---

eip: XXXX

title: Sharding-format blob-carrying transactions

author: Vitalik Buterin (@vbuterin), Dankrad Feist (@dankrad)

discussions-to: https://example.lol

status: Draft

type: Standards Track

category: Core

created: 2022-02-02

Simple Summary

Introduce a new transaction format for "blob-carrying transactions" which contain a large amount of data that cannot be accessed by EVM execution, but whose commitment can be accessed. The format is intended to be fully compatible with the format that will be used in full sharding.

Motivation

Rollups are in the short and medium term, and possibly in the long term, the only trustless scaling solution for Ethereum. Transaction fees on L1 have been very high for months and there is greater urgency in doing anything required to help facilitate an ecosystem-wide move to rollups. Rollups are significantly reducing fees for many Ethereum users: Optimism and Arbitrum frequently provide fees that are ~3-8x lower than the Ethereum base layer itself, and ZK rollups, which have better data compression and can avoid including signatures, have fees ~40-100x lower than the base layer.

However, even these fees are too expensive for many users. The long-term solution to the long-term inadequacy of rollups by themselves has always been data sharding, which would add ~16 MB per block of dedicated data space to the chain that rollups could use. However, data sharding will still take a considerable amount of time to finish implementing and deploying.

This EIP provides a stop-gap solution until that point by implementing the *transaction format* that would be used in sharding, but not actually sharding those transactions. Instead, they would simply be part of the beacon block and would need to be downloaded by all consensus nodes (but can be deleted after only a relatively short delay). There would be a reduced cap on the number of these transactions that can be included, corresponding to a target of ~1 MB per block and a limit of ~2 MB.

Specification

Parameters

| Constant | Value |

| - | - |

| SYSTEM_STATE_ADDRESS | 0x000.....0100|

| TOTAL_BLOB_TXS_STORAGE_SLOT | 0 |

| BLOB_TX_TYPE | Bytes1(0x05) |

| CHUNKS_PER_BLOB | 4096 |

| BLS_MODULUS | 52435875175126190479447740508185965837690552500527637822603658699938581184513 |

| KZG_SETUP_G2 | Vector[G2Point, CHUNKS_PER_BLOB], contents TBD |

| KZG_SETUP_LAGRANGE | Vector[BLSCommitment, CHUNKS_PER_BLOB], contents TBD |

| BLOB_COMMITMENT_VERSION_KZG | Bytes1(0x01) |

| BLOB_VERIFICATION_PRECOMPILE_ADDRESS | TBD |

| BLOB_VERIFICATION_PRECOMPILE_GAS | 1800000 |

| POINT_EVALUATION_PRECOMPILE_ADDRESS | TBD |

| POINT_EVALUATION_PRECOMPILE_GAS | 50000 |

| MAX_BLOBS_PER_BLOCK | 16 |

| TARGET_BLOBS_PER_BLOCK | 8 |

| MAX_BLOBS_PER_TX | 2 |

| GASPRICE_UPDATE_FRACTION_PER_BLOB | 64 |

| MAX_OBJECT_LIST_SIZE | 2**24 |

Type aliases

| Type | Base type | Additional checks |

| - | - | - |

| BLSFieldElement | uint256 | x < BLS_MODULUS |

| Blob | Vector[BLSFieldElement, CHUNKS_PER_BLOB] | |

| VersionedHash | Bytes32 | |

| KZGCommitment | Bytes48 | Same as BLS standard "is valid pubkey" check but also allows 0x00..00 for point-at-infinity |

Helpers

Converts a blob to its corresponding KZG point:

```python
def blob_to_kzg(blob: Vector[BLSFieldElement, CHUNKS_PER_BLOB]) -> KZGCommitment:
computed_kzg = bls.Z1
for value, point_kzg in zip(tx.blob, KZG_SETUP_LAGRANGE):
    assert value < BLS_MODULUS
    computed_kzg = bls.add(
        computed_kzg,
        bls.multiply(point_kzg, value)
    )
return computed_kzg
```

Converts a KZG point into a versioned hash:

```python
def kzg_to_versioned_hash(kzg: KZGCommitment) -> VersionedHash:
return BLOB_COMMITMENT_VERSION_KZG + hash(kzg)[1:]
```

