Blockchain and Smart Contracts
Ethereum Course Notes
ICT 3009

Ingram Bondin, Tony Valentine

July 21, 2025

Contents

1 Introduction

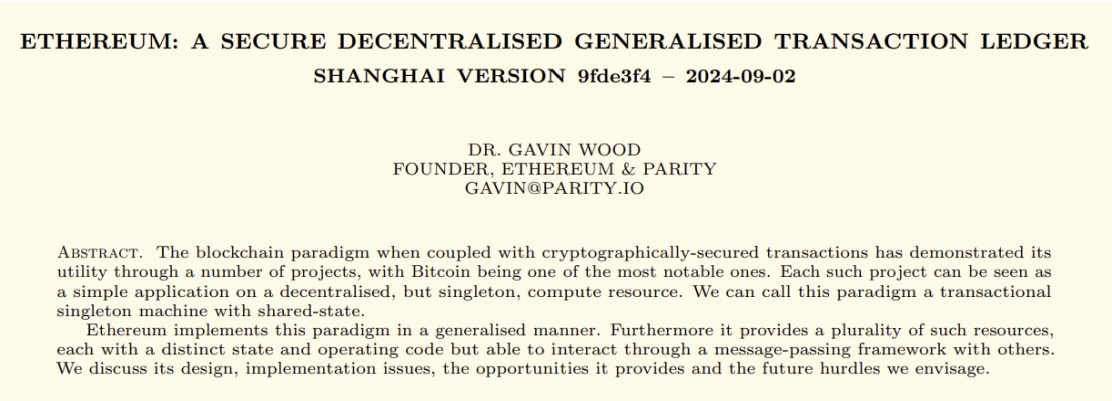
These are the course notes for ICT3009, covering the Ethereum and Smart Contracts part of the course. You should have already covered the Bitcoin part of the course with Dr. Neville Grech.

Some of the material we will be covering depends on concepts which will be explained later in the notes. When this occurs, we will denote this by the following icon 👉. When you come across this symbol, skim over the concept as we will return to it later.

2 Ethereum

Ethereum was conceptualised in 2013 by Vitalik Buterin, as a world-computer. Unlike Bitcoin, which is useful to transfer value but has only limited scripting facilities, Ethereum was supposed to support a Turing complete language which could be used to write Dapps[1] which could run on the blockchain.

The formal specification of Ethereum is defined in the *Yellow Paper*[2], which is periodically updated to reflect the current Ethereum version. However this is not always up to date and the canonical resource is the *Specification for the Execution Layer*[3]. As of the 13th March 2024, we are on the Cancun-Deneb hard fork.



ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER
SHANGHAI VERSION 9fde3f4 – 2024-09-02

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

ABSTRACT. The blockchain paradigm when coupled with cryptographically-secured transactions has demonstrated its utility through a number of projects, with Bitcoin being one of the most notable ones. Each such project can be seen as a simple application on a decentralised, but singleton, compute resource. We can call this paradigm a transactional singleton machine with shared-state.

Ethereum implements this paradigm in a generalised manner. Furthermore it provides a plurality of such resources, each with a distinct state and operating code but able to interact through a message-passing framework with others. We discuss its design, implementation issues, the opportunities it provides and the future hurdles we envisage.

Figure 1: The currently available Yellow Paper is for Shanghai-Capella and is out of date.

[1] Dapps stands for Distributed Applications.
[2] https://ethereum.github.io/yellowpaper/paper.pdf
[3] https://github.com/ethereum/execution-specs

Like Bitcoin, Ethereum has its own native token, which is called Ether, also referred to as ETH. The smallest unit of ETH is the Wei, which is 1×10^{-18} ETH. Therefore 1 ETH = 1×10^{18} Wei. ETH is crucial to the correct functioning of Ethereum. As in Bitcoin, it is a way of rewarding validators for adding blocks to the network (and sometimes to punish them!). It is also a way for paying for computation which is performed on the blockchain on your behalf.

You can obtain ETH by participating in the Ethereum network, or by buying it on the open market. As of the time of writing 1 ETH = 2624.64$.

2.1 Accounts

The first concept in Ethereum which we will consider is called an *account*. There are two types of Ethereum accounts:

- Externally Owned Accounts (EOA) - These are owned by a user and controlled through a private key known only to that user. They can keep a balance of ETH and are able to start transactions.
- Contract Accounts (CA) - These are owned by a smart contract and contain code which can be executed in response to transactions. They can keep a balance of ETH but are not able to start transactions on their own. They do not have a private key associated to them.

Both kinds of account have an *address*, which uniquely identifies the account on the blockchain.

As a user, you can create an EOA by generating a random 256-bit random number, which will be your private key. Using Elliptic Curve Cryptography (secp256k1), it is possible to obtain a public key from this private key. The address of the account is then simply the last 20 bytes of the Keccak256 hash of the public key. This means that addresses in Ethereum are always 20 bytes (160-bits) long.

In order to initiate a transaction, you have to sign it with the private key of your account. The implication of this is that if your private key is leaked, anyone will be able to sign transactions on your behalf (probably to empty it of ETH)! Each EOA has a *nonce* which keeps track of the number of transactions that the EOA has initiated. [4]

Unlike an EOA, Contract Accounts do not have a private key. Their address is determined when the corresponding smart contract is deployed to the blockchain by an EOA[5]

[4]The nonce is also required to prevent replay attacks.

[5]This is called a CREATE operation. There is also the CREATE2 operation, which allows the contract to be deployed at a deterministic address. The CREATE2 operation can only be used by a contract.

This address will be a function of the address of the deployer and its nonce. A smart contract also has its own nonce, but in this case it keeps track of how many other smart contracts our smart contract has deployed.

2.2 Transactions

As we said earlier an EOA can initiate a transaction by signing it using its private key. There are four core transaction types in Ethereum:

- Simple Transaction - this allows an account to transfer ETH to another EOA.
- Message Call Transaction - this allows an account to invoke a function on a given smart contract.
- Contract Creation Transaction - this allows an account to deploy a smart contract, and give it an endowment of ETH if required.

A transaction can have a number of fields:

- Nonce - This corresponds to the current nonce of the account and acts as an identifier for the transaction. It also helps with replay protection.
- To address - The address we are sending the transaction to. This will be empty for a contract creation transaction.
- Value - The amount of ETH that we are sending, if any.
- Signature - An ECDSA signature confirming our identity. The signature contains three pieces of information called V,R and S. The receiver can recover the sender of the transaction from these items using the ECRECOVER operation.
- Data - If we are calling a smart contract, this is where we mention the function to be called and list its parameters. In case of a contract creation transaction, this is where we provide the code we want to deploy. In reality this is the bytecode of a smart contract output by a compiler.
- Gas Price - The amount of Wei we are willing to pay per unit of gas. 👉
- Gas Limit - The maximum amount of gas we want to consume before aborting the transaction. 👉

Aside from the above three types of core transactions, other kinds of transaction have been introduced through various Ethereum Improvement Proposals (EIPs).

2.3 Ethereum State

Ethereum can be seen as a giant state machine. The blockchain starts in its genesis[6] state, and moves between states via its users initiating transactions, such as sending ETH, and deploying or interacting with smart contracts.

The global state of Ethereum at any given time is known as the *world state* of the network, and is simply the current state of all of its accounts. The world state is stored inside the *state trie*, and a node inside this trie is called an *account state*. All of the actions we mentioned previously cause updates to this *state trie*.

An account state keeps track of the state of an individual account. This means that it needs to store the account's nonce and its ETH balance. In addition if the account is a CA, it has to keep track of its contract storage and its code.

Now, every CA keeps its storage in a separate *storage trie*, which is like a key-value store for that smart contract. Inside the account state we keep a *storageRoot*, which is a hash of the root node of its storage trie. We also keep a *codeHash*, which is a hash of its bytecode. These two hashes allow efficient retrieval of the storage and bytecode when needed.

2.4 EVM

In the previous section we said that transactions cause Ethereum to move from one state to another. The new state is the result of a computation carried out by the *Ethereum Virtual Machine* (EVM).

The EVM is a stack-based machine which is capable of executing the bytecode held within a CA. The EVM operates as a sandboxed envrionment for security purposes and is not able to interact with external systems.

The word size[7] of the EVM is 256-bits (32 bytes) and the bytes are encoded in Big Endian order[8]. The EVM has no registers, but instead operates on a stack, which can be 1024 words deep.

Unlike Bitcoin Script, the EVM is Turing complete, because it is able to handle sequential execution, conditional statements, and most importantly unbounded iteration. This introduces the possibility of non-termination, which is limited in practice by the concept of *gas*. If too much gas is consumed, the computation will abort with an error. ☞

[6] In general, the genesis state of blockchains is empty, however a genesis file is often used to allocate funds to early users to enable easier use of the network.
[7] This is the largest amount of bits the EVM can manipulate at a time.
[8] This means that the most significant byte appears at index zero.

The EVM has 4 *data locations* from which data can be read from and written to:

- Memory - This is a temporary storage space which is available during program execution. It is organised as a word-addressed byte array. When reading from memory, one has to read 256-bits (one word) at a time[9], but when writing to it, one can either write 256-bits or 8-bits at a time[10]. Memory is unlimited, but if you exceed a certain threshold, the cost of manipulating it becomes quadratic in the size of the memory allocated.

- Storage - This is a permanent storage space which is stored on the blockchain. It is organised as a key-value store, where both keys and values are 256-bits long. It is possible to both write and read words from storage[11]. Every CA account on the blockchain has a separate storage space for security, and this storage space is not accessible by other CA accounts.

- Stack - This is a storage space used to carry out computations during a transaction, and is cleared after the transaction has ended. It is organised as a last-in-first-out (LIFO) data structure with a maximum depth of 1024 words.[12].

- Calldata - This is the storage space where the data provided by the transaction is held. It is a read only storage space.

In addition to the above, the EVM has access to 2 other areas:

- Code - This is a read only space which holds the bytecode which has been deployed to the blockchain.[13].

- Logs - This is a write only space where the EVM can output logs related to the execution of the code.

Initially, all storage and memory is set to zero.

