Master Thesis

A Quantitative Analysis of the Ethereum Fee Market: How Storing Gas Can Result in More Predictable Prices

Matthias Nadler

Supervised by:
Prof. Dr. Fabian Schär
Credit Suisse Asset Management (Schweiz) Professor for
Distributed Ledger Technologies and Fintech
Center for Innovative Finance, University of Basel

Abstract

Transaction fees on the Ethereum blockchain - measured in units of gas - are very volatile and difficult to predict. So called "gas tokens" use a refund mechanism of the Ethereum protocol to store and release gas. With transaction data from the past 18 months we analyse the temporal association of gas prices and the minting of gas tokens. We then proceed to develop an optimized gas token with an integrated, automated market maker and show that this new Liquid Gas Token (LGT) outperforms the existing gas tokens, especially for transactions which make use of the integrated exchange protocol. With the LGT and the associated dApp we aim at streamlining the benefits of gas storing to increase the accessibility for less experienced users.

Keywords: Ethereum, Smart Contracts, Transaction Fees, Gas.
JEL: G19, C41, C61

Contents

Plagiatserklärung

Ich bezeuge mit meiner Unterschrift, dass meine Angaben über die bei der Abfassung meiner Arbeit benutzten Hilfsmittel sowie über die mir zuteil gewordene Hilfe in jeder Hinsicht der Wahrheit entsprechen und vollständig sind. Ich habe das Merkblatt zu Plagiat und Betrug vom 22. Februar 2011 gelesen und bin mir der Konsequenzen eines solchen Handelns bewusst.

Matthias Nadler

1 Introduction

Ethereum is the largest blockchain with support for complex smart contracts, both by market capitalization (CoinMarketCap (2020)) and by developer activity (Electric Capital (2019)). It hosts a wide range of decentralized applications (dApps) including decentralized financial services (DeFi), games and social platforms.

To interact with smart contracts on Ethereum, the users submit transactions to the network which miners then include in new blocks to reflect the updated state of the blockchain. During 2019 a new block was mined on average every 14.7 seconds and contained on average 113 transactions (7.7 transactions per second). The amount of computation performed by each block is limited, with more complex transactions taking up more space in a block.

State-changing transactions on Ethereum are therefore a limited resource and a fee is associated with sending a transaction. This fee depends primarily on two factors: The complexity of the transaction and the current congestion of the network (the demand for space in a block). The complexity of a transaction is deterministic and can not be changed under normal circumstances; the congestion of the network on the other hand is very volatile and has a large impact on transaction fees. This leads to an uncertainty about costs when interacting with smart contracts. In this thesis we will quantify the volatility of these transaction costs and propose a smart-contract based method which can lead to a reduction in gas price volatility.

1.1 Ethereum Transactions and Fees

According to the Ethereum Yellow Paper by Gavin (2014):

> "In order to avoid issues of network abuse and to sidestep the inevitable questions stemming from Turing completeness,

> all programmable computation in Ethereum is subject to fees. The fee schedule is specified in units of *gas*."

Every transaction has a fixed, deterministic computational cost associated with it, measured in **gas**. For example, adding two numbers costs 3 gas and reading the balance of an account costs 700 gas. A detailed list of gas costs for each operation can be found in Appendix G of the Ethereum Yellow Paper (Gavin (2014)).

When creating a new transaction, the transactor has to specify the following parameters related to fees:

gasLimit The maximum amount of gas that can be consumed by the transaction.

gasPrice The price in Ether the transactor is willing to pay per unit of gas consumed.

The transaction is then signed by the transactor and sent to an Ethereum node. From there it is broadcast to other nodes and distributed across the network. When the transaction reaches a miner node, the miner can add it to their pool of pending transactions (also called "mempool").

The Ethereum Yellow Paper (Gavin (2014)) further states:

> "Transactors are free to specify any **gasPrice** that they wish, however miners are free to ignore transactions as they choose. A higher gas price on a transaction will therefore cost the sender more in terms of Ether and deliver a greater value to the miner and thus will more likely be selected for inclusion by more miners."

There are no rules for miners which transactions they select from their mempool to include in a new block, but there is a limit to how much total gas (the cumulative gas used by all included transactions) can be

in a single block[1]. Additionally, the miner of a block receives all the fees associated with the included transactions. If we assume the miner as an independent, profit-maximizing individual, they are incentivized to include the transactions from their mempool with the highest gas price. We will therefore look at the Ethereum gas fee market as a first price auction market: Transactors submit a bid, and then if they are included they pay exactly the bid that they submit.

1.2 Selecting an Optimal Gas Price

When deciding which gas price to set for a transaction, the transactor needs to consider the time sensitivity of their transaction and the gas price of other transactions currently in the mempool[2].

With a first price auction model, the transactor is incentivized to submit a gas price that reflects their value of the transaction. This is however very inefficient and would lead to gross overpaying in some cases. These inefficiencies are outlined in EIP-1559 by Buterin et al. (2019) with a proposal to "replace this with a mechanism that adjusts a base network fee based on network demand."

A better approach to selecting a gas price in the current system is to try and predict what the minimum gas price to be included within the next n blocks would have to be, where n corresponds to the time sensitivity of the transaction. The most popular resource for predicting gas prices is ETH Gas Station (ETHGasStation (2020)). ETH Gas Station tries to estimate the confirmation time of a transaction given data from the last 10'000 blocks ($\sim$ 41 hours) by running a poisson regression on the gas price and the gas used by the transaction.

In May 2020 the German startup Upvest announced their new model "FeNN" or "Fee estimations using Neural Networks" in an article by Val-

[1]This limit, called block gas limit, it not determined by the protocol, but agreed upon by the mining community.

[2]Each node has its own mempool, but for simplicity we assume a shared mempool that includes all pending transactions.

son (2020). Using state of the art machine learning technology and adding more "factors affecting gas price volatility and network properties signalling demand", they trained a model where "FeNN gas price inferences are overall cheaper by 13.35% compared to [ETH Gas Station]." Their research is currently not published, but they state that "Open-sourcing the entire codebase is among our top priorities." Further research into this topic will help to eliminate some of the inefficiencies of a first price auction market.

1.3 Transaction Gas Refunds

Every state-changing transaction consumes a fixed, deterministic amount of gas which has to be paid for by the transactor. To send a transaction on Ethereum, the transactor has to specify a gas limit T_g, which is the maximum amount of gas that can be consumed by the transaction, and a gas price T_p which is the cost in Ether the transactor is willing to pay per unit of gas consumed.

During the execution of a transaction, the Ethereum Virtual Machine (EVM) will keep track of the remaining gas, g' which is initially equal to the gas available for the transaction g. Before executing the transaction, an amount of gas, called the intrinsic gas $g0$, has to be paid[3] and is deducted from T_g. This gives us the gas available for the transaction: $g \equiv T_g - g0$.

Every operation executed by the transaction will then reduce the remaining gas g' by the amount associated with that operation. A detailed list of gas costs for each operation can be found in Appendix G of the Ethereum Yellow Paper (Gavin (2014)). The idea is to price each operation according to its computational cost.

If at any point g' would be decreased below zero, the transaction immediately fails and the state is reverted. The cost for the consumed gas

[3]See section 6.2. in the Ethereum Yellow Paper (Gavin (2014)). This includes for example the cost to increment the nonce of the sender account.

($\sim T_g T_p$) is still deducted from the transactors Ether balance.

If the transaction uses less gas than provided, $g' > 0$, the remaining gas g' will be refunded. The transactor only pays for the gas used by the transaction.

To provide an incentive to delete unused storage on the blockchain, the Ethereum Yellow Paper (Gavin (2014)) specifies two operations that trigger a gas refund:

- Using SELFDESTRUCT to destroy a contract
- Using SSTORE in order to reset contract storage to zero from some none-zero value

This gas refund is not simply added to the remaining gas g', but it is tracked in the *accrued transaction substate* A. More specifically, the set of selfdestructed contracts is tracked in A_s and the gas refunded through SSTORE is tracked in A_r. The effective refund is caluclated at the end of a transaction using g', A_s and A_r as seen below.

This has the immediate consequence that g must always be greater than the total cost of the transaction, even if gas refunds occur during the execution.

1.3.1 Calculating Gas Refunds

Based on the Ethereum Yellow Paper (Gavin (2014)), after the transaction is processed, the total gas refunded g^* is calculated by first adding up the refunds tracked in A:

$$A'_r \equiv A_r + \sum_{i \in A_s} R_{selfdestruct} \tag{1}$$

Where A_r is the total balance of refunds through SSTORE, A_s is the count of selfdestructed contracts and $R_{selfdestruct}$ is the amount of gas refunded for selfdestructing a contract.

At the time of writing, resetting one contract storage variable via SSTORE[4] will add 15000 gas to A_r and $R_{selfdestruct}$ is priced at 24000 gas.

The total amount refunded g^* consists of the remaining gas g' plus the total refund A'_r which is capped at a maximum of half (rounded down) of the total gas used[5]:

$$g^* \equiv g' + min\left\{\left\lfloor\frac{T_g - g'}{2}\right\rfloor, A'_r\right\} \tag{2}$$

The total amount of Ether the owner of the transaction has to pay is thus:

$$C \equiv (T_g - g' - g^*)T_p \tag{3}$$

Where T_p is the gas price per unit of gas in Ether associated with the transaction.

1.3.2 Consequences for the Miners

The Ether paid for a transaction as defined in (3) is deducted from the balance of the transactor and given to the miner of the block where the transaction is included.

Since refunds will reduce the gas cost of a transaction but not their computational cost, this leads to a situation where the miner has to process a transaction where the income they receive is no longer in relation to the computational cost of the transaction. The miner, and later every node in the network, pay for the computational overhead. Note that the miner's Ether income is not directly affected; they are still able to fill their block and be paid for every gas included. The additional cost for the miner inflicted by refunds is purely computational.

