

Proposed initial abstraction changes for Metropolis #86

[New issue](#)

vbuterin opened this issue on Apr 1, 2016 · 42 comments



vbuterin commented on Apr 1, 2016 · edited ▾

Collaborator

This document is outdated; see [#208](#) for current details.

Specification

If `block.number >= METROPOLIS_FORK_BLKNUM`, then:

1. If the signature of a transaction is $(0, 0, 0)$ (ie. $v = r = s = 0$), then treat it as valid and set the sender address to $2^{160} - 1$
2. Set the address of any contract created through a creation transaction to equal $\text{sha3}(0 + \text{init code}) \ \% \ 2^{160}$, where $+$ represents concatenation, replacing the earlier address formula of $\text{sha3}(\text{rlp.encode}([\text{sender}, \text{nonce}]))$
3. Create a new opcode at `0xfb`, `CREATE_P2SH`, which sets the creation address to $\text{sha3}(\text{sender} + \text{init code}) \ \% \ 2^{160}$. If a contract at that address already exists, fails and returns 0 as if the init code had run out of gas.

Rationale

The goal of these changes is to set the stage for abstraction of account security. Instead of having an in-protocol mechanism where ECDSA and the default nonce scheme are enshrined as the only "standard" way to secure an account, we take initial steps toward a model where in the long term all accounts are contracts, contracts can pay for gas, and users are free to define their own security model.

Under EIP 86, we can expect users to store their ether in contracts, whose code might look like the following (example in Serpent):

```
# Get signature from tx data
sig_v = ~calldataload(0)
sig_r = ~calldataload(32)
sig_s = ~calldataload(64)
# Get tx arguments
tx_nonce = ~calldataload(96)
tx_to = ~calldataload(128)
tx_value = ~calldataload(160)
tx_gasprice = ~calldataload(192)
tx_data = string(~calldatasize() - 224)
~calldataload(tx_data, 224, ~calldatasize())
# Get signing hash
signing_data = string(~calldatasize() - 64)
~mstore(signing_data, tx.startgas)
~calldataload(signing_data + 32, 96, ~calldatasize() - 96)
signing_hash = sha3(signing_data:str)
# Perform usual checks
prev_nonce = ~sload(-1)
assert tx_nonce == prev_nonce + 1
assert self.balance >= tx_value + tx_gasprice * tx.startgas
assert ~ecrecover(signing_hash, sig_v, sig_r, sig_s) == <pubkey hash here>
# Update nonce
~sstore(-1, prev_nonce + 1)
```

Assignees

No one assigned

Labels

[editor-needs-to-review](#)

Projects

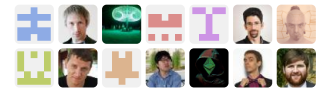
None yet

Milestone

No milestone

Notifications

14 participants



```
# Pay for gas
~send(MINER_CONTRACT, tx_gasprice * tx.startgas)
# Make the main call
~call(msg.gas - 50000, tx_to, tx_value, tx_data, len(tx_data), 0, 0)
# Get remaining gas payments back
~call(20000, MINER_CONTRACT, 0, [msg.gas], 32, 0, 0)
```

This can be thought of as a "forwarding contract". It accepts data from the "entry point" address $2^{160} - 1$ (an account that anyone can send transactions from), expecting that data to be in the format `[sig, nonce, to, value, gasprice, data]`. The forwarding contract verifies the signature, and if the signature is correct it sets up a payment to the miner and then sends a call to the desired address with the provided value and data.

The benefits that this provides lie in the most interesting cases:

- **Multisig wallets:** currently, sending from a multisig wallet requires each operation to be ratified by the participants, and each ratification is a transaction. This could be simplified by having one ratification transaction include signatures from the other participants, but even still it introduces complexity because the participants' accounts all need to be stocked up with ETH. With this EIP, it will be possible to just have the contract store the ETH, send a transaction containing all signatures to the contract directly, and the contract can pay the fees.
- **Ring signature mixers:** the way that ring signature mixers work is that N individuals send 1 coin into a contract, and then use a linkable ring signature to withdraw 1 coin later on. The linkable ring signature ensures that the withdrawal transaction cannot be linked to the deposit, but if someone attempts to withdraw twice then those two signatures can be linked and the second one prevented. However, currently there is a privacy risk: to withdraw, you need to have coins to pay for gas, and if these coins are not properly mixed then you risk compromising your privacy. With this EIP, you can pay for gas straight out of your withdrawn coins.
- **Custom cryptography:** users can upgrade to ed25519 signatures, Lamport hash ladder signatures or whatever other scheme they want on their own terms; they do not need to stick with ECDSA.
- **Non-cryptographic modifications:** users can require transactions to have expiry times (this being standard would allow old empty/dust accounts to be flushed from the state securely), use k -parallelizable nonces (a scheme that allows transactions to be confirmed slightly out-of-order, reducing inter-transaction dependence), or make other modifications.