When the EVM wants to execute some bytecode in response to a function being called, it will retrieve the bytecode which needs to be executed. It will then execute the bytecode instructions, using a program counter to keep track of where it has arrived. Whilst executing instructions, it will use the stack to temporarily store parameters and return values. It can also read and write to memory and storage, and can also output logs. Inputs to the function being executed are accessible through the calldata, and a return value can be provided to the callee of the function through a region of memory called

[9] This is achieved through the MLOAD operation
[10] This is achieved through the MSTORE and MSTORE8 operations
[11] This is achieved through the SLOAD and SSTORE operations.
[12] The stack is manipulated through PUSH and POP operations.
[13] For advanced tasks, code can be copied into memory through CODECOPY and EXTCODECOPY operations.

the returndata. During the execution of instructions, the EVM will keep track of the gas consumed. If it runs out of gas, the computation will be aborted. 👉

2.5 Blocks

In the Ethereum network, there are a number of nodes, called *validators* which have the power to add blocks to the blockchain[14]. Approximately every 12 seconds, the network will choose which validator gets to add such a block. 👉

The validator maintains a *mempool*[15], which consists of a number of user transactions which were seen by the validator after being shared on the network using a gossip protocol.

It will then choose a number of these transactions, and assemble them into a *block*, before proposing to add the block to the blockchain. The block has a finite amount of space, which is limited by the maximum amount of computation which can be carried out in a single block. This maximum is called the *block's gas limit*[16].

When submitting a transaction, a user will specify a certain amount of fees to be paid to the validator[17] 👉. A rational validator will start including the transactions from its mempool which guarantee it the highest fees, until it either reaches the gas limit, or there are no more transactions left in its mempool.

A validator does not have an unlimited amount of time to propose their block, as the network requires new blocks to be proposed within tight windows to maintain the timeliness of the network. Therefore producing a valid block which maximises rewards is an important optimisation problem.

A block in Ethereum consists of a header and a body. The most important fields in the header are the following.

- parentHash - The hash of the previous block's header. This links the current block to its predecessor, forming the blockchain.
- beneficiary - The validator who will get the reward for adding a block.
- stateRoot - The hash of the root of the world state trie after executing the transactions in a block. This is essentially a summary of Ethereum's state after the block has been added to the blockchain.

[14] As we shall see they can also vote on the validity of blocks on the blockchain.
[15] This is also known as a transaction pool.
[16] This is different from the transaction's gas limit specified by the user. 👉
[17] As we shall see this is made up of two components, the fee and the gas cost.

- transactionRoot - The hash of the root of the transactions in the block, organised as a trie. This provides an easy way to verify the integrity of the transactions in the block.
- receiptsRoot - The hash of the root of the receipts of the transactions in the block, organised as a trie. The receipts are data structures which contain information about the execution of the various transactions, such as gas usage, status and logs.
- number - The block's number.
- gasLimit - The block's gas limit, as described above.
- gasUsed - The total amount of gas which was consumed to execute all the transactions in the block. 👉
- timestamp - The Unix timestamp of when the block was proposed by the validator.

In the body of the block we find a list of transactions which were included in the block.

The first block of the Ethereum blockchain is called the Genesis block. This block is special because it is hardcoded into Ethereum. All subsequent blocks were added through either mining[18] or through validators proposing blocks[19].

Once a validator has proposed a block, it will share it with the other Ethereum validators in the network. The other validators will validate the block by checking its integrity. The most basic integrity checks involve checking that the block references a valid parent and that the timestamp satisfies certain constraints. In particular the timestamp should be higher than the one found in the parent block, and less than 15 minutes in the future from the latter.

Following this the transactions will be checked for consistency. This process involves taking the list of transactions included in the block, and re-executing them on the new validator node. The transactions are organised into a transaction trie, the receipts are organised into a receipts trie, and the updated world state into an updated state trie. The roots of all these three tries are then hashed and compared with the block's transactionRoot, receiptsRoot and stateRoot. If they match, then we know that the original validator did not attempt to trick us, and we can accept the block by writing the new transaction, receipt and state tries to the local blockchain database of the new validator.

[18] Before Ethereum moved to Proof-of-Stake, blocks were mined in a way similar to Bitcoin.
[19] Ever since The Merge hard fork took place on the 15th of Sepetember 2022.

2.6 Gas

Earlier we saw that in order for a validator to include a transaction in a block, the user has to pay the validator a fee. If the fee paid is too low, the transaction may be left in the mempool and not be picked up as a result.

In addition to this fee, Ethereum uses the concept of *gas* to keep track of how much computation was performed during a transaction. This is a cost which must also be paid in order to ensure that the transaction is executed.

If the computation is very complex or is too long, we may experience an *out of gas* condition, leading to our transaction being aborted. When a transaction is aborted, it is still included in a block as a failed transaction. The miner still gets paid, and the user does not get any refund.

One reason for using gas is to stop computations from entering infinite loops and not terminating.

When the a user invokes a function of a smart contract, the EVM will execute the bytecode corresponding to that function. Each instruction which is executed has a fixed gas cost. The gas used to execute the function is the sum of the gas costs of the individual instructions for that function. For example calling the function *mint()* on a smart contract may cost 1000 gas.

Now, the gas used is just an integer, but the gas fee has to be paid in ETH. How do we know how much ETH we have to pay for the 1000 units of gas we consumed?

The answer is that we decide how much a unit of gas is worth to us. So for example, we may decide that we are ready to pay 25 Wei for 1 unit of gas. This means that the computation will cost us $25 * 1000 = 25000$ Wei. In general the formula to calculate the gas fee is the following:

$$Gas\ Fee = Gas\ Used \times Gas\ Price$$

What is stopping us from paying the bare minimum for gas? The gas price is the reward we are giving to the validator for executing our instructions. If someone provides a higher gas price than us, the validator will prefer to execute their transaction rather than ours.

This means that submitting transactions involves competing in an auction for limited block space. As long as the validator is rational, it will include the transactions with the highest fees and gas price first, because this maximises its returns. If it runs out of block space, the transaction has to wait for the next block for a chance to be included.

One side effect of this is that there can be periods of high network congestion where most transactions cannot be executed because the gas price is not viable. Similarly in periods where not much is going on on the blockchain, transactions can be executed for a minimal gas price.

One possible way of reducing high gas fees is to have larger blocks, or to produce blocks quicker. However there are always trade offs. These adjustments mean that validators would need to run more powerful machines. As a result less people will participate in validating blocks and the network becomes less secure. There are other ways to scale blockchains, such as so-called Layer 2 solutions[20]. We will mention some of these later if we have the time.

2.7 Consensus

The Ethereum network consists of many validators, and these validators have to reach a consensus about what the current blockchain state is at any point in time. We are now in a position to discuss how this consensus is reached on Ethereum.

We will first mention briefly that Ethereum originally reached consensus through a Proof-of-Work algorithm similar to the one used in Bitcoin, which was called *Ethash*. The major difference from Bitcoin was that Ethash was supposed to be ASIC-resistant, which means that people with specialised hardware could not get an advantage over users who were running commodity hardware. This would allow greater participation and hence greater security for the network. However, ASIC chips were eventually created for mining Ethereum as well. In addition to this, Proof of Work is a very wasteful and resource intensive algorithm and this was another reason why Ethereum changed its consensus algorithm to *Gasper*, which is a Proof-of-Stake algorithm.

The Gasper protocol consists of two parts, called *Casper* and *LMD-GHOST* respectively.

2.7.1 Casper

Under Casper, the Ethereum network is organised into a number of validators. In principle, this network is permissionless, in the sense that anyone can join the network as a validator by running the Ethereum software. Validators are able to both propose new blocks, as well as attesting (validating) blocks proposed by other validators. Validators can earn ETH for proposing new blocks as well as for attesting to blocks correctly.

Casper organises block production into *epochs*, where each epoch has 32 slots and each slot takes 12s. At the start of each epoch, some randomness is generated using the

[20] These have an off-chain component and include rollups and sidechains.

RANDAO algorithm. During each slot, a single validator is chosen randomly. This validator is tasked with proposing a new block and broadcast it to other nodes.

This block is not attested by every validator on the network. Rather, during each epoch, the validators are randomly split into 32 groups (1 for each slot). Each group of validators will be attesting the blocks for that slot, which means that each block will be attested by $\frac{1}{32}$ of the total number of validators.

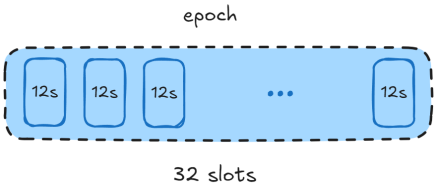


Figure 2: One epoch has 32 slots. Each slot is 12 seconds long.

In order to reduce network congestion[21], not every validator transmits his attestation to the other nodes in the network. Instead, each group attesting a slot is divided into 64 committees, with each committee having a minimum of 128 validators. During each epoch, an aggregator is chosen from each committee. It is the job of this aggregator to collect all attestations in this committee, aggregate the signatures and broadcast the aggregated message to the wider network. The proposer will assemble these aggregated attestations and include them in the new block.

In order to keep validators honest, Casper requires validators to lock up (stake) exactly 32 ETH before being able to participate in the network.

Validators which are dishonest or negligent can lose part of their staked ETH. Validators which go offline are *penalised* and suffer a reduced balance. Validators which are dishonest are *slashed*, which means that part of their funds are burned and that they are ejected from the validator committee.

Validators can be slashed if they commit one of the following dishonest behaviours:

- They propose and sign two different blocks during the same epoch.
- They sign two conflicting attestations (for and against the block)
- They attest to one version of the chain, then attest to a different version of the chain 👉

This arrangement ensures that validators fulfill their functions and makes it very expen-

[21] This was not needed in Bitcoin, because only the node which finds the block needs to communicate with the other nodes. In Ethereum's case, the validators also need to communicate with one another to attest the blocks, which increases network traffic.

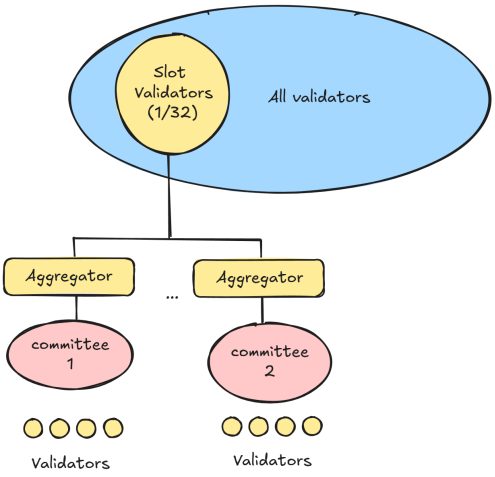


Figure 3: The group validating each slot is divided into committees to reduce communication overheads. An aggregator is just a normal validator which has been tasked with broadcasting the aggregated attestations for the duration of the slot.

sive for validators to attack Ethereum, as malicious validators become poorer with every attack.

