How does Ethereum work

Odds are you've heard about the Ethereum blockchain, whether or not you know what it is. It's been in the news a lot lately, including the cover of some major magazines, but reading those articles can be like gibberish if you don't have a foundation for what exactly Ethereum is. So what is it? In essence, a public database that keeps a permanent record of digital transactions. Importantly, this database doesn't require any central authority to maintain and secure it. Instead it operates as a "trustless" transactional system—a framework in which individuals can make peer-to-peer transactions without needing to trust a third party OR one another.

Still confused? That's where this post comes in. My aim is to explain how Ethereum functions at a technical level, without complex math or scary-looking formulas. Even if you're not a programmer, I hope you'll walk away with at least better grasp of the tech. If some parts are too technical and difficult to grok, that's totally fine! There's really no need to understand every little detail. I recommend just focusing on understanding things at a broad level.

Many of the topics covered in this post are a breakdown of the concepts discussed in the yellow paper. I've added my own explanations and diagrams to make understanding Ethereum easier. Those brave enough to take on the technical challenge can also read the Ethereum yellow paper.

Let's get started!

Blockchain definition

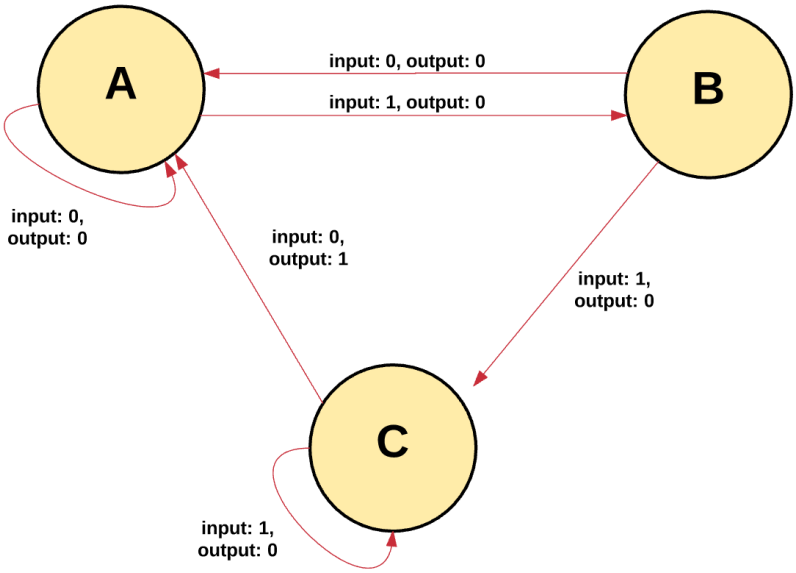
A blockchain is a "**cryptographically secure transactional singleton machine with shared-state."** [1] That's a mouthful, isn't it? Let's break it down.

- **"Cryptographically secure"** means that the creation of digital currency is secured by complex mathematical algorithms that are obscenely hard to break. Think of a firewall of sorts. They make it nearly impossible to cheat the system (e.g. create fake transactions, erase transactions, etc.)
- **"Transactional singleton machine"** means that there's a single canonical instance of the machine responsible for all the transactions being created in the system. In other words, there's a single global truth that everyone believes in.
- **"With shared-state"** means that the state stored on this machine is shared and open to everyone.

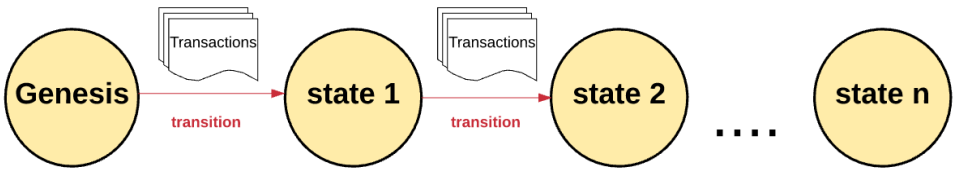
Ethereum implements this blockchain paradigm.

The Ethereum blockchain paradigm explained

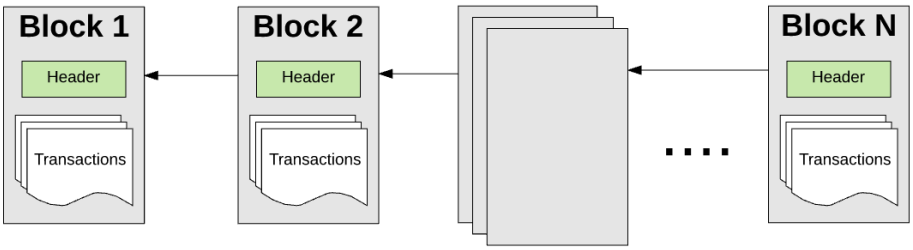
The Ethereum blockchain is essentially a **transaction-based state machine**. In computer science, a *state machine* refers to something that will read a series of inputs and, based on those inputs, will transition to a new state.



With Ethereum's state machine, we begin with a "genesis state." This is analogous to a blank slate, before any transactions have happened on the network. When transactions are executed, this genesis state transitions into some final state. At any point in time, this final state represents the current state of Ethereum.



The state of Ethereum has millions of transactions. These transactions are grouped into "blocks." A block contains a series of transactions, and each block is chained together with its previous block.



To cause a transition from one state to the next, a transaction must be valid. **For a transaction to be considered valid, it must go through a validation process known as mining.** Mining is when a group of nodes (i.e. computers) expend their compute resources to create a block of valid transactions.

Any node on the network that declares itself as a miner can attempt to create and validate a block. Lots of miners from around the world try to create and validate blocks at the same time. Each miner provides a mathematical “proof” when submitting a block to the blockchain, and this proof acts as a guarantee: if the proof exists, the block must be valid.

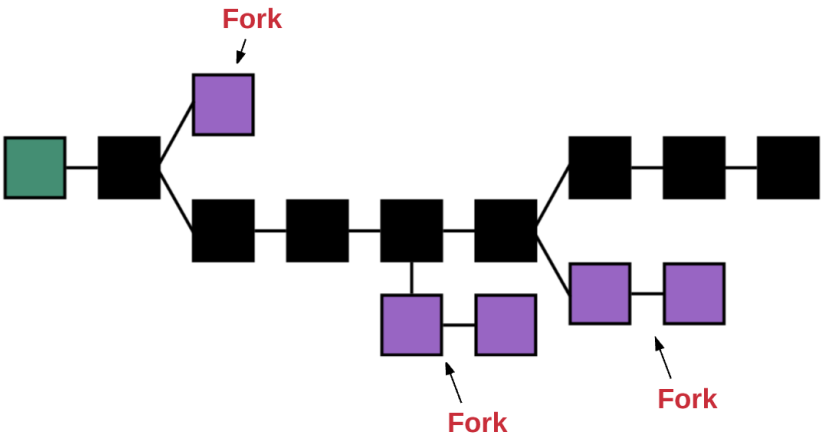
For a block to be added to the main blockchain, the miner must prove it faster than any other competitor miner. The process of validating each block by having a miner provide a mathematical proof is known as a **“proof of work.”**

A miner who validates a new block is rewarded with a certain amount of value for doing this work. What is that value? The Ethereum blockchain uses an intrinsic digital token called “Ether.” Every time a miner proves a block, new Ether tokens are generated and awarded.

You might wonder: what guarantees that everyone sticks to one chain of blocks? How can we be sure that there doesn’t exist a subset of miners who will decide to create their own chain of blocks?

Earlier, we defined a blockchain as a **transactional singleton machine with shared-state**. Using this definition, we can understand the correct current state is a single global truth, which everyone must accept. Having multiple states (or chains) would ruin the whole system, because it would be impossible to agree on which state was the correct one. If the chains were to diverge, you might own 10 coins on one chain, 20 on another, and 40 on another. In this scenario, there would be no way to determine which chain was the most “valid.”

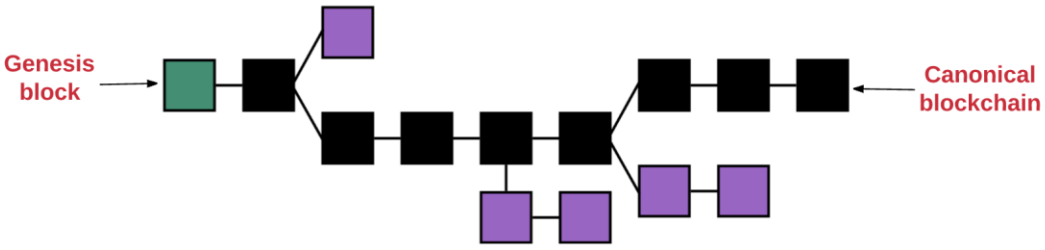
Whenever multiple paths are generated, a “fork” occurs. We typically want to avoid forks, because they disrupt the system and force people to choose which chain they “believe” in.



To determine which path is most valid and prevent multiple chains, Ethereum uses a mechanism called the **“GHOST protocol.”**

“GHOST” = “Greedy Heaviest Observed Subtree”

In simple terms, **the GHOST protocol says we must pick the path that has had the most computation done upon it**. One way to determine that path is to use the block number of the most recent block (the “leaf block”), which represents the total number of blocks in the current path (not counting the genesis block). The higher the block number, the longer the path and the greater the mining effort that must have gone into arriving at the leaf. Using this reasoning allows us to agree on the canonical version of the current state.



