Broken Metre: Attacking Resource Metering in EVM

Daniel Perez
Imperial College London
London, United Kingdom
Email: daniel.perez@imperial.ac.uk

Benjamin Livshits
Imperial College London
London, United Kingdom
Email: b.livshits@imperial.ac.uk

***Abstract*—Metering is an approach developed to assign cost to smart contract execution in blockchain systems such as Ethereum. This paper presents a detailed investigation of the metering approach based on *gas* taken by the Ethereum blockchain. We discover a number of discrepancies in the metering model such as significant inconsistencies in the pricing of the instructions. We further demonstrate that there is very little correlation between the gas and resources such as CPU and memory. We find that the main reason for this is that the gas price is dominated by the amount of *storage* that is used.**

Based on the observations above, we present a new type of DoS attack we call *Resource Exhaustion Attack*, which uses these imperfections to generate low-throughput contracts. Using this method, we show that we are able to generate contracts with a throughput on average 50 times slower than typical contracts. These contracts can be used to prevent nodes with lower hardware capacity from participating in the network, thereby artificially reducing the level of centralization the network can deliver.

I. Introduction

Some blockchain systems support code execution to allow arbitrary programs to take advantage of decentralized trust. Ethereum and its virtual machine, the EVM, are probably the most common approach of this sort. The notion of gas is used for transfer-style transactions on Ethereum.

Additionally, gas-based metering was pioneered in the Ethereum blockchain and was introduced for two reasons: to create monetary incentives for miners who validate smart contracts and to prevent Denial of Service (DoS) attacks on the Ethereum network. The design of the metering system is spelled out in the Ethereum yellow paper [47] and specifies the cost of every instruction. However, the logic for matching costs and instructions not only appears somewhat disconnected to actual costs, but embeds fundamental limitations which have shown cracks with what is known as EIP-150 [9], an update to the Ethereum gas fees in response to several DoS attacks.

This paper is the first attempt to explore the design of the EVM metering system in depth in order to understand both how valid this approach is and how it may be possible to take advantage of EVM design flaws.

A. Contributions

This paper makes the following contributions:

1) **Exploration of metering in EVM**: We identify several important edge cases that highlight inherent flaws in EVM metering; specifically, we identify i) EVM instructions for which the gas fee is too low compared to their resources consumption; and ii) cases of programs where the cache influences execution time by an order of magnitude.
2) **Analysis of Ethereum main net**: We explore the history of executing 2.5 months worth of smart contracts on the Ethereum blockchain and demonstrate that the gas usage is only *marginally correlated* with the usage of resources such as CPU and memory, and that the gas cost is dominated by the EVM storage.
3) **Resource Exhaustion Attacks (REA) contract generation strategy**: We present a code generation strategy able to produce REA attacks of arbitrary length. Some of the complexity comes from the need to produce well formed EVM programs which minimize the throughput. We propose an approach which combines empirical data and genetic programming in order to generate contracts with low throughput. We explore the efficacy of our strategy as a function of the amount of time and effort our program generation approach is allowed to take.
4) **Experimental evaluation**: We show that our REA can abuse the imperfections in EVM's metering approach. Our genetic programming technique is able to generate programs which exhibit a throughput of 1.25M after a single generation. A minimum in our experiments is attained at generation 244 with the block using around 7.9M gas and taking around 78 seconds. We show that our method can generate contracts, which are on average more than 50 times slower than typical contracts. Because node using commodity hardware is unable to keep up under REA, this allows the attacker to further compromise the level of decentralization of the underlying blockchain.

Paper Organization. The rest of the paper is organized as follows. In Section II, we provide background information about Ethereum and its metering scheme, as well as a few examples of how it has been exploited in the past. In Section III, we present a few case studies based on measurements that we obtained by re-executing the Ethereum main chain. In Section IV, we use these measurements to evaluate the correlation between gas and different resources such as CPU, memory and storage. In Section V, we present our Resource Exhaustion Attacks (REA) and the results we obtained. Finally, we present related

work in Section VI and conclude in Section VII.

II. Background

In this section, we briefly describe the Ethereum network and the EVM. Then, we provide an in-depth explanation of how the gas mechanism works and provide additional insights into smart contract execution costs on the Ethereum main network. Finally, we highlight some of the attacks which have been performed by abusing the gas mechanism.

A. Ethereum and the Ethereum Virtual Machine (EVM)

The Ethereum [12] platform allows its users to run "smart contracts" on its distributed infrastructure. Ethereum *smart contracts* are programs which define a set of rules for the governing of associated funds, typically written in a Turing-complete programming language called Solidity [18]. The Solidity code is compiled into EVM bytecode, a low level bytecode designed to be executed by the EVM.

Once the EVM bytecode is generated, it is deployed on the Ethereum blockchain by sending a transaction which only purpose is to create a smart contract with the given code. To execute a smart contract, a user can then send a transaction to this contract. The sender will pay a *transaction fee* which is derived from the contract's computational cost, measured in units of *gas* [47]. The fee itself is paid in Ether (ETH[1]), the underlying currency of the Ethereum blockchain. When a *miner* successfully mines a blocks, he receives the transaction fee of all the transactions included in the block. We will describe exactly how this transaction fee is computed in the part of this section.

B. Metering in EVM

As briefly outlined in Section I, gas is a fundamental component of Ethereum, and generally applicable to permissioned and permissionless blockchain platforms that utilise a distributed virtual machine for contract code execution. [44], [6]. Gas is the main protection against Denial of Service (DoS) attacks based on non-terminating or resource-intensive programs. It is also used to incentivise miners to process transactions by rewarding them with a fee computed based on the resource usage of the transaction.

Gas cost. In the EVM, each transaction has a cost which is computed in and expressed as gas. The cost is split into two parts, a fixed *base cost* of $21,000$ gas, and a variable *execution cost* of the smart contract. Each instruction has a fixed gas cost which has been set by the designers of the EVM [47], who classify the instructions in multiple tiers of gas cost:

- *Zero Tier (0 gas)*: Instructions to stop contract execution: `STOP` and `RETURN`
- *Base Tier (2 gas)*: Most instructions used to read some state of the VM, such as the current value of the program counter, the caller, or value sent to the contract

[1]When converting ETH to USD, we use the exchange rate on 2019-05-12: 1 ETH = 200 USD. For consistency, any monetary amounts denominated in USD are based on this rate.

```
PUSH1 0x02 ; very low tier (3 gas)
PUSH1 0x03 ; very low tier (3 gas)
MUL        ; low tier      (5 gas)
PUSH1 0x05 ; very low tier (3 gas)
SSTORE     ; special tier  (20k gas)
```

Fig. 1: Example gas cost of an EVM program

- *Very Low Tier (3 gas)*: Instructions to operate on the stack, such as `PUSH` and `POP` as well as fast arithmetic operations such as `ADD` or `SUB`
- *Low Tier (5 gas)*: More involved arithmetic instructions such as `MUL`, `DIV` or `MOD`.
- *Mid Tier (8 gas)*: Combined arithmetic operations such as `ADDMOD` and `MULMOD`, as well as the `JUMP` instruction which is used to perform a jump.
- *High Tier (10 gas)*: Only the `JUMPI` instruction, which performs a conditional jump.
- *Special Tier*: Instructions of which the cost needs more complex rules. For example, the cost of `SSTORE`, which is used to store an element in storage, depends on what the previous value was and what the next value will be. If the previous value was zero and the new value is non-zero, the instruction allocates storage and costs 20,000 gas to execute. On the other hand, if the previous value was non-zero and the new value is zero, the instruction frees storage and 15,000 gas is refunded.

The gas cost for a transaction in the EVM is the sum over the cost of each instruction in the contract. For example, given the program in Figure 1, the gas cost will be computed as follow. `PUSH1` is in the Very Low Tier and therefore costs 3 gas. It is called 3 times in total and will therefore consume 9 gas. The arguments of `PUSH1` do not consume any extra gas. The `MUL` instruction is in the Low Tier and hence costs 5 gas. Finally, the `SSTORE` will store the result of 2×3 at location 5 in the storage. `SSTORE` is in the Special Tier and has slightly more complex pricing rules. Assuming the location in the storage was previously 0, the instruction allocates storage and will cost 20,000 gas. Therefore, this program will cost a total of 20,014 gas to execute. Given the current pricing for storage, the cost of the program is clearly dominated by the storage operation.

It is important to note that, as the transaction has a base cost of 21,000 gas, it will cost a total of $21{,}000 + 20{,}014 = 41{,}014$ gas to execute the above transaction.

