

[Article](#) [ethereum](#)

# Transaction Execution - Ethereum Yellow Paper Walkthrough (4/7)

**Lucas Saldanha**

05 Nov 2019 • 9 min read

In this post, we will look at how the Ethereum platform executes transactions. We will learn the transaction validity rules and why they exist. After that, we will deep-dive into the transaction execution and understand the steps taken by the node while processing a transaction.

This post is the fourth in the series Ethereum Yellow Paper Walkthrough. The goal of this series is to demystify the

concepts in the paper, making it accessible to a broader audience. If you missed the previous posts, here they are!

- [The blockchain paradigm - Ethereum Yellow Paper Walkthrough \(1/7\)](#)
- [Merkle Tree and Ethereum Objects \(World State, Transaction, Block, etc.\) - Ethereum Yellow Paper Walkthrough \(2/7\)](#)
- [Gas and Payment - Ethereum Yellow Paper Walkthrough \(3/7\)](#)

*(DISCLAIMER: this post is based on the Byzantium version of the [Yellow Paper](#), version 7e819ec from 20th October 2019)*

## Introduction

Along with the series, we talked about how Ethereum works as a distributed computer. We also talked about how a user interacts with this machine sending transactions to the system (and paying the associated cost of these transactions).

In the first post of the series, we learned about the state transition function, and how a sequence of state transitions can represent the Ethereum computer.

At the simplest level, the state transition function takes the current state and a transaction to compute the next state.



$$\sigma_{t+1} \equiv \mathbf{1}(\sigma_t, \mathbf{I})$$

Ethereum state transition function

Now, it is time to look under the hood and understand how the Ethereum node executes a transaction. In this first part, let's understand how a transaction is validated.

## Transaction Validity

Before executing a transaction, the node validates that a transaction passes a list of essential (intrinsic) rules. If a transaction doesn't pass one of the five rules, the node won't even try to execute the transaction.

The list of intrinsic rules is as follows:

1. The transaction is a well-formed RLP.
2. The transaction signature is valid.
3. The transaction nonce is valid (same as the transaction sender account current nonce).
4. The transaction's intrinsic cost is lower than the transaction gas limit.
5. The transaction sender account balance is equal to or higher than the required upfront payment for the transaction.

There is also one rule that isn't part of the intrinsic group. It

states that the transaction must not be included in a block if, by including it, the total gas limit of all transactions in the block exceeds the block's gas limit.

Let's take a look into each one of the rules to understand how they work and why they exist.

## Well-formed RLP

This rule is probably the most straight forward to understand. RLP or Recursive Length Prefix is an encoding method used to serialize objects in Ethereum. As any other encoding method, if you don't follow the rules when encoding an object, decoding it fails, and you won't be able to retrieve the original object from the encoded data.

The purpose of this rule is to ensure that any Ethereum client that receives the transaction can decode it successfully and execute it.

## Valid transaction signature

Imagine that you have an Ethereum account with lots of Ether. What would happen if someone creates a transfer transactions from your account to their own? I am sure you wouldn't be happy with someone impersonating you and stealing your money. That is why transaction signatures are necessary.

Ethereum uses asymmetric cryptography to ensure that only a user in control of an account can send transactions from it

USER IN CONTROL OF AN ACCOUNT CAN SEND TRANSACTIONS FROM IT.

Also, this same cryptography enables anyone to verify that the transaction has been sent by an authorized user.

I won't get into the details of how ECDSA (Ethereum's asymmetric cryptography algorithm of choice) works. All that we need is the basics.

In asymmetric cryptography, there is a public key and a private key. The private key should be kept a secret while the public key can be shared with anyone. The private key is used to create a signature that can be verified by anyone with the corresponding public key. In the same way that you would sign a letter to prove that you wrote it, you would sign an Ethereum transaction to prove that you created that transaction. Luckily, a cryptographic signature isn't as easy to forge as our hand signatures.

In Ethereum, your account address is derived from your public key. When sending transactions, the private key is used to sign the transaction (remember the values  $v$ ,  $r$  and  $s$  in the transaction object?). All other nodes can then determine if the owner of the private key associated with the sender account actually signed the transaction.

As you can imagine, there is no point in executing transactions that don't have a valid signature, hence why a valid transaction signature is one of the intrinsic validity rules of a transaction.

