

Ethereum Execution Layer Specification

Yellow Paper Spiritual Successor

tl;dr

If you're only going to watch a few seconds of this presentation, this is what you want to know.

tl;dr . where we were

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER
BERLIN VERSION 2bcd2b2d – 2023-08-25

DR. GAVIN WOOD
FOUNDER, ETHEREUM & PARITY
GAVIN@PARITY.IO

$$\begin{aligned} (2) \quad \sigma_{t+1} &\equiv \Pi(\sigma_t, B) \\ (3) \quad B &\equiv (\dots, (T_0, T_1, \dots), \dots) \\ (4) \quad \Pi(\sigma, B) &\equiv \Omega(B; \Upsilon(\Upsilon(\sigma, T_0), T_1) \dots) \end{aligned}$$

$$\ell(\mathbf{x}) \equiv \mathbf{x}[\|\mathbf{x}\| - 1]$$

$$v(x) \equiv x_n \in \mathbb{N}_{256} \wedge x_b \in \mathbb{N}_{256} \wedge x_s \in \mathbb{B}_{32} \wedge x_c \in \mathbb{B}_{32}$$

(152)

$$D_J(\mathbf{c}, i) \equiv \begin{cases} \{\} & \text{if } i \geq \|\mathbf{c}\| \\ \{i\} \cup D_J(\mathbf{c}, N(i, \mathbf{c}[i])) & \text{if } \mathbf{c}[i] = \text{JUMPDEST} \\ D_J(\mathbf{c}, N(i, \mathbf{c}[i])) & \text{otherwise} \end{cases}$$

tl;dr . where we are

```
def calculate_intrinsic_cost(tx: Transaction) -> Uint:
    data_cost = 0
    for byte in tx.data:
        if byte == 0:
            data_cost += TX_DATA_COST_PER_ZERO
        else:
            data_cost += TX_DATA_COST_PER_NON_ZERO
    if tx.to == Bytes0(b''):
        create_cost = TX_CREATE_COST
        create_cost = TX_CREATE_COST + int(init_code_cost(Uint(len(tx.data))))
    else:
        create_cost = 0
    access_list_cost = 0
    if isinstance(tx, (AccessListTransaction, FeeMarketTransaction)):
        for _address, keys in tx.access_list:
            access_list_cost += TX_ACCESS_LIST_ADDRESS_COST
            access_list_cost += len(keys) * TX_ACCESS_LIST_STORAGE_KEY_COST
    return Uint(TX_BASE_COST + data_cost + create_cost + access_list_cost)
```

Specification

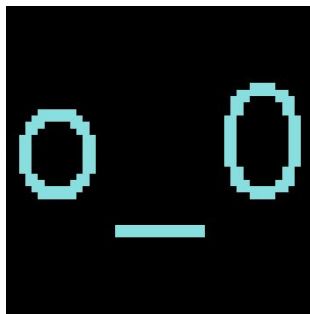
The behaviour of `SELFDESTRUCT` is changed in the following way:

```
# register account for deletion
evm.accounts_to_delete.add(originator)
# Only continue if the contract has been created in the same tx
if originator in evm.env.created_contracts:
```

```
# mark beneficiary as touched
if account_exists_and_is_empty(evm.env.state, beneficiary):
    evm.touched_accounts.add(beneficiary)
# register account for deletion
evm.accounts_to_delete.add(originator)
```


```
# HALT the execution
evm.running = False
# mark beneficiary as touched
if account_exists_and_is_empty(evm.env.state, beneficiary):
    evm.touched_accounts.add(beneficiary)
```

about me



Sam Wilson

 @sam_wilson@moobi.monster

 github.com/SamWilsn

eels

What is an “Ethereum Execution Layer Specification”?

Python reference implementation of most of an Ethereum client.

team . maintainers



Guruprasad Kamath

✕ twitter.com/guruka_math

🐙 github.com/gurukamath

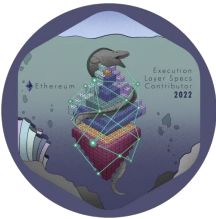
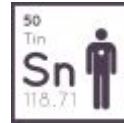
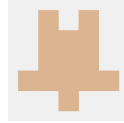


Peter Davies

✕ twitter.com/peter_t_davies

🐙 github.com/petertdavies

team . contributors



— there's a POAP!

