Ethereum Blocks

Saravanan Vijayakumaran
sarva@ee.iitb.ac.in

Department of Electrical Engineering
Indian Institute of Technology Bombay

September 3, 2019

Ethereum Block Header

Block = (Header, Transactions, Uncle Headers)

Block Header

`parentHash`	32	bytes
`ommersHash`	32	bytes
`beneficiary`	20	bytes
`stateRoot`	32	bytes
`transactionsRoot`	32	bytes
`receiptsRoot`	32	bytes
`logsBloom`	256	bytes
`difficulty`	≥ 1	byte
`number`	≥ 1	byte
`gasLimit`	≥ 1	byte
`gasUsed`	≥ 1	byte
`timestamp`	≤ 32	bytes
`extraData`	≤ 32	bytes
`mixHash`	32	bytes
`nonce`	8	bytes

Simple Fields in Block Header

parentHash	32	bytes
ommersHash	32	bytes
beneficiary	20	bytes
stateRoot	32	bytes
transactionsRoot	32	bytes
receiptsRoot	32	bytes
logsBloom	256	bytes
difficulty	≥ 1	byte
number	≥ 1	byte
gasLimit	≥ 1	byte
gasUsed	≥ 1	byte
timestamp	≤ 32	bytes
extraData	≤ 32	bytes
mixHash	32	bytes
nonce	8	bytes

- parentHash = Keccak-256 hash of parent block header
- beneficiary = Destination address of block reward and transaction fees
- stateRoot = Root hash of world state trie after all transactions are applied
- transactionsRoot = Root hash of trie populated with all transactions in the block
- number = Number of ancestor blocks
- timestamp = Unix time at block creation
- extraData = Arbitrary data; Miners identify themselves in this field

gasLimit and gasUsed

`parentHash`	32	bytes
`ommersHash`	32	bytes
`beneficiary`	20	bytes
`stateRoot`	32	bytes
`transactionsRoot`	32	bytes
`receiptsRoot`	32	bytes
`logsBloom`	256	bytes
`difficulty`	≥ 1	byte
`number`	≥ 1	byte
`gasLimit`	≥ 1	byte
`gasUsed`	≥ 1	byte
`timestamp`	≤ 32	bytes
`extraData`	≤ 32	bytes
`mixHash`	32	bytes
`nonce`	8	bytes

- `gasUsed` is the total gas used by all transactions in the block
- `gasLimit` is the maximum gas which can be used
- $|\texttt{gasLimit} - \texttt{parent.gasLimit}| \leq \frac{\texttt{parent.gasLimit}}{1024}$
- Miner can choose to increase or decrease the `gasLimit`

logsBloom and receiptsRoot

`parentHash`	32	bytes
`ommersHash`	32	bytes
`beneficiary`	20	bytes
`stateRoot`	32	bytes
`transactionsRoot`	32	bytes
`receiptsRoot`	32	bytes
`logsBloom`	256	bytes
`difficulty`	≥ 1	byte
`number`	≥ 1	byte
`gasLimit`	≥ 1	byte
`gasUsed`	≥ 1	byte
`timestamp`	≤ 32	bytes
`extraData`	≤ 32	bytes
`mixHash`	32	bytes
`nonce`	8	bytes

- Bloom filter = Probabilistic data structure for set membership queries
  - **Query:** Is x in the set? **Response:** "Maybe" or "No"
- `receiptsRoot` is the root hash of transaction receipts trie
  - Each transaction receipt contains Bloom filter of addresses and "topics"
- `logBloom` is the OR of all transaction receipt Bloom filters
- Light clients can efficiently retrieve only transactions of interest

Mining

Ethash Mining Algorithm

- An epoch lasts 30,000 blocks
- Epoch index EI = block_number / 30000
- At an epoch beginning
  - A list called cache of size $\approx 2^{24} + EI \times 2^{17}$ bytes is created
  - A list called dataset of size $\approx 2^{30} + EI \times 2^{23}$ bytes is created
- The dataset is also called the DAG (directed acyclic graph)

Block Number	Epoch	Cache Size	DAG Size	Start Date
30000	1	16 MB	1 GB	17 Oct, 2015
3840000	128	32 MB	2 GB	21 Jul, 2017
7680000	256	48 MB	3 GB	30 Apr, 2019
192000000	640	96 MB	6 GB	25 Aug, 2024