Now that you’ve gotten the 10,000-foot overview of what a blockchain is, let’s dive deeper into the main components that the Ethereum system is comprised of:

- accounts
- state
- gas and fees
- transactions
- blocks
- transaction execution
- mining
- proof of work

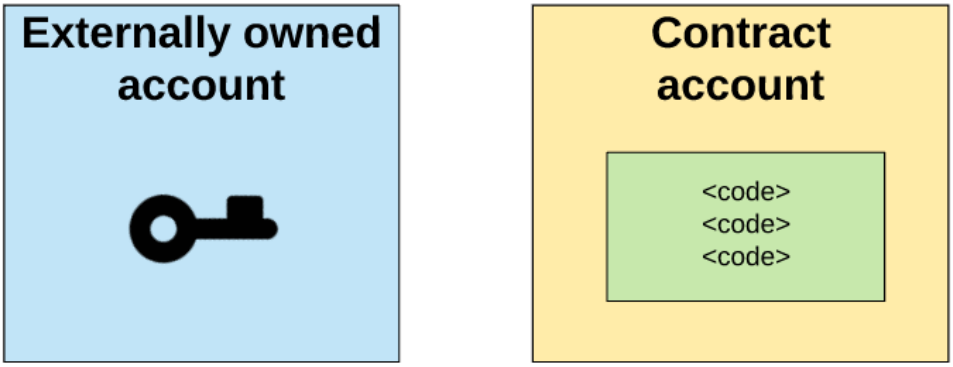
One note before getting started: whenever I say "hash" of X, I am referring to the KECCAK-256 hash, which Ethereum uses.

Accounts

The global "shared-state" of Ethereum is comprised of many small objects ("accounts") that are able to interact with one another through a message-passing framework. Each account has a **state** associated with it and a 20-byte **address**. An address in Ethereum is a 160-bit identifier that is used to identify any account.

There are two types of accounts:

- Externally owned accounts, which are **controlled by private keys** and have **no code associated with them.**
- Contract accounts, which are **controlled by their contract code** and **have code associated with them.**

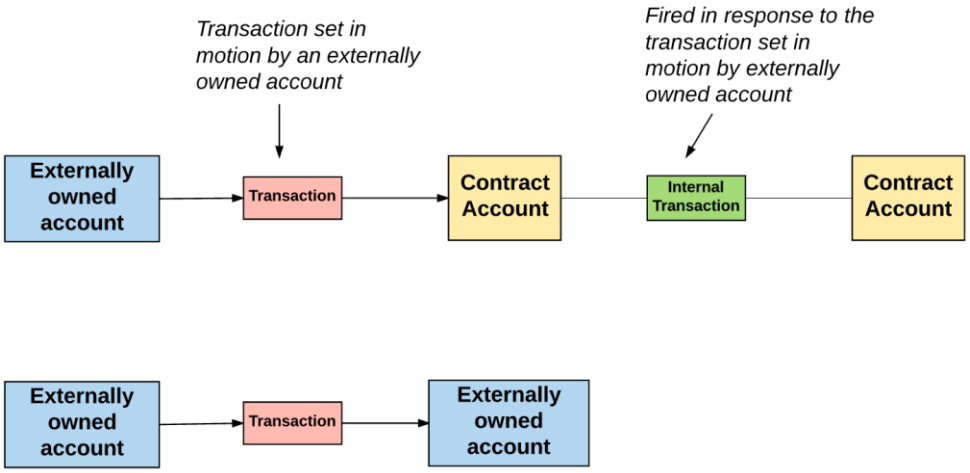


Externally owned accounts vs. contract accounts

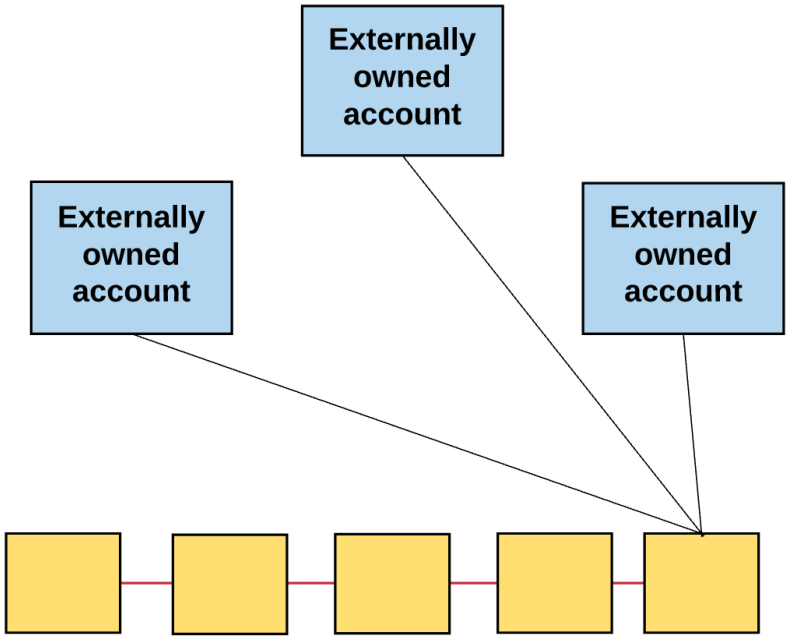
It's important to understand a fundamental difference between externally owned accounts and contract accounts. **An externally owned account can send messages to other externally owned accounts OR to other contract accounts by creating and signing a transaction using its private key.** A message between two externally owned accounts is simply a value transfer. But a message from an externally owned account to a contract account activates the contract account's code, allowing it to perform various actions (e.g. transfer tokens, write to internal storage, mint new tokens, perform some calculation, create new contracts, etc.).

Unlike externally owned accounts, contract accounts can't initiate new transactions on their own. Instead, contract accounts can only fire transactions in response to other transactions they have received (from an externally owned account or

from another contract account). We'll learn more about contract-to-contract calls in the "*Transactions and Messages*" section.



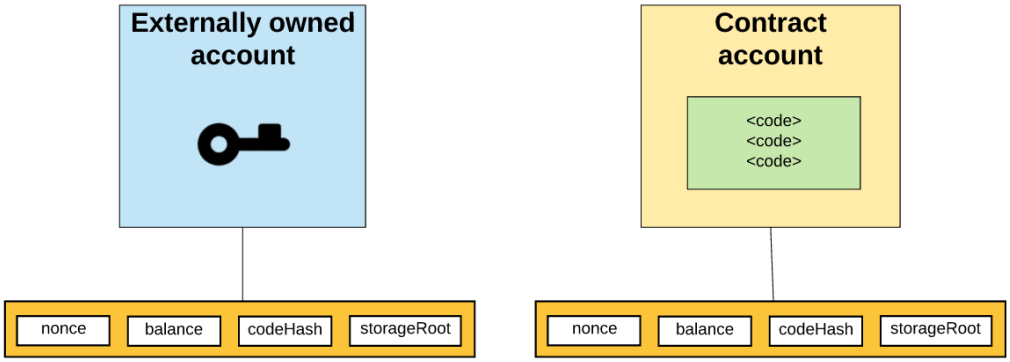
Therefore, any action that occurs on the Ethereum blockchain is always set in motion by transactions fired from externally controlled accounts.



Account state

The account **state** consists of four components, which are present regardless of the type of account:

- **nonce**: If the account is an externally owned account, this number represents the number of transactions sent from the account's address. If the account is a contract account, the nonce is the number of contracts created by the account.
- **balance**: The number of Wei owned by this address. There are 1e+18 Wei per Ether.
- **storageRoot**: A hash of the root node of a Merkle Patricia tree (we'll explain Merkle trees later on). This tree encodes the hash of the storage contents of this account, and is empty by default.
- **codeHash**: The hash of the EVM (Ethereum Virtual Machine—more on this later) code of this account. For contract accounts, this is the code that gets hashed and stored as the **codeHash**. For externally owned accounts, the **codeHash** field is the hash of the empty string.

