (either in EVM, disassembled EVM, or in Solidity source code), (ii) a selection of a cost model, and (iii) a selected public function, and it infers an upper bound for the gas usage of the selected function. As opposed to their prior work [5], the cost model of Gasol is highly configurable. In simple terms, Gasol relies on several tools to extract control flow graphs from smart contracts, which are then decompiled into a high-level representation from which upper bounds can be calculated using a combination of static analyzers and solvers [3]. Gasol is implemented as an Eclipse plug-in, making it suitable for development time use. No evaluation of the proposed tool is conducted.

The second type of study focuses on exact estimations of gas usage (similarly to us). To the best of our knowledge, there is only one study of this type. Das and Qadeer [16] propose a tool called *GasBoX* that takes a smart contract function and an initial gas bound as input (e.g., the gas limit) and verifies that the bound is either exact or returns the program location where the virtual machine would run out of gas. The tool is also designed to be efficient, running with complexity in linear-time in the size of the program. GasBoX works by instrumenting the smart contract code with specific instructions that keep track of gas consumption. GasBoX applies a Hoare-logic style gas-analysis framework to determine gas estimation (the Hoare logic is a formal system with a set of logical rules for reasoning rigorously about the correctness of computer programs). Despite the theoretical soundness of their proposed tool, such a tool has three important limitations: (i) it does not account for function arguments and the state of contracts and (ii) it operates on contracts written in a simplified version of Move[21], which is a programming language in prototypal phase, and (iii) it is evaluated on 13 examples contracts written by the authors.

Differently from all of the aforementioned approaches, our proposed model in RQ3 (i) does not rely on the source code of the contract under investigation, (ii) has been thoroughly evaluated on all successful contract transactions that happened during Ethereum Byzantium (161.6M transactions), and (iii) yields accurate results despite its inherent simplicity.

[20] http://remix.ethereum.org

[21] Move is the smart contract programming language under development by Facebook, to be used in their *Libra* blockchain platform. The language's definition can be seen at https://developers.libra.org/docs/assets/papers/libra-move-a-language-with-programmable-resources/2020-05-26.pdf

Transaction fees. Chen et al. [12] investigated the efficiency of smart contracts by analyzing the bytecode produced by the Solidity compiler. The authors observed that the compiler fails to optimize several gas-costly programming patterns, resulting in higher gas usage (and consequently, higher transaction fees). The authors introduce a tool called GASPER, which can detect several gas costly patterns automatically.

Signer [43] studied the gas cost of different parts of a smart contract code by executing transactions with semi-random data. He deployed and executed the transactions in a local simulation of Ethereum blockchain based on the Truffle IDE and `solc` compiler. With each execution of a transaction, he collected data and mapped to the corresponding section of an abstract syntax tree (AST) of the Solidity source code. He proposed Visualgas, a tool that provides developers with gas cost insights and that directly links to the source code.

Gas-related vulnerabilities. Grech et al. [22] analyzed gas-focused vulnerabilities in Ethereum, which "exploit undesired behavior when a contract (directly or through other interacting contracts) runs out of gas". The authors propose a static analysis program tool called MadMax to automatically detect gas-focused vulnerabilities in Ethereum. The authors analyzed the bytecode of all smart contracts deployed in Ethereum as of Apr. 9th, 2018 and they were able to identify vulnerabilities in contracts that hold together approximately US$2.8B. Through manual inspection of a sample, the authors observed that 81% of the flagged contracts indeed were indeed vulnerable.

Due to the financial aspect of blockchain platforms, Ethereum is constantly subject to hacking activity [6]. To understand the role of gas in these attacks, Chen et al. [13] designed an emulation based framework to automatically measure the gas consumption of EVM operations. They observed that, with a fixed cost of operations, the network can be exploited using under-priced operations. As a solution, they proposed a dynamic cost mechanism for EVM operations that will prevent possible DoS attacks.

Miners perspective on transaction fees. Blockchain platforms such as Bitcoin and Ethereum are designed to discontinue the block reward and keep only the transaction fee at a later point in their development roadmap. To understand the impact of such an event, Carlsten et al. [11] conducted game-theory based simulations to observe the possible outcomes. They concluded that a transaction-fee based system would allow the deviant miners to cause severe instability to the blockchain ecosystem, causing the network to be more vulnerable to a 51% attack.

With further game theory simulations, Tsabary and Eyal [44] investigates the validity of the claims made by Carlsten et al. [11]. They devised a game called *The Gap Game* for their simulation. They found that the claims are valid and that there exist gaps in the mining strategy of the miner, with selfish miners prioritizing these gaps to maximize profit. They also observed that larger mining pools lead to centralization, threatening the fundamental design of a blockchain system. To avoid the selfish mining problem, Abraham et al. [1] proposed a new blockchain platform where a hierarchy is introduced among the miner nodes.