+ many more

yellow paper

- Created around 2014 by Gavin Wood
- Creative Commons Attribution Share-Alike (CC-BY-SA) Version 4.0

yellow paper . tour

a blazingly fast tour of the yellow paper

by someone who doesn't like it

yellow paper . tour . background

1. INTRODUCTION

With ubiquitous internet connections in most places of the world, global information transmission has become incredibly cheap. Technology-rooted movements like Bitcoin have demonstrated through the power of the default, consensus mechanisms, and voluntary respect of the social contract, that it is possible to use the internet to make a decentralised value-transfer system that can be shared across the world and virtually free to use. This system can be said to be a very specialised version of a cryptographically secure, transaction-based state machine. Follow-up systems such as Namecoin adapted this original “currency application” of the technology into other applications, albeit rather simplistic ones.

Ethereum is a project which attempts to build the generalised technology; technology on which all transaction-based state machine concepts may be built. Moreover it aims to provide to the end-developer a tightly integrated end-to-end system for building software on a hitherto unexplored compute paradigm in the mainstream: a trustful object messaging compute framework.

1.1. Driving Factors. There are many goals of this project; one key goal is to facilitate transactions between consenting individuals who would otherwise have no means to trust one another. This may be due to geographical separation, interfacing difficulty, or perhaps the incompatibility, incompetence, unwillingness, expense, uncertainty, inconvenience, or corruption of existing legal systems. By specifying a state-change system through a rich and unambiguous language, and furthermore architecting a system such that we can reasonably expect that an agreement will be thus enforced autonomously, we can provide a means to this end.

1.2. Previous Work. Buterin [2013a] first proposed the kernel of this work in late November, 2013. Though now evolved in many ways, the key functionality of a blockchain with a Turing-complete language and an effectively unlimited inter-transaction storage capability remains unchanged.

yellow paper . tour . blockchain

2. THE BLOCKCHAIN PARADIGM

A valid state transition is one which comes about through a transaction. Formally:

$$(1) \quad \sigma_{t+1} \equiv \Upsilon(\sigma_t, T)$$

where Υ is the Ethereum state transition function.

Mining is the process of dedicating effort (working) to bolster one series of transactions (a block) over any other potential competitor block. It is achieved thanks to a cryptographically secure proof. This scheme is known as a proof-of-work and is discussed in detail in section 11.5.

Formally, we expand to:

$$(2) \quad \sigma_{t+1} \equiv \Pi(\sigma_t, B)$$

$$(3) \quad B \equiv (\dots, (T_0, T_1, \dots), \dots)$$

$$(4) \quad \Pi(\sigma, B) \equiv \Omega(B; \Upsilon(\Upsilon(\sigma, T_0), T_1) \dots)$$

yellow paper . tour . fork choice

2.2. **Which History?** Since the system is decentralised and all parties have an opportunity to create a new block on some older pre-existing block, the resultant structure is necessarily a tree of blocks. In order to form a consensus as to which path, from root (the genesis block) to leaf (the block containing the most recent transactions) through this tree structure, known as the blockchain, there must be an agreed-upon scheme. If there is ever a disagreement between nodes as to which root-to-leaf path down the block tree is the 'best' blockchain, then a *fork* occurs.

yellow paper . tour . state

4.1. World State. The world state (*state*), is a mapping between addresses (160-bit identifiers) and account states (a data structure serialised as RLP, see Appendix [B]). Though not stored on the blockchain, it is assumed that the implementation will maintain this mapping in a modified Merkle Patricia tree (*trie*, see Appendix [D]). The trie requires a simple database backend that maintains a mapping of byte arrays to byte arrays; we name this underlying database the state database. This has a number of benefits; firstly the root node of this structure is cryptographically dependent on all internal data and as such its hash can be used as a secure identity for the entire system state. Secondly, being an immutable data structure, it allows any previous state (whose root hash is known) to be recalled by simply altering the root hash accordingly. Since we store all such root hashes in the blockchain, we are able to trivially revert to old states.