Source: `https://investoon.com/tools/dag_size`

- Mining nodes need to store full dataset (ASIC resistance)
- Light nodes store cache and recalculate specific dataset items

Cache Generation

- The cache is initialized by repeatedly hashing a seed (deriving from the block headers)
- Two rounds of a function called randmemohash are applied

```
1   HASH_BYTES = 64
2   CACHE_ROUNDS = 3
3
4   def mkcache(cache_size, seed):
5     n = cache_size // HASH_BYTES
6
7     # Sequentially produce the initial dataset
8     o = [sha3_512(seed)]
9     for i in range(1, n):
10        o.append(sha3_512(o[-1]))
11
12    # Use a low-round version of randmemohash
13    for _ in range(CACHE_ROUNDS):
14        for i in range(n):
15            v = o[i][0] % n
16            o[i] = sha3_512(map(xor, o[(i-1+n) % n], o[v]))
17
18    return o
```

Dataset Generation

```
HASH_BYTES = 64
WORD_BYTES = 4
DATASET_PARENTS = 256

FNV_PRIME = 0x01000193

def fnv(v1, v2):
    return ((v1 * FNV_PRIME) ^ v2) % 2**32

def calc_dataset_item(cache, i):
  n = len(cache)
  r = HASH_BYTES // WORD_BYTES
  # initialize the mix
  mix = copy.copy(cache[i % n])
  mix[0] ^= i
  mix = sha3_512(mix)
  # fnv it with a lot of random cache nodes based on i
  for j in range(DATASET_PARENTS):
      cache_index = fnv(i ^ j, mix[j % r])
      mix = map(fnv, mix, cache[cache_index % n])
  return sha3_512(mix)
```

Ethash Mining Algorithm

`parentHash`	32	bytes
`ommersHash`	32	bytes
`beneficiary`	20	bytes
`stateRoot`	32	bytes
`transactionsRoot`	32	bytes
`receiptsRoot`	32	bytes
`logsBloom`	256	bytes
`difficulty`	≥ 1	byte
`number`	≥ 1	byte
`gasLimit`	≥ 1	byte
`gasUsed`	≥ 1	byte
`timestamp`	≤ 32	bytes
`extraData`	≤ 32	bytes
`mixHash`	32	bytes
`nonce`	8	bytes

- Cache calculation involves hashing previous cache elements pseudorandomly
- Every dataset element involves hashing 256 pseudorandom cache elements
- Mining loop takes partial header hash, `nonce`, and dataset as input
- 128 dataset elements are used to create 256-bit `mixHash`

$$\text{Mining output} = \text{Keccak256}\left(\text{Keccak512}(\texttt{HdrHash} \| \texttt{nonce}) \| \texttt{mixHash}\right)$$

Mining Difficulty

`parentHash`	32	bytes
`ommersHash`	32	bytes
`beneficiary`	20	bytes
`stateRoot`	32	bytes
`transactionsRoot`	32	bytes
`receiptsRoot`	32	bytes
`logsBloom`	256	bytes
`difficulty`	≥ 1	byte
`number`	≥ 1	byte
`gasLimit`	≥ 1	byte
`gasUsed`	≥ 1	byte
`timestamp`	≤ 32	bytes
`extraData`	≤ 32	bytes
`mixHash`	32	bytes
`nonce`	8	bytes

- Proof of work is valid if `mixhash` and `nonce` lead to

$$\text{Keccak256}\left(\text{Keccak512}(\texttt{HdrHash}\|\texttt{nonce})\|\texttt{mixHash}\right) \leq \frac{2^{256}}{\texttt{difficulty}}$$