Validators also fulfill another important function in that they attest that certain blocks have been finalized. A finalized block is a block which has permanently become part of the main blockchain and is irreversible to all intents and purposes.

Finalization, is achieved through a part of Casper called *Casper-FFG*[22], which works as follows.

To participate in the network, validators have to stake exactly 32 ETH. When a validator attests to a block, the validator is essentially voting for that block using the amount of ETH which it has staked.

Now, the last block of every epoch is called a checkpoint block. Validators will examine pairs of successive checkpoint blocks and attest them if they are valid. Once a checkpoint block has been voted for by validators holding $\frac{2}{3}$ of the total staked ETH, it becomes justified. Justified blocks are considered stable, but can still be reverted. Once the subsequent block has been justified as well, the prior block becomes finalized. A finalized block is considered irreversible. Once a checkpoint is finalized, its epoch is considered finalized as well.

In order for a finalized block to be reversed, more than $\frac{1}{3}$ of the ETH possessed by the

[22]Casper-FFG stands for Casper the Friendly Finality Gadget.

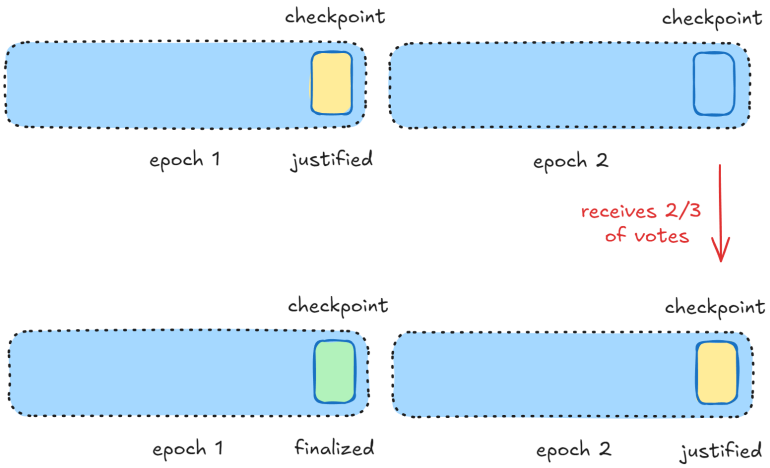


Figure 4: Initially, the checkpoint block of epoch 1 is justified. Once the checkpoint block of epoch 2 receives $\frac{2}{3}$ of the votes, it becomes justified as well, and the preceding block becomes finalized.

validators would have to be put at risk of slashing. This is an incredibly improbable scenario.[23]

One concern is that since finalizing epochs requires a $\frac{2}{3}$ majority of votes, an attacker which controls $\frac{1}{3}$ of the votes could stop the chain from finalizing. In order to ensure the liveness of the chain, if the chain fails to finalize for more than 4 epochs, an *inactivity leak mechanism* will drain ETH away from validators on the wrong chain. The resulting destruction of ETH allows the majority to regain $\frac{2}{3}$ of the votes for the correct chain again.

2.7.2 LMD-GHOST

Another concern that we need to address is whether it is possible for forks to occur on the Ethereum blockchain. Earlier we said that a single validator is nominated as a proposer during every slot, and that only a proposer can add a block to the blockchain. This seems to imply that the Ethereum blockchain evolves linearly, one block at a time, with no possibilities of a fork occurring.

However this is not the case. Suppose that the latest block to be added on the blockchain is block B_1. It is possible that a validator proposes a block B_2 with B_1 as a parent, but

[23] While Bitcoin has probabilistic finality, where a competing chain could overtake the main chain with decreasing probability for each block added, Ethereum has deterministic finality, because the finalised part of the chain cannot be overtaken by another chain unless the network has been compromised with an attacker controlling $\frac{2}{3}$ of the staked ETH.

this block does not reach all the other validators before the next slot starts. If the next validator has not seen B_2, it will think that the validator at the previous slot did not propose a block on time. Hence it will propose block B_3 with B_1 as a parent. The result is that we have a fork on the chain. The third slot comes along and the next validator sees both the chain ending in B_2 and the chain ending in B_3. At this point it has to choose which chain to follow, and to do so it needs a *fork-choice* rule. The fork-choice rule used in Ethereum is called LMD-GHOST[24].

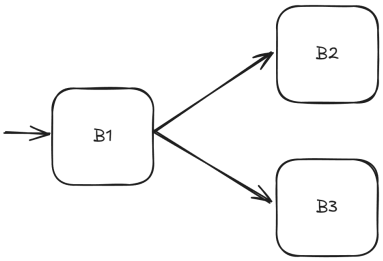


Figure 5: A fork has occurred on the Ethereum blockchain.

LMD-GHOST will first identify the last block which was finalized by Casper-FFG. This will form the root of the tree used to make the fork-choice. The algorithm will then look at the children of this block, and choose the subtree with the most weight. The weight of a subtree is the sum of votes which have been attested on the blocks which make up the subtree. If there is a tie between the weight of two subtrees, the subtree whose root block has the lower hash is chosen to break the tie. This process is then repeated, until a block with no children is reached. This block is the head of the blockchain, and the next block is added to it. This is known as the GHOST rule. The LMD rule says that only the latest attestation from each validator is counted when computing the votes for a subtree. Thus within a subtree, we only count the votes for the deepest block a validator has attested so far. In this way we avoid double counting of votes when calculating the weight of a subtree.

In our motivating example above a fork was caused by an honest validator. However a fork can also be caused by a malicious validator. In fact, there are several ways in which a malicious validator could cause a fork in Ethereum:

- A validator could sumbit two different blocks when it is the proposer.
- A validator could withhold a block when it is the proposer, and release the block shortly after its slot has ended.
- A validator could attest to a block which is not on the main chain.
- A validator could simultaneously attest to blocks on the main chain and to blocks

[24] LMD-GHOST stands for Latest Message Driven, Greediest Heaviest Observed Subtree.

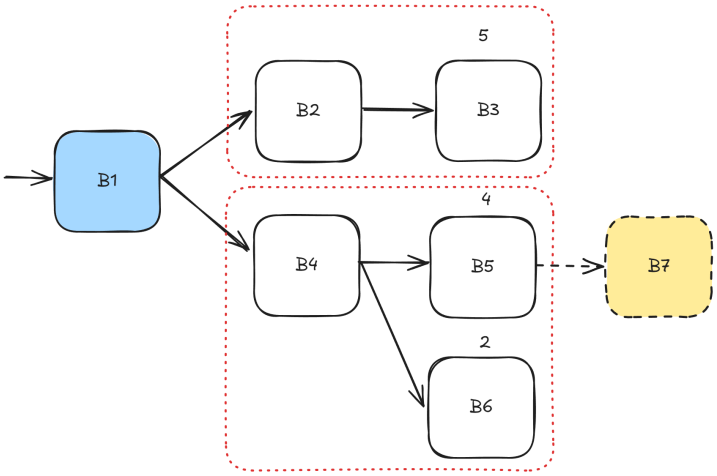


Figure 6: The block B_1 is the last finalized block. The subtree with root B_4 has more weight than the subtree with root B_2. After B_4 is chosen, the leaf block with most votes is B_5. So B_5 is the head of the blockchain, and the new block B_7 is added after B_5.

off the main chain.

As one can see the problem is not a trivial one and all of these scenarios had to be considered when Ethereum's consensus algorithm was being designed.

3 Applications

In this section we will be discussing some sample standards and protocols which can be builton the Ethereum Blockchain. This section assumes that you have already covered the Solidity language. When a new blockchain application is proposed, it is often described in an Ethereum Improvement Proposal (EIP) or in a white paper issued by the team building the application (often known as a protocol).

3.1 ERC-20

ERC-20[25] is a *fungible token standard* on the Ethereum blockchain, meaning that each token created using this standard is identical to every other token of the same type. The standard defines a set of functions that an Ethereum smart contract must implement to create and manage tokens. There are many ERC-20 compliant tokens on the Ethereum blockchain, such as USDC, DAI, LINK and UNI.

There are three main use cases for ERC-20 tokens:

- Currency Tokens: These act as decentralized digital currencies, designed to be a store of value and medium of exchange, similar to traditional fiat currencies but operating on the Ethereum blockchain.
- Utility Tokens: These tokens are used to provide access to a specific product or service within a decentralized platform, often functioning as a payment method for using the platform's features.
- Security Tokens: These tokens represent ownership in an external asset, such as shares in a company or in a bond.
- Stablecoins: These are pegged to the value of a stable asset, such as a fiat currency (e.g., USD), to minimize volatility and provide a stable medium of exchange.

One interesting thing we can do with ERC-20 tokens is to wrap the native ETH token into the ERC-20 token WETH. This allows us to manipulate ETH as if it were an ERC-20 token and avoids protocols having to have two kinds of logic to deal with Native tokens and ERC-20 tokens.

The key idea behind an ERC-20 token is to have a *balances* mapping, which represents the amount of tokens owned by an address at a given time.

```
mapping(address => uint256) balances
```

[25] https://eips.ethereum.org/EIPS/eip-20

The contract allows a user who owns some of the tokens to transfer these to another user through a *transfer* function.

```
transfer(address to, uint256 value)
```

The transfer is achieved by updating the *balances* mapping, subtracting the funds from the balance of the first user and adding them to the balance of the second user.

Another key piece of functionality which the contract supports is to allow a user to approve another user to move funds on their behalf. This is achieved through the *approve* function.

```
function approve(address spender, uint256 value) public
returns (bool success)
```

In order to keep track of approvals we use the following mapping. The first address is the approver, and the second address is the person being approved. The third parameter is the amount being approved.

```
mapping(address => mapping(address => uint256)) approvals
```

When *approve* is called, it will update the above mapping. This allows the person who was approved to move funds on behalf of their owner, using the following function, which updates the *balances* mapping accordingly.

```
function transferFrom(address from, address to, uint256 value) public
returns (bool success)
```

ERC-20 tokens will also have a contract variable which keeps track of the total number of tokens in circulation.

```
uint256 totalSupply
```

This *totalSupply* is updated whenever tokens are minted or burned, although this logic will be specific to the token in question. The variable can also be used for other purposes, such as putting a cap on the maximum amount of tokens which can be minted.

3.2 NFTs

ERC-721[26] is a standard for creating *non-fungible tokens (NFTs)* on the Ethereum blockchain. Unlike ERC-20 tokens, which are fungible, each ERC-721 token is unique and represents

[26] https://eips.ethereum.org/EIPS/eip-721

ownership of a specific asset. This standard defines a set of rules that enable NFTs to be transferred, managed, and tracked on the Ethereum blockchain.