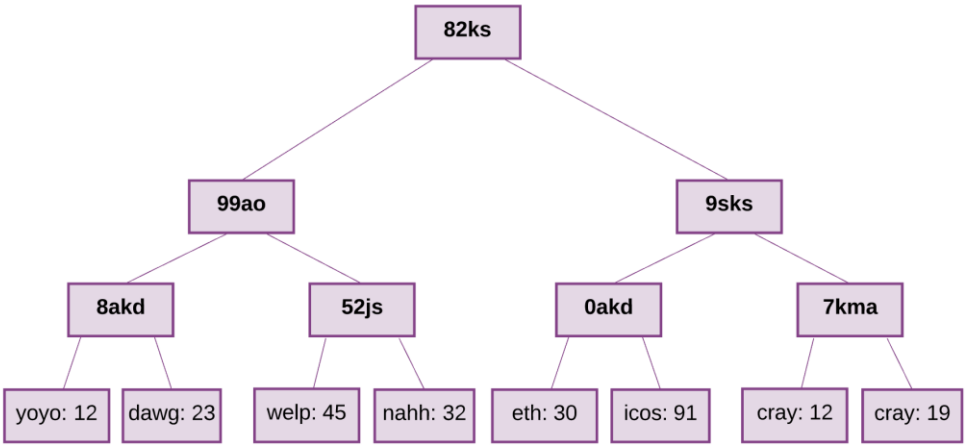


World state

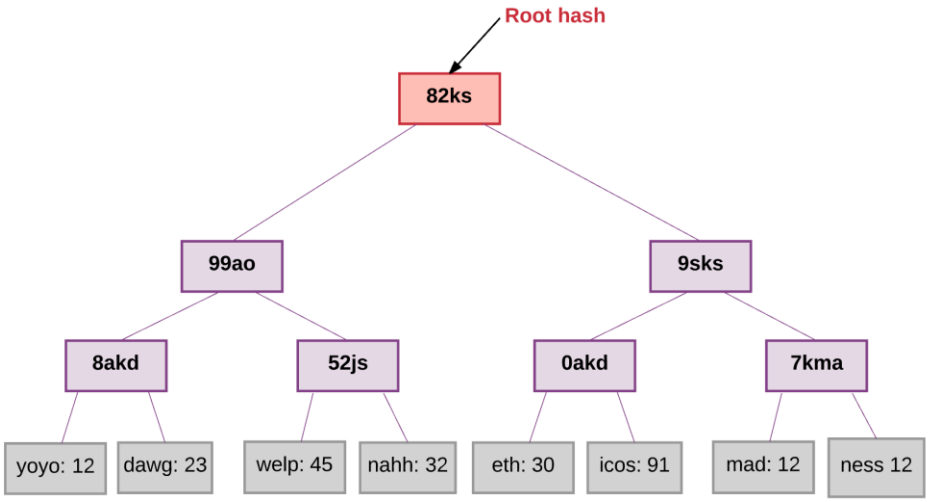
Okay, so we know that Ethereum's global state consists of a mapping between account addresses and the account states. This mapping is stored in a data structure known as a **Merkle Patricia tree**.

A Merkle tree (or also referred as "Merkle trie") is a type of binary tree composed of a set of nodes with:

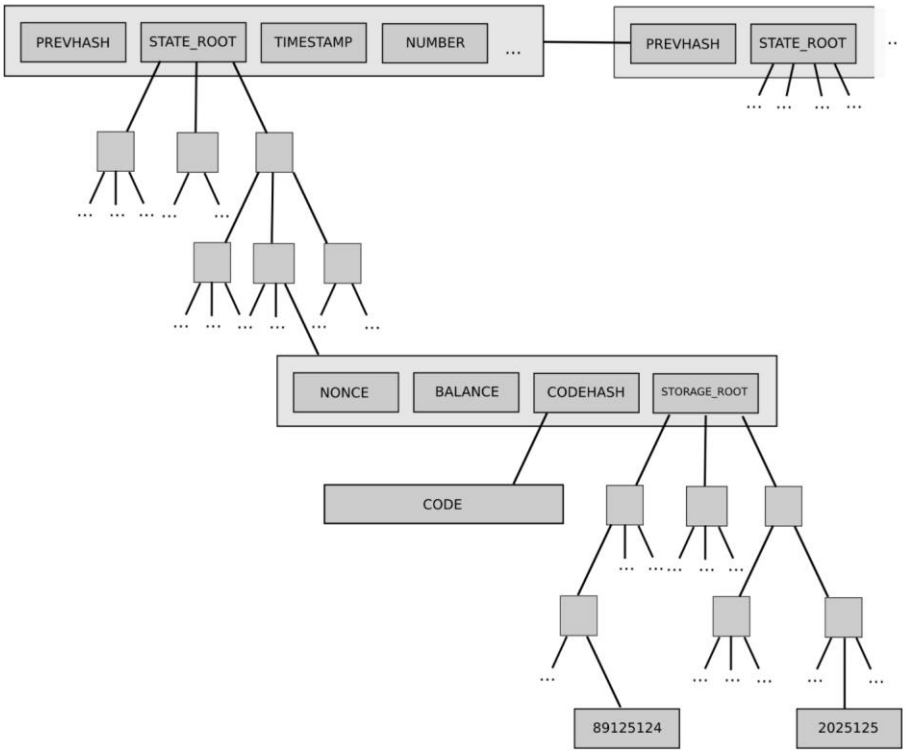
- a large number of leaf nodes at the bottom of the tree that contain the underlying data
- a set of intermediate nodes, where each node is the hash of its two child nodes
- a single root node, also formed from the hash of its two child node, representing the top of the tree



The data at the bottom of the tree is generated by splitting the data that we want to store into *chunks*, then splitting the chunks into *buckets*, and then taking the hash of each bucket and repeating the same process until the total number of hashes remaining becomes only one: **the root hash.**



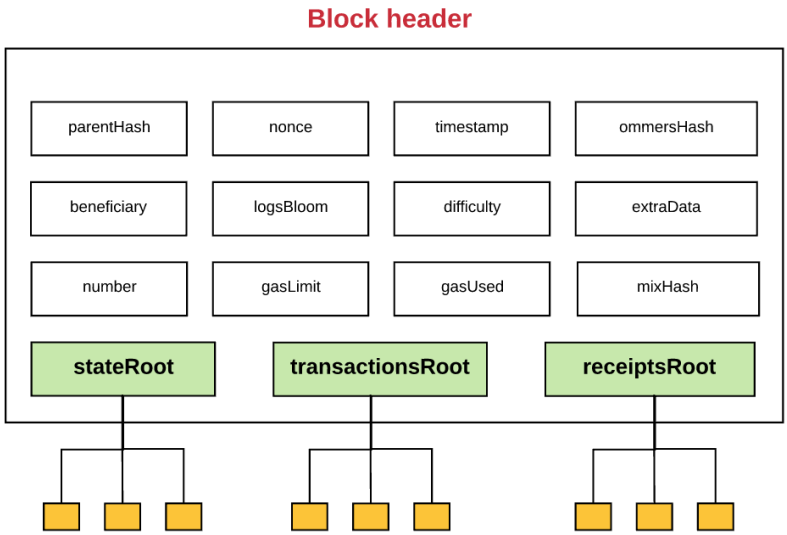
This tree is required to have a key for every value stored inside it. Beginning from the root node of the tree, the key should tell you which child node to follow to get to the corresponding value, which is stored in the leaf nodes. In Ethereum's case, the key/value mapping for the state tree is between addresses and their associated accounts, including the balance, nonce, codeHash, and storageRoot for each account (where the storageRoot is itself a tree).



Source: Ethereum whitepaper

This same trie structure is used also to store transactions and receipts. More specifically, every block has a “header” which stores the hash of the root node of three different Merkle trie structures, including:

1. State trie
2. Transactions trie
3. Receipts trie



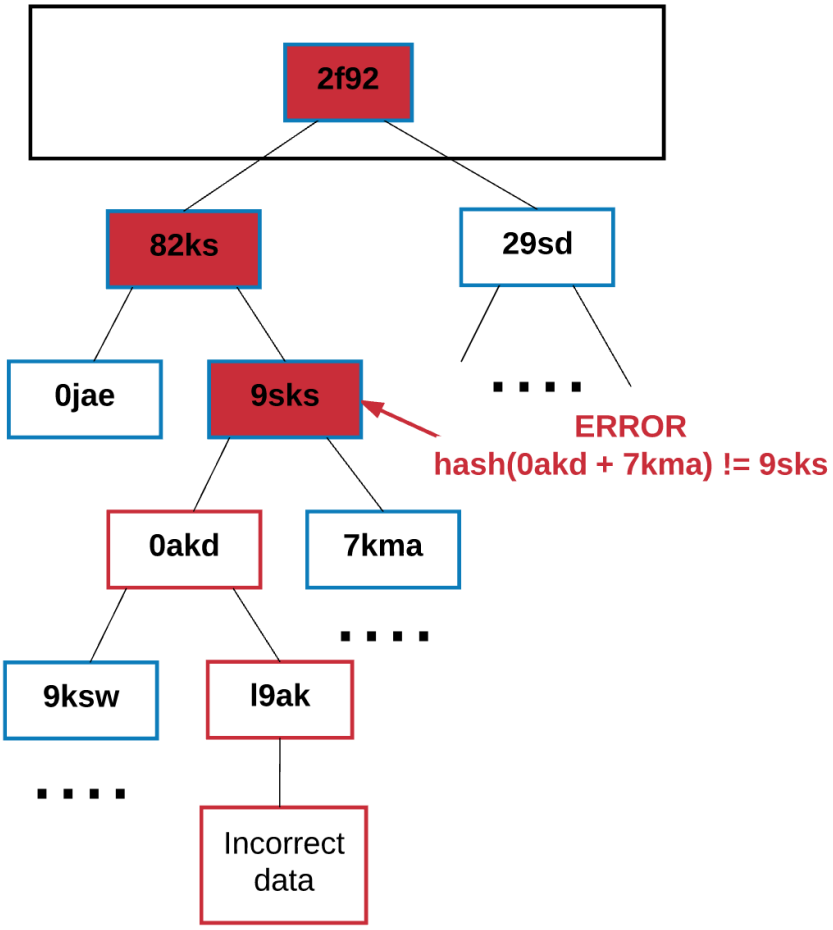
The ability to store all this information efficiently in Merkle tries is incredibly useful in Ethereum for what we call “light clients” or “light nodes.” Remember that a blockchain is

maintained by a bunch of nodes. Broadly speaking, there are two types of nodes: full nodes and light nodes.