(2) and (3) introduce a feature similar to bitcoin's P2SH, allowing users to send funds to addresses that provably map to only one particular piece of code. Something like this is crucial in the long term because, in a world where all accounts are contracts, we need to preserve the ability to send to an account before that account exists on-chain, as that's a basic functionality that exists in all blockchain protocols today.

Miner strategy

Note that miners would need to have a strategy for accepting these transactions. This strategy would need to be very discriminating, because otherwise they run the risk of accepting transactions that do not pay them any fees, and possibly even transactions that have no effect (eg. because the transaction was already included and so the nonce is no longer current). One simple approach is to have a whitelist for the codehash of accounts that they accept transactions being sent to; approved code would include logic that pays miners transaction fees. However, this is arguably too restrictive; a looser but still effective strategy would be to accept any code that fits the same general format as the above, consuming only a limited amount of gas to perform nonce and signature checks and having a guarantee that transaction fees will be paid to the miner. Another strategy is to, alongside other approaches, try to process any transaction that asks for less than 250,000 gas, and include it only if the miner's balance is appropriately higher after executing the transaction than before it.

See

https://www.reddit.com/r/ethereum/comments/5ab69v/metropolis_protocol_change_proposal_highlight_for/ for reddit discussion.

👍 7 | 🎨 3



gavofyork commented on Apr 2, 2016

mixhash and nonce are all substituted with the empty string
unlike for the powhash where they are not present at all :-/



gavofyork commented on Apr 2, 2016

1/2/3 mean state trie will grow indefinitely large (so far there would be an additional ~50MB on it, a substantial increase on the pruned, compressed state db)



gavofyork commented on Apr 2, 2016

key , mstart , msz -> key , mstart , msize or ky , mst , msz . don't mix naming conventions.

👍 1



eth1au commented on Apr 2, 2016

Solid. V, can you show Gav the DK Gas email. JG



gavofyork commented on Apr 2, 2016

for 9, what happens when there's a collision?



gavofyork commented on Apr 2, 2016

it should also be noted that (12) and (13) are *suggested* miner alterations, but do not form part of the consensus protocol.



eth1au commented on Apr 2, 2016

Perhaps. Alas, essential weekend reading:
http://www.mit.edu/~mrognlie/piketty_diminishing_returns.pdf



vbuterin commented on Apr 2, 2016

Collaborator

Fixed the naming conventions for msz.

1/2/3 mean state trie will grow indefinitely large (so far there would be an additional ~50MB on it, a substantial increase on the pruned, compressed state db)

However, note that this growth is constant in tx load; right now, it's 20% of the state size, but in a future environment where we see 5-10 tx/sec, it will be much smaller, and we should optimize for that case. Additionally, if we are still uncomfortable with storing ALL state roots, then there is a compromise solution where we store the state roots at key `block.number % CYCLE_LENGTH` (eg. `CYCLE_LENGTH = 65536`), and in the `STATEROOT` and `BLOCKHASH` opcodes keep the condition that we return 0 if `block.number - target_height >= CYCLE_LENGTH`

for 9, what happens when there's a collision?