ERC-721 tokens have a wide range of use cases due to their non-fungible nature, making them ideal for representing unique digital assets:

- Digital Art and Collectibles: ERC-721 tokens are widely used to represent unique digital artworks or collectibles.
- Gaming: In-game assets, such as rare items, skins, or virtual land, are represented by ERC-721 tokens, providing players with true ownership and the ability to trade or sell these assets outside the game.
- Tokenizing Real-World Assets: ERC-721 tokens are also used to represent real-world assets like real estate or luxury goods, enabling ownership proof, traceability, and transferability on the blockchain.

The key idea behind an ERC-721 contract is to represent each non-fungible token by a number, and then keep track of the address of the owner of that token. This is achieved through the *owners* mapping.

```
mapping(uint256 => address) owners
```

In order to link the token to some real world asset, we will also have a contract variable called *baseURI*, usually immutable and initialised at deployment time.

```
string baseURI
```

By adding the *baseURI* to the token number, we can construct the URL which points to the asset in question, for example an image of the digital art identified by the token number.

Similarly to ERC-20 tokens, the contract will also support functionality to transfer a token number from one owner to another. It will also support approvals for specific tokens as well as the associated transferFrom functionality.

One problem with this design is that if a user owns many assets, and wants to delegate the ability to transfer these assets to some other user, then the first user will will have to approve each asset individually, which will be expensive. To enable this use case, NFT contracts have the concept of an *operator*, which is an address enabled to transfer any of another user's assets.

We can keep track of operator approvals through the *operatorApprovals* mapping. The first parameter is the owner of the assets, the second is an address which represents a potential operator, and the third is a flag representing whether that operator is approved

to manage the assets of a given owner.

```
mapping(address owner => mapping(address operator => bool)) operatorApprovals
```

An owner can do a bulk approve to an operator through the *setApprovalForAll* function, which will update the *operatorApprovals* mapping.

```
function setApprovalForAll(address operator, bool approved) public
```

Needless to say, when *transferFrom* is called, it will check whether the caller is approved to transfer that asset in the traditional way, or whether it has been bulk approved to manage the owner's assets as an operator.

3.3 Decentralised Lending

We will now explore an important DeFi protocol, called Maker. The field of DeFi stands for decentralised finance, and is one of the major usecases of blockchain technology. The main idea behind DeFi is to take use cases in traditional finance (such as Exchanges, Insurance and Lending) and to decentralise them by the deploying them on the blockchain and have smart contracts automate the process as much as possible. Some advantages of DeFi include removing middlemen from the process, reduced costs to operate a service, providing financial services to those excluded from using them and potential ownership of the service by its own users.

In this section we shall be describing the Maker protocol[27], which is decentralised lending protocol. Using Maker you can obtain a loan denominated in the DAI token. The DAI token is a stablecoin, meaning that it tries to keep a price peg (be tradable 1:1) with the US Dollar (USD), instead of experiencing the volatility of the crypto market. By issuing credit in a stablecoin, Maker gives you a stable asset which you can then use for other purposes, including other DeFi applications.

In order to obtain a loan from Maker, you have to deposit some collateral with the system, such as ETH, into what is called a Collateralized Debt Position (CDP, or Vault). This is similar to obtaining a loan to buy a house (the house is collateral and can be repossessed if you don't pay the loan back to the bank). Of course, once you repay the loan back, you will get back access to your ETH.

Now ETH is not a stable coin, and its price in USD is thus volatile. Therefore, if the price of 1 ETH happens to be 2334 USD, the Maker protocol cannot issue you 2334 DAI in return for taking the 1 ETH as collateral. This is because the protocol has to mantain a margin of safety for its loan.

[27] https://makerdao.com/en/whitepaper/

Every type of collateral in maker, will have a collateralisation ratio associated with it, representing the risk of using that asset as a collateral. For instance, if the collateralization ratio for ETH is 150%, we have to provide 150 USD worth of ETH to obtain a loan worth 100 USD of DAI.

The collateralisation ratio is defined as follows:

$$Collateralization\ Ratio = \frac{Collateral\ Value}{Loan\ Value}$$

Therefore, with 1 ETH we are can only get a maximum of $\frac{1}{1.5} \times 2334 = 1556$ DAI. At this point you may be asking yourself what's the point of locking up your 1 ETH with Maker, to not even get back an equivalent amount of DAI. The answer is that you are only giving up control of your ETH temporarily (you will still be able to enjoy any positive price appreciation when you repay the loan and get back your collateral). In the meantime, you can use the extra 1556 DAI to invest in and obtain the return of some other DeFi protocol. Another reason is that you can use this loan to acquire more ETH and then repeat the process as many times as you like, multiplying your exposure to positive ETH volatility. This concept is called leverage.

In reality, you should never borrow the maximum amount. Since the price of ETH can go down, the collateral value can also go down, which means that the size of the loan which that collateral can support can also go down. If the loan value which can be supported goes below the amount which was actually loaned out to you, your position is now under-collateralised and will be liquidated by the protocol. This involves Maker selling off your ETH for DAI. This DAI is then burned (with the effect that the loan has been effectively cancelled). If more DAI is raised than needed to repay the loan, the excess is returned to the user. The flipside of this scenario is that the ETH price goes up, allowing you to borrow even more DAI because your collateral value has increased in this case.

Of course, in return for taking out a loan you will have to pay some interest. This interest is calculated at a yearly rate (say 2% every year), but is accumulated in real time, not at once at the end of the year. This interest is added to the amount which was loaned, and must be repaid as well in order to unlock the collateral. This interest rate is called a stability fee, and it plays multiple purposes in this protocol. The interest paid generates interest for the protocol, but it also plays a major role as a monetary policy took to keep DAI pegged to the USD.

How does this work? If the value of DAI exceeds the value of the USD, then DAI is becoming scarce. The protocol lowers the interest rate to encourage borrowers to borrow more DAI, which mints more DAI in the process, driving the price of DAI down. On the other hand if the value of DAI is below the value of the USD, then there is too much DAI in circulation. So the protocol increases the interest rate. This has the effect of making

borrowers want to close their DAI positions, thus taking DAI out of circulation and making it more scarce, which increases its price.

The Maker protocol also has a governance function, which allows holders of the MKR governance token to vote to control certain parameters of the protocol. Governance is a way to introduce a human decision making element into a protocol driven by automated smart contract rules. For instance, holders of the MKR token can change the collateralisation ratios of different collaterals and even influence the stability fee. Holders of the MKR token are rewarded by the protocol in the following way. The protocol will use some of the stability fees it has collected to buy back MKR tokens and make them more scarce, thus increasing the value of MKR tokens. This incentivises holders of MKR tokens to take decisions which are good for the health of the protocol.

Finally, there could be a disastrous scenario where a sudden collapse in the price of the collateral occurs faster than positions can be liquidated to make up for the lower collateral value. In this case, the collateralisation ratio can go below 100%, meaning that the Maker protocol becomes insolvent. In Emergency Situation, the protocol will mint more MKR tokens and sell them for DAI. It will then burn this DAI, which is the equivalent of closing some amount of loans. In this way the remaining collateral is able to adequately support the remaining outstanding DAI. This loss is of course borne by the holders of the MKR token, who find that the value of their tokens has been diluted.

In a catastrophic situation, the holders of the MKR token can even trigger an Emergency Shutdown of the protocol, which pauses the protocol and allows DAI holders to redeem their tokens for the underlying collateral at a fair global settlement price. This is calculated based on the value of the collateral at the time of the system shutdown. This helps to guarantee an orderly winding down of the protocol.

4 Off-Chain Communication

So far our distributed applications operate in a closed system, and do not interact with the external world in any way. However, this is far from ideal. For instance, to be useful, our application may need to obtain data from the real world (such as prices, or the current temperature outside) and use this as a basis for its computations. In addition, our application may want to interoperate with another system (an application on another blockchain, or a traditional application) and thus needs a way to output information to these systems.

Interactions with the real world will involve traditional software components, which are often referred to as off-chain components. This can be contrasted with smart contracts, which are referred to as on-chain components.

4.1 Oracles

Oracles are used to provide data from the real world to blockchain applications. An oracle is a smart contract which contains three things:

- A contract variable which will be used to store the off-chain data.
- An external setter function which can be called by an EOA to store the off-chain value in this contract variable.
- A getter function to allow other smart contracts to read the off-chain data known by the oracle.

Outside of the blockchain, we can then have a traditional application which access the data periodically (say from an exchange, or a weather station, through its API) and then updates the oracle contract through its control of the EOA.

Our own smart contract application can then call the oracle through its getter function to obtain the latest value reported by the oracle. When a value is read from an oracle, the oracle will often provide additional data, such as the timestamp at which the value was last updated. The smart contract consuming this value needs to check this timestamp, to ensure that the value is not so old as to be stale. It should also make some additional validations, to ensure that the oracle has not failed and is returning some default value such as a zero price.

4.2 Events

While oracles are used to provide real-world inputs to the blockchain, a blockchain application can provide output to the real world by outputting events. Events can then be read by an off-chain component, as these are stored as part of a transaction's logs and can be retrieved from the blockchain. The off-chain component can then execute some code of its own, or even start another transaction with the same or another blockchain. A protocol which uses such a mechanism to perform interactions between different chains is called a *cross-chain* protocol.

4.3 Centralisation Concerns

An important consideration when interacting with off-chain components is that the system becomes less decentralized, since someone is in control of the off-chain component and everyone else needs to trust it (besides being a single point of failure!).

One solution which is often used to remove this centralisation concern is to use a *decentralised oracle network*. This replaces the off-chain component with a network of nodes, controlled by different individuals.

This network will run its own consensus algorithm. Each component will get the off-chain data separately and send it to an aggregator node. The aggregator node will aggregate the various values into a single value (for example by taking the majority value, or an average).

People running the nodes may also be required to stake some tokens, and may be slashed if they report a value which is far outside the acceptable range. Such networks also need to have some mechanism for rewarding honest nodes for the work they are performing, most often by distributing some token to them (otherwise there is no incentive to participate in this network). Chainlink Price Feeds are one example of such a system.

5 Upgradability

Although smart contracts on the blockchain are typically immutable, they can still be designed with upgradability in mind. This is necessary to address changing requirements, fix bugs, or improve functionality. This is typically achieved though the use of a *proxy pattern*, which separates the contract's logic from its storage, allowing for upgrades to the logic without affecting the contract's state.