A full archive node synchronizes the blockchain by downloading the full chain, from the genesis block to the current head block, executing all of the transactions contained within. Typically, miners store the full archive node, because they are required to do so for the mining process. It is also possible to download a full node without executing every transaction. Regardless, any full node contains the entire chain.

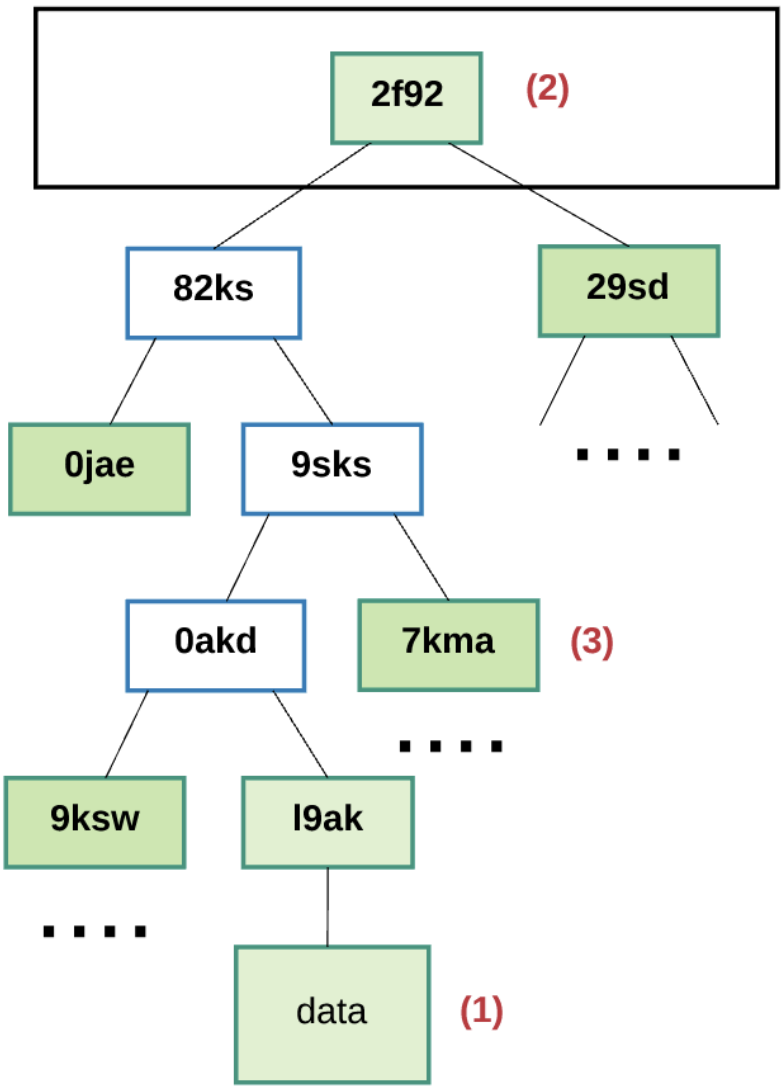
But unless a node needs to execute every transaction or easily query historical data, there's really no need to store the entire chain. This is where the concept of a light node comes in. **Instead of downloading and storing the full chain and executing all of the transactions, light nodes download only the chain of headers, from the genesis block to the current head, without executing any transactions or retrieving any associated state.** Because light nodes have access to block headers, which contain hashes of three tries, they can still easily generate and receive verifiable answers about transactions, events, balances, etc.

The reason this works is because hashes in the Merkle tree propagate upward—if a malicious user attempts to swap a fake transaction into the bottom of a Merkle tree, this change will cause a change in the hash of the node above, which will change the hash of the node above that, and so on, until it eventually changes the root of the tree.



Any node that wants to verify a piece of data can use something called a “Merkle proof” to do so. A Merkle proof consists of:

1. A chunk of data to be verified and its hash
2. The root hash of the tree
3. The “branch” (all of the partner hashes going up along the path from the chunk to the root)



Anyone reading the proof can verify that the hashing for that branch is consistent all the way up the tree, and therefore that the given chunk is actually at that position in the tree.

In summary, the benefit of using a Merkle Patricia tree is that the root node of this structure is cryptographically dependent on the data stored in the tree, and so the hash of the root node can be used as a secure identity for this data. Since the block header includes the root hash of the state, transactions, and receipts trees, any node can validate a small part of state of Ethereum without needing to store the entire state, which can be potentially unbounded in size.

Gas and payment

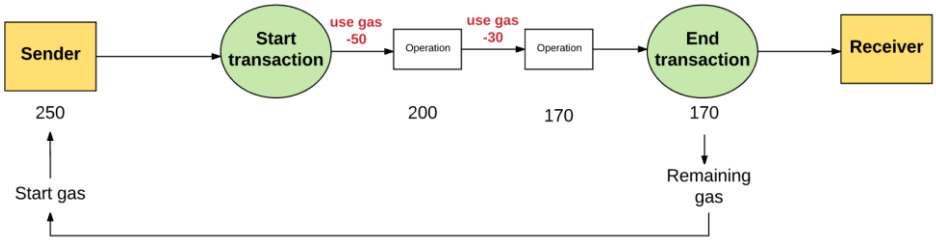
One very important concept in Ethereum is the concept of fees. **Every computation that occurs as a result of a transaction on the Ethereum network incurs a fee—there's no free lunch!** This fee is paid in a denomination called "gas."

Gas is the unit used to measure the fees required for a particular computation. **Gas price** is the amount of Ether you are willing to spend on every unit of gas, and is measured in “gwei.” “Wei” is the smallest unit of Ether, where 1^{018} Wei represents 1 Ether. One gwei is 1,000,000,000 Wei.

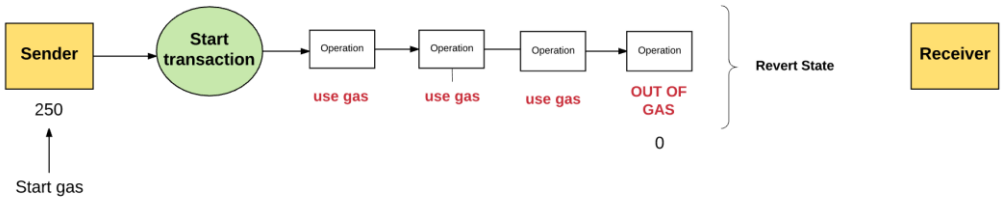
With every transaction, a sender sets a **gas limit** and **gas price**. The product of **gas price** and **gas limit** represents the maximum amount of Wei that the sender is willing to pay for executing a transaction.

For example, let’s say the sender sets the gas limit to 50,000 and a gas price to 20 gwei. This implies that the sender is willing to spend at most 50,000 x 20 gwei = 1,000,000,000,000,000 Wei = 0.001 Ether to execute that transaction.

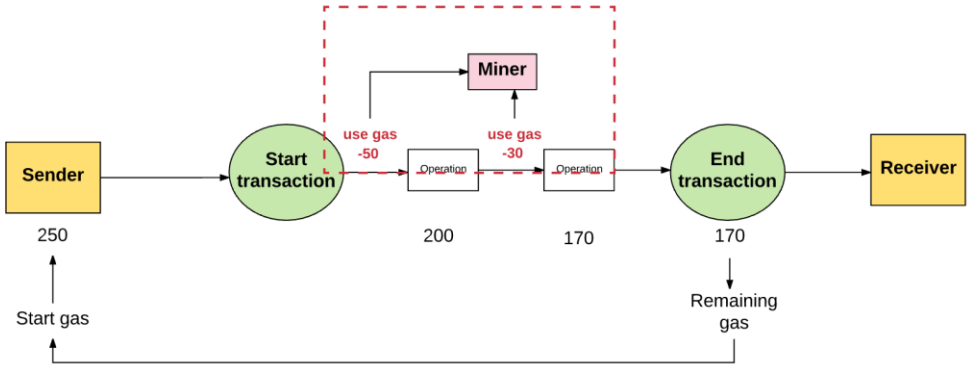
Remember that the gas limit represents the maximum gas the sender is willing to spend money on. If they have enough Ether in their account balance to cover this maximum, they’re good to go. The sender is refunded for any unused gas at the end of the transaction, exchanged at the original rate.



In the case that the sender does not provide the necessary gas to execute the transaction, the transaction runs “out of gas” and is considered invalid. In this case, the transaction processing aborts and any state changes that occurred are reversed, such that we end up back at the state of Ethereum prior to the transaction. Additionally, a record of the transaction failing gets recorded, showing what transaction was attempted and where it failed. And since the machine already expended effort to run the calculations before running out of gas, logically, **none of the gas is refunded to the sender.**



Where exactly does this gas money go? **All the money spent on gas by the sender is sent to the "beneficiary" address, which is typically the miner's address.** Since miners are expending the effort to run computations and validate transactions, miners receive the gas fee as a reward.



Typically, the higher the gas price the sender is willing to pay, the greater the value the miner derives from the transaction. Thus, the more likely miners will be to select it. In this way, miners are free to choose which transactions they want to validate or ignore. In order to guide senders on what gas price to set, miners have the option of advertising the minimum gas price for which they will execute transactions.

