Lesson 1 - Course Introduction

Course Plan

WEEK 1

Course Introduction / Blockchain / Cryptography review
Solidity Review and New Features
EVM Deep Dive
Security

WEEK 2

Solidity Assembly part 1
Solidity Assembly part 2
Layer 2 and Rollups
Gas Optimistation part 1

WEEK 3

Gas Optimistation part 2
MEV
IDEs - Foundry
Tips and tricks

WEEK 4

Auditing / Formal Methods
ETH2 / Research Part 1
Research Part 2
Review Day

Practical Details

All lessons will be conducted via zoom.
The format will usually be 45 mins of theory followed by 45 mins practical
You can work in teams if you wish
We use Sli.do to provide Q&A and polls : link

About us

Free Developer Workshops

- Basic
- Enterprise
- Advanced EVM
- Zero Knowledge Proofs

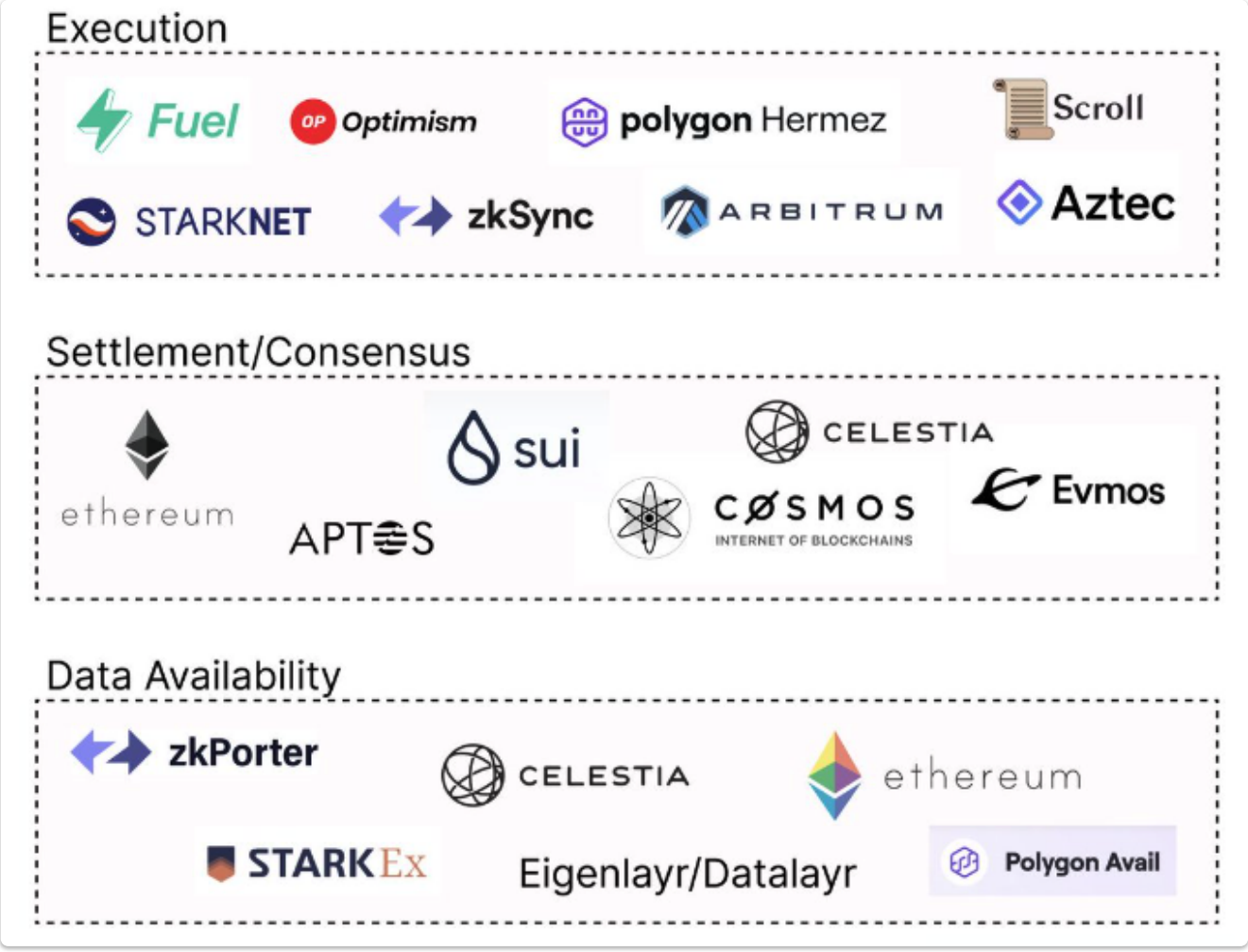
Business Workshops

Website : https://extropy.io

Email : info@extropy.io

Twitter : @extropy

Modular Blockchains



If we follow the principle of separation of concerns, we can use a combination of blockchains to provide the functionality of a L1 and increase scalability.
For further details see Volt article