I would say failure (either do nothing and return zero, or exception). We could remove the issue entirely by adding a nonce back in to the formula, but I want to avoid this because one of my goals with EIP 101 is the desire to remove the notion of in-protocol sequence numbers entirely, moving all of that logic to the protocol level. So if you want to create multiple instances of the same contract, you can just append a nonce after the code, and maintain and increment the nonce yourself in storage; this can all be done with a few EVM operations, and simplifies the protocol further as we don't have to deal with the current logic around when nonces are and aren't incremented within consensus code.



vbuterin commented on Apr 2, 2016

Collaborator

Changed the hash with state root/mixhash/nonce as empty strings into the standard PoW hash, but with the state root replaced with the empty string. Added rationale to post; here it is copied for convenience:

The state root needs to be set to the empty string because (2) is itself a state change, so the actual final state root is not known (and neither are the mixhash and nonce).



chriseth commented on Apr 5, 2016

Contributor

What happens if a slot is SLOAD'ed from that has been SSTOREBYTES'ed before?



vbuterin commented on Apr 5, 2016

Collaborator

Last 32 bytes, left-zero-padded.



chriseth commented on Apr 6, 2016

Contributor

@vbuterin Still many open questions regarding the variable-size slots. Is the general idea that `SSTORE` behaves identical to `SSTOREBYTES` with 32 bytes and `SLOAD` identical to `SLOADBYTES` plus padding? (I would prefer padding at the less-significant end, by the way).

So using `SSIZE` for not yet used slots returns zero, but it returns 32 for slots where we wrote `0`. This is still a bit inconsistent with how solidity uses storage, but it might work.

Furthermore, I am not sure that the memory exceptions for `SLOADBYTES` are a good idea (different case than for `CALL`) because we do know the size.

Also it could be considered whether `SLOADBYTES` should zero-pad to multiples of 32 - that would be more consistent with our ABI.



vbuterin commented on Apr 8, 2016

Collaborator

Is the general idea that SSTORE behaves identical to SSTOREBYTES with 32 bytes and SLOAD identical to SLOADBYTES plus padding?

Yes, with the exception that you can use SSTOREBYTES and not SSTORE to save 32 zero bytes, as you brought up.

Furthermore, I am not sure that the memory exceptions for SLOADBYTES are a good idea (different case than for CALL) because we do know the size.

True. I'm fine either way.

Also it could be considered whether SLOADBYTES should zero-pad to multiples of 32 - that would be more consistent with our ABI.

I would prefer no. It's not consistent with the ABI, but the ABI is a high-level construct. Return slices of CALL and the like are not zero-padded to multiples of 32, and it is consistent with that.



chriseth commented on Apr 20, 2016

Contributor

12: I think we should still perform some static analysis on the target contract, it is easy to spam miners with a lot of txes with small startgas.



chriseth commented on Apr 20, 2016

Contributor


General concern: If we move nonce checking into the contract, to we still enforce the requirement that the chain should never include two transactions with the same hash?



chriseth commented on Apr 20, 2016 • edited ▼

Contributor

9: I would guess that none of the existing contracts relies on the actual way the new address is calculated in a `create`, so changing that is fine. What I'm worried about is that previously, two `create` calls with identical code from the same contract did not result in a collision but now they do, because existing code does not add a nonce to the init code.

★  janx referenced this issue on Apr 20, 2016

Enable cost differences between SSTORE8-248 #93

 Closed



vbuterin commented on Apr 21, 2016 • edited ▼

Collaborator

to we still enforce the requirement that the chain should never include two transactions with the same hash?

I would say no. One of the key positive side effects of an eventual universal adoption of EIP 101 is that transaction *validity* becomes statically evaluable, which means that blocks can be validated independently of having the prior state.

What I'm worried about is that previously, two `create` calls with identical code from the same contract did not result in a collision but now they do, because existing code does not add a nonce to the init code.

True. If we decide that this is a problem, one simple solution would be that if the destination address contains code, we keep incrementing the address by 1 (wrapping around if needed) until it doesn't, and run the code at the first address we find that is empty.

- ★

This was referenced on Apr 26, 2016

Blockhash and state root changes #96

SSTORE/SLOAD for byte arrays #97

🔓 Open

🔓 Open



Smithgift commented on May 4, 2016

👍 on incrementing address on collision. In addition (ha) to all current contract-creating contracts continuing (ha ha) to function, it permanently resolves the question of two contracts accidentally having the same address.

- ★

This was referenced on Jun 26, 2016

Add multisig transactions. #123

Who pays withdraw transaction fee? zcash-hackworks/babyzoe#1

🔒 Closed

🔓 Open



tromer commented on Jul 29, 2016

Letting contracts pay fees is also needed by Zerocash over Ethereum ([zcash/babyzoe#1](#)).