The account state, $\sigma[a]$, comprises the following four fields:

nonce: A scalar value equal to the number of transactions sent from this address or, in the case of accounts with associated code, the number of contract-creations made by this account. For account of address a in state σ , this would be formally denoted $\sigma[a]_n$.

balance: A scalar value equal to the number of Wei owned by this address. Formally denoted $\sigma[a]_b$.

storageRoot: A 256-bit hash of the root node of a Merkle Patricia tree that encodes the storage contents of the account (a mapping between 256-bit integer values), encoded into the trie as a mapping from the Keccak 256-bit hash of the 256-bit integer keys to the RLP-encoded 256-bit integer values. The hash is formally denoted $\sigma[a]_s$.

codeHash: The hash of the EVM code of this account—this is the code that gets executed should this address receive a message call. All such code fragments are contained in the state database under their corresponding hashes for later retrieval. This hash is formally denoted $\sigma[a]_c$, and thus the code may be denoted as \mathbf{b} , given that $\text{KEC}(\mathbf{b}) = \sigma[a]_c$.

Since we typically wish to refer not to the trie's root hash but to the underlying set of key/value pairs stored within, we define a convenient equivalence:

$$(7) \quad \text{TRIE}(L_I^*(\sigma[a]_s)) \equiv \sigma[a]_s$$

The collapse function for the set of key/value pairs in the trie, L_I^* , is defined as the element-wise transformation of the base function L_I , given as:

$$(8) \quad L_I((k, v)) \equiv (\text{KEC}(k), \text{RLP}(v))$$

yellow paper . tour . transaction

4.2. The Transaction. A transaction (formally, T) is a single cryptographically-signed instruction constructed by an actor externally to the scope of Ethereum. The sender of a transaction cannot be a contract. While it is assumed that the ultimate external actor will be human in nature, software tools will be used in its construction and dissemination^[1]. EIP-2718 by Zoltu [2020] introduced the notion of different transaction types. As of the Berlin version of the protocol, there are two transaction types: 0 (legacy) and 1 (EIP-2930 by Buterin and Swende [2020b]). Further, there are two subtypes of transactions: those which result in message calls and those which result in the creation of new accounts with associated code (known informally as ‘contract creation’). All transaction types specify a number of common fields:

type: EIP-2718 transaction type; formally T_x .

nonce: A scalar value equal to the number of transactions sent by the sender; formally T_n .

gasPrice: A scalar value equal to the number of Wei to be paid per unit of *gas* for all computation costs incurred as a result of the execution of this transaction; formally T_p .

gasLimit: A scalar value equal to the maximum amount of gas that should be used in executing this transaction. This is paid up-front, before any computation is done and may not be increased later; formally T_g .

to: The 160-bit address of the message call’s recipient or, for a contract creation transaction, \emptyset , used here to denote the only member of \mathbb{B}_0 ; formally T_t .

value: A scalar value equal to the number of Wei to be transferred to the message call’s recipient or, in the case of contract creation, as an endowment to the newly created account; formally T_v .

r, s: Values corresponding to the signature of the transaction and used to determine the sender of the transaction; formally T_r and T_s . This is expanded in Appendix F.

(16)

$$L_T(T) \equiv \begin{cases} (T_n, T_p, T_g, T_t, T_v, \mathbf{p}, T_w, T_r, T_s) & \text{if } T_x = 0 \\ (T_c, T_n, T_p, T_g, T_t, T_v, \mathbf{p}, T_A, T_y, T_r, T_s) & \text{if } T_x = 1 \end{cases}$$

where

(17)

$$\mathbf{p} \equiv \begin{cases} T_i & \text{if } T_t = \emptyset \\ T_d & \text{otherwise} \end{cases}$$

yellow paper . tour . block

4.3. The Block.

parentHash: The Keccak 256-bit hash of the parent block's header, in its entirety; formally H_p .

ommersHash: The Keccak 256-bit hash of the omers list portion of this block; formally H_o .

beneficiary: The 160-bit address to which all fees collected from the successful mining of this block be transferred; formally H_c .

stateRoot: The Keccak 256-bit hash of the root node of the state trie, after all transactions are executed and finalisations applied; formally H_r .

transactionsRoot: The Keccak 256-bit hash of the root node of the trie structure populated with each transaction in the transactions list portion of the block; formally H_t .

receiptsRoot: The Keccak 256-bit hash of the root node of the trie structure populated with the receipts of each transaction in the transactions list portion of the block; formally H_e .

logsBloom: The Bloom filter composed from indexable information (logger address and log topics) contained in each log entry from the receipt of each transaction in the transactions list; formally H_b .

difficulty: A scalar value corresponding to the difficulty level of this block. This can be calculated from the previous block's difficulty level and the timestamp; formally H_d .

