Broken Metre: Attacking Resource Metering in EVM

Daniel Perez
Imperial College London

Benjamin Livshits
Imperial College London,
UCL Centre for Blockchain Technologies, and Brave Software

Abstract—**Blockchain systems, such as Ethereum, use an approach called "metering" to assign a cost to smart contract execution, an approach which is designed to incentivise miners to operate the network and protect it against DoS attacks. In the past, the imperfections of Ethereum metering allowed several DoS attacks which were countered through modification of the metering mechanism.**

This paper presents a new DoS attack on Ethereum which systematically exploits its metering mechanism. We first replay and analyse several months of transactions, during which 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 execution cost and the utilised resources, such as CPU and memory. Based on these observations, we present a new type of DoS attack we call *Resource Exhaustion Attack*, which uses these imperfections to generate low-throughput contracts. To do this, we design a genetic algorithm that generates contracts with a throughput on average 100 times slower than typical contracts. We then show that all major Ethereum client implementations are vulnerable and, if running on commodity hardware, would be unable to stay in sync with the network when under attack. We argue that such an attack could be financially attractive not only for Ethereum competitors and speculators, but also for Ethereum miners. Finally, we discuss short-term and potential long-term fixes against such attacks. Our attack has been responsibly disclosed to the Ethereum Foundation and awarded a bug bounty reward of 5,000 USD.

I. Introduction

Some blockchain systems support code execution, allowing arbitrary programs to take advantage of decentralised trust. Ethereum and its virtual machine, the Ethereum Virtual Machine (EVM), is probably the most widely used blockchain adopting this approach. However, allowing arbitrary programs from non-trusted users introduces many new challenges. One of these challenges is to prevent users from running code which could negatively impact the performance of the system. To tackle this challenge, Ethereum introduced the notion of "gas", which is a unit used to measure the execution cost of a program, referred to as a "smart contract" in this context. Gas-based metering is used to price the execution of smart contracts, and must ensure that the throughput of the blockchain, in terms of gas per second, remains stable. Metering is therefore critical to keep the Ethereum blockchain safe against Denial of Service (DoS) attacks involving slow running contracts. However, assigning costs to different instructions is a highly non-trivial task, and the costs originally assigned in the Ethereum yellow paper [57], which were designed to maintain a throughput of 1 gas/μs, had many inconsistencies. As a consequence, several DoS attacks have been conducted on Ethereum [13], [12], and the gas cost has also been reviewed several times [11], [40] to increase the cost of the under-priced instructions.

To the best of our knowledge, there has still not been any attempt to try to find and exploit such inconsistencies in a systematic way. In this paper, we design a new DoS attack which exploits inconsistencies of the gas metering mechanism by taking a systematic approach to finding these. We first replay and analyse several months of transactions to discover discrepancies in the gas cost. We then use the data and insight from our analysis to design a genetic algorithm capable of generating low-throughput contracts. We evaluate the contracts generated by our algorithm on all major Ethereum clients and find that they are all vulnerable to our attack.

Contributions. This paper makes the following contributions:

1) **Exploration of metering in EVM**: We explore the history of executing 2.5 months worth of smart contracts on the Ethereum blockchain and 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) **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 minimise the throughput. We propose an approach which combines empirical data and a genetic algorithm in order to generate contracts with low throughput. We explore the efficacy of our strategy as a function of the throughput in terms of gas per second of the generated programs.
3) **Experimental evaluation**: We show that our REA can abuse imperfections in EVM's metering approach. Our genetic algorithm is able to generate programs with a throughput of 1.25M gas per second after a single generation. A minimum in our experiments is attained at generation 243 with a block using around 9.9M gas and taking about 93 seconds. We

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24267
www.ndss-symposium.org

show that our method generates contracts on average more than 100 times slower than typical contracts. Finally, we evaluate our low-throughput contracts on the major Ethereum clients and show that they are all vulnerable. Using commodity hardware, nodes would be unable to stay in sync when under attack.

4) **Disclosure and fixes**: We responsibly disclosed our attack to the Ethereum foundation, and were awarded a bug bounty reward of 5,000 USD. We discussed with the developers about the ongoing efforts as well as some potential fixes, and present some of the short-term and long-term fixes in this paper.

Paper Organisation. The rest of the paper is organised as follows. In Section II, we provide background information about Ethereum and its metering scheme, as well as a few instances of how it has been exploited in the past. In Section III, we present case studies based on measurements that we obtained by re-executing the Ethereum main chain. In Section IV, we present our Resource Exhaustion Attacks (REA) and the results we obtained. In Section V we present short and long-term solutions to gas mispricing issues. 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 [14] 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 [20]. 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* [57]. 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 following 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 [54], [7]. 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.