Ethereum Improvement Proposal (EIP) 150. Although the cost of each instruction was decided when first designing the EVM, the authors found that some costs were poorly aligned with actual resource consumption. Particularly, IO-heavy instructions tended to be to cheap, allowing for DOS attacks on the Ethereum [10] blockchain. As a fix, EIP 150 [9] was proposed and implemented, significantly increasing the gas consumption of instructions which require to perform IO operations, such as `SLOAD` or `EXTCODESIZE`. This change

revised the cost of under-priced instructions and prevented further DoS attacks such as the one seen in September 2016 [11]. This briefly highlights the potential risks rooted in mismatches between instructions and gas costs. While the above cases have been fixed, it is unclear whether all potential issues have been eradicated or not.

Gas price. Up to here, we have explained how the gas cost for executing a contract are computed. However, the gas cost is not the only element needed to compute the total execution cost of a contract. When a transaction is sent, the sender can choose a gas price, namely the amount of *wei* (1wei $= 10^{-18}$ ETH) that the sender is ready to pay per unit of gas. Miners will usually prioritise transactions with high gas prices, as this will increase the final fee they receive for processing a transaction.

Transaction fee. The transaction fee is the total amount of wei that the sender of the transaction has to pay for the transaction. It is obtained by multiplying the gas price by the gas cost. The transaction fee is non-refundable: even if the transaction fails, it will be paid.

C. Gas Statistics

Now that we presented the key points about metering in the EVM, we provide numbers to give a better insight on how much gas is usually consumed and for what price.

In Figure 2a and Figure 2b, we show respectively the average gas consumption per block, the average gas price and the price of Ether in USD. The data in all figures is aggregated by week. We can see that the gas used went up from an average of around 25,000 gas at the beginning of 2016 to an average of around 55,000 gas at the beginning of 2018 with occasional stronger oscillations. We assume the increase in the average gas consumed is mostly due to the increase in the complexity of smart contracts over time [50]. We would think that users would tend to use less gas when the gas price is high but looking at Figures 2a and 2b, the gas usage does not seem to be correlated with the gas price.

To understand the behaviour of the gas price, we also plot the price of Ethereum over time with a log-scaled y-axis in Figure 2c. We can clearly see that the gas price peaks when the Ethereum price increases. For example, there is a first significant increase of the price of Ethereum at the beginning of 2015 and that the gas price rises from 20 Gwei to more than 70 Gwei. The same phenomenon can be observed around January 2018 as well as May 2018. We hypothesise that the reason behind this is that when the price increases, the network activity tends to increase as well, generating higher demand: more transactions to be processed. Given that the supply — the throughput of the network — is mostly constant, the price increases to adjust to this demand.

To give a sense of the transaction fees, we show a variety of typical fees in Figure 3. The fees are divided depending on their gas price and gas consumption. The *Low* gas price is close to the lowest price that can be paid to get the transaction accepted on the Ethereum blockchain. The *High* gas price refers to the price that people would pay when they are extremely eager to get their transaction included, for example when competing with other users to have a transaction included first [36]. The *basic* transaction type refers to transactions consuming only the base amount of gas, without executing any instruction. This is typically the cost to send Ether to a contract or another party. The *gas intensive* transaction type represents computationally expensive transactions, for example, verifying a zero-knowledge proof [41]. At the time of writing, the maximum amount of gas which can be used in a single block is 8,000,000, which means only 16 such transactions could be included in a single block.

In Figure 4, we show the values of the gas price, gas used and transaction fee. In order to obtain results reflecting the current situation, we limit the analysis to recent blocks. We use all the transactions sent to contracts between August 23rd 2018 and March 9th 2019. We find that the median gas price paid by a transaction's sender is around 8.5 Gwei[1], which is around 4 times more than the minimum possible fee. It is worth noting that when paying the minimum possible fee, the probability for the transaction to get included fast is low and the transaction can therefore be delayed for several blocks: at the time of writing, only a little more than 10% of the last 200 blocks accepted a gas price of 2Gwei [17]. This explains that users usually pay a higher fee to get their transaction included faster. The median for the gas consumed is around 50,000 gas, indicating that most transactions perform relatively simple computations. Indeed, the basic fee being 21,000, a simple read followed by an allocation of storage would already result in 46,000 gas. Overall, the median fee paid per transactions is 0.0004 ETH which is around 0.08 USD.

D. Previously Known Attacks

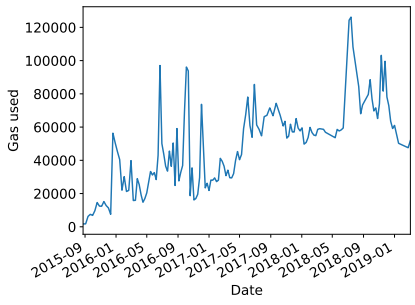
The Ethereum network has been victim of several Denial of Service (DoS) attacks due to instructions being underpriced. We present two considerable DoS attacks which were performed on the Ethereum network.

`EXTCODESIZE` attack. In September 2016, a DoS attack was performed on the Ethereum network by flooding it with transactions containing a very large number of `EXTCODESIZE` instructions [11]. `EXTCODESIZE` is an instruction to retrieve the size in bytes of a given contract's code.

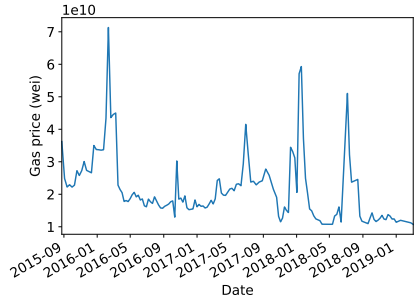
This attack happened because the `EXTCODESIZE` instruction was vastly underpriced. At the time of the attack, a single execution of this instruction cost 20 gas, meaning that one could perform around 1,500 instructions with less than $0.01. Although by itself, this issue might seem benign, `EXTCODESIZE` forces the client to search the contract on disk, resulting in IO heavy transactions. The malicious transactions took around 20 to 80 seconds to execute, compared to a few milliseconds for the average transactions. We show the correlation between the clock time and the gas used by transactions during the period of the attack in Figure 5. Although this did not create any issue on the consensus layer, it reduced the rate of block creation by a factor of more than 2 times, with block creation time peaking to more than 30s [21].

The Ethereum protocol was updated in EIP 150, with all the software running Ethereum, to increase the price of the `EXTCODESIZE` from 20 to 700 gas, making the aforementioned attack considerably more expensive to perform. Some

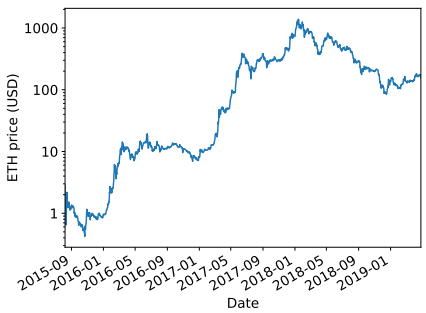
[1] 1Gwei $= 10^9$wei $= 10^{-9}$ETH



(a) Average gas used per block, aggregated per week.



(b) Average gas price per block, aggregated per week.



(c) Log-scaled price of Ethereum over time.

Fig. 2: Statistics of gas usage, gas price and Ethereum price over time.

Transaction type	Gas price Low (2Gwei[1])	High (80Gwei)
Basic (21k gas)	$0.01	$0.34
Gas intensive (500k gas)	$0.20	$8.00

Fig. 3: Fees for different type of transactions. "Low" price is the lowest possible price to have a transaction included while "High" a price someone very eager to have his transaction included would pay.

Number of blocks:	1,133,621
Median gas price:	8.5 Gwei
Median gas used (by contracts):	48,398
Median transaction fee:	0.0004 ETH (0.08 USD)

Fig. 4: Median gas price, gas used and transaction fee from block 6198305 (Aug-23-2018) to block 7331927 (Mar-09-2019).

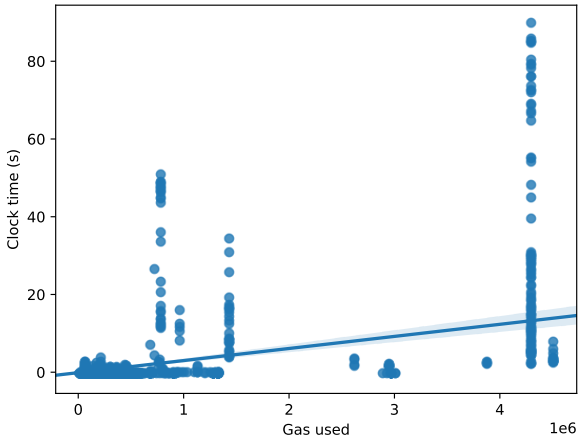


Fig. 5: Correlation between gas and clock time with DoS.

performance improvements were also made at the implementation level, allowing clients to process IO-intensive instructions faster.