number: A scalar value equal to the number of ancestor blocks. The genesis block has a number of zero; formally H_i .

gasLimit: A scalar value equal to the current limit of gas expenditure per block; formally H_l .

gasUsed: A scalar value equal to the total gas used in transactions in this block; formally H_g .

timestamp: A scalar value equal to the reasonable output of Unix's time() at this block's inception; formally H_s .

extraData: An arbitrary byte array containing data relevant to this block. This must be 32 bytes or fewer; formally H_x .

mixHash: A 256-bit hash which, combined with the nonce, proves that a sufficient amount of computation has been carried out on this block; formally H_m .

nonce: A 64-bit value which, combined with the mix-hash, proves that a sufficient amount of computation has been carried out on this block; formally H_n .

yellow paper . tour . the rest

- Gas
- Contracts
- Virtual Machine
- RLP
- Modified MPT
- Precompiles
- EVM Instructions

You get the idea.

yellow paper . problems

$$\sigma_{t+1} \equiv \Upsilon(\sigma_t, T)$$

$$\sigma[a] \quad L_T(T) \equiv \begin{cases} (T_n, T_p, T_g, T_t, T_v, \mathbf{p}, T_w, T_r, T_s) & \text{if } T_x = 0 \\ (T_c, T_n, T_p, T_g, T_t, T_v, \mathbf{p}, T_A, T_y, T_r, T_s) & \text{if } T_x = 1 \end{cases}$$

These formulas were shown on the previous slides.

What do they represent?

$$\text{TRIE}(L_I^*(\sigma[a]_s)) \equiv \sigma[a]_s$$

$$L_I((k, v)) \equiv (\text{KEC}(k), \text{RLP}(v))$$

yellow paper . problems . audience

- The Yellow Paper is inaccessible to most programmers
- Programmers are the ones who need to understand it the most
- Core EIPs (where changes to Ethereum are described) rarely use this notation

yellow paper . problems . untestable

- The Yellow Paper is (mostly) human language
- No way to test the specification itself
- No way to use the specification to fill tests

yellow paper . benefits

To be fair, the Yellow Paper is:

- Succinct
- Formal
- Algorithm Independent

More on that later!

eels

- Created around May 2021 by Consensys' Quilt team
- Most recently maintained by the EELS team at the Ethereum Foundation
- Creative Commons Zero v1.0 Universal

eels . why

- To escape the frustration of trying to understand the Yellow Paper
- No “snapshot” of the current state of Ethereum
 - Only had previous Yellow Paper + EIPs specifying changes
- Be more accessible to programmers
- So the same document that specifies Ethereum also fills the automated tests for it

eels . tour

a blazingly fast tour of EELS

by someone who wrote chunks of it

eels . tour . forks



arrow_glacier



assets



berlin



byzantium



constantinople



crypto



dao_fork



frontier



gray_glacier



homestead



istanbul



london



muir_glacier



paris



shanghai



spurious_dragon



tangerine_whistle



utils



base_types.py



ethash.py



exceptions.py



fork_criteria.py



genesis.py



__init__.py



py.typed



rlp.py

eels . tour . blockchain

BlockChain

History and current state of the block chain.

```
72 @dataclass
class BlockChain:
    blocks
    state
    chain_id
```

blocks

```
78     blocks: List[Block]
```

state

```
79     state: State
```

chain_id

```
80     chain_id: U64
```

```
def state_transition(chain: BlockChain, block: Block) -> None:
    167     parent_header = chain.blocks[-1].header
    168     validate_header(block.header, parent_header)
    169     validate_ommers(block.ommers, block.header, chain)
    170     (
    171         gas_used,
    172         transactions_root,
    173         receipt_root,
    174         block_logs_bloom,
    175         state,
    176     ) = apply_body(
    177         chain.state,
    178         get_last_256_block_hashes(chain),
    179         block.header.coinbase,
    180         block.header.number,
    181         block.header.gas_limit,
    182         block.header.timestamp,
    183         block.header.difficulty,
    184         block.transactions,
    185         block.ommers,
    186         chain.chain_id,
    187     )
    188     ensure(gas_used == block.header.gas_used, InvalidBlock)
    189     ensure(transactions_root == block.header.transactions_root, InvalidBlock)
    190     ensure(state_root(state) == block.header.state_root, InvalidBlock)
    191     ensure(receipt_root == block.header.receipt_root, InvalidBlock)
    192     ensure(block_logs_bloom == block.header.bloom, InvalidBlock)
    193
    194     chain.blocks.append(block)
    195     if len(chain.blocks) > 255:
    196         chain.blocks = chain.blocks[-255:]
```

