The goal of “account abstraction”, as conceived by EIP 86 and other proposals, is to reduce the number of account types from 2 (EOA and contract) to 1 (just contract), and to move functionality such as signature verification, gas payment and replay protection out of the core protocol and into the EVM. However, there are costs to doing this, including development time, transition costs, costs of storing extra code on the blockchain, breaking existing invariants and other issues. Many of these costs are much lower in the context of the current sharding roadmap where the shards can start from scratch and there is no need to upgrade any existing accounts, but some still remain.

Here are some of the possibilities, and their pros and cons.

Lazy full abstraction

- How it works

: the only type of account is a contract. There is one transaction type, which has the following fields: [gas, addr, data]. Executing the transaction consists of playing a message, with msg.sender

being some standard “ENTRY_POINT” address (eg. 0xff…ff), msg.to

being the addr

, and the gas and data being the provided values. Users are expected to store funds in contract accounts, where the code of the contract interprets the provided data as an ABI encoding of [nonce, signature, gasprice, value, data], verifies the nonce and signature, pays gas to the miner, sends a message to the desired address with the desired value, and then asks for a refund for the remaining gas.

- Pros

: makes the protocol very

simple. apply_tx

becomes a very trivial wrapper around apply_msg

.

- Cons

: requires fairly complex code inside of each account to verify the nonce and signature and pay gas. Second, requires fairly complex code in the miner to determine what transactions actually are guaranteed to pay for gas. Third, it requires additional logic for the sender and the miner to create

new accounts. Finally, it introduces the possibility that, with accounts constructed in a “non-standard” way, a transaction with the same hash could get included multiple times.

The miner’s problem is as follows. The miner needs to verify, in O(1) time, that a given transaction actually is guaranteed to pay for gas if the miner decides to process the transaction and try to include it in the block. In an abstraction system, this could involve asking the miner to run some code, say, with a limit of 200000 gas, but the miner would need to be sure that, after this happens, the transaction execution is in a state where the gas is paid for, and the payment cannot be reverted. Currently, the protocol handles this automatically; in full abstraction, this must all be implemented in code, and possibly in a fairly complex way.

Remove nonce abstraction

- How it works

: same as above, except a transaction also has a nonce

field. A rule is added that a transaction’s nonce must equal the account nonce, and that the nonce is incremented upon a successful transaction.

- Pros

: removes the possibility of a transaction appearing in multiple places.

- Cons

: increases base protocol complexity slightly, and remove the possibility of alternative schemes (eg. UTXOs, parallelizable nonces)

Standardize signature scheme

- How it works

: add a byte-array field sig

to the transaction. In the top-level message, add to the transaction's return data sighash ++ sig

, where sighash

is the sha3 of the transaction not including the sig.

- Pros

: makes signature verification much simpler.

- Cons

: increases base protocol complexity slightly.

Add BREAKPOINT opcode

- How it works

: add an opcode BREAKPOINT

, which has the property that if a transaction throws after a breakpoint, it only reverts up to the breakpoint.

- Pros

: makes it much easier for the miner to detect if a transaction pays for gas; the miner's code would only need to be something like "run for N steps or until a breakpoint, see that the breakpoint has been reached, and that gas has been paid for".

- Cons

: deep and fundamental change to the EVM. Historically not the best idea.

Add PAYGAS opcode

- How it works

: takes as input one argument (gasprice), immediately transfers the gasprice * tx.gas to a temporary account, and then at the end of executing the transaction refunds any unused gas.

- Pros

: makes paying for gas simpler, and particularly does not require the transaction to include merkle branches to process a call to the coinbase. Avoids the overhead of a call to the coinbase.

- Cons

: increases base protocol complexity. Also does not allow abstracting gas payment, eg. paying with ERC20 tokens.

Gasprice + PANIC

- How it works

: a transaction adds a parameter gasprice

. At the start of execution, gasprice * startgas

is subtracted. A PANIC opcode is added, which can only be called in a top-level execution context (ie. if msg.sender == ENTRY_POINT

) and where (tx.gas - msg.gas) <= PANICLIMIT

(eg. PANICLIMIT = 200000). If this opcode is triggered, then the entire transaction is invalid. A user account would make sure to check the signature and nonce within the limit, preventing invalid transactions from consuming gas. Miners would run transactions with a sufficient gasprice and reject those that panic.

- Pros

: account code is simple, and miner code is simple, while still adding full signature and nonce abstraction

- Cons

: increases base protocol complexity slightly. Also does not allow abstracting gas payment, eg. paying with ERC20 tokens. Third, does not provide flexibility in how much time signature verification can take (though Casper already enforces a limit, so the limit can be set to be the same value).

One possible variant is to simply make the transaction invalidity behavior be part of the behavior of THROW if called in a top-level context with the given amount of remaining gas.

Combine PANIC and PAYGAS

- How it works

: remove PANIC. Instead, have all exceptions have behavior equivalent to PANIC if PAYGAS has not yet been called.

- Pros

: allows accounts to set their own base verification gas limit. A miner can run the algorithm of running the code for up to N gas, where N is chosen by the miner, until PAYGAS has been called. Also, removes the need for gasprice to be part of the transaction body.

- Cons

: makes the output state of a message slightly more complicated, as it also needs to carry the information of whether or not a PAYGAS opcode was triggered and if so with what gasprice.

Salt + code in transaction

- How it works

: a transaction has two new fields: salt

and code

. If the target account of a transaction is nonempty, then these two fields must be empty [variant: are ignored]. If the target account is empty, then the last 20 bytes of sha3(salt + code) must equal the account, and if this is the case then the code is put into that position in the account code [variant: is used as init code].

- Pros

: creates a standard and clean way to create new accounts.

- Cons

: adds protocol complexity.

Newly created account pays

- How it works

: a transaction can be a contract creation transaction by specifying a salt and code. If the target address is empty, then it takes funds from that address to pay for gas, and then creates the contract.

- Pros

: similar to existing contract creation.

- Cons

: sending the first transaction from an account takes an additional step.

Conclusion

Currently, I favor the gasprice + PANIC approach, including both variants. But there may also be other ways to go.