Verifies a KZG evaluation proof:

```python
def verify_kzg_proof(polynomial_kzg: KZGCommitment,
                x: BLSFieldElement,
                y: BLSFieldElement,
                quotient_kzg: KZGCommitment):
# Verify: P - y = Q * (X - x)
```

```
X_minus_x = bls.add(KZG_SETUP_G2[1], bls.multiply(bls.G2, BLS_MODULUS - x))
P_minus_y = bls.add(polynomial_kzg, bls.multiply(bls.G1, BLS_MODULUS - y))
return bls.pairing_check([
    [P_minus_y, bls.neg(bls.G2)],
    [quotient_kzg, X_minus_x]
])
```

Approximates `2 ** (numerator / denominator)`, with the simplest possible approximation that is continuous and has a continuous derivative:

```python
def fake_exponential(numerator: int, denominator: int) -> int:
cofactor = 2 ** (numerator // denominator)
fractional = numerator % denominator
return cofactor + (
    fractional * cofactor * 2 +
    (fractional ** 2 * cofactor) // denominator
) // (denominator * 3)
```

New transaction type

We introduce a new [EIP-2718](#) transaction type, with the format being the single byte `BLOB_TX_TYPE` followed by an SSZ encoding of the `SignedBlobTransaction` container comprising the transaction contents:

```python
class SignedBlobTransaction(Container):
header: BlobTransaction
signature: ECDSASignature
class BlobTransaction(Container):
nonce: uint64
gas: uint64
max_basefee: uint256
priority_fee: uint256
to: Address # Bytes20
value: uint256
data: Bytes
blob_versioned_hashes: List[VersionedHash, MAX_OBJECT_LIST_SIZE]
class ECDSASignature(Container):
v: uint8
r: uint256
s: uint256
```

When the transaction is passed through the network (see the [Networking](#) section below), it comes in a package that also includes a list `blobs: List[Blob, MAX_OBJECT_LIST_SIZE]`. We expect the i'th versioned hash to match the i'th blob. The execution

layer, however, does not have access to these blobs.

The signature is verified and tx.origin is calculated as follows:

```python
def get_origin(tx: SignedBlobTransaction) -> Address:
    sig = tx.signature
    return ecrecover(ssz.hash_tree_root(tx.header), sig.v, sig.r, sig.s)
```

We also check three restrictions:

- All hashes in `blob_versioned_hashes` must start with the byte `BLOB_COMMITMENT_VERSION_KZG`
- There may be at most `MAX_BLOBS_PER_TX` blob commitments in any single transaction.
- There may be at most `MAX_BLOBS_PER_BLOCK` total blob commitments in a valid block.

In the `ExecutionPayload`, we add a `blob_transactions: List[SignedBlobTransaction, MAX_OBJECT_LIST_SIZE]` list. These transactions should be processed after other transactions.

Beacon-chain-side validation

We add to the `BeaconBlockBody` a new object, `blob_kzgs: List[KZGCommitment, MAX_OBJECT_LIST_SIZE]`. We add a `BeaconBlockAndBlobs` struct:

```python
class BeaconBlockAndBlobs(Container):
    block: SignedBeaconBlock
    blobs: List[Blob, MAX_OBJECT_LIST_SIZE]
```

Nodes broadcast this over the network instead of plain beacon blocks. When a node receives it, it first calls `verify_blobs` and if this call passes it runs the usual processing on the beacon block.

```python
def verify_blobs(block_and_blobs: BeaconBlockAndBlobs):
    kzgs = block_and_blobs.block.message.body.blob_kzgs
    blobs = block_and_blobs.blobs
    assert len(kzgs) == len(blobs)
    for kzg, blob in zip(kzgs, blobs):
        assert blob_to_kzg(blob) == kzg
```

We also add a cross-validation requirement to check equivalence between the `BeaconBlock` and its contained `ExecutionPayload` (this check can be done later than the verifications of the beacon block and the payload):

```python
def cross_validate(block: BeaconBlock):
    body = block.message.body
    all_versioned_hashes = []
    for tx in body.execution_payload.blob_transactions:
        all_versioned_hashes.extend(tx.header.blob_versioned_hashes)
    assert all_versioned_hashes == [
```

```
    kzg_to_versioned_hash(kzg) for kzg in body.blob_kzgs
]
```

Opcode to get versioned hashes

We add an opcode `GET_VERSIONED_HASH_OPCODE` which takes as input one stack argument `index`, and returns `tx.header.blob_versioned_hashes[index]` if `index < len(tx.header.blob_versioned_hashes)`, and otherwise zero.