[1]When converting ETH to USD, we use the exchange rate on 2020-01-07: 1 ETH = 145 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

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 [57], who classify the instructions in multiple tiers of gas cost: zero Tier (0 gas), base tier (2 gas), very low tier (3 gas), low tier (5 gas), high tier (10 gas) and special tier where the cost needs more complex rules. 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 too cheap, allowing for DOS attacks on the Ethereum [12] blockchain. As a fix, EIP 150 [11] 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 [13]. 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

	Gas price	
	Low	High
Transaction type	(1Gwei)	(80Gwei)
Basic (21k gas)	\$0.00304	\$0.2436
Gas intensive (500k gas)	\$0.0725	\$5.8

Fig. 2: Fees for different type of transactions. "Low" price is one of the lowest possible price to have a transaction included while "High" is a price someone very eager to have his transaction included would pay.

Number of blocks:	613,475
Median gas price:	9.1 Gwei
Median gas used (by contracts):	53,787
Median transaction fee:	0.0008 ETH (0.116 USD)

Fig. 3: Median gas price, gas used and transaction fee from block 8,652,096 (Sep-09-2019) to block 9,286,594 (Jan-15-2020).

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. For conciseness, these amounts are often expressed in Gwei, where 1Gwei $=$ 10^9wei. 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 concrete numbers about different aspects of the gas price and transaction fees. In particular, we show the amount of transaction fees that a user would have to pay to have his transaction processed by the main Ethereum network.

To give a sense of the transaction fees, we show a variety of typical fees in Figure 2. 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 [44]. 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 [49]. At the time of writing, the maximum amount of gas which can be used in a single block is 10,000,000, which means only 20 such transactions could be included in a single block.

In Figure 3, 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 September 30, 2019 and January 15, 2020. We find that the median gas price paid by a transaction's sender is around 9.1 Gwei, which is around 9 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 in the next block is relatively low and the transaction can therefore be delayed for several blocks: at the time of writing, about 40% of the last 200 blocks accepted a gas price of 1Gwei [19]. This explains that users usually pay a higher fee to get their transaction included faster. The median for the gas consumed by contracts 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 0.0008 ETH which is around 0.116 USD.

D. Previously Known Attacks

The Ethereum network has been victim of several Denial of Service (DoS) attacks due to instructions being under-priced. 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 [13]. `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 under-priced. 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. While replaying the Ethereum history on our hardware, 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 4. Although this attack did not create any issue at 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 [25].

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 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 [12]. 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

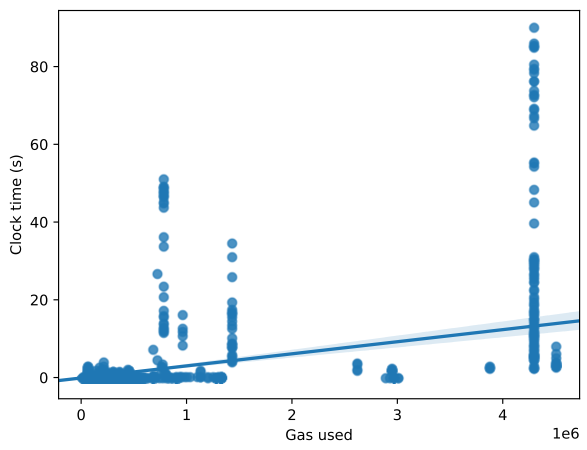
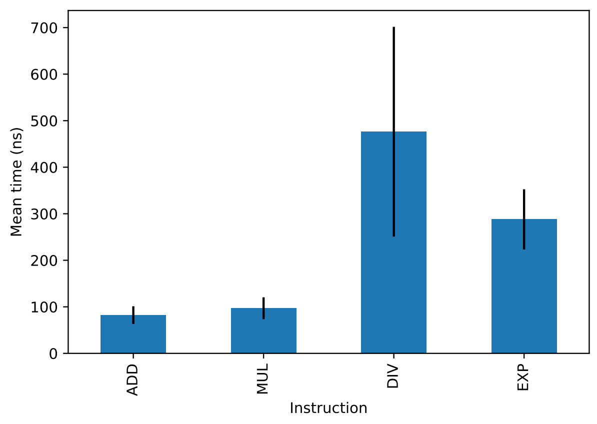


Fig. 4: Correlation between gas and clock time with DoS.



(a) Mean time for arithmetic instructions.

Instruction	Gas cost	Count	Mean time (ns)	Throughput (gas / μs)
ADD	3	453,069	82.20	36.50
MUL	5	62,818	96.96	51.57
DIV	5	107,972	476.23	10.50
EXP	~51	186,004	287.93	177.1

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

Fig. 5: Comparing execution time and gas usage of arithmetic instructions.

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 [30] which was less memory efficient [15].

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* [22], 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) [31] 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. We selected this hardware because it is representative to what has been reported as sufficient to run a full Ethereum node [48], [47], [45].

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 [30]; 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 in III-C where we want to compare after and before EIP-150.

[2]https://github.com/danhper/aleth/tree/measure-gas

Phase	Resource	Pearson score
Pre EIP-150	Memory	0.545
	CPU	0.528
	Storage	0.775
	Storage/Memory	**0.845**
	Storage/Memory/CPU	0.759
Post EIP-150	Memory	0.755
	CPU	0.507
	Storage	0.907
	Storage/Memory	**0.938**
	Storage/Memory/CPU	0.893

Fig. 6: Correlation scores between gas and system resources.