## Transaction nonce matching sender account nonce

In Ethereum, the account nonce represents the number of transactions sent by that account (or, if the account is a contract account, the number of contract creations). Without the nonce, it would be possible to execute the same signed transaction multiple times (this is called replay attack). Also, due to Ethereum's distributed nature, different nodes might try to include the same transaction into different blocks, duplicating the same transaction in the chain. Imagine if this duplicated transaction is sending Ether from your account to someone else's account. How would you feel about sending money to someone twice?

Whenever the user creates a new transaction, they set the transaction nonce to match its current account nonce. During the transaction execution, the node checks that the transaction nonce matches the account nonce.

If for some reason the same transaction is resubmitted to the node, because the account nonce has been incremented, this new transaction won't be valid anymore.

By enforcing that a transaction nonce matches the account nonce, Ethereum ensures that no replay attacks aren't possible and that a transaction can only be executed and change state once.

## **Transaction intrinsic cost lower than transaction gas limit**

In our [previous post](#), we discussed why the user needs to pay to use Ethereum and the concept of gas. In summary, every transaction in Ethereum has a gas cost associated with it. The cost of a transaction has two parts: the **intrinsic cost** and the **execution cost**.

The **execution cost** of a transaction is calculated based on the operations performed by the EVM to execute the transaction. The more operations are required to execute the transaction, the more expensive executing the transaction is.

The **intrinsic cost** of a transaction is calculated based on the transaction payload. The transaction payload can be one of these:

- The EVM code to create a contract, if the transaction is creating a smart contract;
- The input data for a message execution, if the transaction is calling a method in a smart contract;
- Empty, if the transaction is only transferring value between two accounts;

Let  $N_{\text{zeros}}$  be the total number of zero bytes of data in the transaction payload, and  $N_{\text{nonzeros}}$  the total number of non-zero bytes of data in the transaction payload. The following formula represents the calculation of the transaction's intrinsic cost (Section 6.2 in the Yellowpaper, equations 54, 55 and 56):

$$\text{Intrinsic Cost} = \text{CreateTransaction} + \text{CreateDataGas} * N_{\text{nonzeros}}$$

$$\text{INTRINSIC COST} = G_{\text{transaction}} + G_{\text{datazero}} * \text{Nzeros} +$$

$$G_{\text{datanonzero}} * \text{Nnonzeros} + G_{\text{txcreate}}$$

In Appendix G, there is a Fee Schedule that contains all associated costs for creating and executing a transaction. The relevant part for the intrinsic cost is the following:

- $G_{\text{transaction}} = 21,000 \text{ Wei}$
- $G_{\text{txcreate}} = 32,000 \text{ Wei}$
- $G_{\text{txdatazero}} = 4 \text{ Wei}$
- $G_{\text{txdatanonzero}} = 68 \text{ Wei}$  (this changed in the Istanbul release to 16 Wei)

Now that we understand what the transaction's intrinsic cost is, we can understand why a transaction with an intrinsic cost higher than the gas limit is considered invalid. The gas limit of a transaction defines the maximum amount of gas that the transaction execution can spend. If before execution the transaction cost higher than the gas limit, there is no point in trying to execute the transaction.

## **Sender account balance equal to or higher than required upfront payment**

The transaction upfront cost is the amount of gas deducted from the sender's account balance at the beginning of the transaction execution.

The transaction upfront cost is calculated by the following formula:



$$\textit{Upfront Cost} = \textit{gasLimit} * \textit{gasPrice} + \textit{value}$$

The transaction gas limit is the maximum amount of gas that the sender is willing to spend on the transaction execution.

The transaction gas price is the value paid per unit of gas. The transaction value is the amount of Wei sent to the message call's recipient (e.g. transferring value) or as an endowment to the contract being created. To understand better what is gas and why it is used to execute a transaction, check out [our previous post](#).

Because the upfront cost is deducted at the beginning of the transaction execution, it is pointless to start executing a transaction from a sender which account balance is lower than the upfront.

## Transaction gas limit not exceeding block gas limit

This rule isn't one of the intrinsic rules. However, it is an essential rule for the node that is selecting transactions to include in a block. The block gas limit is the total gas that "fit" in a block.

