Ethereum Improvement Proposals

All Core Networking Interface ERC Meta Informational

EIP-1559: Fee market change for ETH 1.0 chain ‹›

Author	Vitalik Buterin, Eric Conner, Rick Dudley, Matthew Slipper, Ian Norden, Abdelhamid Bakhta
Discussions-To	https://ethereum-magicians.org/t/eip-1559-fee-market-change-for-eth-1-0-chain/2783
Status	Final
Type	Standards Track
Category	Core
Created	2019-04-13
Requires	2718, 2930

Table of Contents

Simple Summary

A transaction pricing mechanism that includes fixed-per-block network fee that is burned and dynamically expands/contracts block sizes to deal with transient congestion.

Abstract

We introduce a new EIP-2718 transaction type, with the format `0x02 || rlp([chain_id, nonce, max_priority_fee_per_gas, max_fee_per_gas, gas_limit, destination, amount, data, access_list, signature_y_parity, signature_r, signature_s])`.

There is a base fee per gas in protocol, which can move up or down each block according to a formula which is a function of gas used in parent block and gas target (block gas limit divided by elasticity multiplier) of parent block. The algorithm results in the base fee per gas increasing when blocks are above the gas target, and decreasing when blocks are below the gas target. The base fee per gas is burned. Transactions specify the maximum fee per gas they are willing to give to miners to incentivize them to include their transaction (aka: priority fee). Transactions also specify the maximum fee per gas they are willing to pay total (aka: max fee), which covers both the priority fee and the block's network fee per gas (aka: base fee). The transaction will always pay the base fee per gas of the block it was included in, and they will pay the priority fee per gas set in the transaction, as long as the combined amount of the two fees doesn't exceed the transaction's maximum fee per gas.

Motivation

Ethereum historically priced transaction fees using a simple auction mechanism, where users send transactions with bids ("gasprices") and miners choose transactions with the highest bids, and transactions that get included pay the bid that they specify. This leads to several large sources of inefficiency:

- **Mismatch between volatility of transaction fee levels and social cost of transactions**: bids to include transactions on mature public blockchains, that have enough usage so that blocks are full, tend to be extremely volatile. It's absurd to suggest that the cost incurred by the network from accepting one more transaction into a block actually is 10x more when the cost per gas is 10 nanoeth compared to when the cost per gas is 1 nanoeth; in both cases, it's a difference between 8 million gas and 8.02 million gas.
- **Needless delays for users**: because of the hard per-block gas limit coupled with natural volatility in transaction volume, transactions often wait for several blocks before getting included, but this is socially unproductive; no one significantly gains from the fact that there is no "slack" mechanism that allows one block to be bigger and the next block to be smaller to meet block-by-block differences in demand.
- **Inefficiencies of first price auctions**: The current approach, where transaction senders publish a transaction with a bid a maximum fee, miners choose the highest-paying transactions, and everyone pays what they bid. This is well-known in mechanism design literature to be highly inefficient, and so complex fee estimation algorithms are required. But even these algorithms often end up not working very well, leading to frequent fee overpayment.
- **Instability of blockchains with no block reward**: In the long run, blockchains where there is no issuance (including Bitcoin and Zcash) at present intend to switch to rewarding miners entirely through transaction fees. However, there are known issues with this that likely leads to a lot of instability, incentivizing mining "sister blocks" that steal transaction fees, opening up much stronger selfish mining attack vectors, and more. There is at present no good mitigation for this.

The proposal in this EIP is to start with a base fee amount which is adjusted up and down by the protocol based on how congested the network is. When the network exceeds the target per-block gas usage, the base fee increases slightly and when capacity is below the target, it decreases slightly. Because these base fee changes are constrained, the maximum difference in base fee from block to block is predictable. This then allows wallets to auto-set the gas fees for users in a highly reliable fashion. It is expected that most users will not have to manually adjust gas fees, even in periods of high network activity. For most users the base fee will be estimated by their wallet and a small priority fee, which compensates miners taking on orphan risk (e.g. 1 nanoeth), will be automatically set. Users can also manually set the transaction max fee to bound their total costs.

