GASHOLE - A DECENTRALIZED GAS INFORMATION ORACLE
ZEROTH DRAFT - NOT UNDER REVIEW

ERIC TU & NATHAN RUSH
FOUNDERS, CEO, CFO, CTO, GASHOLE LLC. INCORPORATED GMBH
ERIC.TU@CONSENSYS.NET & NATHAN.RUSH@CONSENSYS.NET

Abstract. This paper describes the first implementation of GasHole, an oracle which provides information on gas statistics that is decentralized, merkle proofifyable. The code can be used by individuals to create new markets that are based on gas price. It can also be used in state channels as a mechanism to provide dynamic locking periods. The first generation of the implemention of this oracle is written in Solidty and running on top of the Ethereum blockchain.

1. Introduction

Black gold. Texas tea.

Currently, the EVM is not aware of any gas related statistics. To some degree, this design decision makes sense; unnecessary bloat in is unnecessary.

However, as we will demonstrate in this paper, there are a variety of useful applications that can be built when the EVM is made aware of information about gas.

In this paper, we present a technique for getting a variety of gas statistics into the blockchain in a fast, efficient, and always-eventually-correct way.

The protocol is incredibly efficient in the average case, and faults are perfectly attributable when they occur.

Below, we explain the protocol, prove that miners participating is an equilibrium, and explore further areas of research and development. We also explore possible interesting applications of the protocol.

2. The Game

In this section, we describe the economic game that leads to the creation of the GasHole oracle.

There are three participants in the game: the *Requester*, the *Provider*, and the *Challenger*.

The game begins when the *Provider* registers with the GasHole contract. To do so, they must provide some large deposit X with a call to the function *register*. The size of the deposit will be specified in later sections.

When there is a registered *Provider*, a *Requester* initiates use of the oracle by requesting a specific type of data for a specific block. The data that can be requested will be defined in later sections.

Then, the *Provider* will provide the statistics that the *Requester* asked for by a call to the function *submitStat* with the claimed statistic.

The *Provider* gives this information under threat of punishment; if they post the wrong statistics, then a *Challenger* may challenge them. When this challenge is initiated, the *Challenger* must also provide some deposit.

If a *Challenger* initiates a challenge, then the oracle will enter the challenged state. This will result in a "challenge game" being played; the winner of such game will receive the deposit of the other.

This challenge game is described below.

3. The Challenge Game

This section will describe the game played by a *Challenger* and a *Provider* in the case of a challenge.

The game is completely deterministic and will absolutely be won by the correct party. The result of the game is the correct value of the statistic that was asked for earlier by the *Requester*.

The reason this game is not played every time a statistic was asked for is because it is very inefficient. As a result, the *Provider's* deposit must be high enough to incentives the *Challenger* to challenge in the case that the *Provider* gives incorrect information.

The game proceeds as follows: 1. An underlying *Database* contract is alerted of the disagreement. 2. The *Challenger* is required to first submit the necessary block headers to the *Database* contract, which is able to check them against *block.blockhash* (which is known in-protocol). 3. The *Challenger* then must provide Merkle Patricia proofs to the contract of every transaction in the relevant blocks, in order. 4. The smart contract is thus able to perform calculations using these transactions and compute the real value of the underlying data requested. 5. After this many-block verification game, the *Database* contract can report the result to the GasHole contract, where it can slash the lying party accordingly.

Note, this Merkle Patricia proof verification is done with proofs generated off-chain with (slightyly modified) JavaScript code written by the wonderful Zac Mitton.

4. Data and Applications

In this section, we describe data that it is possible to derive from this process. Though our oracle cannot currently derive all of this information, all of this information is trivially possible to derive and integrate into the system using the games describe above.

Furthermore, we present some novel applications of these statistics.

Possible Statistics:

(1) Total Gas Used in Block X.
(2) Total Gas Used over Blocks $[X, X + Y]$.
(3) Average Gas Price in Block X.
(4) Average Gas Price over Blocks $[X, X + Y]$.
(5) Change in Gas Limit for each Block.
(6) All information contained in a block header at any point in time.

(7) Any information contained in any transaction or receipt in any block.
(8) Any piece of state at any point in time.

With the aforementioned statistics, we can have a profound impact in the following applications:

(1) Gas price prediction markets.
(2) Elastic time locks.
(3) Gas limit change incentives.
(4) Fraud proofs.
(5) Prediction market integration without any changes to code.

4.1. **Gas Price Prediction Markets.** As the oracle can determine average gas prices over some periods, we are now able to build prediction markets on this average gas price. This effectively allows dApp developers that hold Ether to hedge their risk against falling Ether prices.

Assume there is some dApp developer that holds X ether at some time. If the price of Ether drops in the future, the relative amount of "buying power" that the dApp developer has is less (as gas price will rise w/ the fall of Ether price).

However, if there is a gas price prediction market, this dApp developer could bet on rising gas prices. If Ether price fell and gas prices rose, the dApp developer would be able to take their profits from this market, thus hedging their risk in holding Ether.