When selecting transactions for building the next block, the node should check if by adding the candidate transaction it won't go over the block gas limit. For a transaction to be included in this block, **the sum of the gas used for all other transactions in the block plus the gas limit of the transaction**

**being evaluated must be lower or equal the block gas limit.**

Note that even if a transaction can't be included the current block being created, it might be eligible for a future block.

## Transaction Execution

After validating the transaction, it is time to execute it. In Ethereum, the state is changed by executing transactions. The transactions are grouped in blocks. Each block represents a list of the transactions that, when executed in order, result in a valid new state.

For each transaction execution, the following steps are taken:

1. Increment sender account nonce by one.
2. Reduce the sender balance by part of the up-front cost ( $\text{gasLimit} * \text{gasPrice}$ ).
3. Define the gas available for the execution ( $\text{gasLimit} - \text{intrinsic cost}$ )
4. Execute the transaction operations (value transfer, message call or contract creation)
5. Refund sender for SELFDESTRUCT and SSTORE operations.
6. Refund any remaining gas to the sender.
7. Pay mining fees to the beneficiary account. The miner of the block that includes this transaction chooses the beneficiary account (usually an account that belongs to the miner).

## Increment sender account nonce

Every time a transaction is executed, the sender account nonce must be incremented. This increment happens at the beginning of the execution.

## Reduce sender balance by upfront gas cost

We deduct the upfront gas cost from the sender's account balance. This mechanic is simple: the sender pays for the agreed transaction cost ( $\text{gasLimit} * \text{gasPrice}$ ).

## Calculate gas available for execution

The gas available for the transaction execution is the total of gas available for the transaction ( $\text{gasLimit}$ ) subtracted by the intrinsic cost.

## Execute transaction operations

Executing the transaction also involves executing a list of EVM operations. The only transaction that does not require any EVM operation is a value transfer.

There is a gas cost for each EVM operation executed. During the transaction execution, the operation's cost is deducted from the gas available for execution one-by-one until one of two things happen:

The execution runs out of available gas and fails

- The execution runs out of available gas and fails.
- The execution finishes with zero or more gas available and succeeds.

## Refund sender for SELFDESTRUCT and SSTORE operations

In Ethereum, the SELFDESTRUCT opcode is used to destroy a contract not needed anymore. This operation entitles the caller to receive 24,000 Wei for each destructed contract.

Also, when the SSTORE opcode is used to write zero (effectively deleting value), the user is entitled to 15,00 Wei per SSTORE writing zero.

One interesting aspect of the refund payment is that it is capped by a maximum value. The cap ensures that miners can determine the upper bound on the computational time to execute the transaction (more details on the reason for gas fees and refunds can be found on [Ethereum's design rationale wiki page](#)).

Another critical point is that the refund must happen after the transaction operations execution. Therefore, no refund gas is consumed by the transaction, preventing any exploit where the transaction never runs out of gas.

## Refund any remaining gas to sender

If the user upfront gas payment for the transaction exceeds

the gas used by the transaction, the sender is entitled to receive the remaining gas back after the transaction execution.

## Pay mining fees to the beneficiary account

The miner is entitled to receive all the gas used by the transaction execution as a transaction fee. This mechanism incentivizes miners to keep mining blocks and collaborating in the security of the network.

## Conclusion

In this post, we looked at the details of validating and executing a transaction (section 6 of the Yellow Paper). In sections 7 and 8 have more details on each type of transaction (contract creation and message call). I'll save these sections for a future post.

I believe the best way to understand the details of transaction validation and execution is by reading the source code of one of the clients that implement the protocol. As a Besu contributor, I am familiarized with its implementation. Even if you aren't super familiar with Java, I recommend you to take a look at the code. Start with the MainnetTransactionValidation.java and the MainnetTransactionProcessor.java.

As always, please let me know if you find any mistakes in this post. If you have any questions, please leave a comment down





below.

Catch you guys in the next one!

## References

- [Ethereum Wiki - RLP](#)
- [A closer look at Ethereum signatures](#)
- [Ethereum Design Rationale](#)
- [Why are selfdestructs used in contract programming? \[Stack Exchange question\]](#)
- [What are the limits to gas refund? \[Stack Exchange question\]](#)

Topic   **ethereum**   blockchain

Share            

Show Comments

### Gas and Payment -...

Another day, another Ethereum Yellow Paper blog post! In this...



04 Mar 2019