An important aspect of this fee system is that miners only get to keep the priority fee. The base fee is always burned (i.e. it is destroyed by the protocol). This ensures that only ETH can ever be used to pay for transactions on Ethereum, cementing the economic value of ETH within the Ethereum platform and reducing risks associated with miner extractable value (MEV). Additionally, this burn counterbalances Ethereum inflation while still giving the block reward and priority fee to miners. Finally, ensuring the miner of a block does not receive the base fee is important because it removes miner incentive to manipulate the fee in order to extract more fees from users.

Specification

Block validity is defined in the reference implementation below. The `GASPRICE` (`0x3a`) opcode **MUST** return the `effective_gas_price` as defined in the reference implementation below.

As of `FORK_BLOCK_NUMBER`, a new EIP-2718 transaction is introduced with `TransactionType` 2.

The intrinsic cost of the new transaction is inherited from EIP-2930, specifically `21000 + 16 * non-zero calldata bytes + 4 * zero calldata bytes + 1900 * access list storage key count + 2400 * access list address count`.

The EIP-2718 `TransactionPayload` for this transaction is `rlp([chain_id, nonce, max_priority_fee_per_gas, max_fee_per_gas, gas_limit, destination, amount, data, access_list, signature_y_parity, signature_r, signature_s])`.

The `signature_y_parity, signature_r, signature_s` elements of this transaction represent a secp256k1 signature over `keccak256(0x02 || rlp([chain_id, nonce, max_priority_fee_per_gas, max_fee_per_gas, gas_limit, destination, amount, data, access_list]))`.

The EIP-2718 `ReceiptPayload` for this transaction is `rlp([status, cumulative_transaction_gas_used, logs_bloom, logs])`.