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 5a, 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 5b. `EXP` is the only of these instructions with a variable cost depending on its arguments — the value of the exponent. We use the average gas cost in our measurements to compute the throughput. 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 on average *10 times* the price of `DIV` but executes 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. Gas and System Resources Consumption

In this section, we analyse the gas consumption of Ethereum smart contracts and try to correlate it with different system resources, such as memory, CPU and storage. As described in Section II, EIP-150 influenced the price of many storage related operations, which affected the gas cost of transactions. Therefore, we use a different set of transactions than for other case studies. 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. We assume that the sample of 100,000 blocks, which roughly corresponds to two weeks, is large enough to obtain reliable data.

We use our modified Ethereum client to perform the different measurements. To measure memory, we compute the difference between the total amount of memory allocated and the total amount of memory deallocated. For CPU, we use clock time measurements as a proxy for the CPU usage. Finally, for storage usage, we count the number of EVM words (256 bits) of storage newly allocated per transactions.

We compute the Pearson correlation coefficient[3] [8] between the different resources and the gas usage. We also compute multi-variate correlation between gas consumption and multiple resources. To compute the multi-variate correlation between multiple resources and the gas usage, we first normalise the measurement vector of each targeted resource to have a mean of 0 and a standard deviation of 1. Then, we stack the vectors to obtain a matrix of m resources and n measurements, and transform it in a single vector of n measurements using a principal component analysis [2]. The vector we obtain represents the aggregated usage of the different resources and can be correlated with the gas usage.

We present our results in Figure 6. A first observation is that EIP-150 clearly emphasises the domination of storage in the price of contracts. We can clearly see that storage alone has an extremely high correlation score, with score of 0.907 after EIP-150. Memory usage is not as correlated as storage, but when combining both, they have the highest correlation score of 0.938. Finally, an important point is that CPU time seems completely uncorrelated with gas usage. Although it seems natural that CPU time by itself has a low correlation, as gas cost is dominated by storage cost, adding the CPU time in the multi-variate correlation reduces the correlation. It is not enough to make any conclusion yet but gives a hint that as long as the storage is not explicitly touched, it could be possible for contracts to be both cheap and long to execute.

D. High-Variance Instructions in the EVM

Here, we look at instructions which have a high variance in their execution time. We summarise 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 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 [16]. It discussed 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[4] [35] as it was originally planned to be.

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

[4]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 the highest execution time variance.

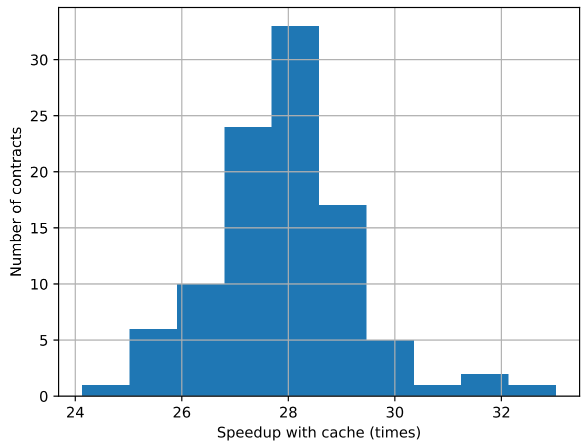


Fig. 8: Comparing throughput with and without page cache: x axis is the relative speed improvement and y axis is the number of contracts.

E. 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 [29]. In these experiments, we fix the block number at height 5,587,480.

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

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 [28], 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 use the first execution to warm-up the node and use the 9 other executions for our measurements. 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 provide anymore speedup. We plot the results for $n = 14$, $n = 15$ and $n = 16$ in Figure 9. When $n = 14$, we see that

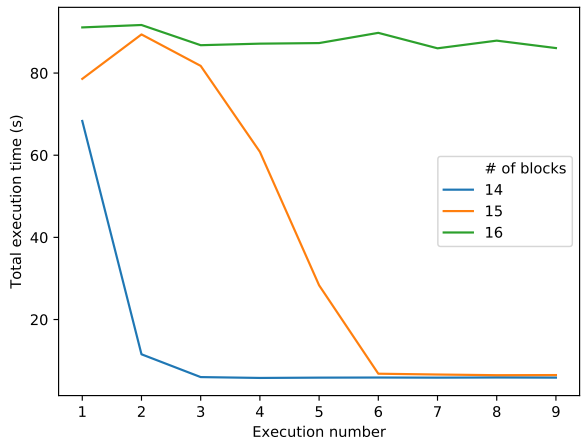


Fig. 9: Measuring block execution speed with and without the effect of cache.

the second execution is much faster than the first one, and that after the third execution, the execution time stabilises at around 6s to execute the 14 blocks. For $n = 15$, the execution time takes longer to decrease, but eventually also stabilises 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.