`SUICIDE` Attack. Shortly after the `EXTCODESIZE` attack, another DoS attack involving the `SUICIDE` instruction was performed [10]. The `SUICIDE` instruction kills a contract and sends all its remaining Ether to a given address. If this particular address does not exist, a new address would be newly created to receive the funds. Furthermore, at the time of the attack, calling `SUICIDE` did not cost any Ether. Given these two properties, an attacker could create and destroy a contract in the same transaction, creating a new contract each time at an extremely low fee. This quickly overused the memory of the nodes, and particularly affected the Go implementation [25] which was less memory efficient [13].

A twofold fix was issued for this attack in EIP 150. First, and most importantly, `SUICIDE` would be charged the regular amount of gas for contract creation when it tried to send Ether to a non-existing address. Subsequently, the price of the `SUICIDE` instruction was increased from 0 to 5,000 gas. Again, these measures would make such an attack very expensive.

III. Case Studies in Metering

In this section, we instrument the C++ client of the Ethereum blockchain, called *aleth* [20], and report some interesting observations about gas dynamics in practice.

A. Experimental setup

Hardware. We run all of the experiments on a Google Cloud Platform (GCP) [26] instance with 4 cores (8 threads) Intel Xeon at 2.20GHz, 8 GB of RAM and an SSD with a 400MB/s throughput. The machine runs Ubuntu 18.04 with the Linux kernel version 4.15.0.

Software. To measure the speed of different instructions, we fork the Ethereum C++ client, *aleth*. Our fork integrates the changes to the upstream repository until Jun-26 2019. We choose the C++ client for two reasons: first, it is one of the two clients officially maintained by the Ethereum Foundation [1] with geth [25]; second, it is the only of the two without runtime or garbage collection, which makes measuring metrics such as memory usage more reliable.

We add compile options to the original C++ client to allow enabling particular measurements such as CPU or memory. Our measurement framework is open-sourced[2] and available under the same license as the rest of *aleth*.

Measurements. For all our measurements, we only take into account the execution of the smart contracts and ignore the time taken in networking or other parts of the software. We use a nanosecond precision clock to measure time and measure both the time taken to execute a single smart contract and the time to execute a single instruction. To measure the memory usage of a single transaction, we override globally the **`new`** and **`delete`** operators and record all allocations and deallocations performed by the EVM execution within each transaction. We ensure that this is the only way used by the EVM to perform memory allocation.

Given the relatively large amount of time it takes to re-execute the blockchain, we only execute each measurement once when re-executing. We ensure that we always have enough data points, where enough in the order of millions or more, so that some occasional imprecision in the measurements, which are inevitable in such experiments, do not skew the data.

In this section, the measurements are run between block 5,171,468 (Feb-28-2018) and block 5,587,480 (May-10-2018), except for the last part where all the benchmarks are run at block 5,587,480.

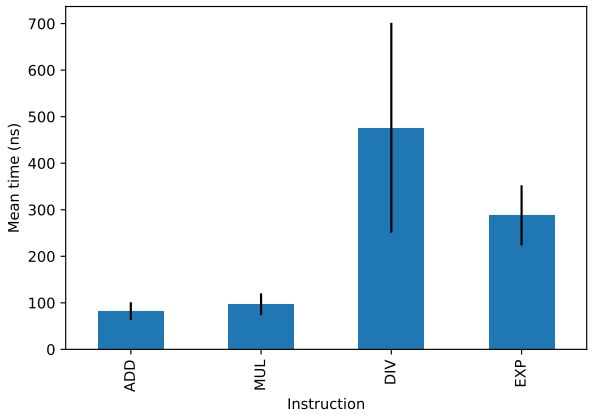
B. Arithmetic Instructions

In this experiment, we evaluate the correlation between gas cost and the execution time for simple instructions which include absolutely no IO access. We use simple arithmetic instructions for measurements, in particular the `ADD`, `MUL`, `DIV` and `EXP` instructions.

In Figure 6a, we show the mean time of execution for these instructions, including the standard deviation for each measurement. We contrast these results with the gas cost of the different instructions in Figure 6b. We see that although in practice `ADD` and `MUL` have similar execution time, the gas cost of `MUL` is 65% higher than the gas cost for `ADD`. On the other hand, `DIV`, which costs the same amount of gas as `MUL`, is around *5 times slower* on average. `EXP` costs twice the price of `DIV` but executes on average 40% faster. Another point to note here is that `DIV` has a standard deviation much higher than the other three instructions. Although we were expecting that for such simple instructions the execution time would reflect the gas cost, this does not appear to be the case in practice. We will show in the coming sections that IO related operations tend to make things worse in this regard.

C. High-Variance Instructions in EVM

Here, we look at instructions which have a high variance in their execution time. We summarize the instructions which had the highest variance in Figure 7. There are two main reasons why the execution time may vastly vary for the execution of the same instruction. First, many instructions take parameters, depending on which, the time it takes to run the



(a) Mean time for arithmetic instructions.

Instruction	Gas cost	Mean time (ns)	Throughput (gas / μs)
`ADD`	3	82.20	36.50
`MUL`	5	96.96	51.57
`DIV`	5	476.23	10.50
`EXP`	10	287.93	34.73

(b) Execution time and gas usage for arithmetic instructions.

Fig. 6: Comparing execution time and gas usage of arithmetic instructions.

particular instructions can vary wildly. This is the case for an instruction such as `EXTCODECOPY`. The second reason is much more problematic and comes from the fact that some instructions may require to perform some IO access, which can be influenced by many different factors such as caching, either at the OS or at the application level. The instruction with the highest variance was `BLOCKHASH`. `BLOCKHASH` allows to retrieve the hash of a block and allows to look up to 256 block before the current one. When it does so, depending on the implementation and the state of the cache, the EVM may need to perform an IO access when executing this instruction, which can result in vastly different execution times. The cost of `BLOCKHASH` being currently fixed and relatively cheap, 20 gas, it results in an instruction which is vastly under-priced. It is worth noting that in the particular case of `BLOCKHASH`, the issue has already been raised more than two years ago in EIP 210 [14]. It discussed of changing the price of `BLOCKHASH` to 800 gas but at the time of writing the proposal is still in draft status and was not included in the Constantinople fork[3] [29] as it was originally planned to be.

D. Memory Caches and EVM Costs

Given the high variance in execution time for some instructions, we evaluate the effects caching may have on EVM execution speed. In particular, we evaluate both the speedup provided by the operating system page cache and the speedup across blocks provided by LevelDB LRU cache [24].

[2]URL anonymised for double blind review

[3]Hard fork which took place on Feb 28 2019 on the Ethereum main network

Instruction	Mean time (μs)	Standard deviation	Measurements Count
BLOCKHASH	768	578	240,000
BALANCE	762	449	8,625,000
SLOAD	514	402	148,687,000
EXTCODECOPY	403	361	23,000
EXTCODESIZE	221	245	16,834,000

Fig. 7: Instructions with highest execution time variance.

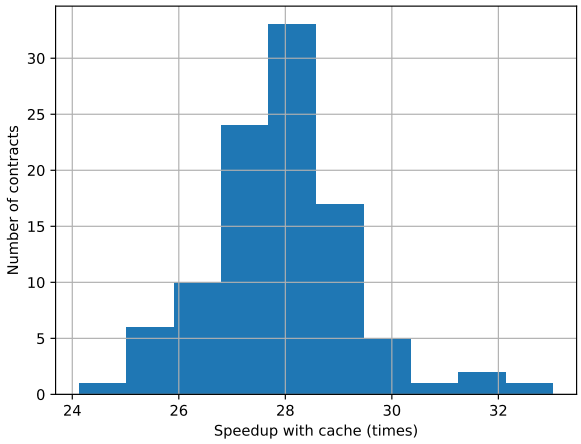


Fig. 8: Comparing throughput with and without page: x axis is the relative speed improvement and y axis is the number of contracts.

Page cache. First, we evaluate how the operating system page cache influences the execution time by reducing the IO latency. We perform the experiment as follows:

1) Generate a contract
2) Run the code of the contract n times
3) Run the code of the contract n times but drop the page cache between each run

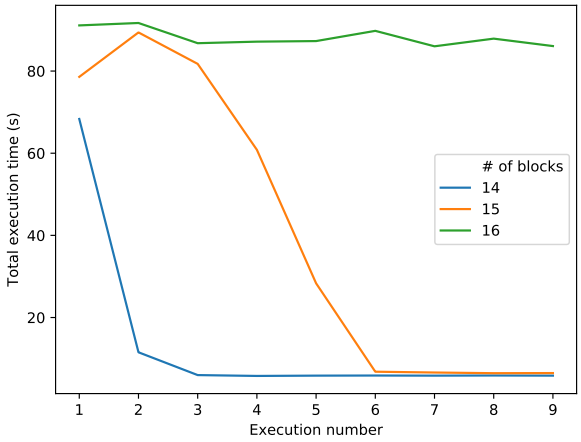


Fig. 9: Measuring block execution speed with and without the effect of cache.