Gas usage parallels in software engineering. Parallels can be drawn between gas usage and several other topics in software engineering for non-blockchain applications. The first parallel that we draw refers to the payment system of cloud computing serves. While several cloud platforms rely on the *pas-as-you-go* mechanism, transaction cost in Ethereum is a function of both gas price and gas usage. Furthermore, gas price influences transaction processing time. Hence, albeit being more flexible, the gas payment system of Ethereum is sensibly more complex than that of typical cloud services.

The second parallel that we draw refers to the energy consumption of software applications. In recent years, the popularity of mobile apps bootstrapped research in energy measurement [7, 24], energy estimation and modeling [2, 14, 18, 36], and code optimizations aimed at lowering energy

consumption [27, 40]. Gas usage, in turn, can be interpreted as a measure of the computing power required to process a given transaction (function call). Given that this computing power needs to be paid for by the transaction issuer, it is also critical to develop mechanisms to measure, estimate, and optimize the gas usage of smart contracts. Indeed, as we discussed before, the survey conducted Zou et al. [47] highlights how important these aspects are for developers of blockchain-powered applications. As also indicated by the survey, the trade-off between code readability and low-level code optimizations is particularly relevant in the blockchain domain.

7 THREATS TO VALIDITY

Construct Validity. In RQ3, we used a threshold of *10 past transactions* to denote the notion of recency. It is possible that other threshold values would yield models with different performances. Nevertheless, in a practical scenario, blockchain-powered application developers can fine-tune this threshold for a set of specific functions that they are interested in. Our key contribution is to show that (i) historical gas usage data can be used as a lightweight approach to predict gas usage and that (ii) *recent* gas usage more closely resembles the current gas usage compared to the entire gas usage history of a function.

Internal Validity. In this study, we are interested in understanding the gas usage of smart contract functions and in trying to predict it. To achieve these goals, we extract, postprocess, and analyze transactional data extracted from the Ethereum blockchain via the Google BigQuery dataset. However, since we identify smart contract functions based on the collected transactional data, we only study the gas usage of those functions that received at least one transaction. As a consequence, in RQ3, we only investigate the accuracy of our method for functions that received at least one transaction. We do not perceive it as a significant limitation, since public/external functions that have never received any transactions are of questionable relevance.

Furthermore, In RQ3 we hypothesize that there are historical patterns in how certain functions burn gas. Indeed, our results from RQ3 indicate that a predictor that operates on recent gas usage outperforms a baseline predictor that operates on all-history gas usage. However, we note that gas usage patterns might exist due to extraneous factors that do not relate to the contract itself (e.g., the amount of data that a given function normally processes). In addition, gas usage patterns might also abruptly change (e.g., a business-driven sudden increase in the amount of data that a given function needs to process daily). Further research is required in order to characterize the different types of historical patterns in gas usage, their rationale, and their susceptibility to change.

External Validity. In this study, we analyzed the Byzantium hard-fork of the Ethereum platform, which was the first hard-fork to contain significant transactional activity (Figure 3). Since hard-forks can introduce different rules to gas consumption, it is possible that our results might not fully hold in more recent hard-forks of Ethereum. We thus encourage future studies to reevaluate our results in more recent hard-fork of the platforms. In addition, we encourage future studies to explore the promising research avenues discussed in Section 5.3. Finally, we emphasize that our results are specific to the Ethereum platform and likely do not generalize to other platforms. Nevertheless, as of June 2020, Ethereum is the most popular platform for the development of blockchain-powered applications according to State of the DApps[22].

8 CONCLUSION

Ethereum is a blockchain platform that supports the development of blockchain-powered applications. When building blockchain-powered applications, developers need to translate requests captured in the frontend of their application (e.g., transfer money from one bank account to another)

[22]https://www.stateofthedapps.com/stats

into one or more smart contract transactions. In particular, developers need to specify the gas price of these contract transactions. Such a task is challenging because (i) miners can prioritize transactions whichever way they wish and (ii) the gas usage of a contract transaction is only known after the transaction is processed and included in a new block.

In this paper, we analyzed the historical transaction metadata from the Byzantium era in order to shed light into the aforementioned challenges. We observed that (i) miners prioritize transactions based exclusively on their gas price in two-thirds of the cases, (ii) 25% of the functions that received at least 10 transactions have an unstable gas usage history, and (iii) the gas usage of contract transactions can be predicted with an RSquared of 0.76 and a median absolute percentage error (APE) of 3.3% by means of a very simple model that relies on the *recent* gas usage of functions.