There are fees for storage, too

Not only is gas used to pay for computation steps, it is also used to pay for storage usage. The total fee for storage is proportional to the smallest multiple of 32 bytes used.

Fees for storage have some nuanced aspects. For example, since increased storage increases the size of the Ethereum state database on *all* nodes, there's an incentive to keep the amount of data stored small. For this reason, if a transaction has a step that clears an entry in the storage, the fee for executing that operation of is waived, AND a refund is given for freeing up storage space.

What's the purpose of fees?

One important aspect of the way the Ethereum works is that **every single operation executed by the network is simultaneously effected by every full node.** However, computational steps on the Ethereum Virtual Machine are very expensive. Therefore, Ethereum smart contracts are best used for simple tasks, like running simple business logic or verifying signatures and other cryptographic objects, rather than more complex uses, like file storage, email, or machine learning, which can put a strain on the network. **Imposing fees prevents users from overtaxing the network.**

Ethereum is a Turing complete language. (In short, a Turing machine is a machine that can simulate any computer algorithm (for those not familiar with Turing machines, check out this and this). This allows for loops and makes Ethereum susceptible to the halting problem, a problem in which you cannot determine whether or not a program will run infinitely. If there were no fees, a malicious actor could easily try to disrupt the network by executing an infinite loop within a transaction, without any repercussions. Thus, fees protect the network from deliberate attacks.

You might be thinking, "why do we also have to pay for storage?" Well, just like computation, storage on the Ethereum network is a cost that the entire network has to take the burden of.

Transaction and messages

We noted earlier that Ethereum is a **transaction-based state machine.** In other words, transactions occurring between different accounts are what move the global state of Ethereum from one state to the next.

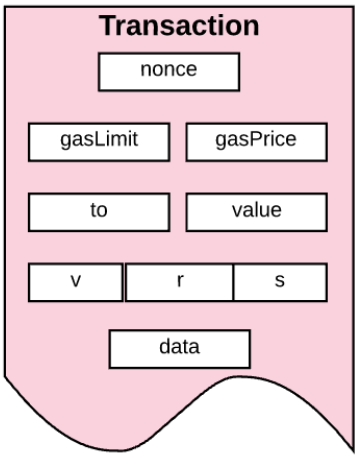
In the most basic sense, a transaction is a cryptographically signed piece of instruction that is generated by an externally owned account, serialized, and then submitted to the blockchain.

There are two types of transactions: **message calls** and **contract creations** (i.e. transactions that create new Ethereum contracts).

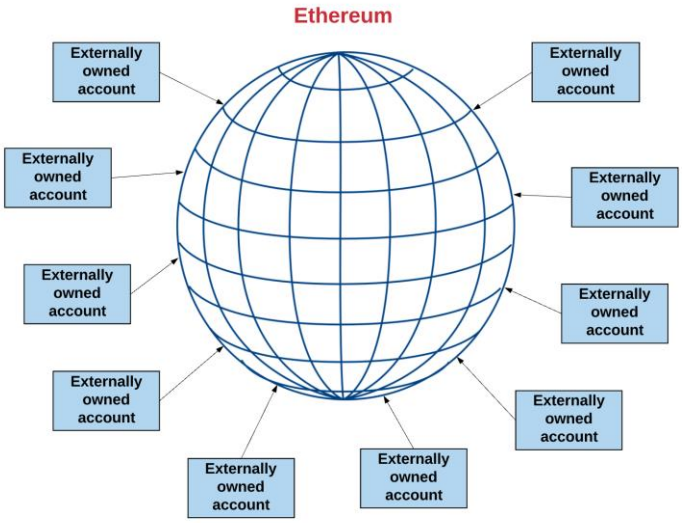
All transactions contain the following components, regardless of their type:

- **nonce**: a count of the number of transactions sent by the sender.
- **gasPrice**: the number of Wei that the sender is willing to pay per unit of gas required to execute the transaction.
- **gasLimit**: the maximum amount of gas that the sender is willing to pay for executing this transaction. This amount is set and paid upfront, before any computation is done.
- **to**: the address of the recipient. In a contract-creating transaction, the contract account address does not yet exist, and so an empty value is used.

- **value**: the amount of Wei to be transferred from the sender to the recipient. In a contract-creating transaction, this value serves as the starting balance within the newly created contract account.
- **v, r, s**: used to generate the signature that identifies the sender of the transaction.
- **init** (only exists for contract-creating transactions): An EVM code fragment that is used to initialize the new contract account. **init** is run only once, and then is discarded. When **init** is first run, it returns the body of the account code, which is the piece of code that is permanently associated with the contract account.
- **data** (optional field that only exists for message calls): the input data (i.e. parameters) of the message call. For example, if a smart contract serves as a domain registration service, a call to that contract might expect input fields such as the domain and IP address.

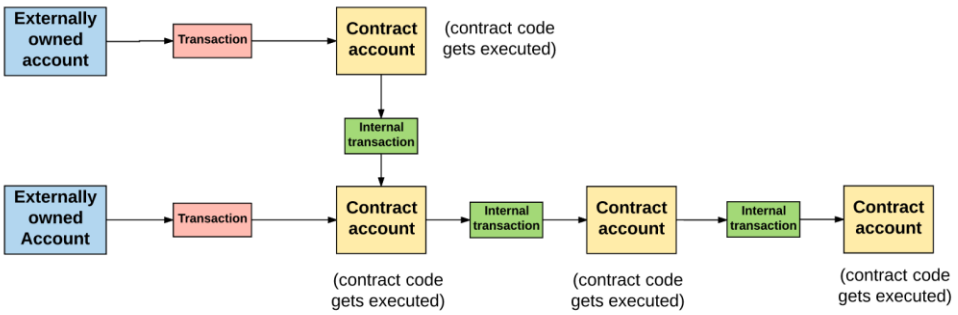


We learned in the "*Accounts*" section that transactions—both message calls and contract-creating transactions—are always initiated by externally owned accounts and submitted to the blockchain. Another way to think about it is that transactions are what bridge the external world to the internal state of Ethereum.



But this doesn't mean that contracts can't talk to other contracts. **Contracts that exist within the global scope of Ethereum's state can talk to other contracts within that same scope. The way they do this is via "messages" or "internal transactions" to other contracts.** We can think of messages or internal transactions as being similar to transactions, with the major difference that they are NOT generated by externally owned accounts. Instead, they are generated by contracts. They are virtual objects that, unlike transactions, are not serialized and only exist in the Ethereum execution environment.

When one contract sends an internal transaction to another contract, the associated code that exists on the recipient contract account is executed.



One important thing to note is that internal transactions or messages don't contain a **gasLimit.** This is because the gas limit is determined by the external creator of the original transaction (i.e. some externally owned account). The gas limit that the externally owned account sets must be high enough to carry out the transaction, including any sub-executions that occur as a result of that transaction, such as contract-to-contract messages. If, in the chain of transactions and messages, a particular message execution runs out of gas, then that message's execution will revert, along with any subsequent messages triggered by the execution. However, the parent execution does not need to revert.

Blocks

All transactions are grouped together into "blocks." A blockchain contains a series of such blocks that are chained together.

In Ethereum, a block consists of:

- the **block header**
- information about the **set of transactions** included in that block
- a **set of other block headers for the current block's ommers**.

Ommers explained

What the heck is an “ommer?” An ommer is a block whose parent is equal to the current block’s parent’s parent. Let’s take a quick dive into what ommers are used for and why a block contains the block headers for ommers.

Because of the way Ethereum is built, block times are much lower (~15 seconds) than those of other blockchains, like Bitcoin (~10 minutes). This enables faster transaction processing. However, one of the downsides of shorter block times is that more competing block solutions are found by miners. These competing blocks are also referred to as “orphaned blocks” (i.e. mined blocks do not make it into the main chain).

The purpose of ommers is to help reward miners for including these orphaned blocks. The ommers that miners include must be “valid,” meaning within the sixth generation or smaller of the present block. After six children, stale orphaned blocks can no longer be referenced (because including older transactions would complicate things a bit).

Ommer blocks receive a smaller reward than a full block. Nonetheless, there’s still some incentive for miners to include these orphaned blocks and reap a reward.

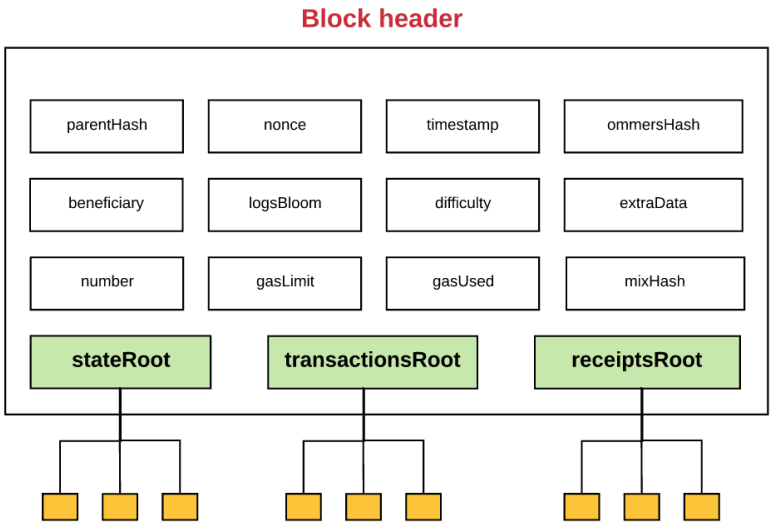
Block header

Let’s get back to blocks for a moment. We mentioned previously that every block has a block “header,” but what exactly is this?