👍 4

- ★

This was referenced on Aug 30, 2016

Creator Contract abstraction #143

RFC: Cross Blockchain Transaction Replay Protection without IAN #127

EIP/ERC process improvement #148

Scratch space #167

Control Protocol Parameters with on-chain governance #182

ERC: Signed Data Standard #191

EIP-98 Remove medstate from receipts paritytech/parity#4264

AllCoreDevs Meeting 9 Agenda ethereum/pm#3

Metropolis Plan Discussion ethereum/pm#4

🔓 Open

🔒 Closed

🔒 Closed

🔓 Open

🔓 Open

🔓 Open

🔒 Closed

🔒 Closed

🔒 Closed



kumavis commented on Jan 27 • edited

Member

here is a rough sketch of an alternate proposal for the tx verification abstraction

If `block.number >= METROPOLIS_FORK_BLKNUM`
and
If the signature of a transaction is `(0, 0, 0)` (ie. `v = r = s = 0`)

send a message to the tx `toAddress` with sender address and origin as `2**160 - 1`

And require that the tx `toAddress` emits a log with topic `ACCEPT_TX` within `TX_VERIFY_GASLIMIT` gas consumed. If the message exits early or the log is not emitted within the limit, it is invalid. If it is emitted, the tx should continue to be processed and all fees from start to finish are paid by the identity contract.

This proposal could be adjusted to use a different signaling mechanism than the log, but it seemed appropriate because verification code continues onto processing code, unifying the gas count for validating and processing the tx.

I would appreciate some criticism @vbuterin @chriseth @gavofyork



cdetrio commented on Feb 1

Member

Would discussion be more organized if 2 (contracts are created at address $\text{sha3}(0 + \text{init code}) \% 2^{160}$) and 3 (new opcode `CREATE_P2SH`) were opened as a separate EIP? The sub-proposals (2) and (3) have no bearing on the account abstraction proposal of (1).



1



cdetrio commented on Feb 1

Member

1. If the signature of a transaction is $(0, 0, 0)$ (ie. $v = r = s = 0$), then treat it as valid and set the sender address to $2^{160} - 1$

Would the nonce for account $2^{160} - 1$ be incremented as done with regular accounts? Or would it have special behavior?

Aren't "forwarding contracts" currently possible without any changes to the transaction protocol? Senders can just use a gas price of zero, and since miner strategy is flexible and left unspecified, its up to miners to choose which zero gas price transactions to include by somehow screening for contracts that send a payment to `COINBASE`.

The only advantage I can see to a special $2^{160} - 1$ account is the possibility for flexible nonce schemes, but that can also be achieved by signing zero-gasprice zero-value transactions from random empty accounts (the empty account bloat that might result could be mitigated by specifying that transactions from empty accounts do not result in a nonce increment).

It seems that the miner strategy is the most important factor to realizing the intended functionality (that contracts can pay for gas). It is not clear how having special behavior for the $2^{160} - 1$ account would help miners discriminate zero-gasprice transactions for inclusion without opening up potential DoS/spam vectors.



1



Nashatyrev commented on Feb 2

Member

@vbuterin Is it up-to-date EIP proposal in the first comment? There are some questions on it.

1. How $(0, 0, 0)$ transactions are supposed to be propagated by peers? Now a peer needs to validate tx and include it to the pending state before propagating it to other peers, how $(0, 0, 0)$ could be validated? Is it propagated without validation? Couldn't the network be DoSed with those transactions for free in that case?
Is #155 applicable to those transactions?
2. Couldn't the same contract be deployed twice as it will obtain the same address? Should the init code be changed for this?



kumavis commented on Feb 2

Member

@cdetrio one minor difference might be that `tx.origin` might be set to the proxy account



axic commented on Feb 6

Member

Here is an implementation of the standard account in Solidity (done a few months back, but forgot to post it here): <https://gist.github.com/axic/528017d2d67801fa669fd75577c2093c>

★ Souptacular referenced this issue in `ethereum/pm` on Feb 6

All Core Devs Meeting 10 Agenda #5

🔒 Closed



chriseth commented on Feb 7

Contributor

@Nashatyrev I guess the propagation rules are the same as the miner strategy. For now, nodes run a simple regular expression on the target contract code and propagate the tx if it matches (and there is enough ether in the contract to pay for the gas).