4.2. **Elastic Time Locks.** Currently, the lock times in predictions are not flexible and does not take into consideration network attacks congestion. With a gas oracle, we can allow for elastic time lock by the following mechanism. If the gas used over the past X blocks is greater than some threshold (or has not been supplied yet), then extend some time lock.

Examples of when this would have been useful was the Status ICO. During the time of the ICO, there was massive network congestion which consequently caused ENS bidders to lose their deposits. It would also prevent certain attacks on state channels where an old state is posted right before a time of high network congestion.

4.3. **Dynamic Block Gas Limits.** With the current implementation of GasHole, we are able to get data of previously mined block and we are able to have new blocks commited to the oracle really quickly.

This allows for participants in the Ethereum network to be able to create on-chain governance structures to dynamically change Block Gas Limits.

The mechanism is quite simple, we can have a smart contract system that allows users to deposit funds into in return for voting power. The participants in this system can then use their voting power to vote on whether they want an increase or decrease in the Block Gas Limit.

W.L.O.G., assume that we have consensus from our governance contract over a higher Block Gas Limit. Then the miners have one of two choices: lower or raise the block gas limit in the next block. If the miners follow the consensus result, they will get a reward based on the bonds posted by the voters in the system. Therefore, miners have direct financial incentive to follow the consensus outcome by the mechanism described above.

4.4. **Fraud Proofs.** As our oracle has access to all historical information about any transaction, state, or receipts in any block at any point in time, we can allow for easy fraud proof contracts to be created.

Essentially, these fraud proof contracts can allow some user to deposit ether and state "this money can be taken from me under some condition," where the condition is a statement about the existence of some transaction, state, or receipt at some point in time.

Using this oracle, proof that this slashing condition has been violated can be proven directly, with easy integration.

4.5. **Prediction Market Integration.** Very similarly to the above-explained fraud proofs, this oracle allows for easy prediction market integration with any contracts that already exist on-chain, that may have not been designed for prediction market integration.

Essentially, succinct state proofs can be provided that prove that some variable has some value in some contract at some point in time (note: all variables have a getter now). This allows for prediction markets to use these variables as oracle reporters at some point in time, even for contracts that were not designed for this.

5. Miners as Providers and Challengers

In this section, we will prove that miners both supplying the data and challenging incorrect data is an equilibrium, and will result in only miners fulfilling both of these roles.

Assume that the economic incentive provided by the *Requesters* is enough to encourage some *Providers* to enter the market. If this is not the case, this system will not be used, and therefore this is a reasonable assumption.

Let P be the set of all *Providers*. $\forall p \in P, t(p) = miner \vee t(p) = nonminer$, where t(x) is a function that returns the role of x in the greater Ethereum ecosystem.

If, $\exists p \in P$, s.t. $t(p) = nonminer$, then it is clear that miners (who may or may not be in P) will be able to front-run the data submission for negligible cost. As our assumptions say there is profit to be made, and the miners cost is strictly less than the producer of the data (as they are front-running rather than doing any actual calculation), they thus will receive profit for doing so.

As miners make profit this way, and cannot make more profit any other way, this is a dominant strategy. The best response to this dominant strategy, for all $p \in P$ where $t(p) = nonminer$, is to leave to market. Else, these non-miners will lose money, due to miner front running.

Consider the case where $\forall p \in P, t(p) = miner$. This case holds trivially.

The proof for C, the set of *Challengers*, is symmetric. Thus, it holds that $\forall x \in P \cup C, t(x) = miner$.

6. Remaining Issues

6.1. **Issues with Miners:** Currently, miners can totally control and manipulate the oracle at essentially zero cost. To do so, they can merely include fake transactions with a high (or low) gas price depending on what type of manipulation they want to perform. The cost of this is very minimal: just the cost of the increase in likelihood of becoming an uncle due to increase block size.

This will be mostly resolved with the introduction of transaction fee burning, a probably-planned feature for future Ethereum improvements. This will make it costly for miners to manipulate our oracle over extended periods of time.

6.2. **Size Attacks.** Assume a *Requester* asks for gas statistics for block X, and a *Provider* gives the wrong information. This results in a challenge by a *Challenger*, which is resolved in, say, block X + 1.

Then, assume a textitRequester asks for gas statistics for block X + 1. It is clear to see that the proof submitted for block X + 1 (in the case of a challenge game) must contain the proofs submitted for block X in the previous challenge game.

Quickly, it will become impossible to submit transactions that give proof of other transactions, due to the recursive nature of this.

There are a couple of solutions to this problem. First, it's worth noting that a solution like Sharding (a proposed future improvement to Ethereum) will solve this issue. Other than that, we can allow transaction proofs to submitted in sections and stored in-between transactions (though this is very expensive). Finally, we could build the gas oracle on another chain.

7. Acknowledgements

We would like to thank everyone at Consensys for their love and support. Non of this would be possible without the Ethereum Foundation as well. Thanks to our parents. All 16 of them.

Huge thanks to Zac Mitton for the library that generate these Merkle Patricia proofs. Thanks to Sam Mayo for his work with RBRelay. Thanks to Dino for his helpful advice. Cheers.