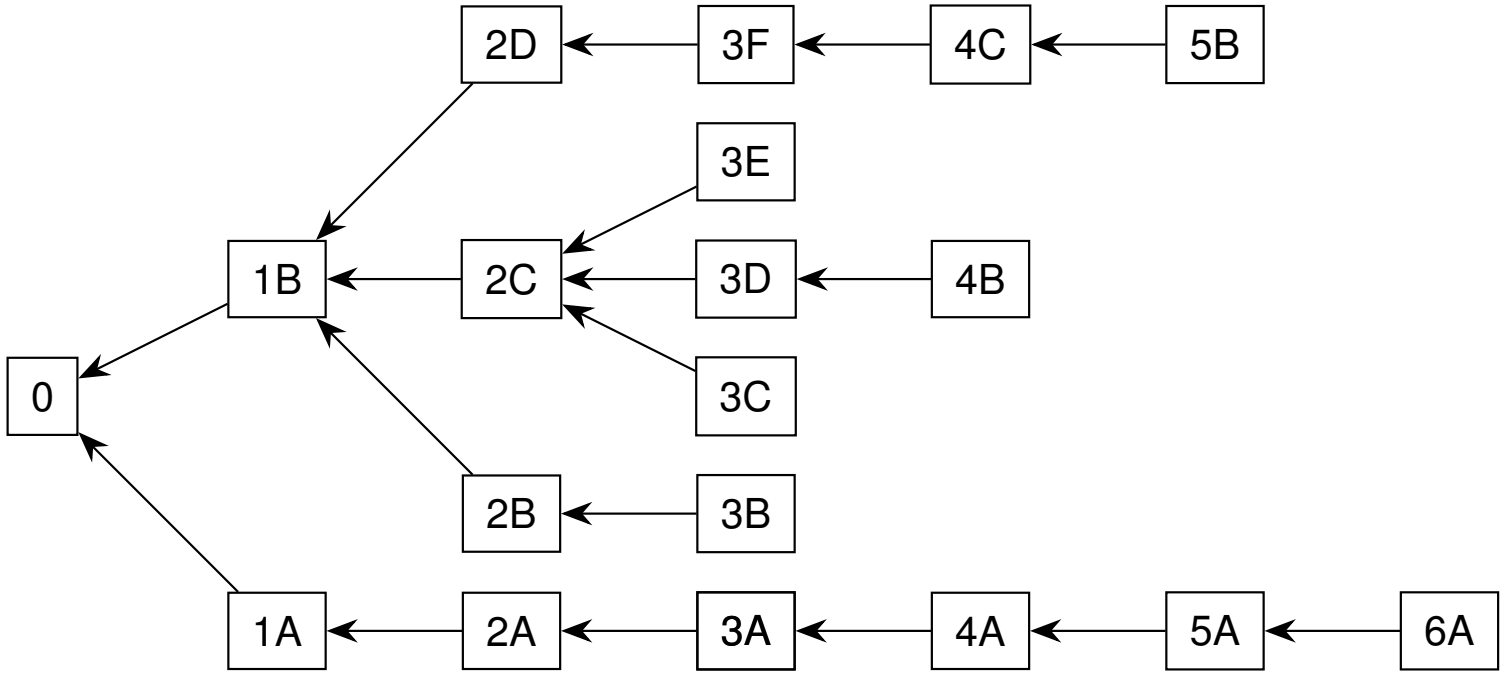
- Partial validation of PoW in block can be done without DAG or cache

Uncle Incentivization

Uncle Blocks

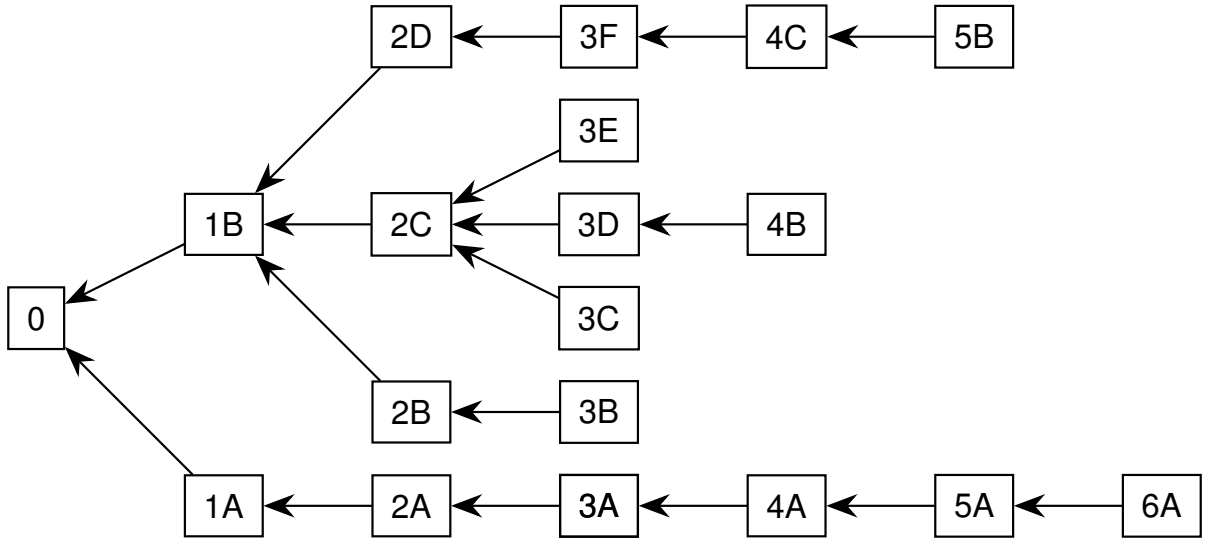
- Block = (Block Header, Transactions List, Uncle Header List)
- `ommersHash` in block header is hash of uncle header list
- **Problem:** Low inter-block time leads to high stale rate
  - Stale blocks do not contribute to network security
- **Solution:** Reward stale block miners and also miners who include stale block headers
- Rewarded stale blocks are called uncles or ommers
  - Transactions in uncle blocks are invalid
  - Only a fraction of block reward goes to uncle creator; no transaction fees
- Greedy Heaviest Observed Subtree (GHOST) protocol proposed by Sompolinsky and Zohar in December 2013
- Ethereum uses a simpler version of GHOST