[4]Note that resetting a variable this way will still incur the costs for SSTORE of 5000 gas which is deducted from g'.

[5]For more details see section 6.2 in the Ethereum Yellow Paper (Gavin (2014)).

This is still not in the interest of the miner and as a profit maximizing individual, they would benefit from including this in their decision making process when selecting which transactions to include into a block. However, after looking at the source code for the most popular mining clients (Geth and Parity), we could not find any evidence of such behaviour.

The most likely explanation for this is that the additional cost of checking if a transaction contains any refunds and then calculating the resulting loss is more expensive than simply ignoring the refunds and accepting that there will be a small loss on some transactions. When used as intended, these refunds will occur rarely and in small numbers, further discouraging checking for them.

2 Storing Gas on Ethereum

Using the insights from section 1.3, we can regard any non-zero contract storage value and every contract itself as a resource that can be freed (destroyed) to trigger a refund. We will focus on contracts and self-destruct to illustrate the concept, but the same applies to contract storage values in a similar way.

Lets assume the simplest possible contract that can be self-destructed:

Algorithm 1: A simple, self-destructible contract.

```
pragma solidity 0.6.9;
contract Storage {
    fallback() external {
        selfdestruct(msg.sender);
    }
}
```

We make use of the fallback function to trigger the self-destruct, so we don't have the overhead of the function selector. Deploying this contract on the Ethereum blockchain costs 69'217 gas.

We can now at any time send an empty call to the contract and trigger the

self-destruct, which will add 24'000 gas to our refund counter. Calling the fallback function on this contract will report the gas used by the transaction to be 13'022 gas. Considering that the intrinsic gas cost $g0$ for this call alone is 21'000 gas, we can see that a refund happened. The reason why we do not see the full 24'000 gas refunded is that the self-destruct call has a cost associated with it and the refund is capped at half the total cost as seen in section 1.3.1.

To make full use of the refund, we need to combine the self-destruct operation with another transaction. This can only be achieved with the aid of a smart contract, as calls from an account can't involve more than a single statement. Algorithm 2 shows such a contract:

Algorithm 2: A contract that sets 5 variables and can claim the refund from destroying another contract.

```
pragma solidity 0.6.9;
contract Incrementer {
    mapping(uint => uint) public counters;
    function _increment() internal {
        for(uint256 i = 0; i < 5; i++) {
            counters[i]++;
        }
    }
    constructor() public {
        _increment();
    }
    function increment(address storageContract) external {
        storageContract.call("");
        _increment();
    }
    function increment() external {
        _increment();
    }
}
```

Note that we can trigger the refund even before we spend the gas. This is possible since the remaining gas and the accruing refund are tracked separately.

Calling Incrementer.increment() on the above contract will cost 50'929

gas, split as follows:

```
Initial call cost [21064 gas]
Incrementer.increment [159 gas]
-- Incrementer._increment [29706 gas]
```

Calling Incrementer.increment(storageAddress) where storageAddress is the Ethereum address of our storage contract will cost 33'181 gas, split as follows:

```
Initial call cost  [21432 gas]
Incrementer.increment  [299 gas]
-- Storage [CALL] [-18256 gas]
-- Incrementer._increment  [29706 gas]
```

The self-destruction of the storage contract refunded 18'256 gas, which means the self-destruct operation itself cost 5'744 gas ($24'000 - 18'256$). This number can be different depending on how efficiently the code is implemented, but the base fee for self-destruct is always 5'000 gas (see the Yellow Paper, Appendix G Gavin (2014)). Furthermore we also have a slight overhead to read the function argument and call an external contract, bringing the total savings down to 17'748 gas.

We spent 69'217 gas to save 17'748 gas. However, the storing and freeing of the gas was done in different transactions. In section 1.2 we saw that the gas price of transactions can be different, depending on the current network load. This implies, that if we can deploy the storage contract during times of low network load with a gas price of P_{low} and self-destruct it during times of high network load with a gas price of P_{high}, the net outcome can be positive. In this example we save Ether as long as $69217P_{low} < 17748P_{high} => \frac{P_{high}}{P_{low}} > 3.9$.

Conclusion: If the spread between P_{high} and P_{low} is sufficiently large, storing gas can be an efficient strategy to reduce transaction costs during network peak times.

This mechanism was developed, formalized and efficiently implemented in 2017 by the team at `https://gastoken.io`, Breidenbach et al. (2017).

2.1 The Gas Token

Breidenbach et al. (2017) wrapped the mechanism described above with an ERC20 contract, tokenizing gas refunds. They did this for both methods of gas storing: SSTORE (GST1) and self-destruct (GST2).

In their findings, the self-destruct variation outperforms the SSTORE variation as long as the volatility $\frac{P_{high}}{P_{low}}$ is above 3.71. Furthermore, the self-destruct token has significantly higher potential savings efficiency. For the remaining thesis, we will exclusively focus on the self-destruct variant of the gas token.

In the GST2 implementation, each token represents the rights to destroy and claim the refunds from 100 self-destructible *child contracts* similar to the one shown in algorithm 1, but with an added check so that only the GST2 *parent contract* can trigger the self-destruct.

The GST2 contract was developed in Solidity and the core mint and free functions are implemented via assembly: A low level programming language to interface directly with the EVM which can produce bytecode that is computationally more efficient than the output of the Solidity compiler.

In late May 2020, 1inch exchange's Anton Bukov introduced the CHI gas token in Bukov (2020*a*) which attempts to improve on the GST2 token.

2.2 Rise of Gas Tokens

The GST2 contract was deployed on the Ethereum main net on September 19th 2017 and the CHI contract was deployed on May 24th 2020.

Figure 1 shows the weekly minting of those tokens since November 2017[6], while figure 2 shows the combined amount of gas tokens minted in the same time frame.

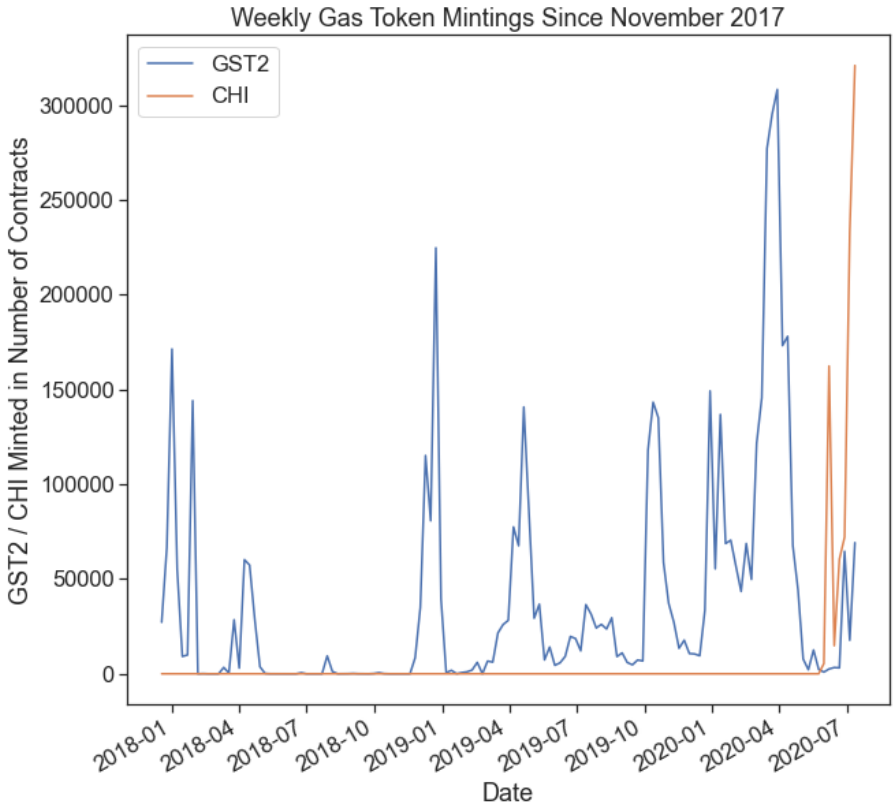

Figure 1: A total of 5'143'049 GST2 and 869'844 CHI contracts were minted in 256'589 resp. 11'236 transactions.

Since the start of 2020 we see an up-tick in gas token usage and especially CHI - the token popularized by and used on 1inch.exchange - has moved the gas tokens into the spotlight.

To find the amount of GST2 tokens freed, we would need access to a

[6]Showing data before November 2017 is unreliable, as we can't perfectly distinguish successful from failed transactions before the Byzantium hardfork.

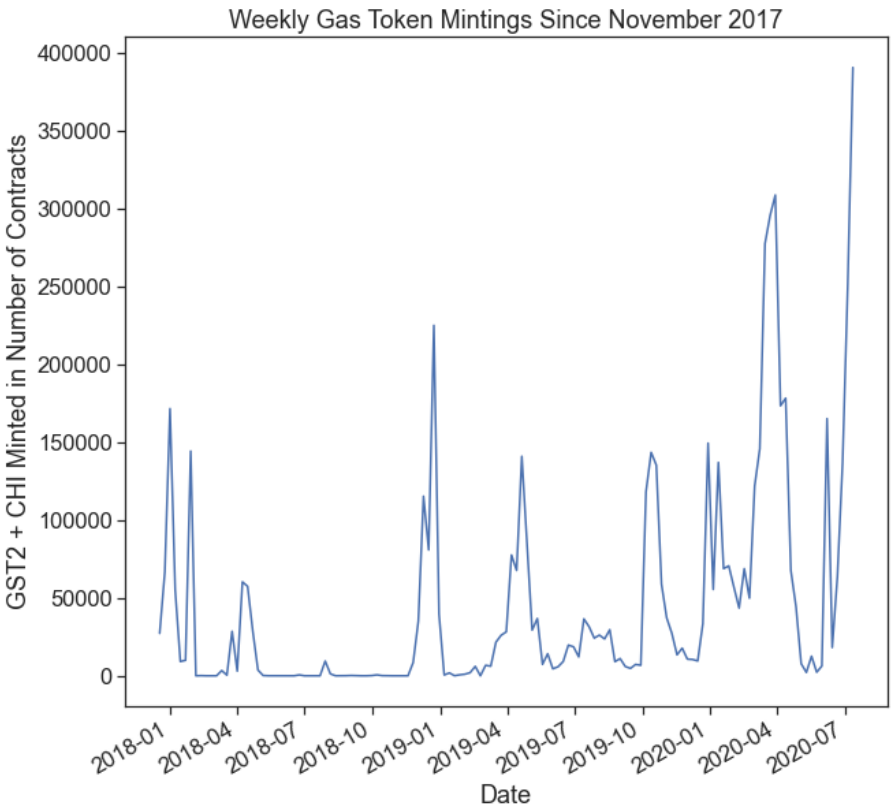

Figure 2: A total of 6'012'893 gas token contracts were minted in 267'825 transactions.

personally hosted[7] archival node to track the supply, since the *free* transactions are sub calls in other contracts and there is no way to reliably read this information from a full node. There are also no events emitted for minting or freeing GST2, presumably to save the associated gas cost.

