EVM: Technical walkthrough

Transaction and Gas

```rust
pub struct TxEnv {
    /// Caller or Author or tx signer
    pub caller: H160,
    pub gas_limit: u64,
    pub gas_price: U256,
    pub gas_priority_fee: Option<U256>,
    pub transact_to: TransactTo,
    pub value: U256,
    pub data: Bytes,
    pub chain_id: Option<u64>,
    pub nonce: Option<u64>,
    pub access_list: Vec<(H160, Vec<U256>)>,
}
```

- Signature is not present.

- Three types of Tx: Legacy, AccessList, eip1559Tx

- TransactTo is zero or contract address.

- Gas is introduced to limit execution, GasPrice for prioritizing transactions (eip1559).

Block

```rust
pub struct BlockEnv {
    pub number: U256,
    /// Coinbase or miner or address that created and signed the block.
    /// Address where we are going to send gas spend
    pub coinbase: H160,
    pub timestamp: U256,
    pub difficulty: U256,
    /// basefee is added in EIP1559 London upgrade
    pub basefee: U256,
    pub gas_limit: U256,
}
```

There are more additional fields but those are not used in EVM execution: **OmnerHash**, **ParentHash**, State/Transaction/Receipt **Root**, **Bloom**, ExtraData,MixHash/Nonce

BlockEnv and TxEnv can be seen as const field in EVM execution.
Additional cfg can be found in CfgEnv that contains ChainId and SpecId.

Beige paper: https://github.com/chronaeon/beigepaper/blob/master/beigepaper.pdf

Database interface

```rust
#[auto_impl(& mut, Box)]
pub trait Database {
    /// Get basic account information.
    fn basic(&mut self, address: H160) -> AccountInfo;
    /// Get account code by its hash
    fn code_by_hash(&mut self, code_hash: H256) -> Bytes;
    /// Get storage value of address at index.
    fn storage(&mut self, address: H160, index: U256) -> U256;
      rakita, 4 months ago • Restructure project
    // History related
    fn block_hash(&mut self, number: U256) -> H256;
}
```

All Block/Transaction data are contained inside environment struct.