This means that if a deployed contract function were re-executed more than 16 blocks after its initial execution, it would execute as slowly as the first time. This shows that not only the cache has a very high impact on execution time but also that the cached information is evicted relatively quickly.

F. *Summary*

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

- 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. Attacking the Metering Model of EVM

In light of the results we obtained in the previous sections, we hypothesise 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 III, 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 minimise 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 minimise $f : P \to \mathbb{R},\ f(p) = g(p)/t(p)$. Our goal is to find the program p_{slowest} such that

$$p_{\text{slowest}} = \arg\min_{p \in P}(f(p)) \tag{1}$$

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 need to approximate the solution.

Although our problem resembles other program synthesis tasks [33], there is a notable difference. Program synthesis usually focuses on generating "meaningful" programs, either from specifications or examples. These tasks often do not have well-defined metrics allowing optimisation techniques (the genetic algorithm in our work). The task we solve is different because we need to define "valid" but not "meaningful" programs and optimise for a well-defined metric: gas throughput.

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 [56]. The reasons for this choice are as follow:

- we have a well-defined fitness function f
- we have promising initialisation 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 detail how we design the initialisation, 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 fulfil 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 parts 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.

Another potential issue would be to run into an infinite loop. However, we decide to explicitly exclude loops out of our program generation algorithm for the following reason: adding loops is unlikely to make the generated programs slower. Indeed, if a piece of code is slow enough, our genetic algorithm will tend to repeat it. The loop version could be faster if a value is already cached but have no reason to be slower.

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 $a : \mathbb{I} \rightarrow \mathbb{N}$ which returns the number of arguments consumed from the stack and $r : \mathbb{I} \rightarrow \mathbb{N}$ which returns the number of elements returned on the stack for an instruction I. We generate 18 sets of instructions using Equation 2.

$$\forall n \in [0, 17],\ \mathbb{I}_n = \{I \mid I \in \mathbb{I} \wedge a(I) \leq n\} \tag{2}$$

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, $uses\_memory : \mathbb{I} \rightarrow \{0, 1\}$ returns 1 only if the given instruction accesses memory in some way. Then, $prepare\_stack : \mathbb{P} \times \mathbb{I} \rightarrow \mathbb{P}$ 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($size$)
    $P \leftarrow (\ )$                          ▷ Initial empty program
    $s \leftarrow 0$                             ▷ Stack size
    for 1 to $size$ do
        $I \leftarrow biased\_sample(\mathbb{I}_s)$
        if $uses\_memory(I)$ then
            $P \leftarrow prepare\_stack(P, I)$
        end if
        $P \leftarrow P \cdot (\ I\ )$                 ▷ Append $I$ to $P$
        $s \leftarrow s + (r(I) - a(I))$
    end for
    return $P$
end function
```

Initialisation. 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 $throughput : \mathbb{I} \rightarrow \mathbb{R}$ 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 3 and 4.

$$W(I \in \mathbb{I}) = \log\left(1 + \frac{1}{throughput(I)}\right) \tag{3}$$

$$P(I \in \mathbb{I}_n) = \frac{W(I)}{\sum_{I' \in \mathbb{I}_n} W(I')} \tag{4}$$

Given that the throughput can have order-of-magnitude differences among instructions, the log in Equation 3 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. $cross\_over : \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P} \times \mathbb{P}$, 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 programs 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.

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.

$$M_I = \{I' \mid I' \in \mathbb{I}_{a(I)} \wedge r(I') = r(I)\} \tag{5}$$

In other words, 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.

Algorithm 2 Cross-over function