The Blockspace Market

See article

Demand Side

The demand side is formed from the collection of transactions supplied by users (via DApps)

Supply Side

The supply side is provided by the miners producing blocks, adding to the Ethereum blockchain.
The 2 sides are mediated by congestion and fees.
The mechanics are complex and give rise to features such as

- MEV
- ASIC development

Consensus on Ethereum

From Ethereum developer documentation
"Now technically, proof-of-work and proof-of-stake are not consensus protocols by themselves, but they are often referred to as such for simplicity. They are actually Sybil resistance mechanisms and block author selectors; they are a way to decide who is the author of the latest block. It's this Sybil resistance mechanism combined with a chain selection rule that makes up a true consensus mechanism."

2 parts

- block producer selection
- block acceptance

From yellow paper
" 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."

Ethash is the PoW algorithm for Ethereum 1.0. It is the latest version of Dagger-Hashimoto

It works as follows

- a seed is computed for each block by scanning through the block headers up until that point.
- From the seed, one can compute a pseudorandom cache. Light clients store the cache.
- From the cache, we can generate a dataset, with the property that each item in the dataset depends on only a small number of items from the cache.
- Full clients and miners store the dataset.
- The dataset grows linearly with time.

Mining involves grabbing random slices of the dataset and hashing them together. Verification can be done with low memory by using the cache to regenerate the specific pieces of the dataset that you need, so you only need to store the cache.
The large dataset is updated once an epoch blocks, so the vast majority of a miner's effort will be reading the dataset, not making changes to it.

BLOCK VALIDATION IN ETHEREUM

The basic block validation algorithm in Ethereum is as follows:

1. Check if the previous block referenced exists and is valid.
2. Check that the timestamp of the block is greater than that of the referenced previous block and less than 15 minutes into the future
3. Check that the block number, difficulty, transaction root, uncle root and gas limit (various low-level Ethereum-specific concepts) are valid.
4. Check that the proof-of-work on the block is valid.
5. Let `S[0]` be the state at the end of the previous block.
6. Let `TX` be the block's transaction list, with `n` transactions.
   For all `i` in `0...n-1`, set `S[i+1] = APPLY(S[i],TX[i])`. If any applications returns an error, or if the total gas consumed in the block up until this point exceeds the `GASLIMIT`, return an error.
7. Let `S_FINAL` be `S[n]`, but adding the block reward paid to the miner.
8. Check if the Merkle tree root of the state `S_FINAL` is equal to the final state root provided in the block header. If it is, the block is valid; otherwise, it is not valid.

Nakamoto Consensus

You can see the Nakamoto consensus mechanism as working by allowing people to build chains and following the longest chain wins rule to decide between them (if needed).
It is important to see who the actors are
Miners - build chains, proposing a state to be decided
Nodes - chose from competing (valid) chains

Note that there are alternatives to a Nakamoto based consensus, DAG based approaches (Avalanche, IOTA) , BFT based (Cosmos, Algorand)

GHOST

See description in white paper and specification paper

The motivation behind GHOST is that blockchains with fast confirmation times currently suffer from reduced security due to a high stale rate - because blocks take a certain time to propagate through the network, if miner A mines a block and then miner B happens to mine another block before miner A's block propagates to B, miner B's block will end up wasted and will not contribute to network security.

Furthermore, there is a centralization issue: if miner A is a mining pool with 30% hashpower and B has 10% hashpower, A will have a risk of producing a stale block 70% of the time (since the other 30% of the time A produced the last block and so will get mining data immediately) whereas B will have a risk of producing a stale block 90% of the time. Thus, if the block interval is short enough for the stale rate to be high, A will be substantially more efficient simply by virtue of its size.
With these two effects combined, blockchains which produce blocks quickly are very likely to lead to one mining pool having a large enough percentage of the network hashpower to have de facto control over the mining process.