eels . tour . fork choice

```
def validate_proof_of_work(header: Header) -> None: hide source
300     header_hash = generate_header_hash_for_pow(header)
303     cache = generate_cache(header.number)
304     mix_digest, result = hashimoto_light(
305         header_hash, header.nonce, cache, dataset_size(header.number)
306     )
307
308     ensure(mix_digest == header.mix_digest, InvalidBlock)
309     ensure(
310         Uint.from_be_bytes(result) <= (U256_CEIL_VALUE // header.difficulty),
311         InvalidBlock,
312     )
```

I'm lying a bit here. Technically EELS assumes it only receives the canonical chain.

eels . tour . state

State

Contains all information that is preserved between transactions.

```
29 @dataclass
class State:
```

Functions

```
close_state
begin_transaction
commit_transaction
rollback_transaction
get_account
get_account_optional
set_account
destroy_account
destroy_storage
mark_account_created
get_storage
set_storage
storage_root
state_root
account_exists
account_has_code_or_nonce
is_account_empty
account_exists_and_is_empty
is_account_alive
modify_state
move_ether
set_account_balance
touch_account
increment_nonce
set_code
create_ether
get_storage_original
```

eels . tour . transaction

LegacyTransaction

Atomic operation performed on the block chain.

```
48 @slotted_freezable
49 @dataclass
class LegacyTransaction:
    nonce
    gas_price
    gas
    to
    value
    data
    v
    r
    s
```

AccessListTransaction

The transaction type added in EIP-2930 to support access lists.

```
66 @slotted_freezable
67 @dataclass
class AccessListTransaction:
    chain_id
    nonce
    gas_price
    gas
    to
    value
    data
    access_list
    y_parity
    r
```

Transaction

```
86 Transaction = Union[LegacyTransaction, AccessListTransaction]
```

eels . tour . block

Header

Header portion of a block on the chain.

```
148 @slotted_freezable
149 @dataclass
class Header:
    parent_hash
    omers_hash
    coinbase
    state_root
    transactions_root
    receipt_root
    bloom
    difficulty
    number
    gas_limit
    gas_used
    timestamp
    extra_data
    mix_digest
    nonce
```

Block

A complete block.

```
172 @slotted_freezable
173 @dataclass
class Block:
    header
    transactions
    omers
```

eels . problems

- Requires Python knowledge
- Because it is an implementation, it requires specific algorithm choices
- Verbose
- Less accessible to academics

eels . benefits

```
def state_transition(chain: Blockchain, block: Block) -> None:
```

```
class Account:  
    nonce  
    balance  
    code
```

```
Transaction = Union[LegacyTransaction, AccessListTransaction]
```

```
def get_storage(state: State, address: Address, key: Bytes) -> U256:
```

More verbose, but easier to understand.

eels . benefits . audience

- EELS is accessible to most programmers (it's normal Python!)
- Core EIPs often use Python-style pseudocode. Now it can be actual Python!

eels . benefits . maintained

- Any programmer can read Python; some can even write it!
- Implemented all the way up to Cancun

eels . benefits . testable

- Can actually sync the chain (very slowly)
- Passes the `ethereum/tests` suite
- Can fill tests for use in production clients

eels . cool features

What have we been doing for a year?

eels . cool features . diffs

```
225     block_parent_hash = keccak256(rlp.encode(parent_header))
226     ensure(header.parent_hash == block_parent_hash, InvalidBlock)
227
239     if (
240         header.number >= FORK_CRITERIA.block_number
241         and header.number < FORK_CRITERIA.block_number + 10
242     ):
243         ensure(header.extra_data == b"dao-hard-fork", InvalidBlock)
244
228     validate_proof_of_work(header)
```

eels . cool features . fuzzing

- Tool generates inputs randomly.
- Outputs compared to all other Ethereum clients.

resources

- [Adding an EVM Instruction to EELS](#)
- [EELS: The Future of Execution Layer Specifications by Peter Davies](#)
- <https://ethereum.github.io/execution-specs/>