There is however a strong assumption that the minting of gas tokens is linked to the volatility and the absolute level of gas prices. In section 4 we will explore this hypothesis.

[7]Infura is not an alternative due to the rate limit - we need millions of API calls - and the generally slow speed compared to a private node.

2.3 Impact of the Gas Token

This section briefly discusses some positive and negative aspects of gas tokens. No definite statement will be made about whether their overall impact is beneficial for the Ethereum ecosystem. More research in this area is required, especially if gas tokens turn out to become more popular and minting them takes up a significant portion of block space.

The presence of gas tokens should guarantee that every block will be filled, as it is always beneficial to mint them for a price of close to zero Ether. Furthermore, it is reasonable to assume that gas tokens lead to a soft cap of how low gas prices can drop as minting below this threshold is generally profitable (this effect becomes more significant as gas tokens become more liquid). The resulting supply of gas tokens can then be used to reduce gas prices during times of high network congestion. Both these effects work to reduce the volatility of gas prices. This effect is desirable for the Ethereum end user with everything else being equal[8].

By lowering the volatility it is not clear how the average gas price will be affected. However, it is very likely that while the volatility is reduced, the average gas price increases through the use of gas tokens, as gas tokens themselves are inherently inefficient (it takes much more gas to mint them than is refunded when they are freed).

As described in section 1.3.2, making use of gas refunds will reduce the gas consumed by a transaction, but not the computation required to process the transaction. For the same block gas limit, an increased use of gas tokens will put more computational strain on the Ethereum network, very similar to an increased block size during peak times. This is not a negative effect per se, and its impact is bounded since at most half the transaction can be refunded, but it needs to be considered when setting the block gas limit. Expressed in different terms: Heavy use of gas tokens will reduce the capacity of the Ethereum network.

If we assume the overall impact of gas tokens to be negative, does it still

[8]Further research is required to quantify the benefit of reduced transaction cost volatility.

make sense to promote and use gas tokens? We think yes, because it is rational to use them. As long as using gas tokens provides a benefit to the people minting and burning them, it is rational to use them even at the detriment of the protocol. It is our belief that a permission-less system like Ethereum must be able to tolerate and withstand any economically rational behaviour inside its ecosystem. Additionally, promoting the use of gas tokens and including them in popular smart contracts is desirable as it levels the playing field between experienced actors who already make use of them and the average user.

Finally, if the effect of the gas token is desirable, there is definitely a more efficient way to implement it on the protocol layer without incurring such massive computational overhad and external cost. We briefly discuss the future of gas tokens and proposed changes on the protocol layer in section 7.

3 Data Preparation

Working with blockchain data is very convenient, since all the data is publicly available for free. However, there are a few challenges to overcome before the data can be used for analyses as presented in this thesis.

The size of the Ethereum blockchain: Ethereum already has well over half a billion transactions recorded on it's public ledger. Working with data tables of this size requires a solid understanding of database- and software technology to process the data in a timely manner.

Ethereum is not optimized for speed: Most conventional databases have their focus on access speed. Ethereum however - to encourage decentralization by keeping the hardware requirements of nodes low - has its focus on minimizing the space required on harddisks. The architecture of a blockchain also means that the data is stored sequentially, each block building on its predecessor and as a further consequence of the architecture, Ethereum does not support indexing of transactions or blocks.

3.1 Extracting the Ethereum Data Into a Database

To make the data workable, we need to load it into a conventional database that supports indexing. Accessing the Ethereum data can be done either through an API provider like Infura (`https://infura.io`), or with direct access to an Ethereum node. To facilitate the analysis of gas prices, we need access to the gas limit and gas price of every single transaction. Most API providers - including Infura - have strict limits on API requests and even for paid plans it would be impossible (or very expensive) to run hundreds of millions of calls to extract every transaction.

The decision to run our own Ethereum node and database was made early on, knowing this will take weeks to fully set up. We wrote a more in-depth article on how to set up a personal node, sync it and configure it to work with data analysis and smart contract testing frameworks (Nadler (2020*b*)) and show that a personal node can be up to 20 times faster at processing transactions than using Infura.

Once set up, we used Ethereum-ETL[9] (Medvedev and the D5.ai Team (2017)) to extract the block- and transaction data to csv-files. From this raw transaction data we set up a MongoDB[10] database that stores the total of 768.7 million transactions (every Ethereum transaction since genesis and up to July 19th 2020). This database could then be indexed for the *block_timestamp* and the *to_address* to vastly improve accessing and filtering for these properties.

3.2 An Efficient Data Structure for Gas Price Data

To analyse the Ethereum fee market, we are interested in historical gas prices between the start of 2019 and July 19th 2020. To our knowledge, no publicly available dataset exists that provides historical gas price data

[9] ETL stands for extract, transform, load.

[10] MongoDB was chosen both for its simplicity (no schemas) and its aptitude to handle big data sets.

with a precision down to every transaction[11].

When analysing historical gas prices, we look at the gas price distribution within each block to for example answer the question what the "lowest gas price was, to be included in a certain block". There is no definite way to answer this question. To leave room for different interpretations we calculated the following gas price statistics for each block: Mean, median, 25th percentile, minimum distinct[12], minimum and standard deviation.

To further optimize the data structure and access speed (reducing the numbers of documents queried), we grouped the block data as described above into hourly bins. This leaves us with 13'560 bins each containing 139 to 309 blocks for a total of 3'391'381 blocks. Loading the data from this database into local memory now takes less than a minute. Note that the grouping into bins does not affect how the data is processed, just how it is retrieved; each block will still be handled independently.

3.3 Treating Gas Price Outliers

Ethereum gas prices are determined via a market mechanism. Therefore, even if the mechanism is not perfectly efficient, gas prices should reflect the current demand for block space and every gas price should be taken at face value. There is however one exception to this: Miners including their own transactions into blocks which they mine. This often takes the form of a block filled with zero-gas transactions. We were able to identify numerous blocks like these; some of them even minting GST2. These transactions would distort our data and to mitigate this we are ignoring any transaction with a gas price of less than 1 GWEI.

[11] If the reader is interested in the data set we built, please contact the author.
[12] The lowest gas price that is higher than the minimum gas price.

4 Analysis of Gas Prices and Volatility

In this section we try to answer two questions: First, does the minting of gas tokens correlate with the absolute level or the spread of gas prices? Second, is there enough weekly spread to make gas tokens profitable in the short term?

Figure 3 shows the 25th percentile gas prices (among gas prices for all transactions within a single block) since the beginning of 2019. We decided to use the 25th percentile price to give a more realistic picture because consistently finding the lowest distinct gas price for a transaction is too ambitious given the difficulty of predicting gas prices.

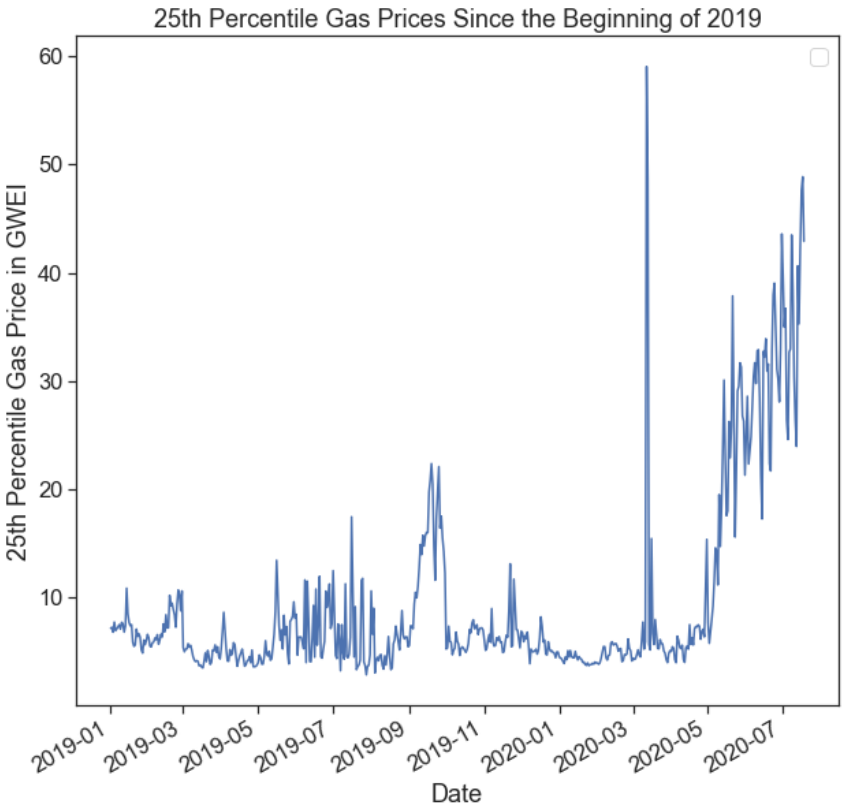

Figure 3: The distinct minimum gas prices since the beginning of 2019.