```
function CreateStackSizeMapping($P$)
    $S \leftarrow$ empty mapping
    $pc \leftarrow 0$
    $s \leftarrow 0$
    for $I$ in $P$ do
        if $s \notin S$ then
            $S[s] \leftarrow \{\}$
        end if
        $S[s] \leftarrow S[s] \cup \{pc\}$
        $s \leftarrow s + (r(I) - a(I))$
        $pc \leftarrow pc + 1$
    end for
    return $S$
end function
function CrossOver($P_1, P_2$)
    $S_1 \leftarrow$ CreateStackSizeMapping($P_1$)
    $S_2 \leftarrow$ CreateStackSizeMapping($P_2$)
    $S \leftarrow S_1 \cap S_2$                    ▷ Intersection on keys
    $s \leftarrow sample(S)$
    $i_1 \leftarrow sample(S_1[s])$
    $i_2 \leftarrow sample(S_2[s])$
    $P_{11}, P_{12} \leftarrow split\_at(P_1, i_1)$
    $P_{21}, P_{22} \leftarrow split\_at(P_2, i_2)$
    $P_1' \leftarrow P_{11} \cdot P_{22}$                    ▷ Concatenate
    $P_2' \leftarrow P_{21} \cdot P_{12}$
    return $P_1',\ P_2'$
end function
```

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 behaviour 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.

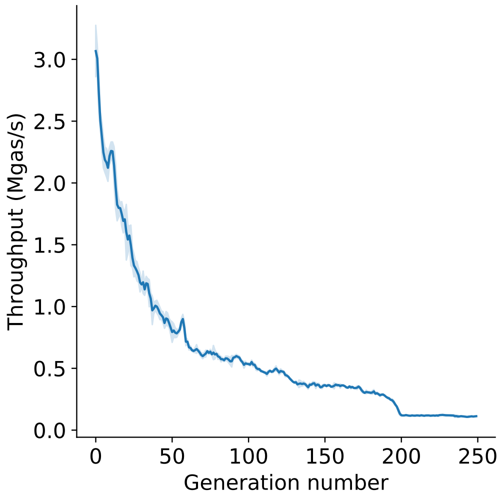


Fig. 10: Evolution of the average contract throughput as a function of the number of generations.

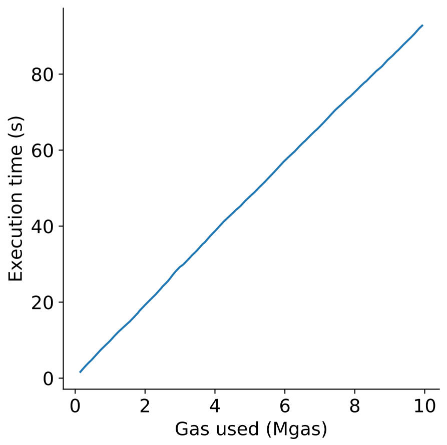


Fig. 11: Execution time as a function of amount of gas used by contracts within a block.

1) Clear the page cache;
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 executes contracts. However, we of course do want to execute the measurements multiple times 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. Once the experiment has finished, 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.

We note that generating a new generation takes on average less than 1 second but the time-consuming part of our algorithm is to compute the throughput of the generated programs. Indeed, we need to wait for the EVM to run the program, which can, as we show see in this section, take more than 90 seconds for a single generation. Furthermore, parallelising this task could bias our measurements, which forces the algorithm to perform the evaluation serially.

Generated bytecode. Before discussing the results further, we show a small snippet of bytecode generated by our genetic algorithm in Figure 12. We highlight the instructions which involve IO operations in bold and show the instructions whose sole purpose is 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` show 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*

```
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. 12: Bytecode snippet generated by our genetic algorithm. Instructions in bold involve some sort of IO operations.

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. We note that our algorithm can generate programs of arbitrary length but in our experiments we set it to create programs of around 4,000 instructions which consume between 100,000 and 150,000 gas.

Generating low-throughput contracts. We show how the throughput of the lowest performing contract evolved with the number of generations in Figure 10. The line represents 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 initialisation is effective. The throughput decreases very quickly in the first few generations, and then steadily decreases down to around 110K gas per second, which is more than 180 slower than the average transaction. After about 200 generations, the throughput more or less plateaus.

Exploring the minimum. The minimum in our experiments is attained at generation 243. At this point, the block uses in total approximately 9.9M gas and takes around 93 seconds to execute, or a throughput of about 107,000 gas per second. We show in Figure 11 how the execution time increases with the amount of gas consumed within the block. It is important to note that the execution time increases perfectly linearly with the gas used, which means that all transactions in the block have almost the same throughput. This implies that an attacker could easily tune the time he wants to delay the nodes depending on his budget. If a block full of malicious

Client	**Throughput** Gas/s	**Time** second	**IO load** MB/s
Aleth	$107,349 \pm 606.6$	93.6 ± 0.53	9.12 ± 4.70
Parity	$210,746 \pm 7,672$	47.1 ± 1.61	10.0 ± 1.36
Parity (metal)	$542,702 \pm 9,487$	18.2 ± 0.23	17.2 ± 1.97
Geth	$131,053 \pm 4,207$	75.6 ± 2.42	6.57 ± 4.13
Geth (fixed)	$3,021,038 \pm 4.67e5$	3.33 ± 0.56	0.72 ± 0.11

Fig. 13: Evaluation of different clients when executing 10M gas worth of malicious transactions. What is presented is the mean across three measurements ± standard deviation. All the measurements are performed on our GCP except the “metal” which is done on our bare-metal server.

transactions were to be processed, given that an Ethereum block is produced roughly every 13 seconds, 7 new blocks would have been created by the time the node finishes to validate the malicious one.

C. Evaluation on Other Ethereum Clients

We used aleth [22] to run our genetic algorithm and find low-throughput contracts. In this section, we show that the contracts crafted using our algorithm are also effective on the two most popular Ethereum clients: geth (v1.9.6) [30] and Parity Ethereum (v2.5.9) [46]. We also show that the fix released in geth following discussions with the development team successfully resolves the issue. Furthermore, although our attack is mainly efficient on less powerful hardware, we include the measurements of Parity on a more powerful bare-metal machine with 4 cores (8 threads) at 2.7GHZ, 32GB of RAM and an SSD with 540MB/s throughput. To benchmark the clients, we use the following procedure, and repeat the measurements three times for each client.