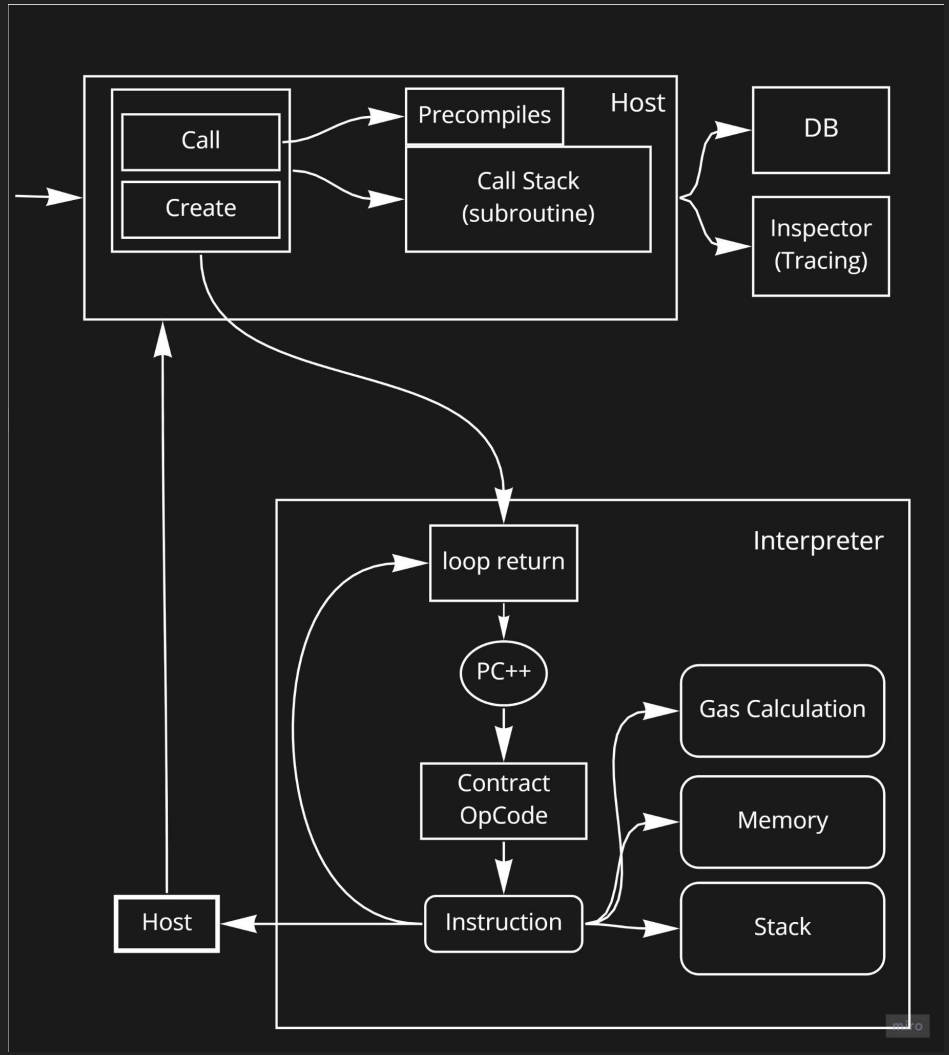
EVM: Host and Interpreter

- EVM Is **stack based** machine
- Transactions in block are executed in **one by one** manner.
- Transaction does transact in two ways: **Call and Create**
- EVM has two main parts: **Host** and **Interpreter**
- Needs to support upgrades in terms of hard forks.
- Precompiles as separate smart contracts written in native language.
- Output of EVM execution is: **Map<H160, Account>, Vec<Log>, ReturnStatus, GasUsed, OutputBytes**

EVM Diagram

Interpreter executes contracts and calls Host for needed information. For example to call another contract.

If revert or selfdestruct happen contract call stops, and all its changes are reverted. Parent caller continue its execution.



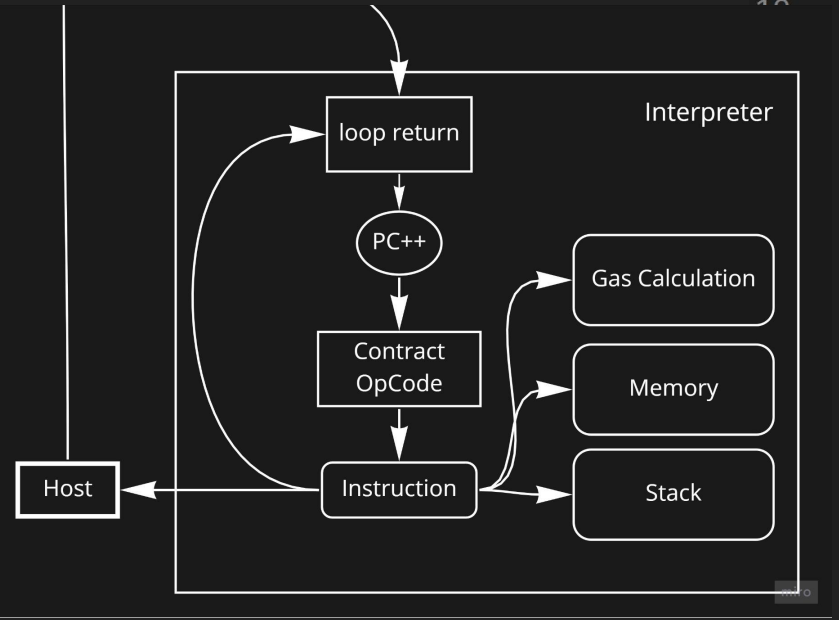
Interpreter

- Is the one that contains instructions and it is one responsible for execution of smart contracts.
- It has two stages. First stage, **Analysis**, goes over smart contract bytecode and checks positions of JUMPDEST opcode and creates JUMPDEST table, this is what all EVM's do (Evmone for optimization, added additional AdvanceAnalasys that for example precalculates GasBlock and adds padding if Bytecode doesn't finish with STOP so that we are safe to iterate and not check length at every step)
- Second stage is **Execution**: one big loop that does **steping** over bytecode, extracts OpCode does match(switch) and executes it depending on the type.
- PUSH(1-15) opcode is special case that allows you to have data embedded inside bytecode and be allowed to push it to **Stack**. All Other OpCodes are just **one byte** sized.

Interpreter contains:

- Memory: continuous unbound chunk of memory. Reserving new parts of memory is paid by gas. (In theory it does not have limit, but in practice you will need a lot of eth to pay for it)
- Stack: 256bit item stack with 1024 limit of items.
- Gas calculation: Spend gas is appended and checked against GasLimit before every instruction is executed. Gas per OpCode depends on the type and can be simple as ADD(priced 3gas) to SSTORE (depends on multiple factors, is new value zero, same as original,cold/hot load). Berlin hardfork introduces cold/hot account/storage loads.
- Host: Interpreter is called by Host but it contains Host interface to get informations that are outside of interpreter, and it allows us to CALL another contract by calling Host.
- Program counter and Contract that we are executing with its Analysis.