Note: `//` *is integer division, round down.*

```
from typing import Union, Dict, Sequence, List, Tuple, Literal
from dataclasses import dataclass, field
from abc import ABC, abstractmethod

@dataclass
```

```
class TransactionLegacy:
	signer_nonce: int = 0
	gas_price: int = 0
	gas_limit: int = 0
	destination: int = 0
	amount: int = 0
	payload: bytes = bytes()
	v: int = 0
	r: int = 0
	s: int = 0

@dataclass
class Transaction2930Payload:
	chain_id: int = 0
	signer_nonce: int = 0
	gas_price: int = 0
	gas_limit: int = 0
	destination: int = 0
	amount: int = 0
	payload: bytes = bytes()
	access_list: List[Tuple[int, List[int]]] = field(default_factory=list)
	signature_y_parity: bool = False
	signature_r: int = 0
	signature_s: int = 0

@dataclass
class Transaction2930Envelope:
	type: Literal[1] = 1
	payload: Transaction2930Payload = Transaction2930Payload()

@dataclass
```

```
class Transaction1559Payload:
	chain_id: int = 0
	signer_nonce: int = 0
	max_priority_fee_per_gas: int = 0
	max_fee_per_gas: int = 0
	gas_limit: int = 0
	destination: int = 0
	amount: int = 0
	payload: bytes = bytes()
	access_list: List[Tuple[int, List[int]]] = field(default_factory=list)
	signature_y_parity: bool = False
	signature_r: int = 0
	signature_s: int = 0

@dataclass
class Transaction1559Envelope:
	type: Literal[2] = 2
	payload: Transaction1559Payload = Transaction1559Payload()

Transaction2718 = Union[Transaction1559Envelope, Transaction2930Envelope]

Transaction = Union[TransactionLegacy, Transaction2718]

@dataclass
class NormalizedTransaction:
	signer_address: int = 0
	signer_nonce: int = 0
	max_priority_fee_per_gas: int = 0
	max_fee_per_gas: int = 0
	gas_limit: int = 0
	destination: int = 0
```

```
        amount: int = 0
        payload: bytes = bytes()
        access_list: List[Tuple[int, List[int]]] = field(default_factory=list)

@dataclass
class Block:
        parent_hash: int = 0
        uncle_hashes: Sequence[int] = field(default_factory=list)
        author: int = 0
        state_root: int = 0
        transaction_root: int = 0
        transaction_receipt_root: int = 0
        logs_bloom: int = 0
        difficulty: int = 0
        number: int = 0
        gas_limit: int = 0 # note the gas_limit is the gas_target * ELASTICITY_MULTIPLIER
        gas_used: int = 0
        timestamp: int = 0
        extra_data: bytes = bytes()
        proof_of_work: int = 0
        nonce: int = 0
        base_fee_per_gas: int = 0

@dataclass
class Account:
        address: int = 0
        nonce: int = 0
        balance: int = 0
        storage_root: int = 0
        code_hash: int = 0
```

```
INITIAL_BASE_FEE = 1000000000
INITIAL_FORK_BLOCK_NUMBER = 10 # TBD
BASE_FEE_MAX_CHANGE_DENOMINATOR = 8
ELASTICITY_MULTIPLIER = 2

class World(ABC):
	def validate_block(self, block: Block) -> None:
		parent_gas_target = self.parent(block).gas_limit // ELASTICITY_MULTIPLIER
		parent_gas_limit = self.parent(block).gas_limit

		# on the fork block, don't account for the ELASTICITY_MULTIPLIER to avoid
		# unduly halving the gas target.
		if INITIAL_FORK_BLOCK_NUMBER == block.number:
			parent_gas_target = self.parent(block).gas_limit
			parent_gas_limit = self.parent(block).gas_limit * ELASTICITY_MULTIPLIER

		parent_base_fee_per_gas = self.parent(block).base_fee_per_gas
		parent_gas_used = self.parent(block).gas_used
		transactions = self.transactions(block)

		# check if the block used too much gas
		assert block.gas_used <= block.gas_limit, 'invalid block: too much gas used'

		# check if the block changed the gas limit too much
		assert block.gas_limit < parent_gas_limit + parent_gas_limit // 1024, 'invalid block: gas limit increa
		assert block.gas_limit > parent_gas_limit - parent_gas_limit // 1024, 'invalid block: gas limit decrea

		# check if the gas limit is at least the minimum gas limit
		assert block.gas_limit >= 5000

		# check if the base fee is correct
```

```
        if INITIAL_FORK_BLOCK_NUMBER == block.number:
                expected_base_fee_per_gas = INITIAL_BASE_FEE
        elif parent_gas_used == parent_gas_target:
                expected_base_fee_per_gas = parent_base_fee_per_gas
        elif parent_gas_used > parent_gas_target:
                gas_used_delta = parent_gas_used - parent_gas_target
                base_fee_per_gas_delta = max(parent_base_fee_per_gas * gas_used_delta // parent_gas_target //
                expected_base_fee_per_gas = parent_base_fee_per_gas + base_fee_per_gas_delta
        else:
                gas_used_delta = parent_gas_target - parent_gas_used
                base_fee_per_gas_delta = parent_base_fee_per_gas * gas_used_delta // parent_gas_target // BASE
                expected_base_fee_per_gas = parent_base_fee_per_gas - base_fee_per_gas_delta
        assert expected_base_fee_per_gas == block.base_fee_per_gas, 'invalid block: base fee not correct'

        # execute transactions and do gas accounting
        cumulative_transaction_gas_used = 0
        for unnormalized_transaction in transactions:
                # Note: this validates transaction signature and chain ID which must happen before we normaliz
                signer_address = self.validate_and_recover_signer_address(unnormalized_transaction)