Hence, blockchain-powered application developers should be aware that a heavy transaction (i.e., high gas usage) will frequently have the same priority as a lightweight transaction (i.e., low gas usage) when these two transactions are issued with the same gas price (despite the difference in transaction fees). Furthermore, blockchain-powered application developers can leverage gas usage prediction models similar to ours to make more informed decisions regarding gas price. Such prediction models are particularly useful for facilitating the integration of third-party smart contracts in a blockchain-powered application. Finally, as actionable insights, we conclude that (i) smart contract developers should provide gas usage information as part of the API documentation and (ii) Etherscan and wallets (e.g., Metamask) should consider providing historical gas usage information for smart contract functions.

REFERENCES

[1] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. 2016. Solidus: An incentive-compatible cryptocurrency based on permissionless byzantine consensus. *CoRR, abs/1612.02916* (2016).

[2] K. Aggarwal, A. Hindle, and E. Stroulia. 2015. GreenAdvisor: A tool for analyzing the impact of software evolution on energy consumption. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 311–320.

[3] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. 2008. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *Static Analysis*, María Alpuente and Germán Vidal (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 221–237.

[4] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2020. GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts. In *Tools and Algorithms for the Construction and Analysis of Systems*, Armin Biere and David Parker (Eds.). Springer International Publishing, Cham, 118–125.

[5] Elvira Albert, Pablo Gordillo, Albert Rubio, and Ilya Sergey. 2019. Running on Fumes. In *Verification and Evaluation of Computer and Communication Systems*, Pierre Ganty and Mohamed Kaâniche (Eds.). Springer International Publishing, Cham, 63–78.

[6] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust*. Springer, 164–186.

[7] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2014. Detecting Energy Bugs and Hotspots in Mobile Apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 588–598. https://doi.org/10.1145/2635868.2635871

[8] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '99)*. Springer-Verlag, Berlin, Heidelberg, 193–207.

[9] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A Scalable Security Analysis Framework for Smart Contracts. *arXiv:1809.03981 [cs]* (Sept. 2018).

[10] Vitalik Buterin et al. 2013. Ethereum white paper. *URL https://github.com/ethereum/wiki/wiki/White-Paper* (2013).

[11] Miles Carlsten, Harry Kalodner, S Matthew Weinberg, and Arvind Narayanan. 2016. On the instability of bitcoin without the block reward. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 154–167.

[12] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 442–446.

[13] Ting Chen, Xiaoqi Li, Ying Wang, Jiachi Chen, Zihao Li, Xiapu Luo, Man Ho Au, and Xiaosong Zhang. 2017. An adaptive gas cost mechanism for ethereum to defend against under-priced DoS attacks. In *International Conference on Information Security Practice and Experience*. Springer, 3–24.

[14] Shaiful Alam Chowdhury and Abram Hindle. 2016. GreenOracle: Estimating Software Energy Consumption with Energy Measurement Corpora. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/2901739.2901763

[15] Christian Colombo, Joshua Ellul, and Gordon J. Pace. 2018. Contracts over Smart Contracts: Recovering from Violations Dynamically. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer International Publishing, Cham, 300–315.

[16] Ankush Das and Shaz Qadeer. 2020. Exact and Linear-Time Gas-Cost Analysis. In *Proceedings of the 27th Static Analysis Symposium (SAS)*. To appear.

[17] G. Destefanis, M. Marchesi, M. Ortu, R. Tonelli, A. Bracciali, and R. Hierons. 2018. Smart contracts vulnerabilities: a call for blockchain software engineering?. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. 19–25.

[18] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. De Lucia. 2017. Software-based energy profiling of Android apps: Simple, efficient and reliable?. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 103–114.

[19] James D. Evans. 1995. *Straightforward Statistics for the Behavioral Sciences*. Brooks/Cole Pub Co.

[20] Josselin Feist, Gustavo Greico, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *Proceedings of the 2Nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB '19)*. IEEE Press, Piscataway, NJ, USA, 8–15.

[21] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy. 2020. Checking Smart Contracts with Structural Code Embedding. *IEEE Transactions on Software Engineering* (2020), 1–1.

[22] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 116.

[23] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *Principles of Security and Trust*, Lujo Bauer and Ralf Küsters (Eds.). Springer International Publishing, Cham, 243–269.

[24] Abram Hindle, Alex Wilson, Kent Rasmussen, E. Jed Barlow, Joshua Charles Campbell, and Stephen Romansky. 2014. GreenMiner: A Hardware Based Mining Software Repositories Software Energy Consumption Framework. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 12–21. https://doi.org/10.1145/2597073.2597097

[25] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, New York, NY, USA, 259–269.