A block header is a portion of the block consisting of:

- **parentHash**: a hash of the parent block’s header (this is what makes the block set a “chain”)
- **ommersHash**: a hash of the current block’s list of ommers
- **beneficiary**: the account address that receives the fees for mining this block
- **stateRoot**: the hash of the root node of the state trie (recall how we learned that the state trie is stored in the header and makes it easy for light clients to verify anything about the state)
- **transactionsRoot**: the hash of the root node of the trie that contains all transactions listed in this block
- **receiptsRoot**: the hash of the root node of the trie that contains the receipts of all transactions listed in this block
- **logsBloom**: a Bloom filter (data structure) that consists of log information
- **difficulty**: the difficulty level of this block
- **number**: the count of current block (the genesis block has a block number of zero; the block number increases by 1 for each each subsequent block)
- **gasLimit**: the current gas limit per block
- **gasUsed**: the sum of the total gas used by transactions in this block
- **timestamp**: the unix timestamp of this block’s inception
- **extraData**: extra data related to this block

- **mixHash**: a hash that, when combined with the nonce, proves that this block has carried out enough computation
- **nonce**: a hash that, when combined with the mixHash, proves that this block has carried out enough computation



Notice how every block header contains three trie structures for:

- state (**stateRoot**)
- transactions (**transactionsRoot**)
- receipts (**receiptsRoot**)

These trie structures are nothing but the Merkle Patricia tries we discussed earlier.

Additionally, there are a few terms from the above description that are worth clarifying. Let's take a look.

Logs

Ethereum allows for logs to make it possible to track various transactions and messages. A contract can explicitly generate a log by defining "events" that it wants to log.

A log entry contains:

- the logger's account address,
- a series of topics that represent various events carried out by this transaction, and
- any data associated with these events.

Logs are stored in a bloom filter, which stores the endless log data in an efficient manner.

Transaction receipt

Logs stored in the header come from the log information contained in the transaction receipt. Just as you receive a receipt when you buy something at a store, Ethereum generates a receipt for every transaction. Like you'd expect, each receipt contains certain information about the transaction. This receipt includes items like:

- the block number
- block hash
- transaction hash
- gas used by the current transaction
- cumulative gas used in the current block after the current transaction has executed
- logs created when executing the current transaction
- ..and so on

Block difficulty

The "difficulty" of a block is used to enforce consistency in the time it takes to validate blocks. The genesis block has a difficulty of 131,072, and a special formula is used to calculate the difficulty of every block thereafter. If a certain block is validated more quickly than the previous block, the Ethereum protocol increases that block's difficulty.

The difficulty of the block affects the **nonce,** which is a hash that must be calculated when mining a block, using the proof-of-work algorithm.

The relationship between the block's **difficulty** and **nonce** is mathematically formalized as:

$$n \leqslant \frac{2^{256}}{H_d}$$

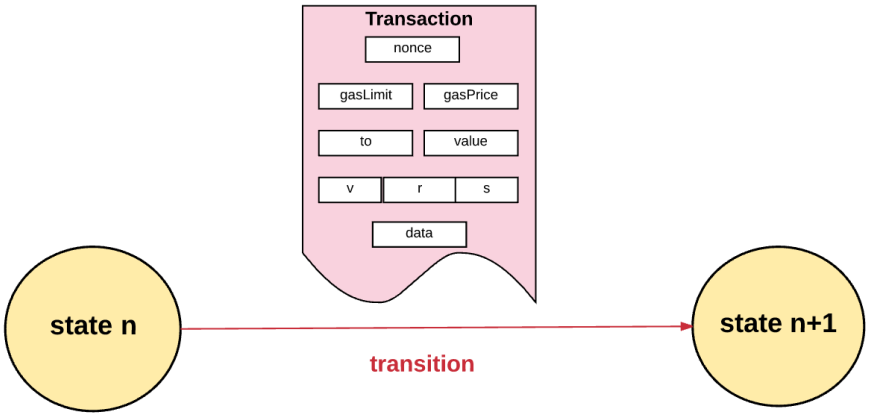
*where **Hd** is the difficulty.*

The only way to find a nonce that meets a difficulty threshold is to use the proof-of-work algorithm to enumerate all of the possibilities. The expected time to find a solution is proportional to the difficulty—the higher the difficulty, the harder it becomes to find the nonce, and so the harder it is to validate the block, which in turn increases the time it takes to validate a new block. **So, by adjusting the difficulty of a block, the protocol can adjust how long it takes to validate a block.**

If, on the other hand, validation time is getting slower, the protocol decreases the difficulty. In this way, the validation time self-adjusts to maintain a constant rate—on average, one block every 15 seconds.

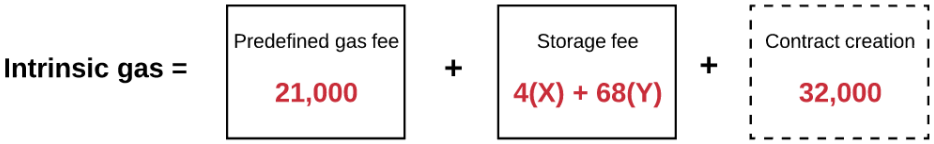
Transaction Execution

We've come to one of the most complex parts of the Ethereum protocol: the execution of a transaction. Say you send a transaction off into the Ethereum network to be processed. What happens to transition the state of Ethereum to include your transaction?

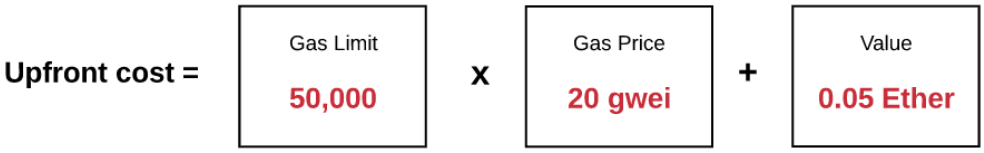


First, all transactions must meet an initial set of requirements in order to be executed. These include:

- The transaction must be a properly formatted **RLP**. "RLP" stands for "Recursive Length Prefix" and is a data format used to encode nested arrays of binary data. RLP is the format Ethereum uses to serialize objects.
- Valid transaction signature.
- Valid transaction nonce. Recall that the nonce of an account is the count of transactions sent from that account. To be valid, a transaction nonce must be equal to the sender account's nonce.
- The transaction's gas limit must be equal to or greater than the **intrinsic gas** used by the transaction. The intrinsic gas includes:
- a predefined cost of 21,000 gas for executing the transaction
- a gas fee for data sent with the transaction (4 gas for every byte of data or code that equals zero, and 68 gas for every non-zero byte of data or code)
- if the transaction is a contract-creating transaction, an additional 32,000 gas

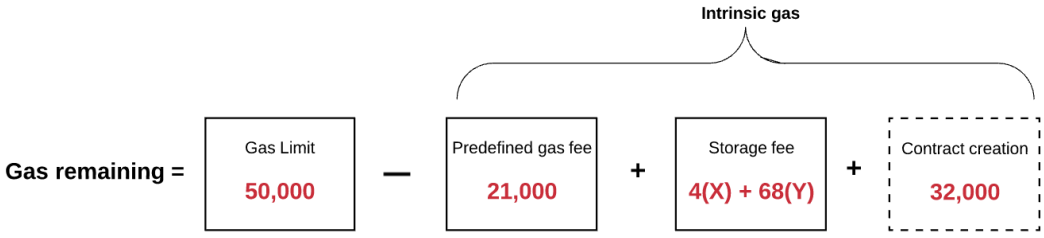


- The sender's account balance must have enough Ether to cover the **"upfront" gas costs** that the sender must pay. The calculation for the upfront gas cost is simple: First, the transaction's **gas limit** is multiplied by the transaction's **gas price** to determine the maximum gas cost. Then, this maximum cost is added to the total value being transferred from the sender to the recipient.



If the transaction meets all of the above requirements for validity, then we move onto the next step.

First, we deduct the upfront cost of execution from the sender's balance, and increase the nonce of the sender's account by 1 to account for the current transaction. At this point, we can calculate the gas remaining as the **total gas limit for the transaction minus the intrinsic gas used.**



Next, the transaction starts executing. Throughout the execution of a transaction, Ethereum keeps track of the "substate." This substate is a way to record information accrued during the transaction that will be needed immediately after the transaction completes. Specifically, it contains:

- **Self-destruct set**: a set of accounts (if any) that will be discarded after the transaction completes.
- **Log series**: archived and indexable checkpoints of the virtual machine's code execution.
- **Refund balance**: the amount to be refunded to the sender account after the transaction. Remember how we mentioned that storage in Ethereum costs

> money, and that a sender is refunded for clearing up storage? Ethereum keeps track of this using a refund counter. The refund counter starts at zero and increments every time the contract deletes something in storage.

Next, the various computations required by the transaction are processed.

Once all the steps required by the transaction have been processed, and assuming there is no invalid state, the state is finalized by determining the amount of unused gas to be refunded to the sender. In addition to the unused gas, the sender is also refunded some allowance from the "refund balance" that we described above.