We perform this for 100 different contracts and measure the execution time for the versions with and without cache. We generate relatively large contracts, which consume on average 800,000 gas each. Although the method is somewhat crude, it provides a good approximation of the extent to which the state of the page cache influences the execution time of a contract. In Figure 8, we show a distribution of the contracts throughput in terms of gas per second, with and without cache. We see that contracts execute between 24 and 33 times faster when using the page cache, with more than half of the contracts executing between 27 and 29 times faster. This vast difference in the execution speed is due to IO operations, which use LevelDB [23], a key-value store database, under the hood. LevelDB keeps only a small part of its data in memory and therefore needs to perform a disk access when the data was not found in memory. If the required part of the data was already in the page cache, no disk access will be required. When keeping the page cache, all the items seen by the contract recently will already be available in cache, eliminating the need for any disk access. On the other hand, if the caches are dropped, many IO related operations will result in a disk access, which explains the speedup. We notice that in the contracts with the highest speedup, BLOCKHASH, BALANCE and SLOAD are in the most frequent instructions. It is worth noting that if the generated contracts are small enough, most of the data will be in memory and dropping the page cache will have much less effect on the runtime. Indeed, when running the same experiment with contracts consuming on average 100,000 gas, only a 2 times average speedup has been observed.

Caching across blocks. In the next experiment, instead of measuring the cache impact by running a single contract multiple times, we evaluate how the cache impacts the execution time across blocks. In particular, we measure how many blocks need to be executed before the data cached during the previous execution of a contract gets evicted from the different caches. To do so, we perform the following experiment.

1) Generate n blocks, with different contracts in each
2) Execute sequentially all the blocks and measure the execution time
3) Repeat the previous step m times in the same process, and record how the execution speed evolves

We set m to 10 and we try different values for n to see how many blocks are needed for the cache not to provide anymore speedup. We find that in our setup, assuming the blocks are full (i.e. close to the gas limit in term of gas), 16 blocks are enough for the cache not to provided anymore speedup. We plot the results for $n = 14$, $n = 15$ and $n = 16$ in Figure 9. When $n = 14$, we see that the second execution is much faster than the first one, and that after the third execution, the execution time stabilizes at around 6s to execute the 14 blocks. For $n = 15$, the execution time takes longer to decrease, but eventually also stabilizes around the same value. It is slightly higher than when $n = 14$ because it has one more block to execute. However, once we reach $n = 16$, we see that the execution time hardly decreases and stays stable at around 85s. We conclude that at this point, almost nothing that was cached during the previous execution of the block is still cached when re-executing the block.

E. Summary

In this section, we empirically analyzed the gas cost and resource consumption of different instructions. To summarize:

- We see that even for simple instructions, the gas cost is not always consistent with resource usage. Indeed, even for instruction with very predictable speed, such as arithmetic operations, we observe that some instructions have a throughput 5 times slower than others.
- We notice that while most instructions have a relatively consistent execution speed, other instructions have large variations in the time it takes to execute. We find that these instructions involve some sort of IO operation.
- Finally, we demonstrate the effect that the page cache has on the execution speed of smart contracts and then show the typical number of blocks for which the page cache still provides speed up.
- Overall, we see that beyond some pricing issue, the metering scheme used by EVM does not allow to express the complexity inherent to IO operations.

IV. Analysis of Gas Consumption

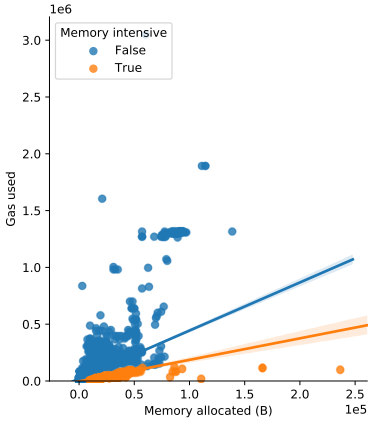
In this section, we analyze the gas consumption of Ethereum smart contracts and try to correlate it with different system resources, such as memory, CPU and storage. To measure the consumption, we re-execute the Ethereum blockchain with a patched version of the Ethereum C++ client [20] and collect information about resources for each contract invocation. As described in Section II, EIP 150 influenced the price of many storage related operations, which affected the gas cost of transactions. Therefore, we perform our experiments on a set of 100,000 blocks before EIP 150 and a set of 100,000 blocks after EIP 150. EIP 150 was introduced at height 2,463,000. We arbitrarily use block 1,400,000 to block 1,500,000 for measurements before EIP 150 and block 2,500,000 to 2,600,000 for measurements after EIP 150. Although the block numbers are arbitrary, we assume that the sample of 100,000 blocks which roughly corresponds to two weeks, is large enough to obtain reliable data.

Below, we denote the gas usage for a transaction as G. All the figures in this section include bands marking a confidence interval of 95%.

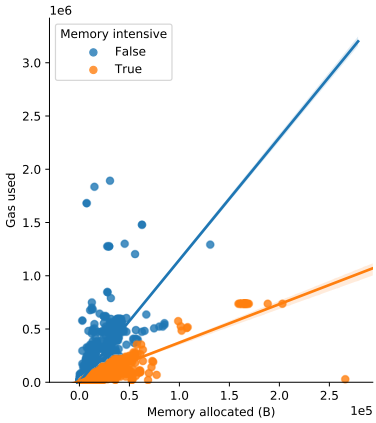
A. Memory Usage

We note M_+ the amount of memory allocated and M_- the amount of memory deallocated. We compute the extra amount of memory allocated $M = M_+ - M_-$ to execute a particular transaction, that is, the difference between the total amount of memory allocated and the total amount of memory deallocated.

An important point is that contracts which allocate storage in any way, may it be with `SSTORE` instructions or `LOG` instructions, will see their memory overhead increase, as this extra storage will stay in memory even after the execution of the contract. Although this is not a perfect measurement, as part of the memory may be released later on during the execution of the transaction, it does give a proxy measure of how a particular contract execution affects the memory usage of the client. It is also worth noting that some contracts may also release more memory than they allocate, if, for example, a contract self-destructs itself. As we want to focus on how execution can consume resources, we filter contracts where $M < 0$.



(a) Pre EIP 150 correlation between gas and memory usage.



(b) Post EIP 150 correlation between gas and memory usage.

Phase	Executions set	Pearson score	Gas/byte
Pre EIP 150	E	0.545	3.96
	E_{MI}	0.773	1.80
	$E \setminus E_{MI}$	0.633	4.96
Post EIP 150	E	0.755	7.67
	E_{MI}	0.902	4.16
	$E \setminus E_{MI}$	0.907	11.7

(c) Relation between allocated memory and gas usage.

Fig. 10: Correlation between gas and memory usage.

We first compute the Pearson correlation coefficient[4] [7] between the extra memory allocated and the gas usage. For contracts prior to EIP 150, we obtain a score 0.545 which shows that, although a positive correlation exists between memory and gas usage, the correlation is fairly weak. As described in Section II, EIP 150 influenced widely the price of all storage related operations, which vastly affect the results obtained for memory usage. Indeed, the Pearson correlation coefficient we obtain for post EIP 150 data is 0.766. This shows that the correlation between memory and gas usage is vastly greater after EIP 150.

The data we obtain suggest that there are different trends in terms of correlation between gas and memory usage. Therefore, we decide to separate the data in two categories to try to isolate the two trends. We first compute the ratio of extra memory allocated per gas used: $R_{M/G} = M/G$. The higher the ratio is, the lower a transaction is paying to consume memory. We then split in 10 quantiles and assign the quantile with the highest ratio $R_{M/G}$, that is, the quantile paying the less gas for memory, as being "Memory intensive". We define E the set of all executions and E_{MI} as the set of "memory intensive" executions.

We plot our results pre and post EIP 150 in Figure 10a and in Figure 10b. We also show the correlation scores we obtain as well as the relation between gas usage and memory usage in Figure 10c. The gas/byte column represents the cost in gas to allocate one extra byte of memory and is inferred by performing a linear regression on the data.