Interpreter machine in code




```rust
pub struct Machine {
    /// Contract information and invoking data
    pub contract: Contract,
    /// Program counter.
    pub program_counter: *const u8,
    /// Memory.
    pub memory: Memory,
    /// Stack.
    pub stack: Stack,
    /// left gas. Memory gas can be found in Memory field.
    pub gas: Gas,
    /// After call returns, its return data is saved here.
    pub return_data_buffer: Bytes,
    /// Return value.
    pub return_range: Range<usize>,
    /// used only for inspector.
    pub call_depth: u64,        rakita, 4 months ago • On sta
}
```

Just look and marvel at that rust code

OpCodes

Can be roughly separated into:

- Arithmetic and logic opcodes (ADD, SUB, MUL, SDIV, GT, LT, AND, OR,...)
- Stack related (POP, PUSH, DUP, SWAP,...)
- Memory opcodes (MLOAD, MSTORE, MSTORE8, MSIZE)
- Program counter related opcodes (JUMP, JUMPI, PC, JUMPDEST)
- Storage opcodes (SLOAD, SSTORE)
- Environment opcodes (CALLER, Transaction and Block info)
- Halting opcodes (STOP, RETURN, REVERT, SELFDESTRUCT,...)
- System opcodes (LOG,CALL, CREATE,CREATE2,CALLSTATIC, …)(next slides)

Full list here: https://github.com/wolflo/evm-opcodes and https://www.evm.codes/

CREATE And CREATE2

CREATE and CREATE2, are OpCodes used to create contract.

They randomly create address where bytecode is going to be added. Bytecode is received as return value of Interpreter after input code is executed.

Only difference between them is how address of contract is going to be created:

- CREATE address: Keccak256(rlp[caller,nonce]
- CREATE2 address: Keccak256([0xff,caller,salt,code_hash])

Call OpCodes

Multiple variants of CALL are called with different call context.Call context contains: **Address**, **Caller**, **ApparentValue**. (It affects SLOAD and SSTORE)

- CALL: **Caller** is present **context.address**. **Address and ApparentValue** are from stack.
- DELEGATECALL: **Address, Caller, ApparentValue** are from present context.
- CALLCODE: **Address** and **Caller** are present **context.addreess**. **ApparentValue** is from stack
- STATICCALL: **Same** as **CALL** but contracts will **fail** if SSTORE, LOG, SELFDESTRUCT, CREATE/2 or CALL if the value sent is not 0 are called

DELEGATECALL was a new opcode that was a bug fix for **CALLCODE** which did not preserve msg.sender and msg.value. If **Alice** invokes **Bob** who does **DELEGATECALL** to Charlie, the **msg.sender** in the **DELEGATECALL** is **Alice** (whereas if **CALLCODE** was used the **msg.sender** would be **Bob**).

More info: https://ethereum.stackexchange.com/questions/3667/difference-between-call-callcode-and-delegatecall

Logs

Logs are a way to log a message that something happened while executing smart contract. It allows smart contract devs to have a nice way to notify users/machine for specific event.

Log contain:

- Contract Address (From Call Context)
- Topics: that are just a list of 256 bit items. Item number depends on if it is LOG0…LOG4. Items are popped from stack.
- Data: Is read from Memory and can be in arbitrary size (of course you pay for every bite of it :))

Gas

Every Opcode is priced in terms of Gas. Every memory extension, DB load or store has some dynamic or base gas calculation.

FeeSpend is representing **GasUsed***GasPrice** and it is what you pay when you execute transaction to miner.

Eip1559 is improvement that introduced **BaseFee** that is taken from FeeSpend and burned (destroyed) rest of Fee is transferred to miner that created the block. And where our **GasPrice** is calculated as **BaseFee**+**PriorityFee**.

There was a way to get refund on gas **GasRefund** to decrease use gas. It is used in SSTORE and SELFDESTRUCT (Idea was okay but was misused and in future probably going to be removed).

Traces

It is utility used for debugging and useful for profiling of contract execution. It contains every step of execution and its **opcode**, used **gas**, **memory**, **stack**.

It can be tied with solidity output to get full view of what is happening.