The large spike in March 2020 was caused by the "Black Thursday" event where Ethereum lost more than half its value over night, disrupting the whole DeFi ecosystem. We also see the meteoric rise in gas costs starting in May 2020. We can only speculate about the reasons for this; an increase in bot activity and/or general activity on the network seems a likely cause and there are currently no signs for this rise to slow down or revert.

By overlaying figure 3 with the volume of gas tokens minted as described in section 2.2 and figure 2 we get an intuition for how the two are related, shown in figure 4.

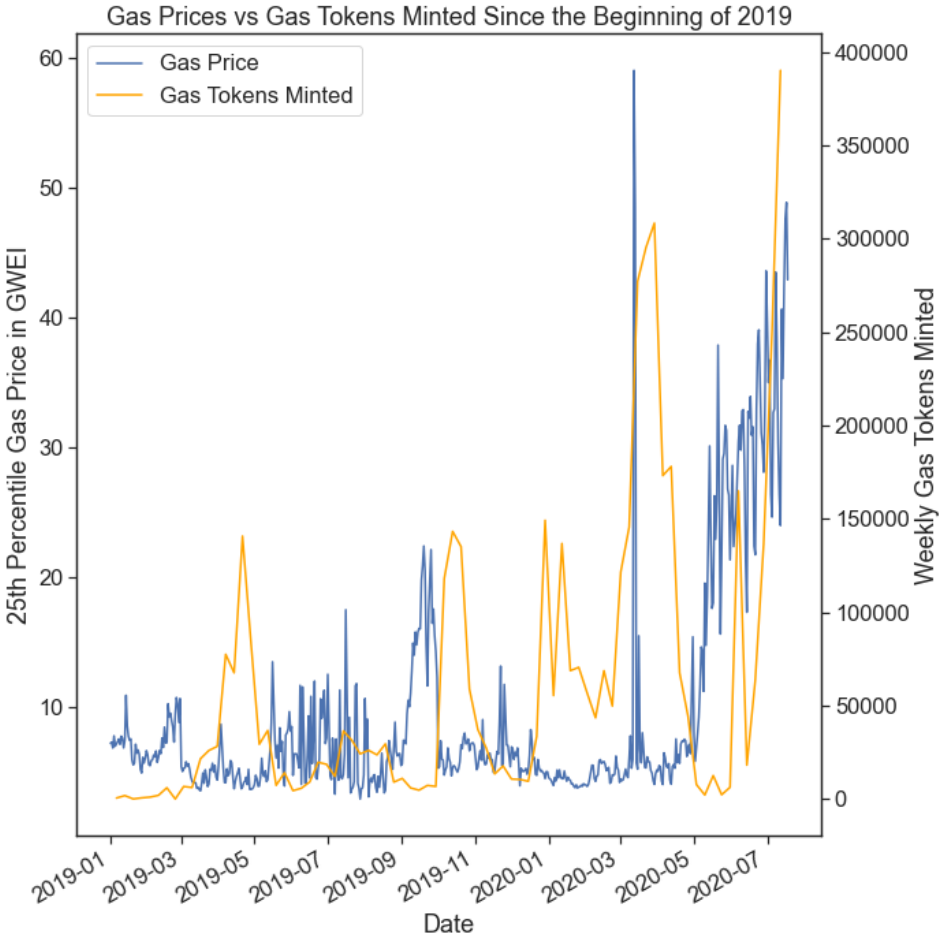

Figure 4: More minting occurs during periods of low gas prices and after spikes.

We see a very distinct increase in the minting of gas tokens during periods of low absolute gas prices, except during the last months. We also see spikes in minting following spikes in gas prices. This is most likely caused by actors who used up their gas tokens during a spike and are rebuilding their supply; and by actors realizing the usefulness of gas tokens after being subjected to a gas price shock and entering the market to mint gas tokens.

The rise in gas token minting since June 2020 can be explained by the introduction of CHI and its use on the popular platform 1inch.exchange, making a gas token available to the broader public for the first time.

Gas token profitability is related to the spread between gas prices over a period of time (see sections 2 and 5.6). Next, we look at the relation between this spread and the amount of tokens minted. The spread is defined as $\frac{P_{high}}{P_{low}}$ over a period of time W (the horizon of the actor). It is not inherently obvious which values should be used for P_{high}, P_{low} and W. We made the following cautious assumptions, which we think fall into the reasonable spectrum[13]:

W = One week (7 days)
P_{high} = The 90th percentile of block-mean gas prices during W
P_{low} = The 10th percentile of block-25th-percentile gas prices during W

This suggests that actors mint tokens during the cheapest 10% of blocks, using low gas prices (25th percentile of block gas prices), and burn tokens during the most expensive 10% of blocks, using higher gas prices (mean block gas prices). Using the higher mean block gas prices for P_{high} implies that the transactions where gas tokens are burned are more time sensitive.

Figure 5 shows the comparison of this spread to the minting of gas tokens with a reference line drawn at a spread of 2.8, which is approximately when gas tokens become profitable (see section 5.6, where this value is derived).

[13]Without access to the data for when gas tokens are burned it is especially difficult to estimate P_{high}.

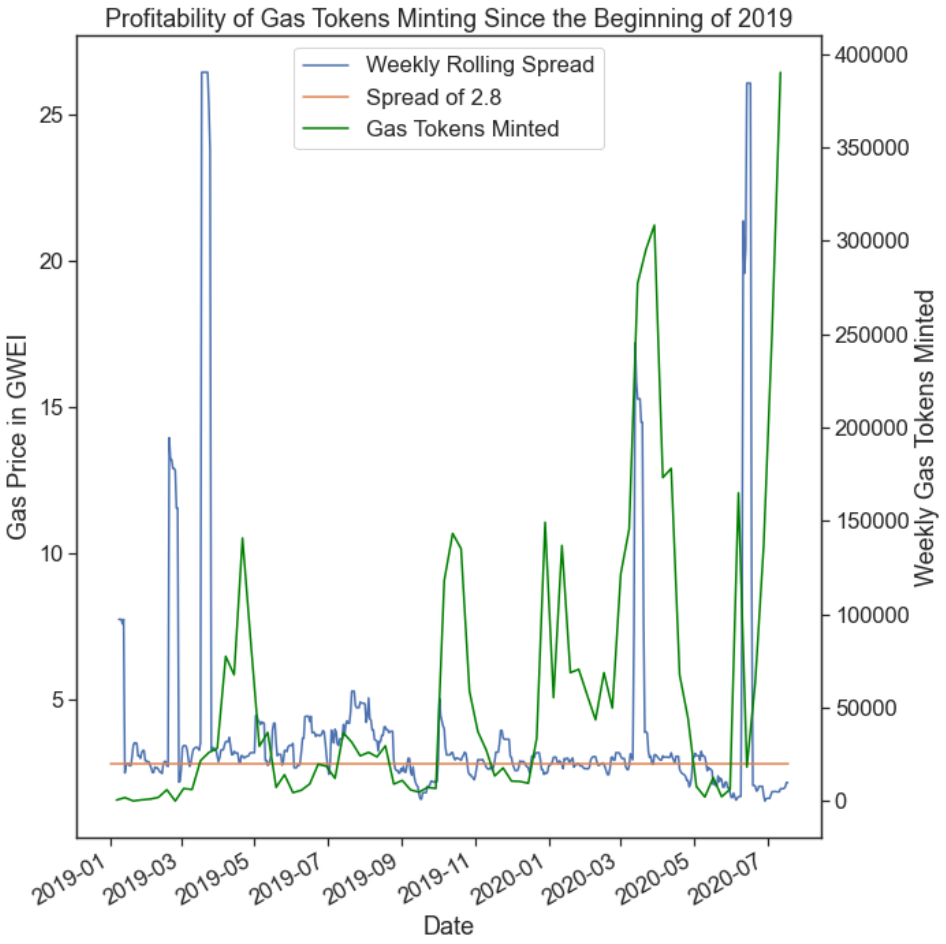

Figure 5: The data shows little correlation between spread and the volume of minted gas tokens.

The actors don't seem to base their decision to mint gas tokens on the observed spread of gas prices. This makes more sense if we consider that the spread can only be observed in retrospect; predicting the future spread is similarly difficult and subject to the same uncertainty as predicting future gas prices. Another factor (amplified by the illiquidity of gas tokens in the past) might be that the actors have a much longer time horizon W than seven days.

Conclusion: The presented data suggest that actors base their decision to mint gas tokens primarily on the absolute levels of the gas prices and react strongly to spikes in gas prices. The spread of gas prices has tight-

ened over the past year, making the use of gas tokens less profitable[14]. Also note that, under the assumed parameters, it is currently not profitable to mint gas tokens. The fact that gas tokens are still minted (and in higher quantities than ever), indicates that they are used for transactions which call for a gas price significantly above the assumed P_{high}.

5 The Liquid Gas Token (LGT)

Gas tokens must be gas efficient in their handling as reducing gas costs of transactions is their primary purpose. Since the inception of GST2, various improvements to the algorithm have been found. Additionally, the need to mint or buy GST2 on an exchange prior to using them is inconvenient and inefficient.

The core goal of this thesis was to present a token that improves the GST2 (and to a lesser extent CHI) in the aspects mentioned above. First, we combined the gas token contract with an integrated liquidity pool and an automated market maker to facilitate highly efficient internal swap transactions and to provide a way to use gas tokens without owning or buying them first. Second, all the proposed and discovered optimizations were tested and implemented only if they led to an overall improvement in the efficiency of the token.