There are several interesting points to note about the results. First, EIP 150 resulted in a significant increase in the price of gas per memory, which had the effect of increasing the correlation between these two variables. Second, after EIP 150, the difference between normal and memory intensive contracts becomes more visible, with the gas per byte cost going from 4.16 for memory intensive contract executions up to 11.7 for others. Nevertheless, both type of contracts have a high correlation score between gas and memory allocated. This shows that there are important discrepancies on how pricing is done for consumed memory depending on the characteristics of the contract being executed.

B. CPU Usage

We analyze the correlation between the gas usage and the CPU usage for each transaction. As for the memory measurements, we use our instrumented C++ client and record the number of clock ticks used by the thread executing the transaction. We use this clock time as a proxy for the CPU usage.

Again, we compute the Pearson correlation score between the CPU and gas usage. The score for contract executions before EIP 150 is 0.528 while the score after EIP 150 is 0.507. This score implies a low positive correlation between the two variables. This comes from the fact that the instructions touching to storage are very expensive, thus dominating the cost of other instructions. The score most likely got lower after EIP 150 because storage related operations got more expensive.

[4]Pearson score of 1 means perfect positive correlation, 0 means no correlation

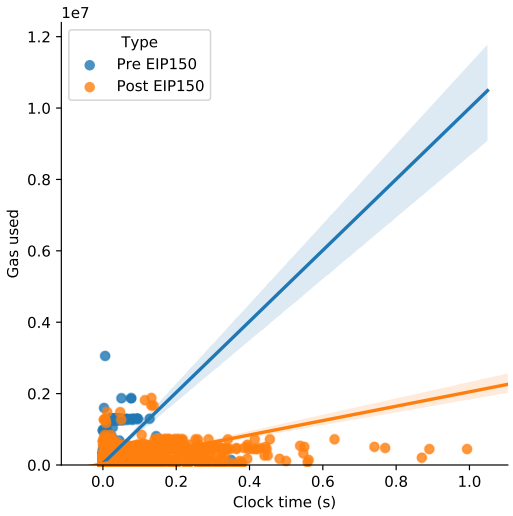


Fig. 11: Correlation between gas and CPU usage.

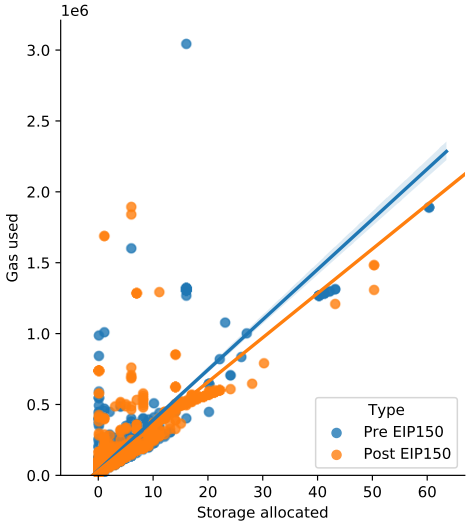


Fig. 12: Correlation between gas and storage usage.

We plot the relation between the CPU and gas usage before and after EIP 150 in Figure 11. Although the correlation is fairly low, it is interesting to see that with EIP 150, as a consequence of storage related operation increasing, the relative cost of CPU has decreased.

C. Storage Usage

To analyze the storage usage, we analyse the number of EVM words (256 bits) of storage allocated per transactions. Here, we define a word to have been allocated if its value before a transaction was 0 and its value after the transaction was not. Conversely, a word has been deallocated if its value before a transaction was non-zero and its value after is 0. Both of these costs are formally defined in Ethereum [47] and the main goal of this experiment is to see by how much the storage allocation dominates the total gas cost. The number of allocations made during a single execution can be negative, if

Phase	Resources	Pearson score
Pre EIP 150	Memory/CPU	0.591
	Storage/CPU	0.725
	Storage/Memory	0.845
	Storage/Memory/CPU	0.759
Post EIP 150	Memory/CPU	0.741
	Storage/CPU	0.837
	Storage/Memory	0.938
	Storage/Memory/CPU	0.893

Fig. 13: Multivariate correlation between gas and resources.

Phase	Resource	Pearson score
Pre EIP 150	Memory	0.773
	CPU	0.528
	Storage	0.775
	Storage/Memory	0.845
Post EIP 150	Memory	0.755
	CPU	0.507
	Storage	0.907
	Storage/Memory	0.938

Fig. 14: Correlation between gas and resources.

more words are deallocated than allocated. Again, as we focus on the resource usage, we filter out this type of executions.

We compute the Pearson's correlation before and after EIP 150 and obtain respectively 0.821 and 0.907. This shows that there is, as expected, a strong positive correlation between the amount of storage allocated and the gas cost during a single execution. We plot the relation between the storage allocated and the gas used before and after EIP 150 in Figure 12. Although EIP 150 increased the price for a lot of storage related instructions, it changed neither the price of allocating a new value, nor the reimbursement received when deallocating a value. This explains that, unlike for memory, the cost does not go up after EIP 150. An interesting point to notice is that the execution cost does not split into two different gas usage trends as they do with memory, which suggests that the storage cost is not what is creating this effect for the extra memory allocated.

D. Multi-variate Correlation

Instead of simply using a bi-variate correlation between gas consumption and a resource, we try to correlate gas consumption to multiple resources at a time. To compute the multi-variate correlation between multiple resources and the gas usage, we proceed as follow.

Given n measures for m resources $r_1, \cdots, r_m$, we note $R_1, \cdots, R_m$ the vectors of measures for each resource, which will each have a dimension of n. We first use equation 1 to normalize each vector R_i to a new vector R'_i so that each vector R'_i has a mean of 0 and a standard deviation of 1. We note $\overline{R_i}$ the mean of R_i and σ_{R_i} its standard deviation.

$$R'_i = \frac{R_i - \overline{R_i}}{\sigma_{R_i}} \tag{1}$$

We then concatenate vectors $R'_1, \cdots, R'_m$ into a $n \times m$ matrix M and use a principal component analysis [2] to reduce the dimension of M to $n \times 1$. The vector we obtain represent the aggregated usage of the different resources. We finally compute the Pearson's correlation score between this vector and the gas used. We show the results we obtain in Figure 13.

Overall we find that for both pre- and post-EIP 150 measurements the aggregation of memory and storage correlates best with gas usage. On the opposite, the CPU usage seem to be adding more noise than information and adding the CPU usage always result in a lower correlation than without it.

E. Summary

Overall, we have seen that the gas usage is dominated by the *storage allocated*, which can also be approximated reasonably well by the memory allocation during a single contract execution. On the other hand, given the high cost of storage, we find that CPU usage and gas usage correlate poorly. We summarize all bi-variate correlations as well as the best performing multivariate correlation in Figure 14.

V. Attacking the Metering Model of EVM

In light of the results we obtained in the previous sections, we hypothesize that it is possible to construct contracts which use a low amount of gas compared to the resources they use.

A. Constructing Resource Exhaustion Attacks

In particular, as we showed in Section IV that the gas consumption is dominated by the storage allocated but is not as much affected by other resources such as the clock time. Therefore, we decide to use the clock time as a target resource and look for contracts which minimize the throughput in terms of gas per second. We can formulate this as a search problem.

Problem formulation. We want to find a program which has the minimum possible throughput, where we define the throughput to be the amount of gas processed per second. Let $\mathbb{I}$ be the set of EVM instructions and P be the set of EVM programs. A program $p \in P$ is a sequence of instructions $I_1, \cdots, I_n$ where all $I_i \in \mathbb{I}$. Let $t : P \to \mathbb{R}$ be a function which takes a program as an input and outputs its execution time and $g : P \to \mathbb{N}$ be a function which takes a program as input and outputs its gas cost. We define our function to minimize $f : P \to \mathbb{R},\ f(p) = g(p)/t(p)$. Our goal is to find the program p_{slowest} such that

$$p_{\text{slowest}} = \arg\min_{p \in P}(f(p)) \tag{2}$$

The search space for a program of size n is $|\mathbb{I}|^n$. Given $|\mathbb{I}| \approx 100$, the search space is clearly too large to be explored entirely for any non-trivial program. Therefore, we cannot

simply go over the space of possible programs and instead approximate the solution.

Search strategy. With the problem formulated as a search problem, we now present our search strategy. We decide to design the search as a genetic algorithm [46]. The reasons for this choice are as follow:

- we have a well-defined fitness function f
- we have promising initialization values, which we will discuss below
- programs being a sequence of instructions, cross-over and mutations can be designed efficiently
- generated programs need to be syntactically correct but do not need to be semantically meaningful, making the cross-over and mutations more straightforward to design

We will now discuss in details how we design the initialization, cross-over and mutations of our genetic algorithm.