1) Synchronise the client to test;
2) Start the client in a private network, so that it does not execute anything else but our contracts;
3) Execute transactions on the client using the `eth_call` RPC endpoint;
    a) Send transactions to warm-up the client
    b) Send enough malicious transactions to consume 10M gas
4) Measure the gas, time, IO, CPU and memory used during the execution of the malicious transactions.

We report our results in Figure 13. Although we measured CPU, memory, and IO usage, most of the used time was related to IO operations and there was no significant increase in either CPU or memory usage during the attack. Therefore, we only report the IO measurements collected during the attack. We express the IO load in terms of MB/s, which we collect using Linux's `iotop` utility.

Before geth's fix, geth takes more than 75 seconds to execute 10M gas worth of malicious transactions. Parity Ethereum is the least vulnerable to our attack, but still takes on average about 47 seconds. Parity has on average a higher, but more constant IO load than geth. Large increase in the IO load tend to increase the IO wait time, which could explain why geth is vastly slower than Parity. Aleth is the slowest of the three clients. There could be two reasons for this: first, our algorithm is optimised on aleth, which makes it more likely to slow it down, second, aleth is less actively developed than the other two clients and might lack some optimisations.

The results of running Parity on a more powerful bare-metal server show that even such machines are relatively vulnerable to our attack. Indeed, Parity, which was the fastest of the tree clients, still took more than 18 seconds to execute the transactions. An important point to notice is that the IO throughput is considerably higher on our bare-metal server, which is most likely one of the main reason for the speedup.

Finally, we ran our attack on an improved version of geth, which the Ethereum developers pointed us to as a result of our interactions with them. This version includes several optimisations to improve the storage access speed. We can see that these improvements drastically reduced the IO load of the client. With these improvements, geth executes the transactions more than 20 times faster, making the execution speed fast enough to counter such an attack. Our interaction with geth developers shows the effectiveness of responsible vulnerability disclosure, as discussed in Section IV-E.

D. REA as a Form of DoS

Malicious contracts crafted using our algorithm could easily be used to perform a DoS attacks on Ethereum. In this section, we will describe the threat model of such an attack, including the implications and feasibility of the attack.

Attack implications. As described in Section II, there have already been several instances of DoS attacks against Ethereum [13], [12]. There are several consequences to such attacks. The most direct one is a high increase in the block production time [26], which in the worst cases more than doubled, significantly decreasing the total throughput of the network. This decrease comes not only from miners who might take more time to validate blocks but also from full nodes who are supposed to relay validated blocks and might take vastly longer to do so. A further indirect consequence of such attacks is the loss of trust in the system, which can lead to a decrease in the price of Ethereum, at least for a short period of time [18].

Probable attacker. Although instances of irrational behaviours with likely no profit to the attacker have been seen on Ethereum [9], we assume that the attacker is rational and wants to profit from such an attack. In this context, there are several ways in which such an attack could be performed.

First, this attack could be beneficial to miners. A miner could use these malicious transactions to perform a sort of selfish-mining [27]. Indeed, if the miner chooses to include a small amount of malicious transactions in the blocks he mines, the propagation time per block is likely to increase and give the miner a head-start on mining the next block. Given that the block arrival time in Ethereum is around 13 seconds, gaining a couple of seconds can be financially interesting for a miner. Furthermore, the only cost for a miner would be the opportunity cost of not including other transactions in the block, as he could include malicious transactions with a gas price of 0.

Another potential motivation for an attack could be to try to reduce the price of the ETH token and the trust in the Ethereum ecosystem. An attacker wanting to make a one-shot profit could spend some amount of money into performing such a DoS attack while taking a short position on ETH, waiting for the price to go down. Other blockchains competing with Ethereum could also potentially use such tactics to try to discredit the reliability of Ethereum.

Attack feasibility. To reason about the feasibility of this attack, we assume that given the same gas price, a malicious transaction has the same chance of being included in a block as any other transaction. We use the time we obtained in our experiments with geth, as it is the Ethereum client with the largest usage share [24].

To find a reasonable gas price, we analyse the gas price of all transactions and blocks from October 1, 2019 (block 8,653,171) to December 31, 2019 (block 9,193,265). We find that the median value of the minimum gas price in a block is around 1.1Gwei and that the average gas price is around 10Gwei with a standard deviation of 11Gwei. These values are in agreement with some other source of gas computation [19]. Finally, we find that at least 2 million gas worth of transactions are included for less than 3Gwei in about 90% of the blocks, and choose this value as the gas price to compute the cost of an attack.

Given that our malicious transactions have a throughput of about 131,000 gas per second, using a price of 3 Gwei, it would cost roughly $131,000 \times 3 \times 10^9 = 3.93 \times 10^{14}$Wei $= 3.93 \times 10^{-4}$ ETH ≈ 0.057 USD to execute code for one second. Consequently, it would cost slightly more than 0.741 USD per block to prevent nodes running on commodity hardware to keep up with the network. This is a very cheap price to pay and could indeed motivate the probable attackers discussed earlier to execute such an attack.