Blob verification precompile

Add a precompile at `BLOB_VERIFICATION_PRECOMPILE_ADDRESS` that checks a blob against a versioned hash. The precompile costs `BLOB_VERIFICATION_PRECOMPILE_GAS` and executes the following logic:

```python
def blob_verification_precompile(input: Bytes) -> Bytes:
# First 32 bytes = expected versioned hash
expected_v_hash = input[:32]
assert expected_v_hash[:1] == BLOB_COMMITMENT_VERSION_KZG
# Remaining bytes = list of little-endian data points
assert len(input) == 32 + 32 * CHUNKS_PER_BLOB
input_points = [
    int.from_bytes(input[i:i+32], 'little')
    for i in range(32, len(input), 32)
]
assert kzg_to_versioned_hash(blob_to_kzg(input_points)) == expected_v_hash
return Bytes([])
```

Its logic is designed so that it is future proof, and new versions of commitments can be added if needed.

Point evaluation precompile

Add a precompile at `POINT_EVALUATION_PRECOMPILE_ADDRESS` that evaluates a proof that a particular blob resolves to a particular value at a point. The precompile costs `POINT_EVALUATION_PRECOMPILE_GAS` and executes the following logic:

```python
def point_evaluation_precompile(input: Bytes) -> Bytes:
# Verify P(z) = a
# versioned hash: first 32 bytes
versioned_hash = input[:32]
# Evaluation point: next 32 bytes
x = int.from_bytes(input[32:64], 'little')
assert x < BLS_MODULUS
# Expected output: next 32 bytes
y = int.from_bytes(input[64:96], 'little')
assert y < BLS_MODULUS
# The remaining data will always be the proof, including in future versions
# input kzg point: next 48 bytes
```

```
data_kzg = input[96:144]

assert kzg_to_versioned_hash(data_kzg) == versioned_hash

# Quotient kzg: next 48 bytes

quotient_kzg = input[144:192]

assert verify_kzg_proof(data_kzg, x, y, quotient_kzg)

return Bytes([])
```

Gas price update rule

We propose a simple independent EIP 1559-style targeting rule to compute the gas cost of the transaction. We use the TOTAL_BLOB_TXS_STORAGE_SLOT storage slot of the SYSTEM_STATE_ADDRESS address to store persistent data needed to compute the cost. Note that unlike existing transaction types, the gas cost is dependent on the pre-state of the block.

For early draft implementations, at the moment we recommend setting `get_blob_gas` to simply return `120000` and not implementing `update_blob_gas`, as this section is more likely to change than the others.

```python
def get_intrinsic_gas(tx: SignedBlobTransaction, pre_state: ExecState) -> int:

return (

    20000 +

    16 * len(tx.header.data) - 12 * len(tx.header.data.count(0)) +

    len(tx.header.blob_versioned_hashes) * get_blob_gas(pre_state)

)
def get_blob_gas(pre_state: ExecState) -> int:

blocks_since_start = get_current_block(pre_state) - FORK_BLKNUM

targeted_total = blocks_since_start * TARGET_BLOB_TXS_PER_BLOCK

actual_total = read_storage(

    pre_state,

    SYSTEM_STATE_ADDRESS,

    TOTAL_BLOB_TXS_STORAGE_SLOT

)

if actual_total < targeted_total:

    return 0

else:

    return fake_exponential(

        actual_total - targeted_total,

        GASPRICE_UPDATE_FRACTION_PER_BLOB

    )
```