Once the sender is refunded:

- the Ether for the gas is given to the miner
- the gas used by the transaction is added to the block gas counter (which keeps track of the total gas used by all transactions in the block, and is useful when validating a block)
- all accounts in the self-destruct set (if any) are deleted

Finally, we're left with the new state and a set of the logs created by the transaction.

Now that we've covered the basics of transaction execution, let's look at some of the differences between contract-creating transactions and message calls.

Contract creation

Recall that in Ethereum, there are two types of accounts: contract accounts and externally owned accounts. When we say a transaction is "contract-creating," we mean that the purpose of the transaction is to create a new contract account.

In order to create a new contract account, we first declare the address of the new account using a special formula. Then we initialize the new account by:

- Setting the nonce to zero
- If the sender sent some amount of Ether as *value* with the transaction, setting the account balance to that value
- Deducting the value added to this new account's balance from the sender's balance
- Setting the storage as empty
- Setting the contract's codeHash as the hash of an empty string

Once we initialize the account, we can actually create the account, using the **init code** sent with the transaction (see the "Transaction and messages" section for a refresher on the init code). What happens during the execution of this init code is varied. Depending on the constructor of the contract, it might update the account's storage, create other contract accounts, make other message calls, etc.

As the code to initialize a contract is executed, it uses gas. **The transaction is not allowed to use up more gas than the remaining gas. If it does, the execution will hit an out-of-gas (OOG) exception and exit. If the transaction exits due to an out-of-gas exception, then the state is reverted to the point immediately prior to transaction. The sender is *not* refunded the gas that was spent before running out.**

Boo hoo.

However, if the sender sent any Ether value with the transaction, the Ether value will be refunded even if the contract creation fails. Phew!

If the initialization code executes successfully, a final contract-creation cost is paid. This is a storage cost, and is proportional to the size of the created contract's code (again, no free lunch!) If there's not enough gas remaining to pay this final cost, then the transaction again declares an out-of-gas exception and aborts.

If all goes well and we make it this far without exceptions, then any remaining unused gas is refunded to the original sender of the transaction, and the altered state is now allowed to persist!

Hooray!

Message calls

The execution of a message call is similar to that of a contract creation, with a few differences.

A message call execution does not include any init code, since no new accounts are being created. However, it can contain input data, if this data was provided by the transaction sender. Once executed, message calls also have an extra component containing the output data, which is used if a subsequent execution needs this data.

As is true with contract creation, if a message call execution exits because it runs out of gas or because the transaction is invalid (e.g. stack overflow, invalid jump destination, or invalid instruction), none of the gas used is refunded to the original caller. Instead, all of the remaining unused gas is consumed, and the state is reset to the point immediately prior to balance transfer.

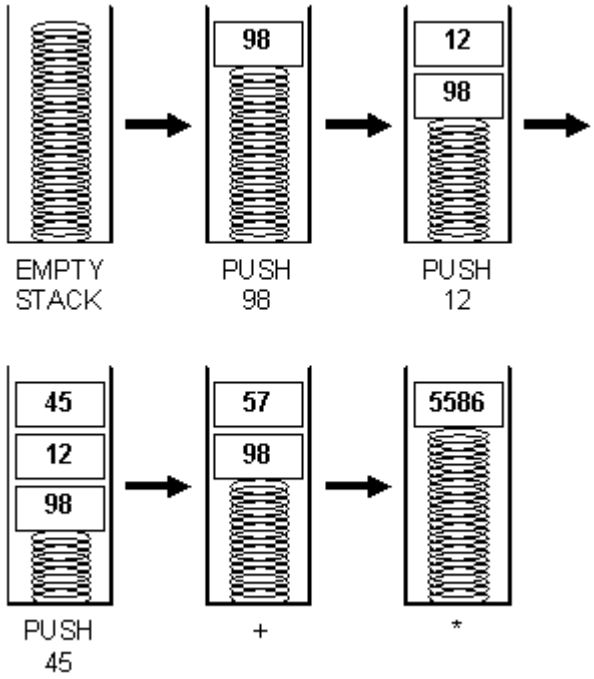
Until the most recent update of Ethereum, there was no way to stop or revert the execution of a transaction without having the system consume all the gas you provided. For example, say you authored a contract that threw an error when a caller was not authorized to perform some transaction. In previous versions of Ethereum, the remaining gas would still be consumed, and no gas would be refunded to the sender. **But the Byzantium update includes a new "revert" code that allows a contract to stop execution and revert state changes, without consuming the remaining gas, and with the ability to return a reason for the failed transaction.** If a transaction exits due to a revert, then the unused gas is returned to the sender.

Execution model

So far, we've learned about the series of steps that have to happen for a transaction to execute from start to finish. Now, we'll look at how the transaction actually executes within the VM.

The part of the protocol that actually handles processing the transactions is Ethereum's own virtual machine, known as the Ethereum Virtual Machine (EVM).

The EVM is a Turing complete virtual machine, as defined earlier. The only limitation the EVM has that a typical Turing complete machine does not is that the EVM is intrinsically bound by gas. Thus, the total amount of computation that can be done is intrinsically limited by the amount of gas provided.



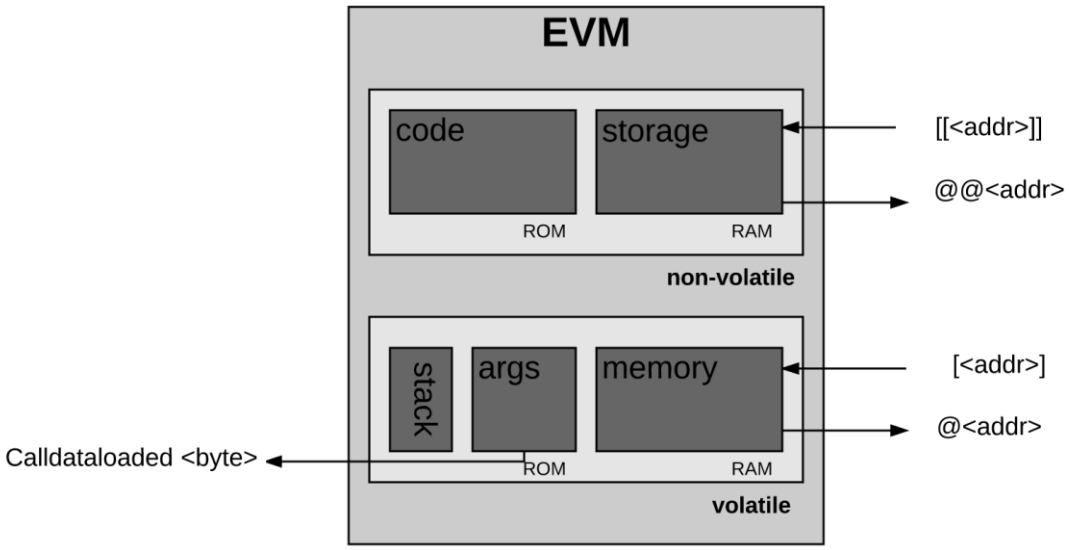
Source: CMU

Moreover, the EVM has a stack-based architecture. A stack machine is a computer that uses a last-in, first-out stack to hold temporary values.

The size of each stack item in the EVM is 256-bit, and the stack has a maximum size of 1024.

The EVM has memory, where items are stored as word-addressed byte arrays. Memory is volatile, meaning it is not permanent.

The EVM also has storage. Unlike memory, storage is non-volatile and is maintained as part of the system state. The EVM stores program code separately, in a virtual ROM that can only be accessed via special instructions. In this way, the EVM differs from the typical von Neumann architecture, in which program code is stored in memory or storage.



The EVM also has its own language: “EVM bytecode.” When a programmer like you or me writes smart contracts that operate on Ethereum, we typically write code in a higher-level language such as Solidity. We can then compile that down to EVM bytecode that the EVM can understand.

Okay, now on to execution.

Before executing a particular computation, the processor makes sure that the following information is available and valid:

- System state
- Remaining gas for computation
- Address of the account that owns the code that is executing
- Address of the sender of the transaction that originated this execution
- Address of the account that caused the code to execute (could be different from the original sender)
- Gas price of the transaction that originated this execution
- Input data for this execution
- Value (in Wei) passed to this account as part of the current execution
- Machine code to be executed
- Block header of the current block
- Depth of the present message call or contract creation stack

At the start of execution, memory and stack are empty and the program counter is zero.

PC: 0 STACK: [] MEM: [], STORAGE: {}

The EVM then executes the transaction recursively, computing the **system state** and the **machine state** for each loop. The system state is simply Ethereum’s global state. The machine state is comprised of:

- gas available
- program counter
- memory contents
- active number of words in memory
- stack contents.

Stack items are added or removed from the leftmost portion of the series.

On each cycle, the appropriate gas amount is reduced from the remaining gas, and the program counter increments.

At the end of each loop, there are three possibilities:

1. The machine reaches an exceptional state (e.g. insufficient gas, invalid instructions, insufficient stack items, stack items would overflow above 1024, invalid JUMP/JUMPI destination, etc.) and so must be halted, with any changes discarded
2. The sequence continues to process into the next loop
3. The machine reaches a controlled halt (the end of the execution process)

Assuming the execution doesn't hit an exceptional state and reaches a "controlled" or normal halt, the machine generates the resultant state, the remaining gas after this execution, the accrued substate, and the resultant output.