Ethereum includes stale blocks in the calculation of which chain is the "longest"; that is to say, not just the parent and further ancestors of a block, but also the stale descendants of the block's ancestor "uncles" or "ommers" are added to the calculation of which block has the largest total proof-of-work backing it.
To solve the second issue of centralization bias, we also provide block rewards to stales blocks.
A stale block receives 87.5% of its base reward, and the 'nephew' that includes the stale block receives the remaining 12.5%. Transaction fees, however, are not awarded to ommers.

- A block must specify a parent, and it must specify 0 or more uncles
- An uncle included in block B must have the following properties:
  - It must be a direct child of the kth generation ancestor of B, where 2 <= k <= 7.
  - It cannot be an ancestor of B
  - An uncle must have a valid block header, but does not need to be a previously verified or even a valid block
  - An uncle must be different from all uncles included in previous blocks and all other uncles included in the same block (non-double-inclusion)
- For every uncle U in block B, the miner of B gets an additional 3.125% added to its coinbase reward and the miner of U gets 93.75% of a standard coinbase reward.

The beacon chain uses Casper as the finality system, we will cover this in more detail in a later lesson.

Ethereum State, Transactions, and Blocks

World State. The world state (state), is a mapping between addresses (160-bit identifiers) and account states
Though not stored on the blockchain, it is assumed that the implementation will maintain this mapping in a modified Merkle Patricia tree

The account state, 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.
- balance: A scalar value equal to the number of Wei owned by this address.
- 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.
- codeHash: The hash of the EVM code of this account—this is the code that gets executed should this address receive a message call; it is immutable and thus, unlike all other fields, cannot be changed after construction.

ACCOUNT ABSTRACTION

See docs from Nethermind
Also this blog from Ethereum and this blog from Gnosis

Why Account Abstraction matters

The shift from EOAs to smart contract wallets with arbitrary verification logic paves the way for a series of improvements to wallet designs, as well as reducing complexity for end users. Some of the improvements Account Abstraction brings include:

- Paying for transactions in currencies other than ETH
- The ability for third parties to cover transaction fees
- Support for more efficient signature schemes (Schnorr, BLS) as well as quantum-safe ones (Lamport, Winternitz)
- Support for multisig transations
- Support for social recovery

Previous solutions relied on centralized relay services or a steep gas overhead, which inevitably fell on the users' EOA.

EIP-4337 is a collaborative effort between the Ethereum Foundation, OpenGSN, and Nethermind to achieve Account Abstraction in a user-friendly, decentralized way.

Instead of transactions, users send 'UserOperation' objects into a separate mempool, then actors called 'bundlers' package these up as transactions to a special contract.
There is a certain synergy here with some of the MEV solutions.

EIP-2930

Adds a transaction type which contains an access list, a list of addresses and storage keys that the transaction plans to access. Accesses outside the list are possible, but become more expensive. Intended as a mitigation to contract breakage risks introduced by EIP 2929 and simultaneously a stepping stone toward broader use of access lists in other contexts.

Digression about gas costs of opcodes
From EIP2929
Generally, the main function of gas costs of opcodes is to be an estimate of the time needed to process that opcode, the goal being for the gas limit to correspond to a limit on the time needed to process a block.
However, storage-accessing opcodes (`SLOAD`, as well as the `*CALL`, `BALANCE` and `EXT*` opcodes) have historically been underpriced. In the 2016 Shanghai DoS attacks, once the most serious client bugs were fixed, one of the more durably successful strategies used by the attacker was to simply send transactions that access or call a large number of accounts.

TRANSACTIONS

A transaction 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. As of the Berlin version of the protocol, there are two transaction types: 0 (legacy) and 1 (EIP-2930

Fields :

- type: EIP-2718 transaction type; formally Tx.
- nonce: A scalar value equal to the number of transactions sent by the sender; formally Tn.
- 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 Tp.
- 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 Tg.
- to: The 160-bit address of the message call's recipient or, for a contract creation transaction, ∅, used here to denote the only member of B0 ; formally Tt.
- 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 Tv.
- r, s: Values corresponding to the signature of the transaction and used to determine the sender of the transaction; formally Tr and Ts.

EIP-2930 (type 1) transactions also have:

- accessList: List of access entries to warm up. Each access list entry E is a tuple of an account address and a list of storage keys: E ≡ (Ea,Es).
- chainId
- yParity (in legacy transactions this is combined with the chain ID)

Bloom Filters

A Bloom filter is a probabilistic, space-efficient data structure used for fast checks of set membership.
The Bloom filter principle: Wherever a list or set is used, and space is at a premium, consider using a Bloom filter if the effect of false positives can be mitigated.