The result is a streamlined gas token that can be directly incorporated into new smart contracts or EOAs (externally owned accounts). Finally, we benchmarked the tokens and show that the LGT outperforms the GST2 in every metric.

5.1 ERC20 With Integrated Liquidity Pool

In the world of DeFi and decentralized exchanges (DEXs), the Uniswap V1 exchange by Adams (2017) occupies a unique spot as the largest truly

[14] It is possible that the increased usage of gas tokens led to this effect.

decentralized exchange without oracles, owner fees or order books. The exchange relies on an automated market maker (AMM) that implements a constant product price model. All its properties make the protocol a perfect starting point when looking to integrate a market maker directly into an ERC20 contract.

5.1.1 Constant Product Market Maker Model

The constant product model as implemented by Uniswap V1 is formally specified in Zhang et al. (2018) and defines a market with two different coins: In our case the native ETH token, and the ERC20 compliant Liquid Gas Token (LGT).

This market has positive reserves of both tokens, $R_{lgt} > 0$ and $R_{eth} > 0$, a constant product $k = R_{lgt}R_{eth}$ and a trading fee $(1 - \gamma)$. According to Angeris et al. (2019), a transaction in this market, trading Δ_{eth} Ether for Δ_{lgt} LGT, must satisfy

$$(R_{lgt} - \Delta_{lgt})(R_{eth} + \gamma\Delta_{eth}) = k, \tag{4}$$

and vice versa. The trading fee γ is applied to every trade and is distributed to the liquidity providers according to their share in the liquidity pool as reward for providing the liquidity.

Providing liquidity to an AMM should not affect the market price of the listed commodities. Since the market price in the Uniswap constant product model is based on the reserves of both listed tokens, providing just one of the tokens as liquidity would shift the market price. When providing liquidity, it is therefore required to provide both tokens at the same time in a ratio corresponding to their reserves. Formally, when adding Δ_{lgt} LGT to R_{lgt}, the user also has to add $\frac{R_{eth}\Delta_{lgt}}{R_{lgt}}$ Ether to R_{eth}.

Angeris et al. (2019) also show that the Uniswap V1 model satisfies many desirable properties related to arbitrage, non-depletion, cost of manipulation, liquidity provider returns and stability under a wide range

of market conditions.

Finally, the Uniswap V1 protocol is unrivalled in its raw gas cost efficiency among similar DEXs, a property which is crucial for a gas token.

5.1.2 Advantages of an Integrated AMM

Ownership changes of ERC20 tokens are expensive. Every operation to change the balance of an ERC20 token for an address or to change the total supply costs at least 5'000 gas[15].

When a token is listed against Ether on Uniswap V1, the exchange contract is the owner of these tokens. Table 1 shows the ownership changes needed to mint or buy GST2 and then burn (free) them.

Operation	Mint and Burn	Buy and Burn
Reduce Balance of Exchange	0	5'000
Increase Balance of Transactor	5'000	5'000
Increase Total Supply	5'000	0
Reduce Balance of Transactor	5'000	5'000
Reduce Total Supply	5'000	5'000
Total SSTORE Cost	20'000	20'000

Table 1: ERC20 + Uniswap - The minimum gas costs incurred by supply and ownership changes when acquiring and burning GST2.

Note that *Mint and Burn* always needs to be split into two transactions as minting and burning a gas token in the same transaction is never profitable. Buying and burning on the other hand is usually performed in the same transaction. This adds at least another 21'000 gas as intrinsic cost to *Mint and Burn*.

When the ERC20 token and the exchange logic (including the AMM) are within the same contract - referred to as a *liquid token* - we don't need to explicitly track the balance of the exchange. In addition to the ERC20 compliant *balances* mapping $A \rightarrow B$ and *totalSupply* S_t, we track the privately owned supply S_o which is the sum of all balances except for

[15] One SSTORE operation is required to change the contract storage variable.

the balance of the exchange. This lets us implicitly define the balance of the exchange B_e as

$$B_e \equiv S_t - S_o. \tag{5}$$

Minting and burning an owned token is now more expensive since we also need to update S_o, but it makes buying and burning a token a lot less expensive as detailed in table 2.

Operation	Mint and Burn	Buy and Burn
Reduce Balance of Exchange	0	0
Increase Balance of Transactor	5'000	0
Increase Owned Supply	5'000	0
Increase Total Supply	5'000	0
Reduce Balance of Transactor	5'000	0
Reduce Owned Supply	5'000	0
Reduce Total Supply	5'000	5'000
Total SSTORE Cost	30'000	5'000

Table 2: Liquid ERC20 - The minimum gas costs incurred by supply and ownership changes when acquiring and burning a liquid token.

Burning (freeing) a token will always reduce S_t to reflect the new token supply. For the integrated case however, reducing S_t will also reduce B_e by the same amount as seen in (5) without the need for an additional SSTORE operation.

The same mechanics apply for other combined transactions such as minting and adding tokens to the liquidity pool or minting and selling tokens to the exchange: With an integrated model, the minted token is created by increasing S_t, which implicitly increases B_e and triggers an Ether transfer to the owner of the *Mint and Sell* transaction. A single storage variable needs to be changed to perform a *Mint and Sell* action.

Conclusion: For tokens where buying, selling, investing[16] or divesting the token is closely tied to minting or burning the token, integrating the

[16]Investing in this context means providing liquidity: Adding the token and Ether to the liquidity pool of the exchange.

exchange logic into the ERC20 contract leads to significantly less contract storage updates and to a much more efficient workflow.

5.1.3 LGT Specific AMM Implementation

Integrating an AMM into an ERC20 contract also allows us to better tailor the AMM to our needs. The Uniswap V1 protocol is audited by multiple sources and very unlikely to contain serious vulnerabilities or bugs. We therefore tried to keep the code of the core functionality as close as possible to the Uniswap V1 implementation [17]

What follows is a summary of changes and digressions from the Uniswap V1 implementation for the LGT implementation with our reasoning behind them.

No token to token swaps: Uniswap V1 exchanges can be either ERC20 to Ether or ERC20 to ERC20. For our application, we only need to consider the ERC20 to Ether case. It is both more efficient[18] and sufficient in its functionality. All the logic to handle token to token transfers was omitted.

No need to check for positive reserves: A new Uniswap exchange is created without liquidity via the Uniswap factory contract. As seen in section 5.1.1, the reserves for both the token and Ether need to be strictly positive. Adding liquidity to an empty reserve also needs to be handled differently. The LGT contract will mint one LGT in its constructor with the Ether that was pre-funded or sent with the contract creation and add them as the initial liquidity reserves. This initial liquidity is owned by a smart contract that has no ability to remove the liquidity; it is therefore guaranteed that the reserves of the LGT contract are always positive which allows us to remove checks for it and streamline the code. These change have no impact on the interface of the contract.

[17] Note that Uniswap V1 was written in Vyper while LGT needs to be written in Solidity to make use of assembly. Fortunately, converting code from Vyper to Solidity is fairly straightforward.

[18] This is also the most important reason why we chose Uniswap V1 over Uniswap V2: The latter only supports ERC20 to ERC20 swaps.

Implicit LGT balance: As described in section 5.1.2, the balance of the LGT exchange B_e is implicitly defined. All functions which transfer owned tokens to or from the exchange update S_o instead of B_e, where S_o is the total supply of externally owned tokens. This has no effect on the interface or the gas costs.

LGT is well behaved: When building a general purpose exchange like Uniswap V1, one needs to consider that the listed tokens don't strictly conform to the expected specifications or even intentionally try to exploit or harm the exchange. For a built-in exchange, we can be sure the token behaves to the specifications we gave it - not accounting for unintended bugs. This allows us to be more lenient with checks - for example we know the LGT supply can never exceed a 256bit unsigned integer - and it allows us to more directly interface with the functionality of the token, improving efficiency.

Liquidity shares are not a token: The Uniswap V1 exchange contract will mint ERC20 compliant liquidity tokens (UNI) that can be traded back for the respective shares of the reserves. LGT already has an ERC20 token interface (LGT itself); the liquidity shares can thus not be ERC20 compatible. Instead of opting for a more complex multi-token contract, LGT offers a reduced pseudo-ERC20 interface for the liquidity shares. They are tracked almost identically to the UNI tokens and offer most of the same functionality like transferring liquidity. However, due to different function- and variable names, they are not ERC20 compatible.

Reduced event coverage and less code re-use: LGT will only emit events for adding, removing and transferring liquidity. All swaps and transfers do not have an event associated with them. The sole reason for this is to save gas in the functions where it is most critical. We made it a priority to keep the minting and freeing of tokens as efficient possible. For the same reason we do not factor out code for these functions, but prefer to re-write the lines to minimize the occurrence and cost of internal function calls.

5.1.4 LGT Exchange Interface

Like Uniswap V1, LGT supports input and output swaps and transfers from LGT to Ether and vice versa, it has the same input and output price query functions and the same add- and remove liquidity functions. The ERC20-like liquidity pool functions have a "pool" prefix added to them. Events are emitted for adding, removing and transferring liquidity.

We made the deliberate decision to use the same function names as the Uniswap V1 protocol when the same behaviour can be expected. This also makes LGT partially compatible with an Uniswap V1 interface or frontend.

The full LGT exchange interface can be seen on the LGT github repository (Nadler (2020*a*)) at `/interfaces/ILiquidERC20.sol`.

5.2 Gas Token Implementation

In the following sections we provide insights into the gas token part of the LGT smart contract, highlight differences from its predecessors GST2 and CHI and provide an overview of the LGT interface related to minting and freeing tokens. The full contract source code with extensive documentation, natspec and accompanying scripts can be found on the LGT github repository (Nadler (2020*a*)).