                transaction = self.normalize_transaction(unnormalized_transaction, signer_address)

                signer = self.account(signer_address)

                signer.balance -= transaction.amount
                assert signer.balance >= 0, 'invalid transaction: signer does not have enough ETH to cover att
                # the signer must be able to afford the transaction
                assert signer.balance >= transaction.gas_limit * transaction.max_fee_per_gas

                # ensure that the user was willing to at least pay the base fee
                assert transaction.max_fee_per_gas >= block.base_fee_per_gas
```

```
                # Prevent impossibly large numbers
                assert transaction.max_fee_per_gas < 2**256
                # Prevent impossibly large numbers
                assert transaction.max_priority_fee_per_gas < 2**256
                # The total must be the larger of the two
                assert transaction.max_fee_per_gas >= transaction.max_priority_fee_per_gas

                # priority fee is capped because the base fee is filled first
                priority_fee_per_gas = min(transaction.max_priority_fee_per_gas, transaction.max_fee_per_gas -
                # signer pays both the priority fee and the base fee
                effective_gas_price = priority_fee_per_gas + block.base_fee_per_gas
                signer.balance -= transaction.gas_limit * effective_gas_price
                assert signer.balance >= 0, 'invalid transaction: signer does not have enough ETH to cover gas
                gas_used = self.execute_transaction(transaction, effective_gas_price)
                gas_refund = transaction.gas_limit - gas_used
                cumulative_transaction_gas_used += gas_used
                # signer gets refunded for unused gas
                signer.balance += gas_refund * effective_gas_price
                # miner only receives the priority fee; note that the base fee is not given to anyone (it is b
                self.account(block.author).balance += gas_used * priority_fee_per_gas

            # check if the block spent too much gas transactions
            assert cumulative_transaction_gas_used == block.gas_used, 'invalid block: gas_used does not equal tota

            # TODO: verify account balances match block's account balances (via state root comparison)
            # TODO: validate the rest of the block

        def normalize_transaction(self, transaction: Transaction, signer_address: int) -> NormalizedTransaction:
            # legacy transactions
            if isinstance(transaction, TransactionLegacy):
                return NormalizedTransaction(
```

```
                signer_address = signer_address,
                signer_nonce = transaction.signer_nonce,
                gas_limit = transaction.gas_limit,
                max_priority_fee_per_gas = transaction.gas_price,
                max_fee_per_gas = transaction.gas_price,
                destination = transaction.destination,
                amount = transaction.amount,
                payload = transaction.payload,
                access_list = [],
        )
# 2930 transactions
elif isinstance(transaction, Transaction2930Envelope):
        return NormalizedTransaction(
                signer_address = signer_address,
                signer_nonce = transaction.payload.signer_nonce,
                gas_limit = transaction.payload.gas_limit,
                max_priority_fee_per_gas = transaction.payload.gas_price,
                max_fee_per_gas = transaction.payload.gas_price,
                destination = transaction.payload.destination,
                amount = transaction.payload.amount,
                payload = transaction.payload.payload,
                access_list = transaction.payload.access_list,
        )
# 1559 transactions
elif isinstance(transaction, Transaction1559Envelope):
        return NormalizedTransaction(
                signer_address = signer_address,
                signer_nonce = transaction.payload.signer_nonce,
                gas_limit = transaction.payload.gas_limit,
                max_priority_fee_per_gas = transaction.payload.max_priority_fee_per_gas,
                max_fee_per_gas = transaction.payload.max_fee_per_gas,
```

```
                destination = transaction.payload.destination,
                amount = transaction.payload.amount,
                payload = transaction.payload.payload,
                access_list = transaction.payload.access_list,
            )
        else:
            raise Exception('invalid transaction: unexpected number of items')

    @abstractmethod
    def parent(self, block: Block) -> Block: pass

    @abstractmethod
    def block_hash(self, block: Block) -> int: pass

    @abstractmethod
    def transactions(self, block: Block) -> Sequence[Transaction]: pass

    # effective_gas_price is the value returned by the GASPRICE (0x3a) opcode
    @abstractmethod
    def execute_transaction(self, transaction: NormalizedTransaction, effective_gas_price: int) -> int: pass

    @abstractmethod
    def validate_and_recover_signer_address(self, transaction: Transaction) -> int: pass

    @abstractmethod
    def account(self, address: int) -> Account: pass
```