Program construction. Although our programs do not need to be semantically meaningful, they need to be executed successfully on the EVM, which means that they must fulfill some conditions. First, an instruction should never try to consume more values than the current number of elements on the stack. Second, instructions should not try to access random part of the EVM memory, otherwise the program could run out of gas straight away. Indeed, when an instruction reads or writes to a place in memory, the memory is "allocated" up to this position and a fee is taken for each allocated memory word. This means that if `MLOAD` would be called with 2^{100} as an argument, it would result in the cost of allocating 2^{100} words in memory, which would result in an out of gas exception. We design the program construction so that all created programs will never fail because of either of the previous reasons.

We first want to ensure that there are always enough elements on the stack to be able to execute an instruction. The instructions requiring the least number of elements on the stack are instructions such as `PUSH` or `BALANCE` which do not require any element, and the element requiring the most number of elements on the stack is `SWAP16` which requires 17 elements to be on the stack. We define the functions function $a : \mathbb{I} \to \mathbb{N}$ which returns the number of arguments consumed from the stack and $r : \mathbb{I} \to \mathbb{N}$ which returns the number of elements returned on the stack for an instruction I. We generate 18 sets of instructions using Equation 3.

$$\forall n \in [0, 17],\ \mathbb{I}_n = \{I \mid I \in \mathbb{I} \wedge a(I) \leq n\} \tag{3}$$

For example, the set $\mathbb{I}_3$ is composed of all the instructions which require 3 or less items on the stack. We will use these sets in Algorithm 1 to construct the initial programs but before, we need to define the functions we use to control memory access. For this purpose, we define two functions to handle these. First, $uses\_memory : \mathbb{I} \to \{0, 1\}$ returns 1 only if the given instruction accesses memory in some way. Then, $prepare\_stack : \mathbb{P} \times \mathbb{I} \to \mathbb{P}$ takes a program and an instruction and ensures that all the arguments of the instruction which influence which part of memory is accessed are below a relatively low value, that we arbitrarily set to 255. To ensure this, $prepare\_stack$ adds `POP` instruction for all arguments of I and add the exact same number of `PUSH1` instructions with a random value below the desired value. For example, in the case of `MLOAD`, a `POP` followed by a `PUSH1` would be generated.

Using the sets $\mathbb{I}_n$, the $uses\_memory$ and $prepare\_stack$ functions, we use Algorithm 1 to generate the program. We assume that the $biased\_sample$ function returns a random element from the given set and will discuss how we instantiate it in the next section.

Algorithm 1 Initial program construction

```
function GeneratePROGRAM(size)
    P ← ( )                                  ▷ Initial empty program
    s ← 0                                    ▷ Stack size
    for 1 to size do
        I ← biased_sample(I_s)
        if uses_memory(I) then
            P ← prepare_stack(P, I)
        end if
        P ← P · ( I )                        ▷ Append I to P
        s ← s + (r(I) − a(I))
    end for
    return P
end function
```

Initialization. As the search space is very large, it is important to start with good initial values so that the genetic algorithm can search for promising solutions. For this purpose, we use the result of the results we presented in Section III, in particular, we use the throughput measured for each instruction. We define a function $throughput : \mathbb{I} \to \mathbb{R}$ which returns the measured throughput of a single instruction. When randomly choosing the instructions with $biased\_sample$, we want to have a higher probability of picking an instruction with a low throughput but want to keep a high enough probability of picking a higher throughput instruction to make sure that these are not all discarded before the search begins. We define the weight of each instruction and then its probability with equations 4 and 5.

$$W(I \in \mathbb{I}) = \log\left(1 + \frac{1}{throughput(I)}\right) \tag{4}$$

$$P(I \in \mathbb{I}_n) = \frac{W(I)}{\sum_{I' \in \mathbb{I}_n} W(I')} \tag{5}$$

Given that the throughput can have orders of magnitude of difference among instructions, the $\log$ in Equation 4 is used to avoid assigning a probability to close to 0 to an instruction.

Cross-over. We now want to define a cross-over function over our search-space, a function which takes as input two programs and returns two programs, i.e. $cross\_over : \mathbb{P} \times \mathbb{P} \to \mathbb{P} \times \mathbb{P}$, where the output programs are combined from the input programs. To avoid enlarging the search space with invalid programs, we want to perform cross-over such that the two output programs are valid by construction. As during program creation, we must ensure that each instruction of the output

program will always have enough elements on the stack and that it will not try to read or write at random memory locations.

For the memory issue, we simply avoid modifying the program before an instruction manipulating memory or one of the `POP` or `PUSH1` added in the program construction phase. For the second issue, we make sure to always split the two programs at positions where they have the same number of elements on the stack.

We show how we perform the cross-over in Algorithm 2. In the CREATESTACKSIZEINDEX function, we create a mapping from a stack size to a set of program counters where the stack has this size. In the CROSSOVER function, we first create this mapping for both program and randomly choose a stack size to split the program. We then randomly choose a location from each program with the selected stack size. Note that here, $sample$ assigns the same probability to all elements in the set. Finally, we split each program in two at the chosen position, and cross the programs together.

Algorithm 2 Cross-over function

```
function CreateStackSizeMapping(P)
    S ← empty mapping
    pc ← 0
    s ← 0
    for I in P do
        if s ∉ S then
            S[s] ← {}
        end if
        S[s] ← S[s] ∪ {pc}
        s ← s + (r(I) − a(I))
        pc ← pc + 1
    end for
    return S
end function
function CrossOver(P1, P2)
    S1 ← CreateStackSizeMapping(P1)
    S2 ← CreateStackSizeMapping(P2)
    S ← S1 ∩ S2                          ▷ Intersection on keys
    s ← sample(S)
    i1 ← sample(S1[s])
    i2 ← sample(S2[s])
    P11, P12 ← split_at(P1, i1)
    P21, P22 ← split_at(P2, i2)
    P1' ← P11 · P22                      ▷ Concatenate
    P2' ← P21 · P12
    return P1', P2'
end function
```

Mutation. We use a straightforward mutation operator. We randomly choose an instruction I in the program where I is not one of the `POP` or `PUSH1` instructions added to handle memory issues previously discussed. We generate a set M_I of replacement candidate instructions defined as follow.

$$M_I = \{I' \mid I' \in \mathbb{I}_{a(I)} \wedge r(I') = r(I)\} \tag{6}$$

In other terms, the replacement must require at most the same number of elements on the stack and put back the same number as the replaced instruction. Then, we replace the instruction I by I', which we randomly sample from M_I. If I had `POP` or `PUSH1` associated with it to control memory, we remove them from the program. Finally, if I' uses memory, we add the necessary instructions before it.

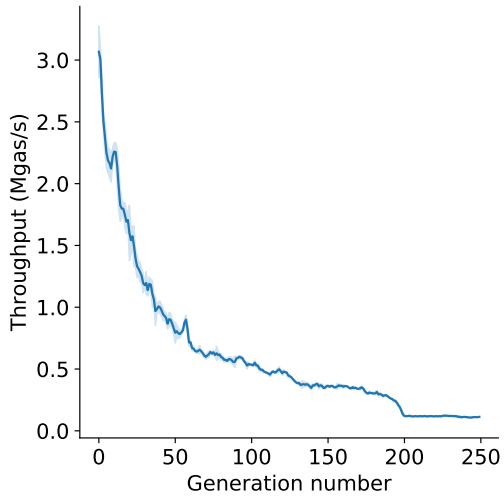


Fig. 15: Evolution of the average contract throughput as a function of the number of generations.

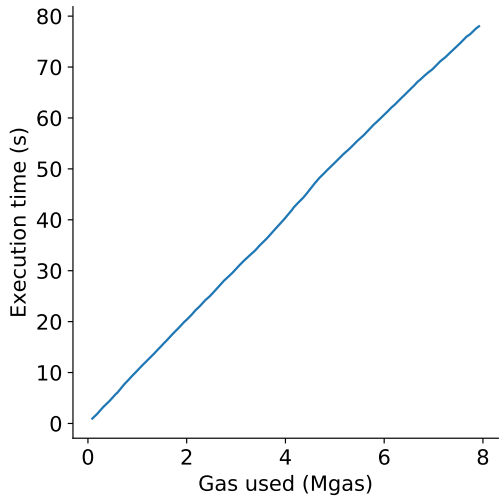


Fig. 16: Execution time as a function of number of most effective synthetic contracts within a block; separate lines for different machines are shown.

B. Effectiveness of Attacks with Synthetic Contracts

We want to measure the effectiveness of our approach to produce Resource Exhaustion Attacks. To do so, we want to generate contracts and benchmark them while mimicking the behavior of a regular full validating node as much as possible. To do so, we execute all the programs produced within every generation of our genetic algorithm, as if they were part of a single block. We use the following steps to run our genetic algorithm.