Bloom filter answers are 'no' and 'maybe'
Bloom filters have the inverse behavior of caches

- bloom filter: miss == definitely not present, hit == probably present
- cache: miss == probably not present, hit == definitely present

The basic idea behind the Bloom filter is to hash each new element that goes into the data set, take certain bits from this hash, and then use those bits to fill in parts of a fixed-size bit array (e.g. set certain bits to 1). This bit array is called a bloom filter.

Later, when we want to check if an element is in the set, we simply hash the element and check that the right bits are1 in the bloom filter. If at least one of the bits is 0, then the element *definitely* isn't in our data set! If all of the bits are 1, then the element *might* be in the data set, but we need to actually query the database to be sure. So we might have false positives, but we'll never have false negatives. This can greatly reduce the number of database queries we have to make.

For an interactive example, see

https://llimllib.github.io/bloomfilter-tutorial/

How are they used ?

Bloom filters are used with logs (events)
Instead of storing events from a transaction, we store the bloom filter, We store a bloom filter for

- each transaction (if needed)
- for each block (for all transactions)

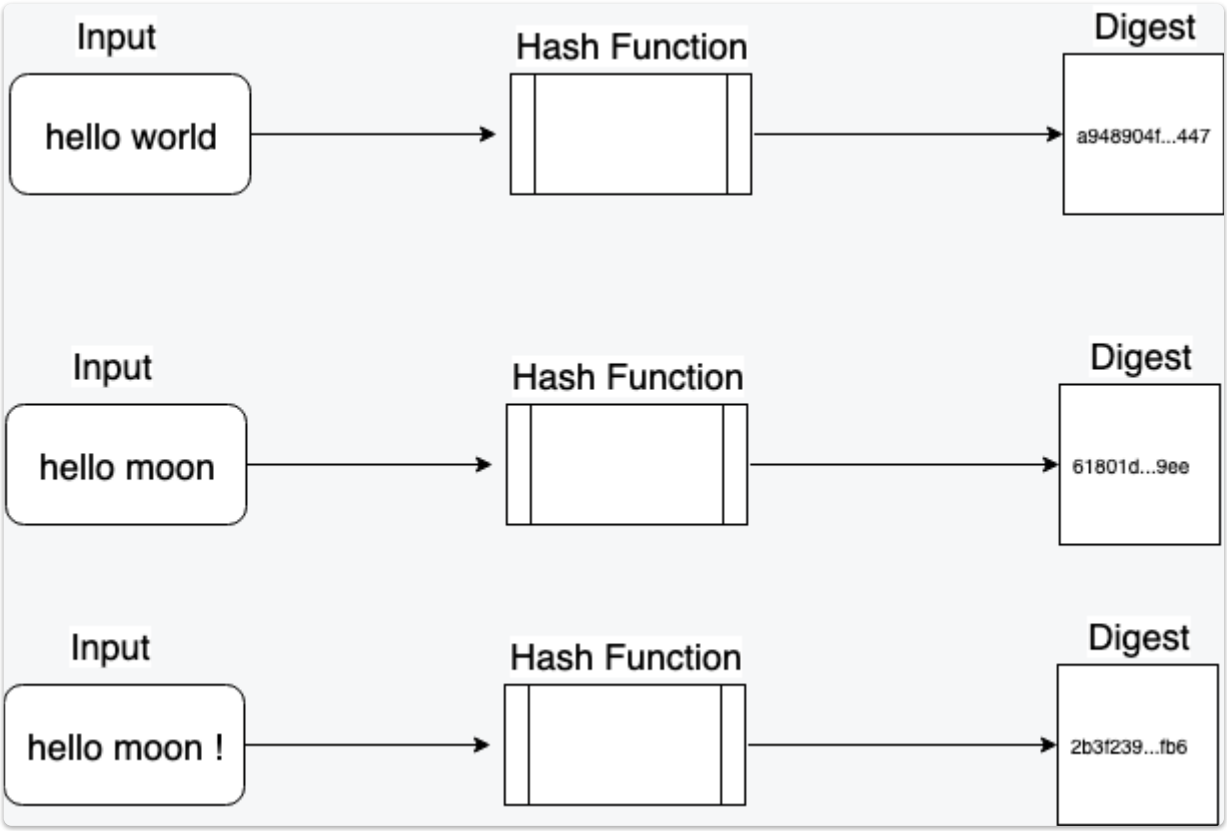
Thus a light client can check for an event in the block bloom filter, and if it probably exists, then check the transaction bloom filters.
A client storing the transaction receipts could then check those for the event.