5.2.1 Creating and Destroying Child Contracts

As discussed in section 2, the core functionality of a GST2 based gas token is to create child contracts that are destructible only by the parent

contract. Such a contract, written in Solidity, is shown in algorithm 3:

Algorithm 3: A child contract for LGT, written in solidity.

```
pragma solidity 0.6.9;
contract Storage {
    fallback() external {
        if (msg.sender == LGT.address) {
            selfdestruct(msg.sender);
        }
    }
}
```

If we break this down to EVM instructions, we get:

```
PUSHX <LGT.ADDRESS> // where X is the length
                    // of the address in bytes
CALLER
XOR
PC
JUMPI
CALLER
SELFDESTRUCT
```

For a normal address, the bytecode for this contract is 27 bytes long. The bytecode to initialize the contract adds another 9 bytes for a total of 36 bytes. There are two interesting properties for this init code:

First, the cost to create this contract scales with the length of the init code: Each byte costs an additional 200 gas. Second, if we can get the init code to 32 bytes or below, it will fit into a single memory slot and further reduce the deployment complexity and cost.

Breidenbach et al. (2017) were quick to notice that there is a way to reduce the length of the init code by using an address for the contract with leading zeros, as these can be omitted in the byte code. An address has a default length of 20 bytes; if we can get at least the first four bytes (8 positions) to all zeroes, we can fit the init code into 32 bytes. At

the time when GST2 was developed, the only way to deploy a contract was the CREATE opcode, which uses the address of the deployer and the nonce of the deployment transaction (this combination is unique) to deterministically calculate the address where the contract is deployed. The team used a brute-force algorithm to find an address and a nonce that will deploy the GST2 contract at an address with five leading zero bytes. The chance to find an address with n leading zero bytes when testing a combination of deployment address and nonce is $\frac{1}{256^n}$. For $n = 5$ this is roughly 1 in a trillion. Our own implementation in Java of such an algorithm took on average three days to find an address with five zero bytes. Finding an address with six zero bytes would take 256 times longer, but save another 200 gas per contract created.

Once the child contract is created, we need a way to retrieve its address to call selfdestruct when we want to claim the gas refund. Storing the addresses in contract storage is not a good option, as this would cost at least 10'000 gas per contract (20'000 to store it and a refund of 10'000 when we later delete the storage slot). Breidenbach et al. (2017) solved this elegantly by calculating the addresses of the child contracts at runtime, using the address of the GST2 contract and the nonce of the contract creation. The nonce can be stored at no additional cost by splitting the **totalSupply** of the GST2 token into **tokensBurned** and **tokensMinted** where

$$totalSupply \equiv tokensMinted - tokensBurned.$$

Whenever a contract is created or burned, the respective variable is incremented by one. This does not increase the gas cost as we still only update one variable to change the **totalSupply**. Since creating these contracts is the only time the nonce of the GST2 contract is incremented, **tokensBurned** is equal to the nonce of the oldest created contract that is not yet self-destructed. This allows GST2 to find the address by calculating it from its own address and **tokensBurned**.

In April 2018, EIP-1014 (Buterin (2018)) introduced CREATE2: A new

way to create contracts by using the address of the deployer, a salt[19] and the bytecode of the contract. Importantly, the method to calculate the address where the new contract is deployed (hashing the input parameters) is significantly cheaper than the RLP method used by CREATE. Switching to CREATE2 for child contract deployment makes burning tokens cheaper and also removes the need for the contract nonce to be accurate since we can use **tokensBurned** as the salt in CREATE2. This was first shown by Bukov (2020*a*) in their implementation of the CHI token.

Using CREATE2 to find short contract addresses: Calculating the address where a contract is deployed with CREATE2 instead of CREATE is not only cheaper, but also faster. We no longer have to generate public/private key-pairs and check the first nonces, but we can simply alternate the salt. With CREATE2 it should be possible to significantly reduce the computational effort to deploy the LGT contract at a 15 byte or even 14 byte address. Johguse (2020) published highly optimized tools which leverage the hashing power of GPUs to find deployment addresses for CREATE and CREATE2. In testing both these tools, we found that brute-forcing short addresses is almost an order of magnitude faster using CREATE2.

Remember that CREATE2 requires the complete contract bytecode as a parameter. Since the LGT contract has its own address hard coded into the child contract init code, this poses a recursive problem where updating this address will change the bytecode, invalidating the CREATE2 deployment address. In order to solve this and use CREATE2, we have to rewrite the LGT contract to be agnostic of its own address. The init code for a child contract with a 14 byte address in LGT is: 0x746d<LGT.ADDRESS>3318585733ff6000526015600bf30000. To remove the dependency from a hard coded address, we construct the

[19] A salt is random data; in this case with a length of 32 bytes.

init code in the EVM memory using assembly, as shown in algorithm 4:

Algorithm 4: Constructing the address agnostic init code for child contracts.

```
mstore(0,
    add(
        add(
            0x746d000...54x0...000, // pad to 32 byte length
            shl(0x80, address())
        ),
        0x3318585733ff6000526015600bf30000
    )
)
```

The same principle is used when retrieving the addresses of child contracts with a dynamic parent contract address. The details of the implementation can be seen in the function *computeAddress2(salt)* of the LGT smart contract.

Once the bytecode for the LGT contract was address agnostic and finalized, we started mining for a deployment address with 6 leading zero bytes using three Nvidia RTX2080 GPUs[20] with a combined hashing power of ∼3 gigahashes per second. The estimated time to compute the address can be calculated as:

$$\frac{256^n}{3 * 10^9/s} = 93825s,$$

or roughly 26 hours. On June 25th, the LGT smart contract was successfully deployed at: 0x000000000000c1cb11d5c062901f32d06248ce48. We checked every transaction for the past 6 months to find transactions from or to 14 bytes or shorter addresses. Of the 85 addresses found[21], most of them were burn addresses and only a handful had a contract deployed

[20] Thankfully the author's flat mates are gamers and offered their GPUs for this project. No external hardware was required.

[21] Fun fact: The shortest owned address we found that sent at least one transaction had 13 leading zeroes (6.5 bytes): 0x0000000000000d9054f605ca65a2647c2b521422

(for example CHI and blue.dex). All of them deployed the contract via CREATE. Bukov (2020*b*) states in a video talking about the CHI token, that it was "quite expensive" to mine the address for their contract. As far as we know, LGT is the first smart contract deployed at a 14 bytes or shorter address using CREATE2.

Further optimizations: Bukov (2020*a*) saved gas by unrolling the for-loop to mint multiple tokens in one transaction for their CHI gas token implementation using assembly. We were able to further improve this unrollment by using **tokensMinted** directly as the control variable and with a similar approach we improved the loop to destroy child contracts by using **tokensBurned** as the control variable.

5.2.2 Minting and Freeing LGT

GST2 and CHI offer the same minting and freeing functions:

```
mint(amount)     // Mint amount tokens
free(amount)     // Free amount tokens and claim the refund
freeFrom(amount) // Free from a different, approved wallet
freeUpTo(amount) // Free all tokens if amount > balance
freeFromUpTo(amount)
```

Of these functions, LGT implements only the first three since the focus is not on minting or freeing owned tokens. Using the integrated liquid exchange, LGT offers the following additional minting and freeing functions. Where present, the deadline parameter prevents the execution of a transaction if it is past a certain date and time. This helps to mitigate frontrunning as described in the Uniswap V1 whitepaper by Adams (2017).

mintFor(amount, recipient): A variation on **mint()** that creates the tokens for a different recipient address.

mintToLiquidity(maxTokens, minLiquidity, deadline, recipient) payable: This function mints up to *maxTokens* LGT and adds both the

tokens and the sent Ether to the liquidity reserves. As described in 5.1.1, adding Ether and LGT to the liquidity pool needs to be in the same ratio as the current ratio of the reserves as to not influence the market price. These parameters might have changed since the transaction was submitted; to ensure the best outcome for the user, the amount of tokens to mint and the amount of Ether to invest is calculated and any excess Ether (if more than *maxTokens* would need to be created) is refunded or the amount of tokens minted (if more than the sent Ether would be required) is reduced. This function is a lot more efficient than minting and later investing the tokens because the tokens can be created in an unowned state and no balances need to be updated. By simply increasing the **totalSupply** we add the tokens to the balance of the exchange (see 5.1.2). The *recipient* is the address that receives the liquidity shares. This is useful to have multiple pending mint transactions from different accounts[22] to build liquidity on a single account.

mintToSellTo(amount, minEth, deadline, recipient): This function enables very easy and almost risk-free gas arbitrage (see section 5.6 for details). *Amount* LGTs are minted and immediately sold to the exchange, transferring the received Ether to the *recipient*. The integrated exchange allows us to take huge shortcuts; the core transaction boils down to just three statements as seen in algorithm 5:

Algorithm 5: Minting and immediately selling LGTs.

```
ethBought = getInputPrice(amount, tokenReserve, ethReserve);
_createContracts(amount, totalMinted);
recipient.call{value: ethBought}("");
```

The tokens are created in an unowned state by increasing the **totalSupply**, implicitly adding them to the balance of the exchange. Not a single additional SSTORE operation is required. To further illustrate just how efficient this function is, lets look at the detailed gas breakdown of a call to mint and sell 3 tokens:

[22] It is a bad idea to have multiple transactions with similar gas prices pending from the same account, as there is no way to guarantee the order in which they will be included; but the nonce requires a certain order to be maintained.

```
Gas used: 147'043 gas
Initial call cost  [21'912 gas]
LiquidGasToken.mintToSellTo [10'468 gas]
--LiquidERC20.getInputPrice [551 gas]
--LiquidGasToken._createContracts [5'301 gas]
----<UnknownContract>.<CREATE2> [36'224 gas]
----<UnknownContract>.<CREATE2> [36'224 gas]
----<UnknownContract>.<CREATE2> [36'224 gas]
```

The intrinsic cost, creating the child contracts and updating the **totalSupply**, accounts for 135'584 gas. According to the Yellow Paper (Gavin (2014), Appendix G), a call with non-zero value transfer (sending the Ether received from the sale) costs 9'700 gas. This leaves just 1'759 gas as the total overhead for the function. This is in contrast to an overhead in excess of 50'000 gas which it would take to mint and sell three GST2 to Uniswap.