1) Clear the page cache;

```
PUSH9 0x57c2b11309b96b4c59
BLOCKHASH
SLOAD
CALLDATALOAD
PUSH7 0x25dfb360fa775a
BALANCE
MSTORE8
PUSH10 0x49f8c33edeea6ac2fe8a
PUSH14 0x1d18e6ece8b0cdbea6eb485ab61a
BALANCE
POP       ; prepare call to CALLDATACOPY
POP
POP
PUSH1 0xf7
PUSH1 0xf7
PUSH1 0xf7
CALLDATACOPY
PUSH7 0x421437ba67fe0e
ADDRESS
BLOCKHASH
```

Fig. 17: Bytecode snippet generated by our genetic programming. Instructions in bold involve some sort of IO operations.

2) Warm up caches by generating and executing randomly-generated contracts
3) Generate the initial set of program;
4) Run the genetic algorithm for n generation.

An important point here is that when running the genetic algorithm, we only want to execute each program once, otherwise every IO access will already be cached and it will invalidate the results, as this is not what would happen when a regular validating node execute contracts. However, we of course do want to execute the measurements multiples time to be able to measure the execution time standard deviation. To work around these two requirements, we save all the programs generated while we run the experiment and once the experiment has finish running, we re-run all the programs in the exact same order. We combine these results to compute the mean and standard deviation of the execution time.

Generated bytecode. Before discussing the results further, we show a small snippet of bytecode generated by our genetic programming method in Figure 17. We highlight the instructions which involve IO operations in bold and show the instructions which have for only purpose to keep the stack consistent in a smaller font. We can see that there is a large number of IO related instructions, in particular `BLOCKHASH` and `BALANCE` shows up multiple times. Although the fee of `BALANCE` has been revised from 20 to 400 in EIP 150, this suggests that the instruction is still under-priced. In the snippet, we also see that the stack is cleared and replaced with small values before calling `CALLDATACOPY`. This corresponds to the $prepare\_stack$ function described in the program construction section: to avoid `CALLDATACOPY` to read very far away in memory, which would make the program run out of gas, the arguments are replaced with small values.

Generating low-throughput contracts. We show how the throughput of the lowest performing contract evolved with the number of generations in Figure 15. The line represent the mean of the measurements and the band represents the standard deviation of the measurements. The measurements are run 3 times. Except from one point in the first measurements, overall the standard deviation remains relatively low.

We can see that during the first generations, the throughput is around 1.25M gas per second, which is already fairly low given that the average throughput for a transaction on the same machine is around 20M gas per second. This shows that our initialization is effective. The throughput decreases very quickly in the first few generations, and then steadily decreases down to around 300 K gas per second, which is more than 60 slower than the average transaction. After about 20 generations, the throughput more or less plateaus.

Exploring the minimum. The minimum in our experiments is attained at generation 244. At this point, the block uses in total approximately 7.9 M gas and takes around 78 seconds to execute. We show in Figure 16 how the execution time increases with the amount of gas consumed within the block. Given that an Ethereum block is produced roughly every 13 seconds, this means that 6 new blocks would have been created by the time the node finished validating this one.

C. REA as a Form of DoS

One of the key value proposition of Ethereum, and the base of its security model, is *decentralisation* of full nodes (and miners but this is out of scope) [12], [22]. The current requirements to run a full node on the Ethereum main net are low enough for most commodity hardware to be able to keep up without issue. The only point mentioned by the Ethereum developers is that running a full node requires an SSD [42]. Although there is currently no official documentation on other requirements, other sources estimate the minimum required memory to be about 8GB [39], [38], [37].

The main consequence of such an attack would be that full nodes running commodity hardware, with specs similar as the one we used in our experiments, would not be able to stay into sync with the network. Assuming a gas price of 2Gwei, an attacker could put such full nodes out of sync for almost 78 seconds by spending only about $3.20. Although it is hard to predict how miners would react, in theory this means that a budget of roughly $148.00 is enough to put full nodes out of sync for more than one hour. Even if the full nodes would eventually catch up, this would make them unusable for activities where being up to sync is a requirement. Given the incentive of running a full node are already limited [40], we hypothesise that this could reduce even further the number of full nodes and thereby increase the centralisation of the network.

VI. Related Work

There has been a great deal of attention focused on the correctness of smart contracts on blockchains, especially, the Ethereum blockchain. Some of the vulnerability types have to do with gas consumption, but not all. There has been relatively little attention given to the organization of metering for blockchain systems. We will first present research focusing on smart contract issues, and then highlight the work that focuses on metering at the smart contract level. We will then

present research focusing on metering at the virtual machine level — EVM in the case of Ethereum.

A. Smart Contracts

Major contracts vulnerabilities have been observed in recent years [5] with sometimes multiple millions of dollars worth of Ether at stake [43], [34]. One of the most famous exploit on the Ethereum blockchain was The DAO exploit [35], where an attacker used a re-entrancy vulnerability [32], [31] to drain funds out of The DAO smart contract. The attacker managed to drain more than 3.5 million of Ether, which would now be worth more than 700.00 million USD. Given the severity of the attack, the Ethereum community decided to hard-fork the blockchain, preventing the attacker to benefit from the Ether he had drained.

In order to prevent such exploits, many different tools have been developed over the years to detect vulnerabilities in smart contracts [28]. One of the first tools which have been developed is Oyente [32]. It uses symbolic execution to explore smart contracts execution pass and then uses an SMT solver [19] to check for several classes of vulnerabilities. Many other tools covering the same or other classes of vulnerabilities have also been developed [31], [8], [45], [30] and are usually based either on symbolic execution or static analysis methods such as data flow or control flow analysis. Some smart contract analysis tools have also focused more on analyzing vulnerabilities related to gas [27], [15], [4]. We present some of these tools in the next subsection.

B. Gas Usage and Metering

Recent work by Yang et al. [48] have recently empirically analyzed the resource usage and gas usage of the EVM instructions. They provide an in-depth analysis of the time taken for each instructions both on commodity and professional hardware. Although our work was performed independently, the results we present in Section III seem to concur mostly with their findings.

Other related themes have also been covered in the literature. One of the large theme is optimization of gas usage for smart contracts. Another one is estimating, preferably statically, the gas consumption of smart contracts.

Gas Usage Optimization: Gasper [15] is one of the first paper which has focused on finding gas related anti-patterns for smart contracts. It identifies 7 gas-costly patterns, such as dead code or expensive operations in loops, which could potentially be costly to the contract developer in terms of gas. Gasper builds a control flow graph from the EVM bytecode and uses symbolic execution backed by an SMT solver to explore the different paths that might be taken.

MadMax [27] is a static analysis tool to find gas-focused vulnerabilities. Its main difference with Gasper from a functionality point of view is that MadMax tries to find patterns which could cause out-of-gas exceptions and potentially lock the contract funds, rather than gas-intensive patterns. For example, it is able to detect loops iterating on an unbounded number of elements, such as the numbers of users, and which would therefore always run out of gas after a certain number of users. MadMax decompiles EVM contracts and encodes properties about them into Datalog to check for different patterns. It is performant enough to analyze all the contracts of the Ethereum blockchain in only 10 hours.

Gas Estimation: Marescotti et al. [33] propose two algorithms to compute upper-bound gas consumption of smart contracts. It introduces a "gas consumption path" to encode the gas consumption of a program in its program path. It uses an SMT solver to find an environment resulting in a given path and computes its gas consumption. However, this work is not implemented with actual EVM code and is therefore not evaluated on real-world contracts.

Gastap [4] is a static analysis tool which allows to compute sound upper bounds for smart contracts. This ensures that if the gas limit given to the contract is higher than the computed upper-bound, the contract is assured to terminate without out-of-gas exception. It transforms the EVM bytecode and models it in terms of equations representing the gas consumption of each instructions. It then solves these equations using the equation solver PUBS [3]. Gastap is able to compute gas upper bound on almost all real world contracts it is evaluated on.

C. Virtual Machines and Metering

Zheng et al. [49] propose a performance analysis of several blockchain systems which leverage smart contracts. Although the analysis goes beyond smart contracts metering, with metrics such as network related performance, it includes an analysis about smart contracts metering at the virtual machine level. Notably, it shows that some instructions, such as `DIV` and `SDIV`, consume the same amount of gas while their consumption of CPU resource is vastly different.

Chen et al. [16] propose an alternative gas cost mechanism for Ethereum. The gas cost mechanism is not meant to replace completely the current one, but rather to extend it in order to prevent DoS attacks caused by under-priced EVM instructions. The authors analyze the average number of execution of a single instruction in a contract, and model a gas cost mechanism to punish contracts which excessively execute a particular instruction. This gas mechanism allows normal contracts to almost not be affected by the price changes while mitigating spam attacks which have been seen on the Ethereum blockchain [11].