It is worth noting that if an attacker wanted to fill a larger portion of the block with malicious transactions, he would need to increase the gas price. Indeed, to fill half of the block with malicious transactions, it would require to pay around 15Gwei, or 5 times more per gas, than to fill only 20% of the block. This would result in a cost of $10,000,000 \times 50\% \times 1.5 \times 10^{10} = 0.075$ ETH ≈ 10.875 USD. Nevertheless, this remains a very low price to pay for an attacker with financial incentives such as the ones described earlier.

Attack limitations. 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 any issue. The only point mentioned by the Ethereum developers is that running a full node requires an SSD [50]. Although there is currently no official documentation on other requirements, other sources estimate the minimum required memory to be about 8GB [48], [47], [45]. However, there is very little information about the typical hardware setup of full nodes. Therefore, it is very difficult to accurately evaluate how many nodes would be affected by such an attack. Nevertheless, the attack was judged severe enough by the Ethereum developers to react very promptly (within less than 24 hours for the first reply and within four days for them to test the fix) after our disclosure.

E. Responsible Disclosure

Given that the attack is very easy and cheap to execute, and worked on all major clients, we went through a responsible disclosure process. The Ethereum Foundation has an official bug bounty program [23] to report vulnerabilities. With the help of colleagues[5], we wrote a report summarising our main findings, including a minimal script to execute our attack, and sent it to the bug bounty program on October 3, 2019. We received a reply the next day from the Ethereum Security Lead, acknowledging the issue and pointing us to some ongoing efforts to improve some of the inefficiencies exploited by our attack. The Ethereum foundation team also let us know that they would coordinate with Parity developers. After discussions about the ongoing efforts and some other potential solutions, we have been confirmed that our report had been awarded a reward of 5,000 USD on November 17, 2019. Finally, the official announcement was published on the bounty program website on January 7, 2020.

V. Towards a Better Approach

Gas metering and pricing is a difficult but fundamental problem in Ethereum and other blockchains which use a similar approach to price contract execution. Mispricing of gas instructions has been a concern for a long time and improvements have been included in several hard forks [11], [53]. However, there remain issues in the current Ethereum pricing model, allowing attacks such as the one we presented in the previous sections. In this section, we will discuss short-term fix which can be used to prevent DoS such as the one presented in this paper, and then briefly present longer-term potential solutions which are still being actively researched.

The main attack vector presented in this paper comes from the low speed of searching for an account which is not currently cached. One of the main issues is that the state of Ethereum gets larger with time. This means that operations accessing the state get more expensive with time in terms of resource usage.

Short-term fixes. Short-term fixes for slow IO related issues can be categorised in the two following classes: increase in the gas cost of IO instructions, as seen in EIP-150 [11] and EIP-2200 [53], and improvements in the speed of Ethereum clients.

Increasing the cost of IO instructions improves the fairness of the gas costs yet is often not sufficient to protect against DoS attacks, albeit it does increase their cost. The attack we present in this paper uses mainly instructions whose prices have increased in EIP-150 or EIP-2200, but remains relatively cheap to execute.

Improvements involve adding more layers of caches to reduce the number of IO accesses, which are typically the bottleneck. However, this requires to keep more data in memory and therefore creates a trade-off between memory consumption and execution speed. Regarding account lookup, two cases must be considered: when the looked up account exists and when it does not. Naively caching all the accounts could allow an attacker to easily evict existing accounts from the cache and is therefore dangerous. To check whether a particular account exists, a Bloom filter can be utilised as a first test. This eliminates the need for most of the IO accesses in case the queried address does not exist, while keeping a relatively low memory footprint [43]. The next case which needs to be handled is the fast lookup of existing accounts. The current attempt to do this keeps an on-disk dynamic snapshot of the accounts state [52], which allows to perform an on-disk look up of an account in $\mathcal{O}(1)$, at the cost of increasing the storage usage of the node. This indeed solves the bottleneck of accessing account data but is very specific to this particular issue.

Long-term fixes. Long-term fixes are likely to only arrive in Ethereum 2.0, as most of them will require major and breaking changes. There have been several solutions discussed by the community and other researchers, which can mostly be categorised as either a) changing the gas pricing mechanism or b) changing the way clients store data.

Current proposals to change the gas mechanism involve making the pricing more dynamic that it is currently. Chen et al. [18] propose a mechanism where contracts using a single instruction in excess would be penalised. The threshold is set using historical data in order to penalise only contracts which diverge too much from what would be a regular usage. Although the approach has some advantages over the current pricing mechanism, it is unclear how well it would be able to prevent attacks taking this mechanism into account.

