A Non-Hard-Fork Approach to Ethereum Precompiled Contracts

Wei Tang

September 29, 2017

Abstract

Ethereum's precompiled contacts create "shortcuts" in the Ethereum Virtual Machine to speed up common computations like elliptic curve and SHA256. In the past, adding a new precompiled contract requires a hard-fork of the whole network. In this essay, we show that there is a way to adding new precompiled contracts without needing a hard-fork, and why this approach is safe.

1 Introduction

In the Ethereum blockchain [1] category, precompiled contracts are native code acting as "shortcuts" in Ethereum Virtual Machine (EVM) – when you issue a CALL-like instruction to any of the precompiled contract address, instead of trying to evaluate the actual code in that address, the EVM will execute a pre-defined native codes and return the results. Those precompiled contracts adds additional functionality such as elliptic curve and SHA256 that allows various interesting features.

In the past, adding new precompiled contracts require a hard fork of the network. In Ethereum's Metropolis upgrade, the zkSNARK precompiled contract, which adds privacy features to the blockchain, requires a new hard fork. A hard fork, however, requires consensus from the whole network, and if carried out incorrectly, can result in a community split.

We show here that you can add new precompiled contracts without requiring a hard fork. This can be done by a modification of the client to include a special rule for including transactions in a block called Gas Price Reduction. To safely detect whether a Gas Price Reduction is valid, one call use the way described in this essay to static call a contract.

2 Static Call

A static call to a contract is a series of opcodes that allow the analyzer to know its callee address, the input size, and the gas limit without executing the code, defined as the series below:

```
PUSH20 <contract address>
PUSH<n> <input size>
SWAP5
```

```
PUSH<n> <gas>
CALL
```

The `CALL` instruction takes 7 items from the stack, they are gas, to, value, input offset, input size, output offset and output size. In the above, we are only interested in 3 of them and it is expected that previous opcodes push all other parameters needed for this contract call.

We make an assumption here that the gas used by a precompiled contract is only dependent on its gas limit and input size. All the existing precompiled contracts, `ECREC`, `SHA256`, `RIP160` and `ID` follow this scheme.

3 Gas Reduction

Precompiled contracts execute native code in the EVM, and thus the computation becomes cheaper. We can consider this as the gas G is reduced by an amount R dependent on its gas limit L and input size I.

$$R = \begin{cases} F(I), & \text{if } L \geq L_{constant} \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

For a precompiled contract, if the gas limit provided is not enough, then the contract fails and thus no gas reduction is applied. Otherwise, we apply a gas reduction based on the input size.

3.1 Nested Gas Reduction

Gas Reduction can be nested, in this case, for a normal contract that the parent contract calls, it executes a precompiled contract. The contract that the parent contract calls must also be from a static call. Thus its input size and gas limit is also known.

In that case, we have.

$$R_{all} = \begin{cases} sum(R_{child}), & \text{if } L \geq sum(R_{child}) \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

4 Gas Price Reduction

Gas is a fixed number in consensus, thus it cannot actually be reduced. We can instead reduce the gas price to archive the same effect.

$$R_{gas} = T_{gas} - sum_{<contract>}(N_{<contract>}) \tag{3}$$

$$R_{price} = \frac{T_{gas} * N_{price}}{R_{gas}} \tag{4}$$

Where T_{gas} is the actual gas limit specified in the transaction, R_{gas} is the reduced gas, R_{price} is the reduced gas price, and N_{price} is the normal gas price.

5 Reduction Proof

Given a transaction, we need to validate that the gas price reduction (if any) embedded within is valid. EVM is turing complete, so instead of validating directly on the client, we can delegate the resposibility to the transaction signer. In the networking layer, this makes it so that when a transaction with gas price reduction is broadcasted, it will also need to broadcast a supplimental piece of information in the form of a gas price reduction proof.

The proof is a list of static call stacks with its code locations. Each call stack, specifically, is represented as:

```
[<pc_loc> <pc_loc> <pc_loc> ...]
```

The last item of the call stack must be a CALL to a non-HF precompiled contract. The client then simply follows this list of proof to check that a non-HF precompiled contract must have been called for at least N times to compensate for its gas price reduction. This check is done in linear time.

The client first goes to the code location (if it is a message call transaction, then the code is from the to address of the transaction, and if it is a contract creation transaction, then the code is from the data field of the transaction) of the first item in the call stack, and checks that it is a static call defined above.

5.1 Proof in the Network Layer

Instead of broadcasting the transaction with gas price reduction using the `Transactions message` in the Ethereum sub-protocol, we define a new message `PriceReducedTransactions` message with the following format:

```
[+0x0n: P, [[nonce: P, receivingAddress: B_20, ...], [[callStackItem: P, ...], ...]], ...]
```

That is, each transaction is also supplied by a list of call stack proofs to demonstrate that the gas price reduction is valid.

6 Procedure

Using the above method, one can implement a new non-HF precompiled contract with the procedure below.

6.1 Implement the Functionality with EVM Opcodes

The first step is to implement the functionality of the proposed precompiled contract in EVM opcodes. This provides a slow version of what is desired. Implementation might be a little bit tricky, but given EVM's similarity to other VMs like WebAssembly, it is doable.

The resulting contract A might be quite slow, and might also require a large amount of gas to execute. If the gas required exceeds the block gas limit, miners will need to vote to raise the block gas limit to the level that is the originally comfortable level plus the gas of the newly created contract A. They should still only accept transactions up to the originally comfortable level – the raised level is totally just for the new contract.

You can see, in this way, no hard fork happens when we get the functionality. So old clients are not affected. Next, the only thing we need to do is to make it so that the contract A runs reasonably fast.

6.2 Replace the Contract with Native Codes

To do this, we create an equivalent native Rust/Go code as the above contract A. When the client tries to run anything that invokes the contract A (either in a transaction or through CALL* opcodes), instead of executing the EVM codes, it executes the native code. This returns the same result, and costs the same gas.

Clients solely uses the raised block gas limit if it finds that a block has a transaction that executes contract A.

6.3 Mark the Contract to Allow Gas Price Reduction

With the above, we then mark the contract as Gas Price Reductable and then accept the method described in this essay when executing the contract.

7 Discussions

7.1 Block Gas Limit Attack

If an attacker controls a significant amount of hash power, it can abuse the raised block gas limit to include other transactions, which may cause a slow down in miners importing new transactions.

This attack can be prevented by doing a soft fork that allows the chain to keep a secondary block gas limit (for example, as a RLP item in the extra data field of the block). If a block does not contain any transactions invoking contract A but its total used gas exceeds the secondary block gas limit, then that block is rejected.

7.2 Contract Call Failure

The client must reject the transaction or compensate the gas price reduction when a contract call fails, because in that case the precompiled contract may not have been executed.

References

[1] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151, 2014.