We update at the end of a block, as follows:

```python
def update_blob_gas(state: ExecState, blob_txs_in_block: int):

current_total = read_storage(

    state,
```

```
    SYSTEM_STATE_ADDRESS,
    TOTAL_BLOB_TXS_STORAGE_SLOT
)
# Don't let the new total fall behind the targeted total to avoid
# accumulating a long period of very low fees
blocks_since_start = get_current_block(pre_state) - FORK_BLKNUM
targeted_total = blocks_since_start * TARGET_BLOB_TXS_PER_BLOCK
new_total = max(current_total + blob_txs_in_block, targeted_total)
write_storage(
    state,
    SYSTEM_STATE_ADDRESS,
    TOTAL_BLOB_TXS_STORAGE_SLOT,
    new_total
)
```

Networking

Transactions are passed around the network in this format:

```python
class BlobTransactionNetworkWrapper(Container):
tx: SignedBlobTransaction
# KZGCommitment = Bytes48
blob_kzgs: List[KZGCommitment, MAX_OBJECT_LIST_SIZE]
# BLSFieldElement = uint256
blobs: List[Vector[BLSFieldElement, CHUNKS_PER_BLOB], MAX_OBJECT_LIST_SIZE]
```

We do network-level validation of `BlobTransactionNetworkWrapper` objects as follows:

```python
def validate_blob_transaction_wrapper(wrapper: BlobTransactionNetworkWrapper):
commitments = wrapper.tx.header.blob_commitments
kzgs = wrapper.blob_kzgs
blobs = wrapper.blobs
assert len(commitments) == len(kzgs) == len(blobs)
for commitment, kzg, blob in zip(commitments, kzgs, blobs):
    assert kzg == blob_to_kzg(blob)
    assert commitment == kzg_to_commitment(kzg)
```

Rationale

On the path to sharding

This EIP introduces blob transactions in the same format in which they are expected to exist in the final sharding specification. This provides temporary scaling relief for rollups by allowing them to scale to 2 MB per slot, with a separate fee market allowing fees to be very low while usage of this system is limited.

The core goal of rollup scaling stopgaps is to provide temporary scaling relief, without imposing extra development burdens on rollups to take advantage of this relief. Today, rollups use calldata. In the future, rollups will have no choice but to use sharded data (also called "blobs") because sharded data will be much cheaper. Hence, rollups cannot avoid making a large upgrade to how they process data at least once along the way. But what we *can* do is ensure that rollups need to *only* upgrade once. This immediately implies that there are exactly two possibilities for a stopgap: (i) reducing the gas costs of existing calldata, and (ii) bringing forward the format that will be used for sharded data, but not yet actually sharding it. EIP-4488 is a solution of category (i); this is a solution of category (ii).

The main tradeoff in designing this EIP is that of implementing more now versus having to implement more later: do we implement 25% of the work on the way to full sharding, or 50%, or 75%?

The work that is already done in this EIP includes:

- A new transaction type, of the exact same format that will need to exist in "full sharding"
- *All* of the execution-layer logic required for full sharding
- *All* of the execution / consensus cross-verification logic required for full sharding
- Layer separation between `BeaconBlock` verification and data availability sampling blobs
- Most of the `BeaconBlock` logic required for full sharding
- A self-adjusting independent gasprice for blobs (multidimensional EIP 1559 with an exponential pricing rule)

The work that remains to be done to get to full sharding includes:

- A low-degree extension of the `blob_kzgs` in the consensus layer to allow 2D sampling
- An actual implementation of data availability sampling
- PBS (proposer/builder separation), to avoid requiring individual validators to process 32 MB of data in one slot
- Proof of custody or similar in-protocol requirement for each validator to verify a particular part of the sharded data in each block

This EIP also sets the stage for longer-term protocol cleanups:

- It adds an SSZ transaction type which is slightly gas-advantaged (1000 discount) to nudge people toward using it, and paves the precedent that all new transaction types should be SSZ
- Its (cleaner) gas price update rule could be applied to the primary basefee. It also starts the trend toward using the state to store system state instead of having "implied state" such as historical blocks and hashes.

How rollups would function

Instead of putting rollup block data in transaction calldata, rollups would expect rollup block submitters to put the data into blobs. This guarantees availability (which is what rollups need) but would be much cheaper than calldata.

Optimistic rollups only need to actually access the underlying data when fraud proofs are being submitted. The fraud proof submission function would require the full contents of the fraudulent blob to be submitted as part of calldata. It would use the blob verification function to verify the data against the versioned hash that was submitted before, and then perform the fraud proof verification on that data as is done today.