GHOST Protocol



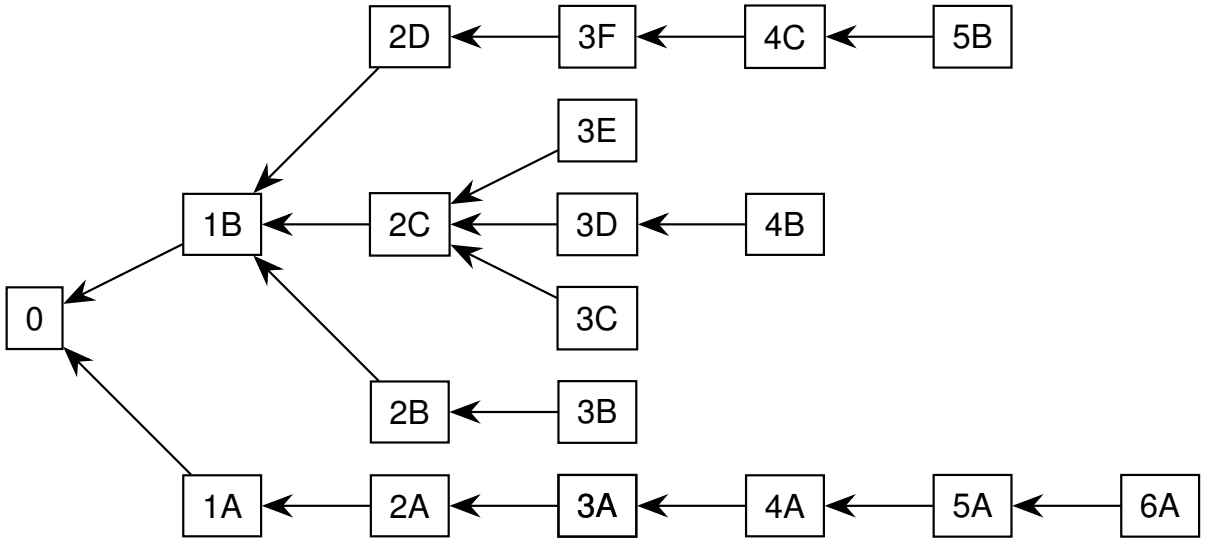
- A policy for choosing the main chain in case of forks
- Given a block tree T, the protocol specifies GHOST(T) as the block representing the main chain
- Mining nodes calculate GHOST(T) locally and mine on top of it
- Heaviest subtree rooted at fork is chosen

GHOST Protocol



function CHILDREN$_T(B)$
    return Set of blocks with B as immediate parent
end function
function SUBTREE$_T(B)$
    return Subtree rooted at B
end function
function GHOST(T)
    $B \leftarrow$ Genesis Block
    while True **do**
        if CHILDREN$_T(B) = \emptyset$ **then return** B and exit
        else $B \leftarrow \text{argmax}_{C \in \text{CHILDREN}_T(B)} |\text{SUBTREE}_T(C)|$
        end if
    end while
end function

GHOST Protocol Example



- Suppose an attacker secretly constructs the chain 1A, 2A,. . . , 6A
- All other blocks are mined by honest miners
- Honest miners' efforts are spread over multiple forks
- Longest chain rule gives 0,1B,2D,3F,4C,5B as main chain
  - Shorter than attacker's chain
- GHOST rule gives 0,1B,2C,3D,4B as main chain

