Preamble

```
EIP: <to be assigned>
Title: Abstraction of transaction origin and signature
Author: Vitalik Buterin
Type: Standard Track
Category: Core
Status: Draft
Created: 2017-02-10
```

Summary

Implements a set of changes that serve the combined purpose of "abstracting out" signature verification and nonce checking, allowing users to create "account contracts" that perform any desired signature/nonce checks instead of using the mechanism that is currently hard-coded into transaction processing.

Parameters

- METROPOLIS_FORK_BLKNUM: TBD
- CHAIN_ID: same as used for EIP 155 (ie. 1 for mainnet, 3 for testnet)
- NULL_SENDER: 2**160 - 1

Specification

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

1. If the signature of a transaction is `(CHAIN_ID, 0, 0)` (ie. `r = s = 0`, `v = CHAIN_ID`), then treat it as valid and set the sender address to `NULL_SENDER`
2. Transactions of this form MUST have gasprice = 0, nonce = 0, value = 0, and do NOT increment the nonce of account NULL_SENDER.
3. Create a new opcode at `0xfb`, `CREATE2`, with 4 stack arguments (value, salt, mem_start, mem_size) which sets the creation address to `sha3(sender + salt + sha3(init code)) % 2**160`, where `salt` is always represented as a 32-byte value.
4. Add to *all* contract creation operations, including transactions and opcodes, the rule that if a contract at that address already exists and has non-empty code OR non-empty nonce, the operation fails and returns 0 as if the init code had run out of gas. If an account has empty code and nonce but nonempty balance, the creation operation may still succeed.

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)
# 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 our 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 and transaction replaying 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 strategy is to have a set of regexps that the to address of an account would be checked against, each regexp corresponding to a "standard account type" which is known to be "safe" (in the sense that if an account has that code, and a particular check involving the account balances, account storage and transaction data passes, then if the transaction is included in a block the miner will get paid), and mine and relay transactions that pass these checks.

One example would be to check as follows:

1. Check that the to address has code which is the compiled version of the Serpent code above, with `<pubkey hash here>` replaced with any public key hash.
2. Check that the signature in the transaction data verifies with that key hash.
3. Check that the gasprice in the transaction data is sufficiently high
4. Check that the nonce in the state matches the nonce in the transaction data
5. Check that there is enough ether in the account to pay for the fee

If all five checks pass, relay and/or mine the transaction.

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.