Remember a Transaction receipt is a tuple of five items comprising:
-the type of the transaction
-the status code of the transaction
-the cumulative gas used in the block containing the transaction receipt as of immediately after the transaction has happened

-the set of logs created through execution of the transaction
-the Bloom filter composed from information in those logs

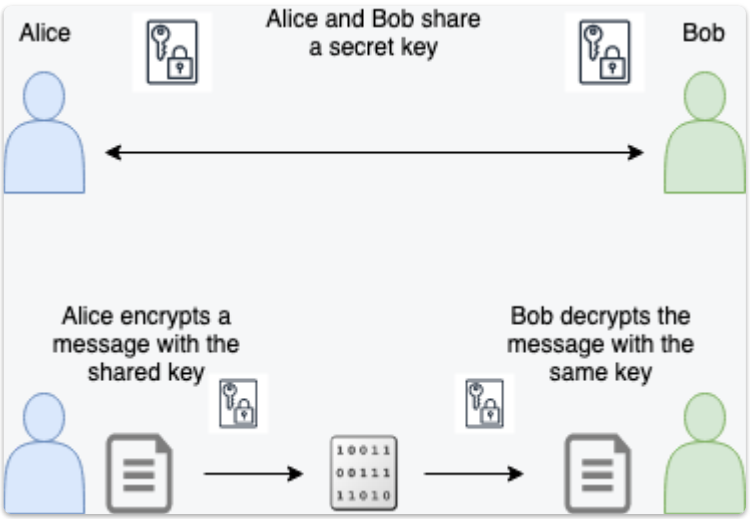
Cryptography Review

Hash Functions



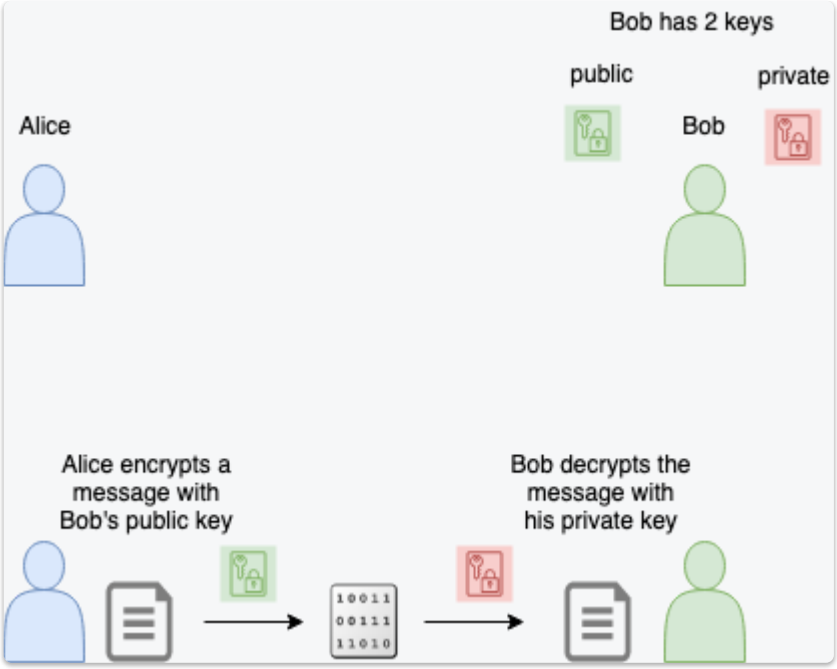
Encryption

SYMMETRIC ENCRYPTION

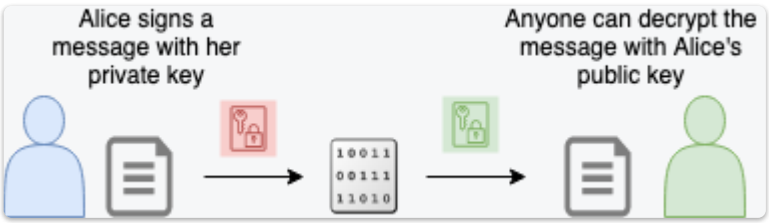


ASYMMETRIC ENCRYPTION

Sending a secret message

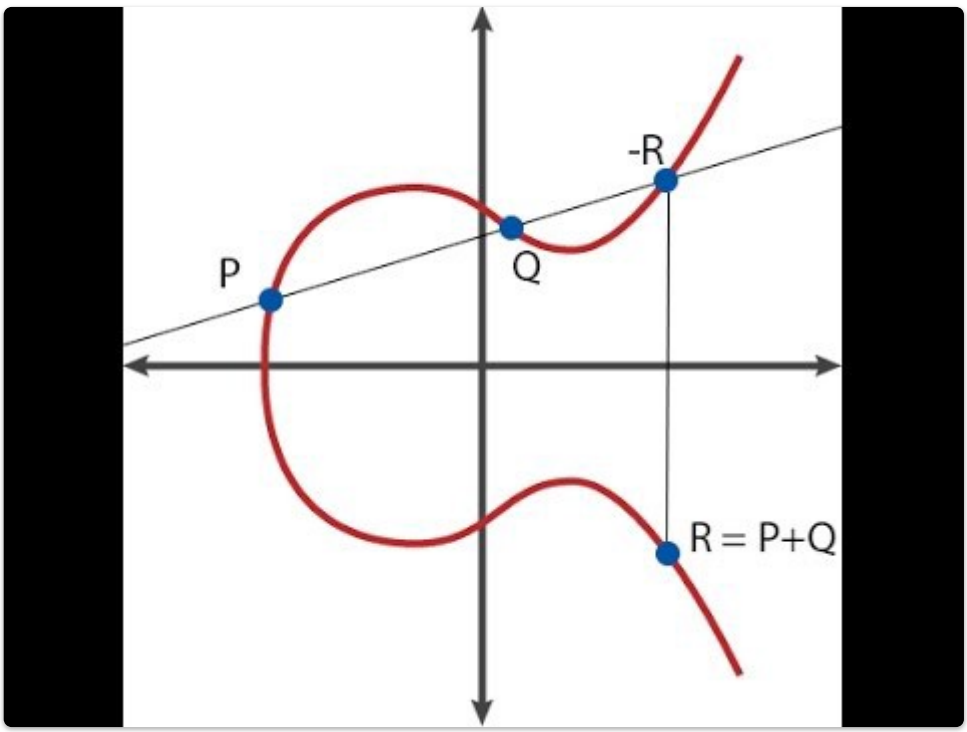


Proving ownership (knowledge of) of a private key



Elliptic Curves

The defining equation for an elliptic curve is for example $y^2 = x^3 + ax + b$



Ethereum uses the SECP-256k1 curve.

Cryptographic Signatures

See appendix F of the Ethereum Yellow Paper

Inputs

- Input message
- Private Key
- Random secret

Output

- Digital signature

The process can be checked even though the private key and random secret remain unknown.

Ethereum uses Elliptic Curve Digital Signature Algorithm, or ECDSA.
Note that ECDSA is only a signature algorithm. Unlike RSA and AES, it cannot be used for encryption.

Signing and verifying using ECDSA

ECDSA signatures consist of two numbers (integers): `r` and `s`.
Ethereum also uses an additional `v` (recovery identifier) variable. The signature can be notated as `{r, s, v}`.