In order to implement a proxy pattern we need to split our design into two contracts, a logic contract and a proxy contract.

The logic contract will contain all the contract variables and business logic required by the application. However, its contract variables are there to provide structure to the code, and will not actually be holding any state in production. When an upgrade is required, we will deploy a new logic contract, but we will not lose the old state, because this is not held inside the logic contract.

The proxy contract serves as the interface through which users interact with our Dapp. It will keep a reference to the address of the logic contract currently in use. The proxy contract does not declare any contract variables and will simply forward any calls it receives to the logic contract. However the twist here is that it will call the logic contract using delegate call, and not using a normal call.

What delegate call does is that it allows the proxy contract to execute a function from the logic contract, but within its own storage space. This means that any state changes occurring as a result of the function call will be recorded inside the proxy contract, not the logic contract.

If the logic contract needs to be upgraded, we can simply instruct the proxy to point to the address of the new logic contract. This means that going forward any call to the proxy will be forwarded to the new logic contract, and thus the new business logic will execute in the context of the existing state.

When upgrading contracts, a critical issue to consider is the *storage clash problem*. Since both the Proxy and Logic Contracts have the same storage layout, any change in the layout (e.g., adding or reordering storage variables) in the Logic Contract can result in a *storage clash*.

A storage clash occurs when storage variables in the new logic contract conflict with the proxy contract's layout. This leads the business logic of the new logic contracts modifying the wrong variables, when it is delegate called, leading to data corruption in the proxy contract.

To prevent storage clashes, developers can:

- Ensure that storage variables remain in the same order in all logic contract upgrades.
- Use unstructured storage patterns such as reserved storage slots, to avoid conflicts. This is especially important when our contracts are part of an inheritance hierarchy, because adding a variable to a contract in the middle of the hierarchy will affect the storage locations of contracts further down the chain of inheritance. To avoid this, we can pre-reserve empty storage slots inside contracts which we will inherit from, allowing us to replace these slots with real contract variables after an upgrade without affecting contracts further down the inheritance chain.

Only owner accounts should be able to set the address of the new logic contract during an upgrade (preferably, these accounts should be controlled by a multisig)! If any user could change the logic of a contract, anyone could point the proxy to a malicious contract, potentially bringing the system down. In general there is a tension between decentralisation and upgradability, in the sense that making a system upgradable means that you have to trust the person with the power to make an upgrade.

6 Scalability

Ethereum is constrained by relatively low transaction speeds, processing between 12–15 transactions per second (TPS). In comparison, traditional payment networks like Visa handle approximately 2,000 TPS on average, with a peak capacity of up to 24,000 TPS, while PayPal processes around 200 TPS. These differences highlight the need for greater transaction capacity on blockchains to meet demand and reduce transaction fees.

Blockchain scaling, however, is more complex than it may initially appear. One might assume we could simply reduce the block time from 12 seconds to 6 seconds to double TPS, or alternatively, double the gas limit while keeping the block times the same. Unfortunately, both approaches have cascading effects on the decentralization and/or the security of the network.

These scaling trade-offs are often referred to as the *Blockchain Trilemma*, a term originally coined by Vitalik Buterin. The Blockchain Trilemma states that a blockchain can scale any one of the following three properties relatively easily, but scaling all three simultaneously is extremely challenging (though not an impossibility, as it is with the CAP theorem):

- **Scalability**: Increasing the throughput of the network, generally measured in Transactions Per Second (TPS).
- **Security**: Strengthening the network against takeover or compromise.
- **Decentralization**: Distributing control across the widest possible variety of participants.

To illustrate this concept more concretely, let's revisit the scaling solution we mentioned earlier—larger blocks[28]. What are the consequences of increasing block size? First, larger blocks require greater processing power to perform state transitions in a timely manner, and increased bandwidth for propagating larger blocks through the network.

As block size increases, so do computational and bandwidth demands, which in turn reduces the number of nodes that can participate. The extreme outcome would be increasing TPS to such a high level that only one node could realistically process it, effectively eliminating the blockchain's decentralization—one of its core principles.

An important principle we need to keep in mind is that when a block is added to the blockchain, all nodes in the network perform the **same** state transition function. Increasing the number of nodes participating does not increase the compute capacity of the network. To make use of these additional nodes we would need a solution which is compatible with horizontal scaling.

[28]This approach is actually known as the *big block hypothesis*

The original plan to scale Ethereum was known as sharding, which intended to split the network into 32 sub-networks, where-in validators would process a portion of the chain. This would decrease the security of each individual shard, however it would increase the throughput since shards would be processing **different** transactions.

The sharding proposal was eventually abandoned in favor of the new Rollup centric scaling solution. The rationale behind this is manifold, with some of the more high-level concerns being the additional complexity, reduced security of each shard, and the incompatibility with existing workflows and tooling.

6.1 Rollups

The rollup-centric roadmap aims to scale Ethereum through the use of a separate system known as a Layer 2 (L2s). Users can submit transactions to the L2 instead of the L1, and the L2 will mantain its own state and perform its own internal state transitions. Periodically, the L2 will batch the transactions it has received, and will synchronise its state will the L1. The core idea here is that L2s can achieve higher transaction throughputs by relaxing their decentralisation and security parameters, while L1's still exert control over certain key L2 functions, thus allowing L2s to inherit security properties from the L1.

There are two primary types of rollups: Optimistic Rollups and ZK Rollups. Though they take significantly different approaches, both hinge on solving a central challenge known as data availability ☞. To understand the broader vision, let's briefly explore these two approaches.

6.1.1 Optimistic Rollups

Optimistic Rollups, such as Arbitrum and Optimism, rely on an *optimistic* assumption: that state transitions are valid by default. This optimistic finality assumption allows the chain to process a higher number of transactions, as it does not need to concern itself with expensive consensus operations.

Blindly trusting the L2 to process the state transition function (STF) correctly is not a true scaling solution, and we need some mechanism to ensure that this is being done correctly. Thankfully we have a solution for this called *fraud proofs*, which allows users to dispute transitions of the STF and rollback any fraudulent changes ☞.

Two important concepts here are those of *sequencer* and *state root*. The sequencer is a dedicated L2 component which is responsible for batching, ordering and executing transactions on the L2. For simplicity, we shall assume that the L2 has one designated

sequencer.[29] When the sequencer tries to sync with the L1, it will submit a state root to the L1, which is a summary of all transactions which have happened on the L2. This is used to guarantee the integrity of the work done by the L2 and ensure that the L2 does not submit a false state to the L1.

The Rollup Contract The current state of the rollup is recorded on the L1 by using a smart contract known as the Rollup contract. This contract has two key responsibilities:

- Accept new state roots from the L2 sequencer
- Rollback state roots if a valid fraud proof is submitted.

Fraud Proofs The sequencer has a very powerful role, as it has sole control over everything happening on the L2. What would happen if it tried to submit a fraudulent batch of transactions, say one which included a transaction giving it all of the users money?

Users can protect themselves from this by submitting something known as a fraud proof. The general idea behind why fraud proofs work is that the STF is a deterministic algorithm (ie: for a given input we always get the same output), therefore if we were to process the same transactions claimed by the L2 and arrive to a different state root then we would know that the sequencer has submitted a fraudulent batch.

This therefore requires two things. Firstly, users need to have access to all the transactions that went into a batch, and secondly, we need a trustless mechanism to show that a batch was processed incorrectly. The former is generally accomplished by taking all the transactions, compressing them, and posting the data on-chain alongside the state root. There are also many ways to accomplish the latter, however one general scheme is the following.

First we compile the state transition function of the L2 to a work on a simple architecture such as MIPS of RISC-V. Then we implement an emulator for the chosen architecture on the L1, as a smart contract. In order to sumbit a fraud proof, the user doing this work will call the emulator with the code of the state transition and the necessary inputs. The emulator will then execute this code and determine the resulting state root after this update. If this does not match the state root held in the rollup contract, the smart contract rolls back the current state root, invalidating the transactions submitted by the L2 in the process.

[29] Although different types of sequencing such as decentralised or self-sequencing are also possible.

Transaction Compression As mentioned above, we need to make the transaction data for rollup batches available to users for the purpose of allowing them to validate the transactions and create fraud proofs. Naively posting all the L2 data onto the L1 is not an ideal strategy, as this would not only be costly, but it would also significantly increase the rate of state growth on the L1.

Therefore we need to devise a schema to compress the transactions on the L2, before posting them to L1. Whilst we could use an off-the-shelf compression algorithm, such as gzip or LZ4, we can actually be significantly more efficient by first modifying our encoding format to remove as much redundant information as possible.

The following text outlines a simple transaction encoding which achieves almost a 10x reduction.[30].

A simple Ethereum transaction (to send ETH) takes approximately 110 bytes. But using the aforementioned compression scheme, an ETH transfer on a rollup takes only around 12 bytes. Table 1 about the compression scheme can be found below.

Parameter	Ethereum (bytes)	Rollup (bytes)
Nonce	~3	0
Gasprice	~8	0–0.5
Gas	3	0–0.5
To	21	4
Value	~9	~3
Signature	~68 (2 + 33 + 33)	~0.5
From	0 (recovered from sig)	4
Total	~112	~12

Table 1: Comparison of Ethereum and Rollup Transaction Sizes

- **Nonce**: The nonce prevents replays. For an Ethereum transaction, the current nonce of an EOA (e.g., 5) must match the transaction's nonce. Once processed, the account's nonce increments, blocking replay of the transaction. In the rollup the nonce can be omitted, as it can be recovered from the pre-state; if someone tries replaying a transaction with an earlier nonce, the signature check would fail since it would be checked against data that includes the higher nonce.

- **Gasprice**: A fixed range of gas prices can be used, such as 16 consecutive powers of two. Alternatively, gas payment could occur outside the rollup protocol, with transactors paying batch creators through an external channel.

- **Gas**: Similarly to gas price, total gas can be restricted to consecutive powers of two, or limited only at the batch level.

[30]See Vitalik *Buterin's Incomplete Guide to Rollups* at https://vitalik.eth.limo/general/2021/01/05/rollup.html

- **To**: Rather than a 20-byte address, we use an index (e.g., if an address is the 4527th added to the L2 state tree, the index 4527 is used).
- **Value**: The value can be stored in scientific notation, as most transfers only need 1-3 significant digits.
- **Signature**: BLS aggregate signatures enable aggregation of many signatures into a single 32-96 byte signature, depending on the protocol. This signature verifies the entire batch's messages and senders at once. The ~0.5 in the table reflects a per-batch signature cost, given limits on signatures that can be aggregated for single-block verification.