Backwards Compatibility

Legacy Ethereum transactions will still work and be included in blocks, but they will not benefit directly from the new pricing system. This is due to the fact that upgrading from legacy transactions to new transactions results in the legacy transaction's `gas_price` entirely being consumed either by the `base_fee_per_gas` and the `priority_fee_per_gas`.

Block Hash Changing

The datastructure that is passed into keccak256 to calculate the block hash is changing, and all applications that are validating blocks are valid or using the block hash to verify block contents will need to be adapted to support the new datastructure (one additional item). If you only take the block header bytes and hash them you should still correctly get a hash, but if you construct a block header from its constituent elements you will need to add in the new one at the end.

GASPRICE

Previous to this change, `GASPRICE` represented both the ETH paid by the signer per gas for a transaction as well as the ETH received by the miner per gas. As of this change, `GASPRICE` now only represents the amount of ETH paid by the signer per gas, and the amount a miner was paid for the transaction is no longer accessible directly in the EVM.

Test Cases

TODO

Security Considerations

Increased Max Block Size/Complexity

This EIP will increase the maximum block size, which could cause problems if miners are unable to process a block fast enough as it will force them to mine an empty block. Over time, the average block size should remain about the same as without this EIP, so this is only an issue for short term size bursts. It is possible that one or more clients may handle short term size bursts poorly and error (such as out of memory or similar) and client implementations should make sure their clients can appropriately handle individual blocks up to max size.

Transaction Ordering

With most people not competing on priority fees and instead using a baseline fee to get included, transaction ordering now depends on individual client internal implementation details such as how they store the transactions in memory. It is recommended that transactions with the same priority fee be sorted by time the transaction was received to protect the network from spamming attacks where the attacker throws a bunch of transactions into the pending pool in order to ensure that at least one lands in a favorable position. Miners should still prefer higher gas premium transactions over those with a lower gas premium, purely from a selfish mining perspective.

Miners Mining Empty Blocks

It is possible that miners will mine empty blocks until such time as the base fee is very low and then proceed to mine half full blocks and revert to sorting transactions by the priority fee. While this attack is possible, it is not a particularly stable equilibrium as long as mining is decentralized. Any defector from this strategy will be more profitable than a miner participating in the attack for as long as the attack continues (even after the base fee reached 0). Since any miner can anonymously defect from a cartel, and there is no way to prove that a particular miner defected, the only feasible way to execute this attack would be to control 50% or more of hashing power. If an attacker had exactly 50% of hashing power, they would make no Ether from priority fee while defectors would make double the Ether from priority fees. For an attacker to turn a profit, they need to have some amount over 50% hashing power, which means they can instead execute double spend attacks or simply ignore any other miners which is a far more profitable strategy.

Should a miner attempt to execute this attack, we can simply increase the elasticity multiplier (currently 2x) which requires they have even more hashing power available before the attack can even be theoretically profitable against defectors.

ETH Burn Precludes Fixed Supply

By burning the base fee, we can no longer guarantee a fixed Ether supply. This could result in economic instability as the long term supply of ETH will no longer be constant over time. While a valid concern, it is difficult to quantify how much of an impact this will have. If more is burned on base fee than is generated in mining rewards then ETH will be deflationary and if more is generated in mining rewards than is burned then ETH will be inflationary. Since we cannot control user demand for block space, we cannot assert at the moment whether ETH will end up inflationary or deflationary, so this change causes the core developers to lose some control over Ether's long term quantity.

Copyright

Citation

Please cite this document as:

Vitalik Buterin, Eric Conner, Rick Dudley, Matthew Slipper, Ian Norden, Abdelhamid Bakhta, "EIP-1559: Fee market change for ETH 1.0 chain," *Ethereum Improvement Proposals*, no. 1559, April 2019. [Online serial]. Available: https://eips.ethereum.org/EIPS/eip-1559.