VII. Conclusion

In this work, we investigated the validity of the metering approach based on gas consumed by the Ethereum blockchain. We first re-executed the Ethereum blockchain for 2.5 months and showed some significant inconsistencies in the pricing of the EVM instructions. We confirmed that although discussed by the community, `BLOCKHASH` remains vastly under-priced. We further explored various other design weaknesses, such as gas costs for arithmetic EVM instructions and cache dependencies on the execution time. Additionally, we demonstrated that there is very little correlation between gas and resources such as CPU and memory. We found that the main reason for this is that the gas price is dominated by the amount of *storage* used.

Finally, we presented a new attack called *Resource Exhaustion Attack* which uses these imperfections to generate

low-throughput contracts. Our genetic programming technique is able to generate programs which exhibit a throughput of around 1.25M gas after a single generation. A minimum in our experiments is attained at generation 244 with the block using around 7.9M gas and taking around 78 seconds. We showed that we are able to generate contracts with a throughput on average 50 times slower than typical contracts. These contracts can be used to prevent nodes with lower hardware capacity from participating in the network entirely, thereby artificially reducing the level of centralization the network can deliver.

REFERENCES

[1] Ethereum - github. https://github.com/ethereum, 2019. [Online; accessed 08-September-2019].

[2] Hervé Abdi and Lynne J Williams. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics*, 2(4):433–459, 2010.

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

[4] Elvira Albert, Pablo Gordillo, Albert Rubio, and Ilya Sergey. GASTAP: A Gas Analyzer for Smart Contracts. *CoRR*, abs/1811.1, nov 2018.

[5] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts (SoK). In *POST*, 2017.

[6] Block.one. About EOSIO. https://eos.io/about-us/, 2019. [Online; accessed 04-June-2019].

[7] Sarah Boslaugh. *Statistics in a nutshell: A desktop quick reference*. " O'Reilly Media, Inc.", 2012.

[8] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, François Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A scalable security analysis framework for smart contracts. *CoRR*, abs/1809.03981, 2018.

[9] Vitalik Buterin. EIP 150: Gas cost changes for IO-heavy operations . https://eips.ethereum.org/EIPS/eip-150. [Online; accessed 05-June-2019].

[10] Vitalik Buterin. Geth nodes under attack again. https://www.reddit.com/r/ethereum/comments/55s085/geth_nodes_under_attack_again_we_are_actively/?st=itxh568s&sh=ee3628ea. [Online; accessed 4-April-2019].

[11] Vitalik Buterin. Transaction spam attack: Next Steps. https://blog.ethereum.org/2016/09/22/transaction-spam-attack-next-steps/. [Online; accessed 4-April-2019].

[12] Vitalik Buterin. A next-generation smart contract and decentralized application platform. *Ethereum*, (January):1–36, 2014.

[13] Vitalik Buterin. Geth nodes under attack again (geth issue). https://www.reddit.com/r/ethereum/comments/55s085/geth_nodes_under_attack_again_we_are_actively/d8ebsad/, 2016. [Online; accessed 05-September-2019].

[14] Vitalik Buterin. EIP 210. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-210.md, 2019. [Online; accessed 20-July-2019].

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

[16] Ting Chen, Xiaoqi Li, Ying Wang, Jiachi Chen, Zihao Li, Xiapu Luo, Man Ho Au, and Xiaosong Zhang. An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. In Joseph K. Liu and Pierangela Samarati, editors, *Information Security Practice and Experience*, pages 3–24, Cham, 2017. Springer International Publishing.

[17] Concourse Open Community. Eth gas station. https://ethgasstation.info/calculatorTxV.php, 2019. [Online; accessed 09-September-2019].

[18] Chris Dannen. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Apress, Berkely, CA, USA, 1st edition, 2017.

[19] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[20] Ethereum community. cpp-ethereum. http://www.ethdocs.org/en/latest/ethereum-clients/cpp-ethereum/. [Online; accessed 1-May-2019].

[21] Etherscan. Ethereum average block timechart. https://etherscan.io/chart/blocktime, 2019. [Online; accessed 09-September-2019].

[22] Adem Efe Gencer, Soumya Basu, Ittay Eyal, Robbert van Renesse, and Emin Gün Sirer. Decentralization in bitcoin and ethereum networks. *CoRR*, abs/1801.03998, 2018.

[23] Sanjay Ghemawat and Jeff Dean. Leveldb. https://github.com/google/leveldb, 2011. [Online; accessed 05-August-2019].

[24] Sanjay Ghemawat and Jeff Dean. Leveldb documentation. https://github.com/google/leveldb/blob/master/doc/index.md#cache, 2011. [Online; accessed 05-August-2019].

[25] The go-ethereum Authors. Official go implementation of the ethereum protocol. https://github.com/ethereum/go-ethereum/, 2019. [Online; accessed 25-August-2019].

[26] Google. Google compute engine documentation. https://cloud.google.com/compute/docs/, 2019. [Online; accessed 08-September-2019].

[27] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts. *SPLASH 2018 Oopsla*, 2(October), 2018.

[28] Dominik Harz and William Knottenbelt. Towards Safer Smart Contracts: A Survey of Languages and Verification Methods. *arXiv preprint arXiv:1809.09805*, 2018.

[29] Hudson Jameson. Ethereum Constantinople Upgrade Announcement. https://blog.ethereum.org/2019/01/11/ethereum-constantinople-upgrade-announcement/, 2019. [Online; accessed 05-July-2019].

[30] Bo Jiang, Ye Liu, and W. K. Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 259–269, New York, NY, USA, 2018. ACM.

[31] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: analyzing safety of smart contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[32] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making Smart Contracts Smarter. In *CCS*, 2016.

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

[34] Max Galka. Multisig wallets affected by the Parity wallet bug. https://github.com/elementus-io/parity-wallet-freeze. [Online; accessed 21-January-2019].

[35] Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M Kim, and Marek Laskowski. Understanding a revolutionary and flawed grand experiment in blockchain: The dao attack. *Journal of Cases on Information Technology (JCIT)*, 21(1):19–32, 2019.

[36] Kevin Owocki. A brief history of gas prices on ethereum. https://gitcoin.co/blog/a-brief-history-of-gas-prices-on-ethereum/, 2018. [Online; accessed 05-August-2019].

[37] Palau, Albert. Analyzing the hardware requirements to be an ethereum full validated node. https://medium.com/coinmonks/analyzing-the-hardware-requirements-to-be-an-ethereum-full-validated-node-dc064f167902, 2019. [Online; accessed 08-September-2019].

[38] PegaSys. Pantheon ethereum client system requirements. http://docs.pantheon.pegasys.tech/en/latest/HowTo/Get-Started/System-Requirements/, 2019. [Online; accessed 08-September-2019].

[39] Petrov, Andrev. An economic incentive for running ethereum full nodes. https://medium.com/vipnode/an-economic-incentive-for-running-ethereum-full-nodes-ecc0c9ebe22, 2018. [Online; accessed 08-September-2019].

[40] Pitts, Jamie. Incentives for running full ethereum nodes. https://ethresear.ch/t/incentives-for-running-full-ethereum-nodes/1239, 2019. [Online; accessed 08-September-2019].

[41] Dani Putney. The aztec protocol: A zero-knowledge privacy system on ethereum. https://www.ethnews.com/the-aztec-protocol-a-zero-knowledge-privacy-system-on-ethereum, 2018. [Online; accessed 23-August-2019].

[42] Schmideg, Adam. go-ethereum faq. https://github.com/ethereum/go-ethereum/wiki/FAQ, 2018. [Online; accessed 08-September-2019].

[43] Us Securities and Exchange Commission. Report of Investigation Pursuant to Section 21(a) of the Securities Exchange Act of 1934: The DAO. Technical report, U.S. Securities and Exchange Commission, 2017.

[44] Tezos. About Tezos. https://tezos.com/learn-about-tezos, 2019. [Online; accessed 04-June-2019].

[45] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 67–82, New York, NY, USA, 2018. ACM.

[46] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[47] Gavin Wood. Ethereum yellow paper, 2014.

[48] Renlord Yang, Toby Murray, Paul Rimba, and Udaya Parampalli. Empirically Analyzing Ethereum's Gas Mechanism. *CoRR*, abs/1905.0, 2019.

[49] P Zheng, Z Zheng, X Luo, X Chen, and X Liu. A Detailed and Real-Time Performance Monitoring Framework for Blockchain Systems. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 134–143, may 2017.

[50] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. Erays: Reverse Engineering Ethereum's Opaque Smart Contracts. 2018.