A promising and actively researched approach is the use of stateless clients and stateless validation. The key idea is that instead of forcing clients to store the whole state, entity emitting transactions must send the transaction, the data needed by the transactions, and a proof that this data is correct. The proof can be fairly trivially constructed as a Merkle proof, as the block headers hold a hash of the root of the state and the state can be represented as a Merkle tree. This allows such clients to verify all transactions without accessing IO resources at all, making execution and storage much cheaper, at the cost of an increased complexity when creating transactions and a higher bandwidth usage.

Another active area of research which should help making things better in this direction is sharding [3]. Although sharding does not address the fundamental issue of gas pricing in the presence of IO operations, it does help to keep the state of the nodes smaller, as different shards will be responsible for storing the state of different parts 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 organisation 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 [6] with sometimes multiple millions of dollars worth of Ether at stake [51], [41]. One of the most famous

[5]Matthias Egli and Hubert Ritzdorf from PwC Switzerland

exploit on the Ethereum blockchain was The DAO exploit [42], where an attacker used a re-entrancy vulnerability [38], [37] 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 507.5 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 [34]. One of the first tools which have been developed is Oyente [38]. It uses symbolic execution to explore smart contracts execution pass and then uses an SMT solver [21] to check for several classes of vulnerabilities. Many other tools covering the same or other classes of vulnerabilities have also been developed [37], [10], [55], [36] 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 analysing vulnerabilities related to gas [32], [17], [5]. We present some of these tools in the next subsection.

B. Gas Usage and Metering

Recent work by Yang et al. [58] have recently empirically analysed 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 optimisation of gas usage for smart contracts. Another one is estimating, preferably statically, the gas consumption of smart contracts.

Gas Usage Optimisation: Gasper [17] 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 [32] 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 analyse all the contracts of the Ethereum blockchain in only 10 hours.

Gas Estimation: Marescotti et al. [39] 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 [5] 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 [4]. 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. [59] 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. [18] 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 analyse 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 [13].

VII. Conclusion

In this work, we presented a new DoS attack on Ethereum by exploiting the metering mechanism. We first re-executed the Ethereum blockchain for 2.5 months and showed some significant inconsistencies in the pricing of the EVM instructions. 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.

Based on our observations, we presented a new attack called *Resource Exhaustion Attack* which systematically exploits these imperfections to generate low-throughput contracts. Our genetic algorithm is able to generate programs which exhibit a throughput of around 1.25M gas per second after a single generation. A minimum in our experiments is attained at generation 243 with the block using around 9.9M gas and taking around 93 seconds. We showed that we are able to generate contracts with a throughput as low as 107,000 gas per second, or on average more than 100 times slower than typical contracts, and that all major Ethereum clients are vulnerable. We argued that several attackers such as speculators, Ethereum competitors or even miners could have financial incentives to perform such an attack. Finally, we discussed about short-term and potential long-term fixes for gas mispricing. Our attack went through the a responsible disclosure process and has been awarded a bug bounty reward of 5,000 USD by the Ethereum foundation.

Acknowledgment

The authors would like to thank Matthias Egli and Hubert Ritzdorf from PwC Switzerland for their insightful feedback and their help with responsible disclosure and experiments.

The authors would also like to thank the Tezos Foundation for their financial support.

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] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform. *arXiv preprint arXiv:1708.03778*, 2017.

[4] 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.

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

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

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

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

[9] Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. An In-Depth Look at the Parity Multisig Bug.

[10] 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.

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

[12] 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].

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

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

[15] 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].

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

[17] 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.

[18] 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.

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

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

[21] 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.

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

[23] Ethereum Foundation. Ethereum bounty program. https://bounty.ethereum.org/, 2020. [Online; accessed 05-January-2020].

[24] ethernodes.org. Ethereum mainnet statistics. https://www.ethernodes.org/, 2020. [Online; accessed 10-January-2020].

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

[26] Etherscan. Ethereum average block time chart. https://etherscan.io/chart/blocktime, 2020. [Online; accessed 10-January-2020].

[27] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International conference on financial cryptography and data security*, pages 436–454. Springer, 2014.

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

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

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

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

[32] 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.

[33] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

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

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

[36] 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.

[37] 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.

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

[39] 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.

[40] Martin Holst Swende. EIP 1184. https://eips.ethereum.org/EIPS/eip-1884, 2019. [Online; accessed 15-Jan-2020].

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

[42] 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.

[43] Michael Mitzenmacher. Compressed bloom filters. *IEEE/ACM Transactions on Networking (TON)*, 10(5):604–612, 2002.

[44] 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].

[45] 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].

[46] Parity Technologies. Parity ethereum. https://www.parity.io/ethereum/, 2020. [Online; accessed 05-January-2020].

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

[48] 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].

[49] 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].

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

[51] 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.

[52] Szilágyi, Péter. Dynamic state snapshot. https://github.com/ethereum/go-ethereum/pull/20152, 2019. [Online; accessed 05-January-2020].

[53] Wei Tang. EIP 2200: Structured Definitions for Net Gas. https://eips.ethereum.org/EIPS/eip-2200. [Online; accessed 10-January-2019].

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

[55] 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.

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

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

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

[59] 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.