Call Traces are for some use cases eve more needed, it represent what contracts are called.

```
$ ./evm --json statetest eip1559.json
{"pc":0,"op":58,"gas":"0x3c9ebc","gasCost":"0x2","memory":"0x","memSize":0,"stack":[],"ret
urnData":"0x","depth":1,"refund":0,"opName":"GASPRICE","error":""}
{"pc":1,"op":96,"gas":"0x3c9eba","gasCost":"0x3","memory":"0x","memSize":0,"stack":["0x3f2
"],"returnData":"0x","depth":1,"refund":0,"opName":"PUSH1","error":""}
```

```
depth:1, PC:0, gas:0x3c9ebc(3972796), OPCODE: "GASPRICE"(58)  refund:0x0(0) Stack:[], Data:
depth:1, PC:1, gas:0x3c9eba(3972794), OPCODE: "PUSH1"(96)  refund:0x0(0) Stack:[1010], Data:
depth:1, PC:3, gas:0x3c9eb7(3972791), OPCODE: "SSTORE"(85)  refund:0x0(0) Stack:[1010, 0], Data:
depth:1, PC:4, gas:0x3c5097(3952791), OPCODE: "BASEFEE"(72)  refund:0x0(0) Stack:[], Data:
depth:1, PC:5, gas:0x3c5095(3952789), OPCODE: "PUSH1"(96)  refund:0x0(0) Stack:[1000], Data:
```

Inspector

-Implementation detail but for traces to be obtain there are need to have some kind of hooks that will allows us to inspect internal state in runtime.

Forge (upcoming tool for solidity devs) are using something similar with Sputnik to obtain traces and apply cheatcodes that help with debugging.

It mostly does hooking on Host part and on every **step** inside Interpreter.

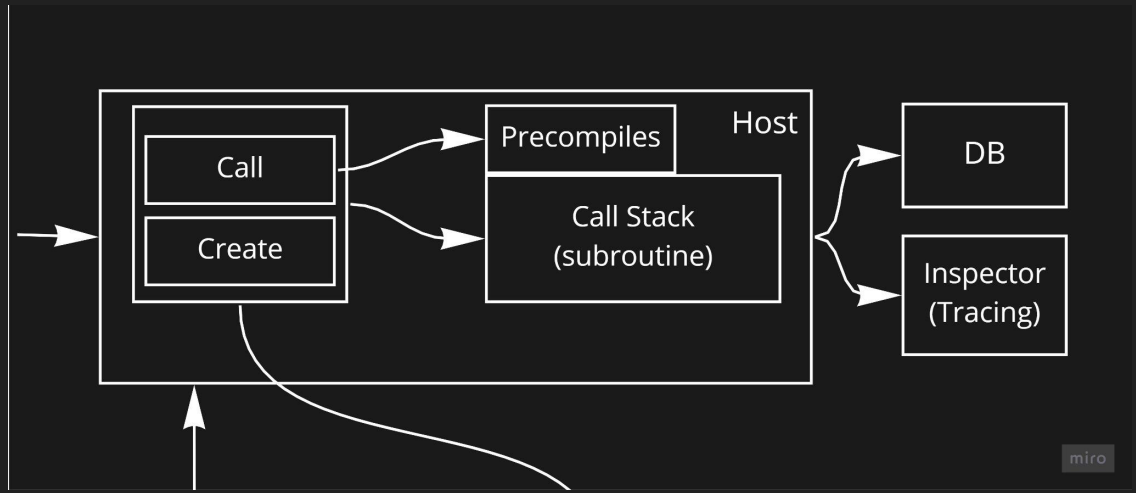
Interpreter code exploration

Host

- Is **starting** point of execution. It creates and calls Interpreter(Machine).
- As we already said, transaction can do: **CALL** and **CREATE** to EVM. so we have inner_call and inner_create functions for recursive calls from Interpreter.
- Additionally Host acts as **binding** between Interpreter and needed data from **outside** of EVM (database, environment, SLOAD,SSTORE).
- It handles contract calls and call stack. It needs to have ability to **revert** changes that happened inside one contract call. Including created **Logs**. Needs to handle selfdestruct storage reset.
- Reverts happen on **OutOfGas**, **StackOverflow** and **StackUnderflow** errors.
- Chooses if precompile contracts needs to be called if **0x00..01** to **0x00..09** addresses are called

Host contains:

- Subroutine: call stack with changes of every call. (Next slide)
- Precompiles: list of native hashes and curves.(Little bit later)
- DB: fetching account info, code, and storage from database.
- Environments: Transaction and Block information.
- *Inspector: Implementation dependent part for hooking of evm execution, main usage is tracing




```
21  pub struct EVMData<'a, DB> {
22      pub env: &'a mut Env,        You, 2 mont
23      pub subroutine: SubRoutine,
24      pub db: &'a mut DB,
25  }
26
    3 implementations | You, 2 months ago | 2 authors (rakita and
27  pub struct EVMImpl<'a, GSPEC: Spec, DB: Da
28      data: EVMData<'a, DB>,
29      precompiles: Precompiles,
30      inspector: &'a mut dyn Inspector<DB>,
31      _phantomdata: PhantomData<GSPEC>,
32  }
```