★ chriseth referenced this issue in `ethereum/cpp-ethereum` on Feb 7

Account abstraction - EIP 86 #3546

🔓 Open

📋 2 of 2 tasks complete



Nashatyrev commented on Feb 8

Member

@chriseth Thanks for clarification!
Still couldn't get how 'regexp' can determine if the concrete contract is going to pay for the concrete Tx? M.b. you can give some simple example?
In reddit discussion I saw the proposal to execute this Tx with some very limited gas to check if the contract would pay within this gas bound. This sounds more realistic to me.



chriseth commented on Feb 8

Contributor

@Nashatyrev the EIP description contains an example contract. If you replace the address by a wildcard, you have the regular expression.



1



vbuterin commented on Feb 10

Collaborator

How (0,0,0) transactions are supposed to be propagated by peers?

Basically, peers would follow the same heuristics as miners. That is, they would have a set of whitelist rules and only propagate txs if they match at least one of them. Some whitelist rules might include:

- Transactions whose to address fits a particular regexp, and whose data includes a sufficiently high fee
- Transactions which consume less than 200000 gas, and which when executed pay a sufficiently large fee to a miner

The general rule is "propagate the tx if you can make an argument that there is at least some probability p that it will be included in the next block and lead to the sender paying the miner a sufficiently high fee".

Couldn't the same contract be deployed twice as it will obtain the same address? Should the init code be changed for this?

In general, *user contracts* should not be an issue because user contracts would include the user's public key somewhere in the code, and so be unique per user. Contracts could be prevented from being deployed twice; if you want to create multiple instances, then the recommendation is to add an incrementing nonce to the end of the code as a dummy variable.



vbuterin commented on Feb 10

Collaborator

Would discussion be more organized if 2 (contracts are created at address $\text{sha3}(0 + \text{init code}) \% 2^{160}$) and 3 (new opcode CREATE_P2SH) were opened as a separate EIP? The sub-proposals (2) and (3) have no bearing on the account abstraction proposal of (1).

@cdetrio possible, though they are highly complementary. You can't easily use EIP 86 accounts if there is not a way to securely send money to them before they exist, which is what (2) and (3) do.

Would the nonce for account $2^{160} - 1$ be incremented as done with regular accounts? Or would it have special behavior?

Don't increment it IMO.

Senders can just use a gas price of zero, and since miner strategy is flexible and left unspecified, it's up to miners to choose which zero gas price transactions to include by somehow screening for contracts that send a payment to COINBASE.

The problem is that if you have a forwarding contract without EIP 86, then you are wasting 96 bytes and a signature recovery operation.



vbuterin commented on Feb 10 • edited ▼

Collaborator

And require that the tx toAddress emits a log with topic ACCEPT_TX within TX_VERIFY_GASLIMIT gas consumed. If the message exits early or the log is not emitted within the limit, it is invalid. If it is emitted, the tx should continue to be processed and all fees from start to finish are paid by the identity contract.

That's doable, but imo more cumbersome to implement. Also, specifying a specific log topic and a specific verification gas limit seems cumbersome. All of this logic could instead be implemented by the account contracts themselves. Also, keeping fee payment in-protocol makes it more difficult to later turn ether into an ERC20 token.



Nashatyrev commented on Feb 10

Member

@vbuterin Thanks, that became more clear now.

What mechanism is supposed to be used for distributing those whitelist rules among peers/miners? Will it be the form of foundation recommendations which are configured via the client config (like gas limit)? Or will they be a part of current and future forks?

Souptacular added the `editor-needs-to-review` label on Feb 10

pirapira referenced this issue in [ethereum/yellowpaper](#) on Feb 10

Byzantium changes #229

Open

 12 of 12 tasks complete

Smithgift commented on Feb 10

I would like it if, rather than a contract's address being at `sha3(<0 or sender> + initcode)`, it was actually `sha3(<0 or sender> + sha3(initcode))`. Reason: It's far more gas efficient for a contract to verify another contract has a given initcode if it can just hash the hash, rather than having to hash the entire initcode every time.

 This was referenced on Feb 13**[WIP] EIP 86: initial abstraction changes** ethereum/yellowpaper#231 Open**[consensus uncertain] EIP86: initial account abstraction**
ethereum/yellowpaper#249 Closed

pirapira commented on Mar 1

Member

#208 seems to be the new version.