The *minEth* parameter can be set to make this arbitrage opportunity almost risk-free: Calculate the total expected cost of the transaction given the desired gas price and require that the payoff is greater than this amount. The transaction is then only executed if it is profitable, and minimal gas losses are incurred otherwise.

mintToSell(amount, minEth, deadline): Same as above, but the Ether is sent to the transaction owner instead of the *recipient*.

buyAndFree(amount, deadline, refundTo) payable: This function can be called from within another transaction and will buy and free up to *amount* LGTs. If not enough Ether is sent, the full amount of Ether will be returned and nothing happens; if too much Ether is sent, all excess Ether is refunded. To allow more lenient inclusion into other smart contracts, this function does not revert under normal circumstances but instead sends back the Ether and returns 0 (indicating failure) on any failed check. Similar to **mintToSell**, this function takes full advantage of the built-in exchange. Burning unowned tokens implicitly reduces the LGT balance of the exchange (the token reserves) and sending Ether

to the contract increases the Ether reserves. No other state changing operations are required to process the swap. We forego a detailed analysis of the gas cost, but refer to the benchmark section (5.3) where we see that the overhead was reduced by more than 60'000 gas as compared to buying GST2 on Uniswap and freeing them.

buyMaxAndFree(deadline) payable: This function will buy as many LGTs as possible using the Ether sent with the transaction and immediately free them. No refunds take place for partial tokens, but the transaction is reverted on any error. Apart from these differences, it functions similar to **buyAndFree**. This function is recommended when the precise amount of Ether to send is calculated on-chain prior to calling the LGT contract.

deploy(tokenAmount, deadline, bytecode) payable: Contract deployments are often associated with very high gas requirements. LGT offers two functions to deploy a contract while simultaneously buying and freeing **tokenAmount** LGTs to reduce the deployment cost. This first deployment function uses CREATE to deploy the contract. Any excess Ether sent with the transaction not spent on buying tokens is refunded, but sending insufficient Ether will revert the transaction. In a similar fashion as the functions above, using this function is more efficient than including a call to **free()** in the constructor of the contract. Note that you can't send Ether with the deployment and any use of *msg.sender* in the constructor will refer to the LGT contract, not the initial sender address.

create2(tokenAmount, deadline, salt, bytecode) payable: This function works identical to **deploy()** above, but uses CREATE2 to deploy the contract instead. By setting the salt it is possible to deploy the contract at a previously known address. This is useful for pre-funding the contract or deploying at an address with specific properties.

Highly optimized functions: For advanced users, we provide more optimized (but also more dangerous) versions of the three most crucial functions: **mintToSell**, **mintToSellTo** and **buyAndFree**. These func-

tions save gas in three ways: First, most checks have been removed and lay the burden of submitting valid parameters on the caller. Second, some parameters have been omitted or replaced by defaults to save gas. Third, we constructed the function names in a specific way to save more gas, as detailed below.

Every byte of a transaction's calldata costs 68 gas per non-zero byte and 4 gas per zero byte. The first four bytes of every transaction is called the *function selector* (FS) and specifies which function in the contract is called. These four bytes are obtained by hashing the signature (unique combination of name and input types) of the function. By brute-forcing we can find a function name that will generate a function selector with three or even all four zero bytes, saving 192 or 256 gas per function call respectively. If we go back to our above example for **mintToSellTo(amount, minEth, deadline, recipient)**, calling **mintToSellTo25630722(amount, recipient)**[23] with the same amount brings the total gas cost down from 147'043 gas to 146'503 gas, saving 540 gas. Small gas savings add up, and we are committed to provide the most efficient implementation of a gas token so far. The optimized functions are as follows:

```
mintToSell9630191(amount)                // FS: 0x00000079
mintToSellTo25630722(amount, recipient) // FS: 0x00000056
buyAndFree22457070633(amount)            // FS: 0x00000000
```

5.3 LGT Benchmarks

In this section we will compare the efficiency - measured in gas cost - of the three gas tokens discussed in this thesis: GST2, CHI and LGT. We are comparing the total cost for the following operations:

Mint Minting 25 owned gas tokens.

Free Using 1'000'000 gas and freeing 25 owned gas tokens.

[23]Note that we can no longer specify a deadline or minEth.

Mint and sell Minting 25 gas tokens and selling them to an exchange.[24]

Buy and free Using 1'000'000 gas, buying 25 tokens from an exchange and freeing them.

The metric of comparison will be the total gas used to perform the respective operation. The benchmarks are measured on a forked Ethereum mainnet run in Ganache, meaning we use the deployed live versions of all the contracts (including Uniswap). Care is taken that all contract state variables are initialized[25] and that the exchanges are sufficiently funded. The script to perform the benchmark - which can be run locally - is located at `scripts\benchmarks\gas_token_comparison.py` in the LGT github repository (Nadler (2020*a*)). Table 3 compares the absolute gas costs while table 4 shows the deviations from the best performing gas token.

Gas Token	Mint	Free	Mint and sell	Buy and free
GST2	947'009	609'156	1'012'805	661'427
CHI	* 942'679	599'767	1'008'764	651'093
LGT	946'314	* 598'078	* 944'483	* 592'697

Table 3: Comparing GST2, CHI and LGT for their absolute gas efficiency. The best performing token in each category is marked with an asterisk.

Gas Token	Mint	Free	Mint and sell	Buy and free
GST2	4'330	11'078	68'322	68'730
CHI	0	1'689	64'281	58'396
LGT	3'635	0	0	0

Table 4: Comparing GST2, CHI and LGT for their gas efficiency. Values are deviations from the best performing token.

As a consequence of the many optimizations detailed in section 5.2, the precursor token GST2 performs worse than CHI and LGT in every category. This is amplified by the fact that the addresses for CHI and LGT

[24]For GST2 and CHI, we sell the tokens to Uniswap V1 in a separate transaction; for LGT we sell them to the integrated exchange.

[25]Changing an uninitialized variable would incur an additional cost of 15'000 gas.

are both one byte shorter than GST2's. It would be expected that LGT performs worse than CHI when dealing with owned tokens since an additional SSTORE operation is used; however, thanks to further improvements we discovered (especially for freeing tokens), LGT is competitive with CHI even when looking at owned tokens.

Conclusion: For transactions that combine minting or freeing with a swap (or investment), LGT vastly outperforms both alternatives. Optimizing this functionality was the primary goal of the LGT token.

5.4 LGT Exchange Application

We provide a decentralized application (dApp) to interact with the LGT smart contract at `https://lgt.exchange`. It integrates with a metamask wallet and currently supports minting tokens for personal use, to sell and to invest as well as deployment of contracts via CREATE.

Additionally it displays detailed statistics for the reserves and prices of the exchange including personal shares and has an integrated calculator to find out if using LGT is profitable given the gas used and the gas price of a transaction.

The dApp works on the Ethereum mainnet as well as on the Kovan and Ropsten testnets.

Further functionality such as swaps, transfers and deployment via CREATE2 will be added in the future.

5.5 Usage of the LGT

The Liquid Gas Token can be used in four distinct ways:

As a transactor: Deploying contracts via the LGT interface can reduce the deployment cost. Using a Relayer[26] smart contract can reduce the

[26]See `https://github.com/matnad/liquid-gas-token/blob/master/contracts/LGTRelayer.sol` for an example.

cost of transactions that don't rely on `msg.sender`. Finally, it is possible to set up an EOA with the ability to reduce the cost of any transaction it performs.

As an arbitrage actor: When the gas price is sufficiently low, any account can mint and sell LGT for an immediate profit. This functionality was designed at least partially with arbitrage bots in mind: It can be automated very easily and multiple parameters can be set to eliminate all but a very small risk. Also see the next section for more details.

As a liquidity provider: By minting directly to the liquidity pool (or buying and adding the tokens), an investor gets a proportional share of the 0.5% transaction fee that is applied to every LGT transaction. The return on this investment depends on the share of the total liquidity pool and the number of swap and transfer transactions on the LGT smart contract.

As a developer: LGT can be directly integrated into a new smart contract to let users save gas on transactions. A detailed write-up can be found at `https://lgt.exchange/integration`.

5.6 Gas Price Arbitrage

As shown in section 2, the LGT can be efficient if the spread between high gas prices P_{high} and low gas prices P_{low} is sufficiently large. The exact spread $\frac{P_{high}}{P_{low}}$ depends on too many factors to formalize and fluctuates slightly as contract storage variables take different lengths. We can formulate an approximate lower bound based on empirical values:

$$\begin{aligned} cost_{mint}(n) &\leq 39'414 + 36'224n \\ cost_{free}(n) &\leq 18'736 + 5'722n \\ refund(n) &\equiv 24'000n, \end{aligned}$$

where n is the number of contracts minted or freed. A maximum of 330

contracts can be minted at once as to not exceed the current 12 million gas block limit:

$$spread_{min} = \frac{cost_{mint}(330)}{profit_{free}(330)} = \frac{11'993'334}{6'013'004} = 1.9946.$$

For most cases however, less than 330 tokens will be minted or freed at the same time and the required spread to be profitable is around a range of 2.5 to 2.8.

