BLS signature aggregation is a powerful technology that allows many signatures for many messages signed by many keys to be compressed into a single signature, which can be verified against the entire set of (message, pubkey) pairs that it represents simultaneously.

That is, if there are:

- A set of private keys k_1, k_2 ... k_n

(held by different users)

- The corresponding pubkeys K_1, K_2 ... K_n
- Messages m_1, m_2 ... m_n

, where m_i

is signed by the corredponding k_i

and the signatures are S_i = k_i * H(K_i, m_i)

Then you can make an aggregate signature S = S_1 + S_2 + ... + S_n

, where S

has a fixed size (usually 32-96 bytes depending on configuration), and S

can be verified against the entire set of pairs [(K_1, m_1), (K_2, m_2) ... (K_n, m_n)]

(the messages and the public

keys), confirming that S

is a valid aggregate of signatures for those key and message combinations.

The challenge when deploying this in a ethereum context, however, is that signature aggregation is at its most powerful when we aggregate many signatures together, and this implies somehow aggregating together signatures across many transactions within a block. Unfortunately, the EVM is not suited to this, because each EVM execution's scope is limited to being within a particular transaction.

I propose a simple extension to the EVM to alleviate this difficulty. We add a new opcode, EXPECT_BLS_SIGNATURE

, which takes two arguments off the stack: (i) a memory slice representing the pubkey (represented on the stack by the starting position in memory, as we know how many bytes a pubkey contains), and (ii) the message (standardized to be a 32 byte hash). The block verification procedure processes this opcode by adding the pair (pubkey, message)

to an expected_signatures

list (this list starts off empty at the start of every block). We also add to the block header a field bls_aggregate

. At the end of block processing, we take the whole expected_signatures =

[(K_1, m_1), (K_2, m_2) ... (K_n, m_n)]

list, and perform the BLS aggregate signature check:

e(K_1, H(K_1, m_1)) * e(K_2, H(K_2, m_2)) * ... * e(K_n, H(K_n, m_n)) ?= e(S, G_2)

Where S

is the bls_aggregate

and G_2

is the elliptic curve generator. If the check does not pass, the entire block is invalid.

At network layer, we add a wrapper to the Transaction

class, where each transaction can come with a signature that covers the expected signatures during execution. A block proposer (ie. miner) would receive the transaction, attempt to verify it by running the transaction, computing the expected_signatures

array just for that transaction

, and seeing if the signature provided with the transaction matches that array. If it does, and the transaction passes the other usual validity and fee sufficiency checks, the block proposer incldues the transaction, and adds (using elliptic curve addition) the transaction's provided BLS signature to the block's bls_aggregate

; if it does not, then the block proposer ignores the transaction.

Note that this breaks the invariant that a transaction cannot be made to be invalid

as a result of things that happen during execution (which is important toprevent DoS attacks). Hence, if deployed on the base layer, it should be combined with account abstraction, with the EXPECT_BLS_SIGNATURE

opcode only usable before the PAYGAS

opcode is called.