Subroutine (State and reverts)

It contains:

- State: current state of accounts and storages.
- Logs: Called OpCodes LOG1-4 are stored here.
- Depth: limit call stack to 1024
- Changelog: List of changes that happened in current changeset (contract call).
    - Checkpoint is created at every call and and it gets its own ID that is incremented over time. If some of contracts failed it’s checkpoint with its ID gets reverted and every ID that is higher.
    - If contract executed correctly usually its changelog should be merged with parent changelog, but we are just leaving it and in return just continue using current changelog without merging.

Host Trait

```rust
pub trait Host {
    const INSPECT: bool;        rakita, 3 months ago • SPEC rework
    type DB: Database;
    fn step(&mut self, machine: &mut Machine, is_static: bool) -> Return;
    fn step_end(&mut self, ret: Return, machine: &mut Machine) -> Return;
    fn env(&mut self) -> &mut Env;
    /// load account. Returns (is_cold,is_new_account)
    fn load_account(&mut self, address: H160) -> (bool, bool);
    /// Get environmental block hash.
    fn block_hash(&mut self, number: U256) -> H256;
    /// Get balance of address.
    fn balance(&mut self, address: H160) -> (U256, bool);
    /// Get code of address.
    fn code(&mut self, address: H160) -> (Bytes, bool);
    /// Get code hash of address.
    fn code_hash(&mut self, address: H160) -> (H256, bool);
    /// Get storage value of address at index.
    fn sload(&mut self, address: H160, index: U256) -> (U256, bool);
    /// Set storage value of address at index. Return if slot is cold/hot access.
    fn sstore(&mut self, address: H160, index: U256, value: U256) -> (U256, U256, U256, bool);
    /// Create a log owned by address with given topics and data.
    fn log(&mut self, address: H160, topics: Vec<H256>, data: Bytes);
    /// Mark an address to be deleted, with funds transferred to target.
    fn selfdestruct(&mut self, address: H160, target: H160) -> SelfDestructResult;
    /// Invoke a create operation.
    fn create<SPEC: Spec>(...) -> (Return, Option<H160>, Gas, Bytes);
    /// Invoke a call operation.
    fn call<SPEC: Spec>(...) -> (Return, Gas, Bytes);
}
```

Precompile Name	Address	Type
Secp256k1::ecrecovery	0x00…01	Curve signature recovery
sha256	0x00…02	Hash
ripemd160	0x00…03	Hash
Identity	0x00…04	Utility
bigModExp	0x00…05	Math
Bn128::add	0x00…06	Curve
Bn128::mul	0x00…07	Curve
Bn128::pair	0x00…08	Curve
Blake2	0x00…09	Hash

More info: https://docs.klaytn.com/smart-contract/precompiled-contracts

Host code exploration

Hard Forks

- Arrow Glacier: Dec-09-2021
  - EIP-4345 – delays the difficulty bomb until June 2022
- London: Aug-05-2021
  - EIP-1559 – improves the transaction fee market
  - EIP-3198 – returns the BASEFEE from a block
  - EIP-3529 - reduces gas refunds for EVM operations
  - EIP-3541 - prevents deploying contracts starting with 0xEF
  - EIP-3554 – delays the Ice Age until December 2021
- Berlin: Apr-15-2021
  - EIP-2565 – lowers ModExp gas cost
  - EIP-2718 – enables easier support for multiple transaction types
  - EIP-2929 – gas cost increases for state access opcodes
  - EIP-2930 – adds optional access lists
- Muir Glacier: Jan-02-2020
  - EIP-2384 – delays the difficulty bomb for another 4,000,000 blocks, or ~611 days.

More on it here: https://ethereum.org/en/history/

Optimizations

Use u64 for gas calculations, in spec it is U256: Spending u256 gas is not something that is going to happen, for comparison current eth Block limit is 30M gas.

Memory calculation for u64, u256 does not make sense. There is no hard limit on memory used, but for every 32bit you use you pay for gas that acts as soft limiter. Usually memory is specified as offset+size and memory is paid as `max(offset+size)` number

Ethereum uses big-endian encoding and all PUSH values are in bigendian format, this can be slow on most machines that uses little endian and have support for u64 items. So in EVM stack is basically U256 that is [u64;4] (list of four u64 numbers) and we always convert those things back and forth.

Q&A