Thanks to LGT's integrated AMM users doesn't need to perform complicated calculations. The market value of an LGT token in relation to Ether is given by the exchange. When attempting to make arbitrage profits during times when the gas price is low, the arbitrageur can call the LGT price query function with the amount of tokens a they wish to mint and sell:
`LGT.getTokenToEthInputPrice(a)`. This returns the guaranteed profit should the `LGT.mintToSell(a,...)` call succeed. A contract to make use of gas arbitrage for LGT could look like described in pseudo-code algorithm 6:

Algorithm 6: Pseudocode for a simple on-chain arbitrage "bot".

```
function lgtArbitrage(amount) {
    profit = LGT.getTokenToEthInputPrice(amount)
    gasCost = (39141 + 36224 * amount + overhead) *
     tx.gasprice
    if(profit > gasCost) {
        LGT.mintToSell(amount, gasCost, deadline)
    }
}
```

Where overhead refers to any gas used other than for minting the contracts (such as calling the arbitrage function itself).

Alternatively, the arbitrageur can use the provided dApp (section 5.4) which will query and calculate the expected profit, the cost of the trans-

action[27] and the expected profits automatically as shown in figure 6.

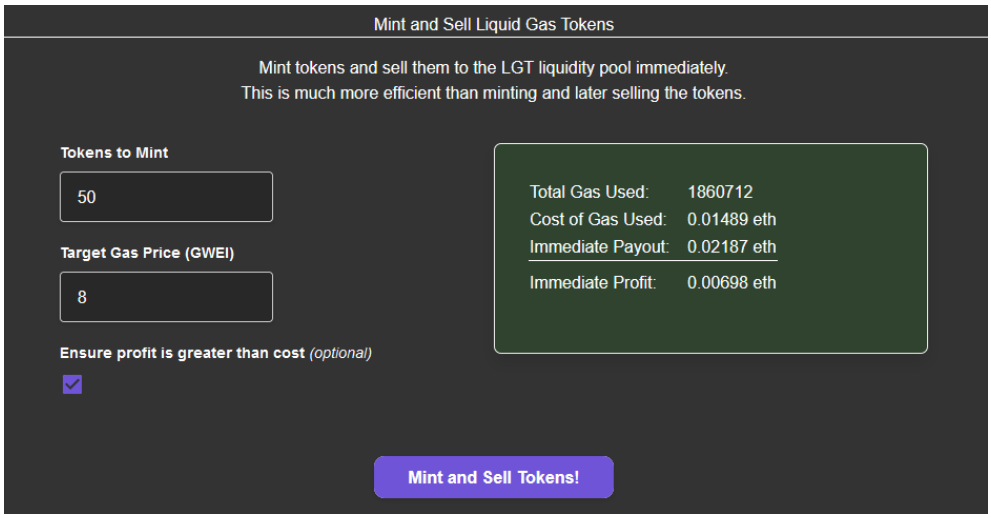

Figure 6: The LGT arbitrage interface on the lgt.exchange dApp. Clicking the button will broadcast the transaction from the linked account.

As long as a minimum required profit ($minEth$) is passed to the `mintToSell` function, the only risk the arbitrageur takes is that the transaction will fail due to changed market prices (too many other arbitrage transactions were included before their transaction). This loss associated with the failed transaction usually does not exceed 26'000 gas.

Conclusion: If during a certain time frame the gas price spread is $\frac{P_{high}}{P_{low}} \gtrapprox 2.8$, we expect arbitrageurs to mint and sell LGT when the gas price is $\leq P_{low}$ and transactors to buy and free LGT when the gas price is $\geq P_{high}$. If the gas price spread is $\frac{P_{high}}{P_{low}} \lessapprox 2$ over an extended period of time, we expect no usage of the LGT contract.

5.7 Development and Testing

The LGT smart contract was developed in the Brownie framework (Hauser (2019)) using PyCharm and Webstorm as IDEs of choice for python and react (frontend) development respectively.

[27] Gas estimation is performed via the web3 function estimateGas.

The LGT smart contract uses openZeppelin's safeMath library and is split into three modules:

ERC20PointerSupply: An ERC20 contract based on openZeppelin's ERC20 template that splits **totalSupply** into **totalMinted** and **totalBurned** (see section 5.2.1) while maintaining full ERC20 compatibility.

LiquidERC20: Extends the ERC20PointerSupply contract to add the integrated market maker as described in section 5.1.

LiquidGasToken: Extends the LiquidERC20 contract to add and integrate the gas token functionality as described in section 5.2.

All three contracts are extensively tested[28] with a reported test coverage of 100%. The tests are split into unit- and integration tests and can be run locally after cloning the repo (Nadler (2020*a*))and installing Brownie.

No independent audit was performed for the LGT smart contract to date.

6 Open Source Contributions

Brownie is a relatively new smart contract framework that is still under heavy development. While working on the LGT smart contract we came across features that would greatly add to our developer experience like a more detailed gas cost analysis. After contacting the Brownie developer, we decided to take part in the development of the framework and started submitting fixes and new features for Brownie.

Over the following months, we made around 15 major contributions to advance the Brownie framework such as:

Detailed gas reports: Breaking down the transaction trace to show detailed gas usage of internal and external function calls within a transaction. This was immensely helpful to optimize the gas costs for the LGT smart contract.

[28] We use state of the art testing with a modified pytest framework. See the Brownie documentation for more details.

Expose more ganache parameters: Especially the unlock feature to take over any arbitrary address on a local forked mainnet was helpful to borrow CHI's 14 byte address before we mined our own.

Integration with react frontends: Outputting deployment artifacts than can be used by a react based frontend to integrate smart contract and dApp development.

Furthermore, while working on the detailed gas analysis, we discovered a major bug in ganache that shifted the whole debug trace by one step and led to inaccurate gas values being reported. This issue was brought to the attention of the ganache developers and has since been fixed.

7 The Future of Gas Tokens

Section 2.3 discusses the impact of gas tokens on the Ethereum ecosystem and concludes, that even if the effect would be desirable, a more efficient implementation must exist at the protocol layer. We are not aware of an EIP that formalizes how this could be implemented, but one naive idea would be to track stored gas in a variable which doesn't require any additional computation and let transactions use this stored gas.

There are however multiple EIPs pending which, when implemented, would change how gas tokens function such as EIP-1559 Buterin et al. (2019) which propeses a change to the gas auction mechanism or EIP-87 Buterin (2016) which proposes a time based cost for storage ("renting storage"). These and other similar EIPs would drastically change how gas tokens work and would most likely render the current gas token contracts unusable.

Looking even further ahead to the planned switch of Ethereum to proof of stake makes it very difficult to predict if and in which form gas tokens may remain relevant. This area requires further research.

List of Figures

List of Tables

List of Algorithms

References

Adams, H. (2017), 'Uniswap Whitepaper'.
URL: *https://hackmd.io/@HaydenAdams/HJ9jLsfTz?type=view*

Angeris, G., Kao, H.-T., Chiang, R., Noyes, C. and Chitra, T. (2019), 'An Analysis of Uniswap Markets', *Cryptoeconomic Systems Journal* .
URL: *https://papers.ssrn.com/abstract=3602203*

Breidenbach, L., Daian, P. and Tramèr, F. (2017), 'GasToken.io - Cheaper Ethereum Transactions, Today'.
URL: *https://gastoken.io/*

Bukov, A. (2020*a*), '1inch introduces Chi Gastoken. The 1inch team has launched Chi, a... | by 1inch | Jun, 2020 | Medium'.
URL: *https://medium.com/@1inch.exchange/1inch-introduces-chi-gastoken-d0bd5bb0f92b*

Bukov, A. (2020*b*), 'Chi Gastoken and deployer.eth by #1inch'.
URL: *https://www.youtube.com/watch?v=-uJyLK9BGpo&feature=youtu.be*

Buterin, V. (2016), 'Blockchain rent: exponential rent-to-own edition · Issue #87 · ethereum/EIPs · GitHub'.
URL: *https://github.com/ethereum/EIPs/issues/87*

Buterin, V. (2018), 'EIP 1014: Skinny CREATE2'.
URL: *https://eips.ethereum.org/EIPS/eip-1014*

Buterin, V., Conner, E., Dudley, R., Slipper, M. and Norden, I. (2019), 'EIP 1559: Fee market change for ETH 1.0 chain'.
URL: *https://eips.ethereum.org/EIPS/eip-1559*

CoinMarketCap (2020), 'Cryptocurrency Market Capitalizations | CoinMarketCap'.
URL: *https://coinmarketcap.com/*

Electric Capital (2019), 'Developer Report'.

ETHGasStation (2020), 'ETH Gas Station | Consumer oriented metrics for the Ethereum gas market'.
URL: *https://ethgasstation.info/*

Gavin, W. (2014), Ethereum: A secure decentralised generalised transaction ledger - petersburg version, PhD thesis.

Hauser, B. (2019), 'GitHub - eth-brownie/brownie: A Python-based development and testing framework for smart contracts targeting the Ethereum Virtual Machine.'.
URL: *https://github.com/eth-brownie/brownie*

Johguse (2020), 'johguse · GitHub'.
URL: *https://github.com/johguse*

Medvedev, E. and the D5.ai Team (2017), 'Ethereum ETL'.
URL: *https://github.com/blockchain-etl/ethereum-etl*

Nadler, M. (2020*a*), 'GitHub - matnad/liquid-gas-token: Liquid Gas Token (LGT) for Ethereum'.
URL: *https://github.com/matnad/liquid-gas-token*

Nadler, M. (2020*b*), 'Supercharged Ethereum Main Net Testing on your own Node with Brownie. | by matnad | Medium'.
URL: *https://medium.com/@matnad/supercharged-ethereum-main-net-testing-on-your-own-node-with-brownie-eb4cb886de7c*

Valson, V. U. (2020), 'Transaction Fee Estimations: How To Save On Gas? Part 2 | by Pranay Valson | Upvest | Medium'.
URL: *https://medium.com/upvest/transaction-fee-estimations-how-to-save-on-gas-part-2-72f908b13d67*

Zhang, Y., Chen, X. and Park, D. (2018), Formal specification of constant product (xy=k) market maker model and implementation, Technical report.