ZK rollups would provide two commitments to their transaction or state delta data: the kzg in the blob and some commitment using whatever proof system the ZK rollup uses internally. They would use the commitment proof of equivalence protoco, using the point evaluation precompile, to prove that the kzg (which the protocol ensures points to available data) and the ZK rollup's own commitment refer to the same data.

Versioned hashes

We use versioned hashes (rather than kzgs) as references to blobs in the execution layer to ensure forward compatibility with future changes. For example, if we need to switch to Merkle trees + STARKs for quantum-safety reasons, then we would add a new version, allowing the precompiles to work with the new format. Rollups would not have to make any EVM-level changes to how they work; sequencers would simply have to switch over to using a new transaction type at the appropriate time.

Blob gasprice update rule

The blob gasprice update rule is intended to approximate the formula `blob_gas = 2**(excess_blobs / GASPRICE_UPDATE_FRACTION_PER_BLOB)`, where `excess_blobs` is the total "extra" number of blobs that the chain has accepted relative to the "targeted" number (`TARGET_BLOB_TXS_PER_BLOCK` per block). Like EIP 1559, it's a self-correcting formula: as the excess goes higher, the `blob_gas` increases exponentially, reducing usage and eventually forcing the excess back down.

The block-by-block behavior is roughly as follows. If in block `N`, `blob_gas = G1`, and block `N` has `X` blobs, then in block `N+1`, `excess_blobs` increases by `X - TARGET_BLOB_TXS_PER_BLOCK`, and so the `blob_gas` of block `N+1` increases by a factor of `2**((X - TARGET_BLOB_TXS_PER_BLOCK) / GASPRICE_UPDATE_FRACTION_PER_BLOB)`. Hence, it has a similar effect to the existing EIP 1559, but is more "stable" in the sense that it responds in the same way to the same total usage regardless of how it's distributed.

Backwards Compatibility

Blob non-accessibility

This EIP introduces a transaction type that has a distinct mempool version (`BlobTransactionNetworkWrapper`) and execution-payload version (`SignedBlobTransaction`), with only one-way convertibility between the two. The blobs are in the `BlobTransactionNetworkWrapper` and not in the `SignedBlobTransaction`; instead, they go into the `BeaconBlockBody`. This means that there is now a part of a transaction that will not be accessible from the web3 API.

Mempool issues

Blob transactions are unique in that they have a variable intrinsic gas cost. Hence, a transaction that could be included in one block may be invalid for the next. To prevent mempool attacks, we recommend a simple technique: only propagate transactions whose `gas` is at least twice the current minimum.

Additionally, blob transactions have a large data size at the mempool layer, which poses a mempool DoS risk, though not an unprecedented one as this also applies to transactions with large amounts of calldata. The risk is that an attacker makes and publishes a series of large blob transactions with fees `f9 > f8 > ... > f1` , where each fee is the 10% minimum increment higher than the previous, and finishes it off with a 21000-gas basic transaction with fee `f10`. Hence, an attacker could impose millions of gas worth of load on the network and only pay 21000 gas worth of fees.

We recommend a simple solution: both for blob transactions and for transactions carrying a large amount of calldata, increase the minimum increment for mempool replacement from 1.1x to 2x, decreasing the number of resubmissions an attacker can do at any given fee level by ~7x.

Test Cases

Security Considerations

This EIP increases the size of the `BeaconBlockBody` by a maximum of ~2 MB. This is equal to the theoretical maximum size of a block today (30M gas / 16 gas per calldata byte = 1.875M bytes), and so it will not greatly increase worst-case bandwidth. Post-merge, block times are expected to be static rather than an unpredictable Poisson distribution, giving a guaranteed period of time for large blocks to propagate.

The *sustained* load of this EIP is much lower than alternatives (eg. EIP-4488 and EIP-4490), because there is no existing software that stores the blobs and there is no expectation that they need to be stored for as long as execution payload. This makes it easier to implement a policy that these blobs should be deleted after eg. 30-60 days (a much shorter delay than the delay after which EIP-4444 intends to delete `ExecutionPayload` history).

Copyright