 This was referenced on Mar 21**New transaction formats #232** Open**consensus, core/*, params: metropolis preparation refactor** ethereum/go-ethereum#14336 Merged**Abstraction of transaction origin and signature.md #208** Open

Dexaran commented on Apr 29

As I understand there will be a special contract at `sha3(sender + init code) % 2**160` address. This special contract will be linked to the account. A special contract can be called when a transaction occurs to an account (the address associated with that contract). Is it right?
It seems like I'm missing the point.



Smithgift commented on Apr 30

@Dexaran: The point is that it's much safer to say "Pay this uncreated contract's address's X ether" when that address is stable. Currently doing so requires never accidentally using the right nonce.



Dexaran commented on Apr 30 • edited ▼

I'm asking about realization details. I need to adapt ERC #223 implementation to Metropolis. I need to know how should token transaction work with Externally Owned Account(EOA) in Metropolis.

I'm now assembling receivers code size to know if receiver is a contract. If receiver is contract (code size > 0) `tokenFallback` function should be executed. If the receiver is an EOA, the `tokenFallback` function should not be executed.

I need to know:

1. Will each EOA be assigned with a special contract that must be executed for each transaction in the EOA?

2. Can the EOA-assigned contract contain the `tokenFallback` function, which should be executed?

I'm ready to implement any useful suggestion about adaptation of ERC #223 to Metropolis.



Smithgift commented on Apr 30

Ah. As far as I know, this version does not *force* externally owned accounts to be contracts, but that will eventually become the case (in future hardforks).

When that occurs, what's in each contract will be up to the users. So they might have a given function implemented, or they might not.



Dexaran commented on Apr 30

Thank you for explanation @Smithgift

Now my questions are:

1. How can I check whether the recipient is a contract or an EOA with a bytecode in Metropolis?
2. How can I check if the `tokenFallback` function is implemented by the recipient? Is there a way to check if the contract implements a certain function?



Smithgift commented on May 1

1. I don't think there's any theoretical difference between a contract and an EOA-with-bytecode. You might be able to examine the code to see if it matches common patterns for an EOA.
2. I think you might be able to examine the dispatcher of a contract to see if a given function with a given signature is present. It's a little beyond me exactly how to do that, though. The Ethereum stackexchange or solidity gitter might be of more help.



Dexaran commented on May 1 • edited ▼

There is a great difference between contract and EOA-with-bytecode for ERC #223 token transaction:

If the receiver is an EOA with the implementation of `tokenFallback`, then `tokenFallback` must be executed.

If receiver is an EOA without `tokenFallback` implementation, then token transaction must submit without a try to execute `tokenFallback` (a try to execute `tokenFallback` will fail a transaction in this case).

If the receiver is a contract, `tokenFallback` must be executed without distinctions whether `tokenFallback` is implemented or not. (if there is no `tokenFallback` implementation on this receiver contract then transaction must fail)



MicahZoltu commented on May 12

Is this still slated for Metropolis? Is "mid-summer" still looking reasonably likely for Metropolis?

I ask because I have a need to create a proxy-contract with a non-trivial ownership pattern and I'm wondering if I should try to create it now, and deal with all of the UI complexities that come with trying to interact with dApps through a proxy contract (I believe at the moment this is basically impossible if I want to use their UI) or if I should wait for Metropolis, which it sounds like will allow for this without having to write a custom UI for every dApp out there.

I believe uPort also has this problem, where if you want to use a dApp with uPort the dApp needs to be built to support uPort or you need to use uPort as your transaction processor (instead of a local node) so they can handle all of the complexities server-side with wrapping up your transaction calls and piping them through their proxy.

★ This was referenced on May 24

- change of tx.origin for account abstraction #637

🔓 Open
- Feature/146382715 metatx controller uport-project/uport-identity#38

🔒 Closed
- Metropolis aggregate PR ethereum/go-ethereum#14726

🔒 Closed
- On EIP 86 and EIP 648: account abstraction, DoS vectors, and parallelizability #678

🔓 Open
- Prevent overwriting contracts #684

🔓 Open
- ECIP-? Atomic Transactions ethereumproject/ECIPs#40

🔓 Open
- ERC223 token standard #223

🔓 Open