Withdrawals A critical component of L2 systems is a bridge which allows for transfers of funds between the L1 and the L2 and vice versa. Transfers from L1 to the L2 are seamless, with a bridging time of around 15 minutes, which is the time required to achieve L1 finality. Bridging from L2 back to L1, however, takes approximately 7 days.

The reason for this significant delay is the fraud proof mechanism mentioned earlier. Users cannot submit fraud proofs at any time for any batch, rather they are given a 7 day window, known as a challenge period, to submit their fraud proofs for a given batch.

If a challenge were to occur, then any transfers that happened, say depositing funds into the L2 bridge, would be reverted. Therefore to prevent the bridge from losing funds, and the L2 becoming insolvent, we do not allow users to withdraw funds before the challenge period is closed.

6.1.2 ZK-Rollups

Zero-Knowledge (ZK) Rollups share many similarities with Optimistic Rollups. The use a rollup contract on the L1 to store the current state root, the L2 posts batches of transactions to the L1, and there is a bridge between the L1 and L2 to handle fund transfer. The key distinction between the two lies in their finality mechanism.

ZK-Rollups utilize either ZK-STARKs or ZK-SNARK to perform what is known as *verifiable computation* (VC). The specific differences between STARKs and SNARKs are beyond the scope of this unit, but VC provides a reliable way to prove that a certain computation was completed correctly.

The zero-knowledge properties of these techniques isn't something we particularly care about when implementing rollups, and we mainly rely on their ability to produce short proofs. It should be noted however that we do very much still need to send transaction data from the L2 to the L1, because of issues related to data availability.

Verifiable Computing One of the original motivations for VC was distributed computing networks, such as the Berkeley Open Infrastructure for Network Computing (BOINC). BOINC enables individuals to donate a portion of their hardware resources to a global distributed computing network. You might have heard of a similar project, Folding@Home, which performs protein folding simulations to support medical research, including cancer and drug development. These systems operate by assigning each computer a unique simulation to run, then collecting a large dataset of results to analyze for significant patterns.

Now, suppose we wanted to reward participants for their contributions, thus incentivizing more people to offer their computing power. If we simply paid users for submitting results, some malicious actors might send back random, meaningless data as quickly as possible just to earn rewards. This would undermine the entire initiative, wasting resources on useless data. The only way to verify that a simulation was run correctly would be to rerun it ourselves, defeating the purpose of outsourcing the computations. Keep this analogy in mind as it illustrates the usefulness of verifiable computation.

What VC enables us to do is to take our program (in this case, the simulation software) and transform it into something known as a *circuit*. We can then execute this circuit with our inputs, and it will return the same outputs as the original program. So far, this may sound convoluted, as we have simply transformed our program from one representation to another before executing it. However, the key insight with circuits is that we can take an *execution trace* of the circuit and highly compress it using advanced mathematics. We can then send this compressed execution trace along with the output to the other party.

This compressed execution trace allows any party with the original circuit to verify that the program was indeed executed and that the provided output was derived from the given inputs using the program. Crucially, verifying this proof is much faster than re-executing the original program. For example, if verification is 10 times more efficient, then a 4-core machine could verify outputs as if they were generated by a 40-core machine. This scalability is precisely what makes verifiable computation so valuable in a distributed supercomputer setting.

Using Zero-Knowledge Proofs for State Transitions ZK-Rollups make use of verifiable computation by applying it to their state transition function (STF). Instead of submitting a state root, and waiting 7 days until someone disproves it, the L2 instead instead publishes the state root and a proof to verify that the state root was calculated correctly. This proof can even be verified on-chain inside a smart contract, which gives us 100% certainty that the state transition was performed correctly.

It should be noted however that the proof by itself is not enough. What if the L2 sequencer were to disappear? We could elect a new sequencer to pick up where the last one left off, however the new sequencer doesn't know the entire state. When we execute

transactions on the L2, we make changes to the state trie, a balance here, a storage value there. Without knowing these values we cannot process any new transactions. Therefore either the transactions or the state updates must also be posted alongside the proof.

Table 2 shows a comparison between Opimistic Rollups and ZK-Rollups.

6.1.3 State Growth, Blobs and the Data Availability Problem

Regardless of the rollup solution chosen, we need to make some amount of data available to the public, so as to allow people to recreate the state of the L2 in case it goes down. Initially this was achieved by having the sequences post "bogus" transactions, where-in they send 1 Wei to a specified account, and put all the data they need to make available as calldata.

Whilst this sounds like a fine solution, it causes issues for state growth. We mentioned earlier that as we increase the compute requirements of the network, we decrease the decentralization of it. Storage is a measure of compute like any other, so a rapidly growing state will cause some problems.

We know that for optimistic rollups, we actually only need to keep the transaction data around for 7 days, the length of the contest period. What if there were some temporary store of data we could use for Ethereum?

This is the central idea behind blobs, originally detailed under the proto-danksharding proposal (EIP 4844). The entire EIP is quite complicated, but essentially we make a handful of 1MB blobs available for purchase at each slot of the Ethereum chain. These blobs are an arbitrary storage area, which is persisted for 2 weeks on the Ethereum chain. After those 2 weeks nodes are no longer required to store the data, and are free to delete it.

The advantage of this approach is that we only suffer a known constant storage overhead, which will not be added to the permanent space. Assuming 3 blobs per slot, and 12s per slot, in 2 weeks we would need to store an additional 300GB of storage space. Whilst this is quite a bit of data, node operators do not incur that much of an additional cost to store it, and they also only incur this cost once as the space is "rolling".

Property	Optimistic Rollups	ZK Rollups
Fixed gas cost per batch	~40,000 (a lightweight transaction that mainly just changes the value of the state root)	~500,000 (verification of a ZK-SNARK is quite computationally intensive)
Withdrawal period	~1 week (withdrawals need to be delayed to give time for someone to publish a fraud proof and cancel the withdrawal if it is fraudulent)	Very fast (just wait for the next batch)
Complexity of technology	Low	High (ZK-SNARKs are very new and mathematically complex technology)
Generalizability	Easier	Harder (ZK-SNARK proving general-purpose EVM execution is much harder than proving simple computations, though there are efforts (e.g., Cairo) working to improve on this)
Per-transaction on-chain gas costs	Higher	Lower (if data in a transaction is only used to verify, and not to cause state changes, then this data can be left out, whereas in an optimistic rollup it would need to be published in case it needs to be checked in a fraud proof)
Off-chain computation costs	Lower (though there is more need for many full nodes to redo the computation)	Higher (ZK-SNARK proving especially for general-purpose computation can be expensive, potentially many thousands of times more expensive than running the computation directly)

Table 2: Comparison of Optimistic Rollups and ZK Rollups

6.1.4 Rollup Stages and Risk Analysis

Rollups can be categorized into various 'stages', determined by a number of security properties they satisfy, which indicate how centralised or decentralised the rollup is. The initial work to formalise this was spearheaded by L2Beat, which developed a comprehensive system for breaking down the different security properties of rollups.

The security properties considered by L2Beat are the following:

- Data Availabilty: Whether the data required to construct the state/proofs is published on-chain
- State Validation: Whether fraud proofs or zk proofs are being used
- Sequencer Failure: Whether users can sequence their own transactions, or whether they can force the sequencer to include their transactions
- Proposer Failure: Whether users can post their own state roots and withdraw their funds from the L2
- Exit Window: Whether the system is either not upgradeable or upgrades to the system have enshrined delays to allow users to exit

Using these security properties, L2Beat categorises rollups into 3 stages:

- Stage 0 — Full Training Wheels: At this stage, the rollup is effectively run by the operators. Still, there is an source-available software that allows for the reconstruction of the state from the data posted on L1, used to compare state roots with the proposed ones.

- Stage 1 — Limited Training Wheels: In this stage, the rollup transitions to being governed by smart contracts. However, a Security Council might remain in place to address potential bugs. This stage is characterized by the implementation of a fully functional proof system, decentralization of proof submission, and provision for user exits without operator coordination. The Security Council, comprised of a diverse set of participants, provides a safety net, but its power also poses a potential risk.

- Stage 2 — No Training Wheels: This is the final stage where the rollup becomes fully managed by smart contracts. At this point, the proof system is permissionless, and users are given ample time to exit in the event of unwanted upgrades. The Security Council's role is strictly confined to addressing soundness errors that can be adjudicated on-chain, and users are protected from governance attacks.

At the time of writing, November 2024, Arbitrum and Optimism have reached stage 1 as Optimistic Rollups and ZKSync has reached stage 1 as a ZK-Rollup.

7 Appendix

7.1 Merkle Patricia Tries

Ethereum uses *Merkle-Patricia tries* [31] to store certain items of information, such as the state trie.

A Merkle-Patricia trie can be seen as an optimisation of a data structure known as a radix tree, which is a special case of a prefix-tree. In addition this data structure is augmented with functionality found in Merkle trees.

Thus to explain the concept of a Merkle Patricia Trie, we will have to first understand Merkle trees, then Prefix Trees, and then Radix Trees. Then we will finally combine these concepts into a Merkle-Patricia Trie.

7.1.1 Merkle Tree

Merkle trees are a tree based data structure, where data is stored in the leaves, and leaves can be accessed by taking paths through the branches of the tree. The main innovation in Merkle trees lies in their use of a hashing function to both guarantee the integrity of the tree as well as to allow efficient verification of whether a piece of data is part of the data structure or not.

A hashing function is a one-way function, which takes an arbitrary amount of data in and returns a fixed size piece of data known as a digest. Given a digest it should not be possible to reverse the function and retrieve the original data[32]. The security properties of hash functions are determined by their construction, and are usually classified by the size of the output digest.

In a Merkle tree, data is stored in the leaves, with each non-leaf (branch) node containing a hash of its child nodes. If h denotes the hash function, we can visualise a Merkle tree as follows:

[31] Patricia stands for Practical Algorithm To Retrieve Information Coded in Alphanumeric.