[26] Johannes Krupp and Christian Rossow. 2018. TEETHER: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18)*. USENIX Association, Berkeley, CA, USA, 1317–1333.

[27] Mario Linares-Vásquez, Christopher Vendome, Michele Tufano, and Denys Poshyvanyk. 2017. How developers micro-optimize Android apps. *Journal of Systems and Software* 130 (2017), 1 – 23. https://doi.org/10.1016/j.jss.2017.04.018

[28] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. ReGuard: Finding Reentrancy Bugs in Smart Contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18)*. ACM, New York, NY, USA, 65–68.

[29] Matteo Marescotti, Martin Blicha, Antti E. J. Hyvärinen, Sepideh Asadi, and Natasha Sharygina. 2018. Computing Exact Worst-Case Gas Consumption for Smart Contracts. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer International Publishing, Cham, 450–465.

[30] Anastasia Mavridou and Aron Laszka. 2017. Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. *arXiv:1711.09327 [cs]* (Nov. 2017).

[31] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. 2019. VeriSolid: Correct-by-Design Smart Contracts for Ethereum. In *Financial Cryptography and Data Security*, Ian Goldberg and Tyler Moore (Eds.). Springer International Publishing, Cham, 446–465.

[32] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*. ACM, New York, NY, USA, 653–663.

[33] Gustavo A. Oliva, Ahmed E. Hassan, and Zhen Ming (Jack) Jiang. 2020. An exploratory study of smart contracts in the Ethereum blockchain platform. *Empirical Software Engineering* 25 (2020). Issue 3.

[34] Reza M. Parizi, Ali Dehghantanha, Kim-Kwang Raymond Choo, and Amritraj Singh. 2018. Empirical Vulnerability Analysis of Automated Smart Contracts Security Testing on Blockchains. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering (CASCON '18)*. IBM Corp., Riverton, NJ, USA, 103–113.

[35] Reza M. Parizi, Amritraj Singh, and Ali Dehghantanha. 2018. Smart Contract Programming Languages on Blockchains: An Empirical Evaluation of Usability and Security. In *Blockchain – ICBC 2018*, Shiping Chen, Harry Wang, and Liang-Jie Zhang (Eds.). Springer International Publishing, Cham, 75–91.

[36] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. 2011. Fine-Grained Power Modeling for Smartphones Using System Call Tracing. In *Proceedings of the Sixth Conference on Computer Systems (EuroSys '11)*. Association for Computing Machinery, New York, NY, USA, 153–168. https://doi.org/10.1145/1966445.1966460

[37] C. Peng, S. Akca, and A. Rajan. 2019. SIF: A Framework for Solidity Contract Instrumentation and Analysis. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. 466–473.

[38] S. Porru, A. Pinna, M. Marchesi, and R. Tonelli. 2017. Blockchain-Oriented Software Engineering: Challenges and New Directions. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 169–171.

[39] J. Romano, J.D. Kromrey, J. Coraggio, and J. Skowronek. 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys?. In *Annual meeting of the Florida Association of Institutional Research*. 1–3.

[40] Cagri Sahin, Lori Pollock, and James Clause. 2016. From benchmarks to real apps: Exploring the energy impacts of performance-directed changes. *Journal of Systems and Software* 117 (2016), 307 – 316. https://doi.org/10.1016/j.jss.2016.03.031

[41] Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. 2018. Writing Safe Smart Contracts in Flint. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming (Programming18 Companion)*. ACM, New York, NY, USA, 218–219.

[42] Ilya Sergey and Aquinas Hobor. 2017. A Concurrent Perspective on Smart Contracts. In *Financial Cryptography and Data Security*, Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson (Eds.). Springer International Publishing, Cham, 478–493.

[43] Christopher Signer. 2018. *Gas Cost Analysis for Ethereum Smart Contracts*. Master's thesis. ETH Zurich, Department of Computer Science.

[44] Itay Tsabary and Ittay Eyal. 2018. The gap game. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 713–728.

[45] Gavin Wood, Nick Savers, and Community. 2018. Ethereum: A Secure Decentralised Generalised Transaction Ledger - Byzantium Version. https://github.com/ethereum/yellowpaper/tree/byzantium. [Online; accessed 02-March-2020].

[46] K. Yamashita, Y. Nomura, E. Zhou, B. Pi, and S. Jun. 2019. Potential Risks of Hyperledger Fabric Smart Contracts. In *2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. 1–10.

[47] W. Zou, D. Lo, P. S. Kochhar, X. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu. 2019. Smart Contract Development: Challenges and Opportunities. *IEEE Transactions on Software Engineering* (2019), 1–1. Early Access.