Phew. We got through one of the most complex parts of Ethereum. Even if you didn't fully comprehend this part, that's okay. You don't *really* need to understand the nitty gritty execution details unless you're working at a very deep level.

How a block gets finalized

Finally, let's look at how a block of many transactions gets finalized.

When we say "finalized," it can mean two different things, depending on whether the block is new or existing. If it's a new block, we're referring to the process required for mining this block. If it's an existing block, then we're talking about the process of validating the block. In either case, there are four requirements for a block to be "finalized":

1) Validate (or, if mining, determine) ommers
Each ommer block within the block header must be a valid header and be within the sixth generation of the present block.

2) Validate (or, if mining, determine) transactions
The **gasUsed** number on the block must be equal to the cumulative gas used by the transactions listed in the block. (Recall that when executing a transaction, we keep track of the block gas counter, which keeps track of the total gas used by all transactions in the block).

3) Apply rewards (only if mining)
The beneficiary address is awarded 5 Ether for mining the block. (Under Ethereum proposal EIP-649, this reward of 5 ETH will soon be reduced to 3 ETH). Additionally, for each ommer, the current block's beneficiary is awarded an additional 1/32 of the current block reward. Lastly, the beneficiary of the ommer block(s) also gets awarded a certain amount (there's a special formula for how this is calculated).

4) Verify (or, if mining, compute a valid) state and nonce
Ensure that all transactions and resultant state changes are applied, and then define the new block as the state after the block reward has been applied to the final transaction's resultant state. Verification occurs by checking this final state against the state trie stored in the header.

Mining proof of work

The "*Blocks*" section briefly addressed the concept of block difficulty. The algorithm that gives meaning to block difficulty is called Proof of Work (PoW).

Ethereum's proof-of-work algorithm is called "Ethash" (previously known as Dagger-Hashimoto).

The algorithm is formally defined as:

$$(m, n) = \mathrm{PoW}(H_{\not n}, H_n, \mathbf{d})$$

*where **m** is the **mixHash**, **n** is the **nonce**, **Hn** is the new block's header (excluding the **nonce** and **mixHash** components, which have to be computed), **Hn** is the nonce of the block header, and **d** is the DAG, which is a large data set.*

In the "*Blocks*" section, we talked about the various items that exist in a block header. Two of those components were called the **mixHash** and the **nonce**. As you may recall:

- **mixHash** is a hash that, when combined with the nonce, proves that this block has carried out enough computation
- **nonce** is a hash that, when combined with the mixHash, proves that this block has carried out enough computation

The PoW function is used to evaluate these two items.

How exactly the **mixHash** and **nonce** are calculated using the PoW function is somewhat complex, and something we can delve deeper into in a separate post. But at a high level, it works like this:

A "seed" is calculated for each block. This seed is different for every "epoch," where each epoch is 30,000 blocks long. For the first epoch, the seed is the hash of a series of 32

bytes of zeros. For every subsequent epoch, it is the hash of the previous seed hash. Using this seed, a node can calculate a pseudo-random “cache.”

This cache is incredibly useful because it enables the concept of “light nodes,” which we discussed previously in this post. The purpose of light nodes is to afford certain nodes the ability to efficiently verify a transaction without the burden of storing the entire blockchain dataset. A light node can verify the validity of a transaction based solely on this cache, because the cache can regenerate the specific block it needs to verify.

Using the cache, a node can generate the DAG “dataset,” where each item in the dataset depends on a small number of pseudo-randomly-selected items from the cache. In order to be a miner, you must generate this full dataset; all full clients and miners store this dataset, and the dataset grows linearly with time.

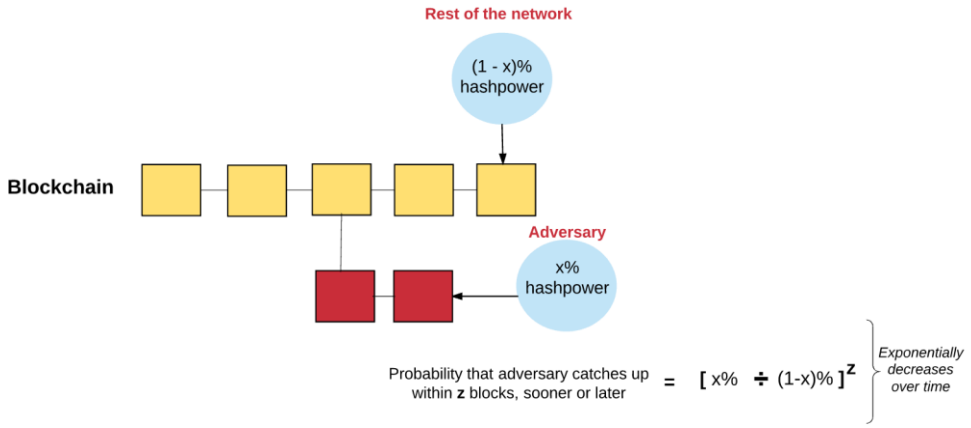
Miners can then take random slices of the dataset and put them through a mathematical function to hash them together into a “**mixHash.**” A miner will repeatedly generate a **mixHash** until the output is below the desired target **nonce**. When the output meets this requirement, this nonce is considered valid and the block can be added to the chain.

Mining as a security mechanism

Overall, the purpose of the PoW is to prove, in a cryptographically secure way, that a particular amount of computation has been expended to generate some output (i.e. the **nonce**). **This is because there is no better way to find a nonce that is below the required threshold other than to enumerate all the possibilities.** The outputs of repeatedly applying the hash function have a uniform distribution, and so we can be assured that, on average, **the time needed to find such a nonce depends on the difficulty threshold.** The higher the difficulty, the longer it takes to solve for the nonce. In this way, **the PoW algorithm gives meaning to the concept of difficulty, which is used to enforce blockchain security.**

What do we mean by blockchain security? It’s simple: we want to create a blockchain that EVERYONE trusts. As we discussed previously in this post, if more than one chain existed, users would lose trust, because they would be unable to reasonably determine which chain was the “valid” chain. In order for a group of users to accept the underlying state that is stored on a blockchain, we need a single canonical blockchain that a group of people believes in.

This is exactly what the PoW algorithm does: it ensures that a particular blockchain will remain canonical into the future, making it incredibly difficult for an attacker to create new blocks that overwrite a certain part of history (e.g. by erasing transactions or creating fake transactions) or maintain a fork. To have their block validated first, an attacker would need to consistently solve for the nonce faster than anyone else in the network, such that the network believes their chain is the heaviest chain (based on the principles of the GHOST protocol we mentioned earlier). This would be impossible unless the attacker had more than half of the network mining power, a scenario known as the majority 51% attack.



Mining as a wealth distribution mechanism

Beyond providing a secure blockchain, PoW is also a way to distribute wealth to those who expend their computation for providing this security. Recall that a miner receives a reward for mining a block, including:

- a *static block reward* of 5 ether for the “winning’” block (soon to be changed to 3 ether)
- the cost of gas expended within the block by the transactions included in the block
- an extra reward for including ommers as part of the block

In order to ensure that the use of the PoW consensus mechanism for security and wealth distribution is sustainable in the long run, Ethereum strives to instill these two properties:

- Make it accessible to as many people as possible. In other words, people shouldn’t need specialized or uncommon hardware to run the algorithm. The purpose of this is to make the wealth distribution model as open as possible so that anyone can provide any amount of compute power in return for Ether.
- Reduce the possibility for any single node (or small set) to make a disproportionate amount of profit. Any node that can make a disproportionate amount of profit means that the node has a large influence on determining the canonical blockchain. This is troublesome because it reduces network security.

In the Bitcoin blockchain network, one problem that arises in relation to the above two properties is that the PoW algorithm is a SHA256 hash function. The weakness with this type of function is that it can be solved much more efficiently using specialized hardware, also known as ASICs.

In order to mitigate this issue, Ethereum has chosen to make its PoW algorithm (Ethhash) sequentially memory-hard. This means that the algorithm is engineered so that calculating the nonce requires a lot of memory AND bandwidth. The large memory requirements make it hard for a computer to use its memory in parallel to discover

multiple nonces simultaneously, and the high bandwidth requirements make it difficult for even a super-fast computer to discover multiple nonce simultaneously. This reduces the risk of centralization and creates a more level playing field for the nodes that are doing the verification.

One thing to note is that Ethereum is transitioning from a PoW consensus mechanism to something called “proof-of-stake”. This is a beastly topic of its own that we can hopefully explore in a future post. ☺

Conclusion

…Phew! You made it to the end. I hope?

There’s a lot to digest in this post, I know. If it takes you multiple reads to fully understand what’s going on, that’s totally fine. I personally read the Ethereum yellow paper, white paper, and various parts of the code base many times before grokking what was going on.

Nonetheless, I hope you found this overview helpful. If you find any errors or mistakes, I’d love for you to write a private note or post it directly in the comments. I look at all of ’em, I promise ;)

And remember, I’m human (yep, it’s true) and I make mistakes. I took the time to write this post for the benefit of the community, for free. So please be constructive in your feedback, without unnecessary bashing.

原文地址：https://www.preethikasireddy.com/post/how-does-ethereum-work-anyway