[32] There are some other important security properties which hashing functions need to satisfy. The one mentioned above is called pre-image resistance. There are also second pre-image resistance, collision resistance, and pseudo-randomness

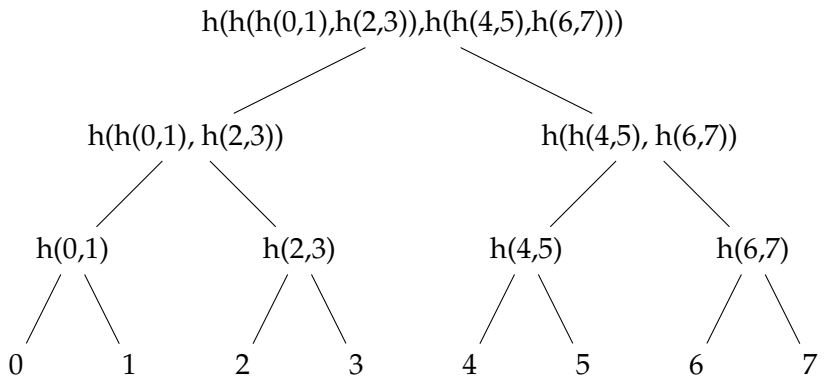


Figure 7: A Merkle Tree

Let us denote the root of the tree as N_r and all the other nodes as N_s, where s is a binary string. Each bit of the binary string will represent taking either the left or right path from the root, read from left to right. For instance N_{000} represents the left most leaf, N_{111} represents the right most leaf, and N_{011} represents the node with the value 3 in the diagram.

The value of the root N_r (in this case: $h(h(h(0,1),h(2,3)),h(h(4,5),h(6,7))))$ is called the root hash, and it can be used to summarise the entire tree.

Importantly, none of the leaf values can change without changing the root hash of the tree, and thus knowledge of the root hash prevents tampering with the tree. For instance, if we wish to replace the leftmost node of the tree with a value $x \neq 0$, we would have to find an x such that $h(h(h(x,1),h(2,3)),h(h(4,5),h(6,7))) = h(h(h(0,1),h(2,3)),h(h(4,5),h(6,7)))$, which is computationally infeasible.

Using this root hash we can also prove that a certain item of data lies in a particular leaf node of the tree. We can also do so efficiently, in the sense that we do not need to reproduce the entire tree to show that this is the case. Instead, we can provide a short proof of this fact. To do so we need to provide the root hash as well as a number of intermediate nodes which enable to compute the hashes leading from that leaf node to the root of the tree.

For instance, assuming that we have already shared the root hash with someone, proving that $N_{000} = 0$ to that person only requires us to share the following, rather than all the data in the leaves.

- $N_{000} = 0$
- $N_{001} = 1$
- $N_{01} = h(N_{010}, N_{011})$
- $N_1 = h(h(N_{100}, N_{101}), h(N_{110}, N_{111}))$

The key insight here is that nodes N_{01} and N_1 are able to summarise the values in their subtrees thanks to the hash digest.

If we assume a 32 byte word size, N_{01} saves 32 bytes[33], and N_1 saves 96 bytes[34]. Therefore our proof that $N_{000} = 0$ has been compressed from sharing 256 bytes to sharing 128 bytes, saving 50% of data transmission costs. It is trivial to see how scaling this system up results in even more data savings.

7.1.2 Prefix Trees, Radix Trees and Patricia Tries

Prefix Trees, sometimes referred to as *tries*, are a tree based data structure optimised for efficient storage and search of data. Consider the following two diagrams, which show three items of data (cat, car and dog) stored in a binary tree and a prefix tree respectively.

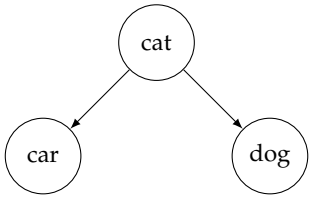


Figure 8: Binary Tree

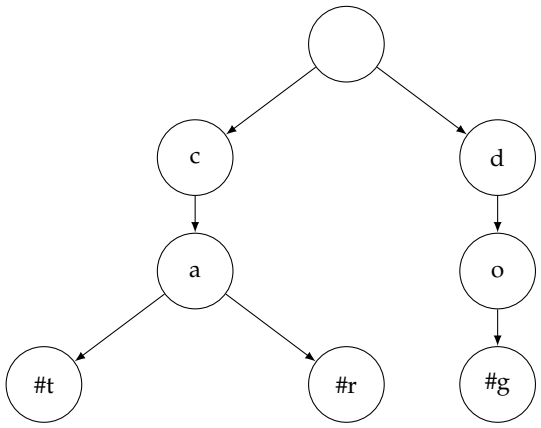


Figure 9: Prefix Tree (Trie)

In the binary tree on the left, each node contains a value along with two pointers to its left and right children. To search for an element in the tree, we can use binary search until we find the desired element.

One drawback of this storage method is data duplication. For example, the values "cat" and "car" share the letters "c" and "a", or more specifically the prefix "ca". In contrast, the prefix tree on the right stores a single letter in each node. Whilst this results in using more nodes, the storage cost is amortized because the letters "c" and "a" are only stored once.

In order to see the true benefit of prefix trees, one has to consider larger datasets. The space savings in a prefix tree increase as the number of items increase, since we find

[33] two 32 byte elements, summarised into 32 bytes
[34] four 32 byte elements, summarised into 32 bytes

more common prefixes. The search performance of a prefix tree is slightly different from that of a tree as it is $\mathcal{O}(n)$ in the length of the key, whereas trees are $\mathcal{O}(\log n)$ in the number of elements. As the number of elements increases, the performance of the prefix tree becomes more appealing.

From the representation above you may have noticed that our encoding for the word "dog" is rather inefficient. We do not need a node for each letter if we are only representing one word, as we do not have any common prefixes with this word.

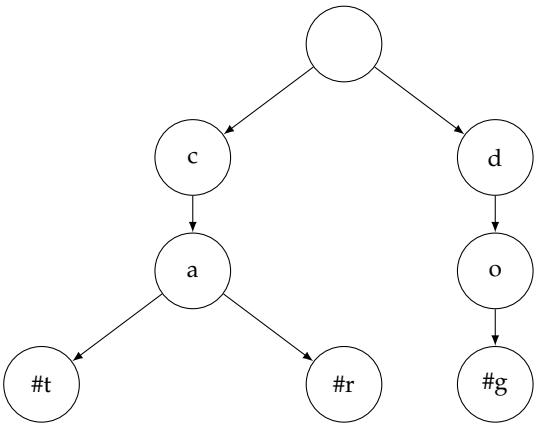


Figure 10: Prefix Tree (Trie)

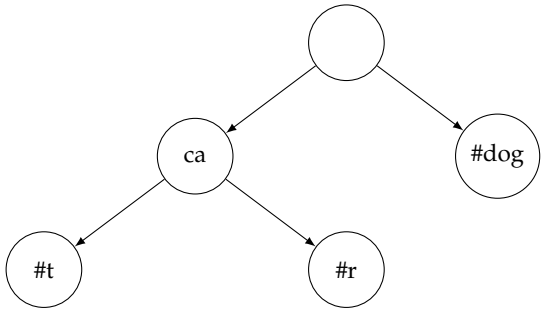


Figure 11: Radix Tree (Trie)

A radix tree resolves this problem by merging single child nodes with their parent. There are some disadvantages to this approach, namely that inserting in or updating the tree may require splicing nodes, however we do save a significant amount of space and improve search performance (in the practical case, not in computational complexity terms).

Patricia Tries are a special case of the Radix Tree. In this case keys are encoded in binary and each node is either a zero or one value from the keys stored in the trie.

7.1.3 Ethereum's Merkle-Patricia Trie

We now have all the building blocks to understand Merkle-Patricia Tries as they are used in Ethereum.

Similarly to Merkle Tree, the leaves of the Merkle-Patricia Trie will be the ones to hold the data (for example Ethereum accounts). All other nodes will keep a hash of their respective subtrees.

On the other hand, from the Patricia tries we take the concept of being able to being able to lookup these leaves using a key (often an identifier, such as the address of the

account). We store this key in the intermediate nodes of the tree, forming a path from the root of the tree to the leaf nodes. These paths are stored exploiting the optimisations which a Radix tree makes available. Instead of encoding the keys in binary like a normal Patricia tree, we encode them in hexadecimal, since this is the perfect format for storing addresses (which are represented as 40 hexadecimal values).

The resulting Merkle-Patricia trie has three types of nodes:

- Branch Nodes - Branch nodes consists of a 16 element array corresponding to the hexadecimal characters from 0 to F. These point to other nodes in the tree. They also contain the hash of the node's subtrees.
- Extension Nodes - Extension nodes function as radix-optimized nodes within the trie. They are used when a branch node has only one child node. Instead of having a long sequence of nodes, this is compressesed into an extension node. The extension node holds the prefix which it is representing, and the hash of its subtrees.
- Leaf Nodes - Leaf nodes contain the data of the tree.

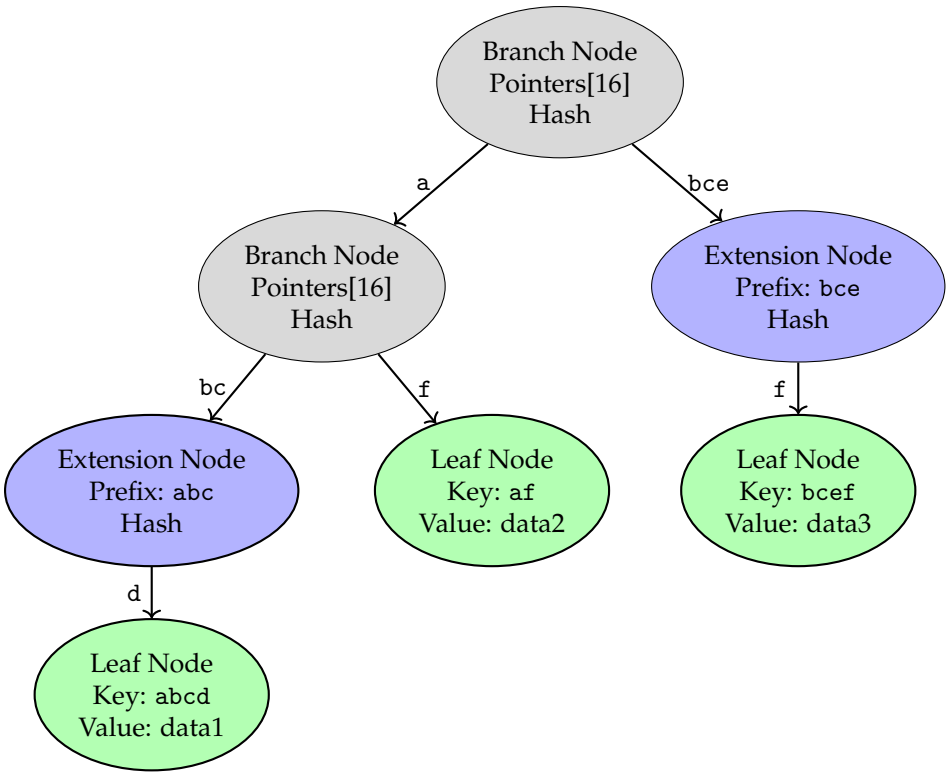


Figure 12: A Merkle Patricia Trie with 3 leaf nodes. The annotations on the edges are for readability only. So is the Key inside the leaf node.