Main Chain Selection and Uncle Rewards

- Chain with maximum total difficulty is chosen
  - Total difficulty is sum of block difficulty values
- Uncles contribute to difficulty since Oct 2017 (Byzantium)
- A uncle block of a given block satisfies the following
  - Cannot be a direct ancestor of given block
  - Cannot already be included as an uncle block in the past
  - Has to be the child of given block's ancestor at depth 2 to 7
- Mining reward
  - Block reward = 3 ETH, Nephew reward = $\frac{3}{32}$ ETH
  - Total reward to block miner is

$$\text{Block reward} + \text{NumUncles} \times \text{Nephew reward}$$

  - NumUncles can be at most 2
  - Uncle miner gets

$$\text{Block reward} \times \frac{(8 + \text{UncleHeight} - \text{BlockHeight})}{8}$$

Difficulty Adjustment

Difficulty Adjustment Algorithm Evolution

Frontier Release, July 2015

```
1  MIN_DIFF = 131072
2
3  def calc_difficulty(parent, timestamp):
4    offset = parent.difficulty // 2048
5    sign = 1 if timestamp - parent.timestamp < 13 else -1
6    return int(max(parent.difficulty + offset * sign, MIN_DIFF))
```

- If difference between current timestamp and parent's timestamp is less than 13 seconds, difficulty is increased
- Otherwise, difficulty is decreased
- Quantum of change is $\frac{1}{2048}$ of parent block's difficulty
- Difficulty is not allowed to go below a fixed minimum

Difficulty Adjustment Algorithm Evolution

Patch to Frontier Release, August 2015

```
1   MIN_DIFF = 131072
2   EXPDIFF_PERIOD = 100000
3   EXPDIFF_FREE_PERIODS = 2
4
5   def calc_difficulty(parent, timestamp):
6     offset = parent.difficulty // 2048
7     sign = 1 if timestamp - parent.timestamp < 13 else -1
8     o = int(max(parent.difficulty + offset * sign, MIN_DIFF))
9     period_count = (parent.number + 1) // EXPDIFF_PERIOD
10    if period_count >= EXPDIFF_FREE_PERIODS:
11      o = max(o + 2**(period_count - EXPDIFF_FREE_PERIODS),
            MIN_DIFF)
12    return o
```

- Difficulty time bomb was added to force move to proof-of-stake
- Bomb term added to every block's difficulty double every 100,000 blocks
- Ice age = Blocks too difficult to find

Ethereum Difficulty Chart

Image credit: `https://etherscan.io/chart/difficulty`

- Byzantium release (Oct 2017) delayed ice age by approximately 42 million seconds to account for PoS transition delays
- Other tweaks also done to target mean block time of 15 seconds

Blockchain Forks

- Temporary Forks
  - When two miners mine a block at almost the same time
- Soft forks and hard forks
  - Caused by changes to the consensus rules
  - Consensus rules = Rules determining validity of blocks and transactions
- Soft forks
  - Backward compatible rule changes
  - Nodes which do not upgrade still consider blocks produced under new rules valid
  - **Example:** Block size limit reduced to 500 KB from 1 MB
    - Sub-500 KB blocks produced by upgraded miners will be considered valid by non-upgraded nodes
    - Blocks with size larger than 500 KB produced by non-upgraded miners will be rejected by upgraded nodes
  - Soft fork success requires nodes controlling a majority of the hashpower to upgrade to new rules
- Hard forks
  - Not backward compatible rule changes
  - Hard fork success requires all nodes to upgrade

References

- Yellow paper https://ethereum.github.io/yellowpaper/paper.pdf
- Light client protocol https://github.com/ethereum/wiki/wiki/Light-client-protocol
- Ethash https://github.com/ethereum/wiki/wiki/Ethash
- Randmemohash http://www.hashcash.org/papers/memohash.pdf
- GHOST paper https://eprint.iacr.org/2013/881
- Uncle calculations https://github.com/ethereum/pyethereum/blob/develop/ethereum/pow/consensus.py
- Ethereum difficulty chart https://etherscan.io/chart/difficulty
- Byzantium difficulty adjustment https://blog.ethereum.org/2017/10/12/byzantium-hf-announcement/
- Article on forks www.mycryptopedia.com/hard-fork-soft-fork-explained/