To create a signature you need the message to sign and the private key (d_a) to sign it with.
The "simplified" signing process looks something like this:

1. Calculate a hash (`e`) from the message to sign.
2. Generate a secure random value for `k`.
3. Calculate point (x_1, y_1) on the elliptic curve by multiplying `k` with the `G` constant of the elliptic curve.
4. Calculate $r = x_1 \bmod n$. If `r` equals zero, go back to step 2.
5. Calculate $s = k^{-1}(e + rd_a) \bmod n$. If `s` equals zero, go back to step 2.

In Ethereum in step 1 we usually add
`Keccak256("\x19Ethereum Signed Message:\n32")`
to the beginning of the hashed message.

To verify the message you need

- the original message
- the address associated with the private key
- the signature `{s,r,v}`

v is either 27 or 28 in Bitcoin and Ethereum before EIP 155, since then, the chain ID is used in the calculation of v, to give protection against replaying transactions

`v = {0,1} + CHAIN_ID * 2 + 35`

Why do we need the v value ?
There can be up to 4 different points for a particular x coordinate modulo n
2 because each X coordinate has two possible Y coordinates (reflection in x axis), and
2 because r+n may still be a valid X coordinate
The v value is used to determine which one of the 4.

From the yellow paper
Recovery :

`ECDSARECOVER(e , v , r , s ) ≡ pu`

Where pu is the public key, assumed to be a byte array of size 64
e is the hash of the transaction, h(T).
v, r, s are the values taken from the signature as above.

References

Ethereum yellow paper
Ethereum white paper