7.2 Cryptography

Cryptography underpins much of modern infrastructure and is crucial to blockchain technologies. In this course, we won't dive into the rigorous mathematical theory behind cryptography—that requires an entire course on its own. Instead, these next sections will introduce key cryptographic building blocks and provide an intuition for how and why they work.

A central concept in cryptography is *computational infeasibility*, the idea that certain tasks require so much computational power that they are practically impossible. A simple analogy is the lottery: while it's not impossible to win, it's economically infeasible to guarantee a win by purchasing enough tickets.

For example, if a lottery ticket has a one in a million chance of winning (0.0001%), buying more tickets increases your odds, but at a cost. If I have $5000 and tickets cost $2, I can buy 2500 tickets and increase my odds to 0.25%. Even then, the likelihood of winning is low, and I've used all my resources. With $2 million, I might boost my odds to 50%, but still have no guarantee of winning (and will still not recoup my losses).

Computational infeasibility operates in a similar way. Imagine I choose a random 64-bit number, and you attempt to guess it. You would have a 1 in 2^{64} chance of guessing correctly on the first try. Let's say you have a 4.0 GHz CPU and can guess one number per cycle, allowing you to guess approximately 10^9 numbers per second, or 2^{32} numbers. It may seem like you're covering significant ground, but that's not the case.

After two seconds, you've guessed 2^{33} numbers. Each increment of the exponent represents a doubling, so reaching 2^{63}, which is half the search space, would take:

$$\frac{2^{63}}{2^{30}} = 2^{33} \text{ seconds} \approx 68 \text{ years}.$$

The number of bits in a number determines how computationally feasible it is to brute-force. As the bit length increases, so does the time required to break it by brute force. While we can accelerate this with better hardware like GPUs or using parallel processing, today's rule of thumb for secure systems is to use at least 80 bits of security, with 128 bits being more commonly recommended. In post-quantum contexts, this increases to 256 bits, as in the case of Ethereum.

7.2.1 Hash Functions

Hash functions can be viewed as deterministic, one-way compression functions. Their purpose is to take data of arbitrary length and produce a fixed-size output called a digest.

This process should be deterministic, meaning that the same input will always result in the same digest.

There are two primary types of hash functions: regular hash functions and cryptographically secure hash functions. You may already be familiar with regular hash functions from data structures like hash maps. A hash function is considered cryptographically secure when it satisfies the following properties:

- **Preimage Resistance**: Given an output z, it should be computationally infeasible to find any input x such that $h(x) = z$. In other words, the hash function is one-way.
- **Second Preimage Resistance**: Given an input x_1 and its hash $h(x_1)$, it should be computationally infeasible to find any x_2 where $h(x_1) = h(x_2)$.
- **Collision Resistance**: It should be computationally infeasible to find two distinct inputs $x_1 \neq x_2$ such that $h(x_1) = h(x_2)$.

While second preimage resistance and collision resistance might sound similar, they are distinct due to a concept called the birthday paradox (or birthday attack). The paradox reveals that although the probability of a specific pair of elements colliding is low, the probability of *any* two elements colliding is much higher. This occurs because as more elements are generated, they must map to a shrinking set of possible values, increasing the chance of collisions.

In the birthday paradox problem, the probability of n people not having a common birthday is the following:

$$P(\text{no shared birthday}) = 1 \times \frac{364}{365} \times \frac{363}{365} \times \frac{362}{365} \times \cdots \times \frac{365-(n-1)}{365} = \prod_{k=0}^{n-1} \frac{365-k}{365}$$

In the collision resistance case we have 2^m instead of 365, where m is the size of the digest. As we consider larger groups of inputs n and check whether they have a hash collision, the probability of this occurring will quickly increase.

Efficiency is also a consideration when using hash functions. For applications that don't require security, we aim to balance the distribution of the hash with execution speed. When a hash collision occurs, we must iterate through all elements mapped to the same bucket. Regular hash functions strive for efficiency because the cost of scrambling data (to minimize collisions) may outweigh the benefits, making it cheaper to simply iterate through the elements.

Cryptographically secure hash functions, on the other hand, prioritize collision resistance over speed. Highly optimized hash functions can actually reduce security, as adversaries

may exploit the efficiency to brute force more hashes. While the hash function must execute in a reasonable time (waiting five minutes for a digest would be impractical), improving efficiency too much can weaken security by enabling faster attacks.

The security of a hash function is measured by its bit length. Preimage resistance typically requires 2^n operations, while collision resistance requires about $2^{\frac{n}{2}}$ operations.

7.2.2 Asymmetric Cryptography

Asymmetric cryptography, also known as public-key cryptography, is a method of encryption where two distinct but mathematically related keys are used: a public key and a private key. Unlike symmetric cryptography, where the same key is used for both encryption and decryption, asymmetric cryptography uses one key for encryption (the public key) and a separate key for decryption (the private key).

The public key can be shared freely and is used to encrypt data, while the private key is kept secret and is used to decrypt the data. This ensures that even if someone has access to the public key, they cannot decrypt messages without the corresponding private key.

Asymmetric cryptography relies on trapdoor functions, problems which are easy to compute one way, but hard to compute in reverse. The most common example for this, and the one used in schemes such as RSA is prime factorization. Given two prime numbers p and q we can multiply them together to get a new prime number n, this operation is trivial. Going from a given n into its corresponding p and q requires checking all the prime numbers up to $\sqrt{n}$.

If we wanted to build a cryptographically secure scheme using the above, we would need to guarantee at least 128 bits of security. This means that we need to ensure that an attacker needs to go through at least 2^{128} values to break our cryptography.

To achieve this, our key length needs to have n bits, such that $\sqrt{2^n} = 2^{128}$, which means that n has to be at least 256 bits. In practice we can break this scheme much more efficiently, through algorithms such as the General Number Field Sieve (GNFS). To counter this algorithm, the number of key bits to security bits needs to scale non-linearly, and 128-bits of security can only be achieved with a 3072-bit key.

There are two key properties that asymmetric cryptography must ensure:

- **Confidentiality**: Only the holder of the private key can decrypt a message encrypted with the public key, ensuring the secrecy of the message.
- **Authentication**: The private key can also be used to sign a message, and anyone with the corresponding public key can verify that the message originated from the

owner of the private key.

These properties make asymmetric cryptography suitable not only for secure message transmission but also for digital signatures, where the private key is used to sign a document and the public key is used to verify its authenticity.

7.2.3 Signatures

Signatures have been mentioned numerous times in the text so far, but we have yet to explain them properly. In practice, they function much like real-world signatures (though they are actually more secure) and can be used to verify that a particular party has issued a message.

There are various ways to construct digital signatures, some using symmetric cryptography and others asymmetric. With an understanding of encryption and hash functions, you should be able to conceive of basic schemes for both.

One way to create a signature scheme using asymmetric cryptography is to hash the message and encrypt the resulting digest as follows:

$$s = e_{\text{priv}}(h(x)).$$

The receiving party can verify this signature by checking that

$$e^{-1}_{\text{pub}}(s) \equiv h(x),$$

where s is the signature and x is the message. If the decrypted signature does not match the hashed message, then the signature is invalid.

In this notation, the superscript e^{-1} denotes decryption, while e alone represents encryption. The subscript indicates whether it is the public key (e_{pub}) or the private key (e_{priv}). It is important to note that for an encryption function, decryption can only be performed using the complementary key, i.e., $e^{-1}_{\text{pub}}(e_{\text{priv}})$ or $e^{-1}_{\text{priv}}(e_{\text{pub}})$.

7.2.4 Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) isn't a major departure from regular (a)symmetric cryptography, it simply uses a different trap door function based on the mathematics of elliptic curves[35]. The choice for ECC over traditional cryptography lies in its reduced key size, allowing us to preserve the same security properties whilst being more efficient. The latter is especially important for lower power devices such as mobile phones.

[35]The problem is known as the discrete log problem

7.2.5 BLS Signatures

Boneh-Lynn-Shacham (BLS) signatures are a form of cryptographic signature based on pairings on elliptic curves, specifically designed to create short, verifiable digital signatures. BLS signatures rely on a mathematical operation called a *bilinear pairing*, which maps points on elliptic curves in a way that enables unique functionality, particularly efficient aggregation and verification of signatures. The full mathematical details of how this works are beyond the scope of this course.

The main advantage of BLS signatures is that they provide a concise way to allow multiple signers for a given message. Consider a message m that needs to be signed by several signers $e_1, e_2, \ldots, e_n$. Without aggregation, we would need to produce individual signatures as follows:

$$s_1 = e_1(h(m)), \quad s_2 = e_2(h(m)), \quad \ldots, \quad s_n = e_n(h(m))$$

If each signature is 32 bytes, transmitting all signatures would require $n \times 32$ bytes of data. For example, with n as one million (the current number of Ethereum validators), this would be 32 MB of data!

Ideally, we want a way to compress these signatures while still being able to validate them. BLS signatures allow us to *aggregate* both the signatures and the corresponding public keys so that an aggregated signature can be validated using an aggregated public key. In essence, we produce:

$$s_a = s_1 \oplus s_2 \oplus \cdots \oplus s_n$$

and an aggregated public key:

$$e_{\text{pub a}} = e_{\text{pub 1}} \oplus e_{\text{pub 2}} \oplus \cdots \oplus e_{\text{pub n}}$$

such that $e^{-1}_{\text{pub a}}(s_a) \equiv e^{-1}_{\text{pub a}}(e_{\text{priv a}}(h(m))) \equiv h(m)$. Here, $e_{\text{priv a}}$ represents a hypothetical "super private key" for all combined signers.

In reality, the operations involved are more complex, but this construction gives a general working model. Under this setup, if we want to prove that a specific set of signers $(e_1, e_2, \ldots, e_n)$ all signed a message m, we only need to send m, the aggregated signature s_a, and the list of signers.

The recipient can construct $e_{\text{pub a}}$ and verify the signature s_a. This scheme allows us to send just one 32-byte signature and a bitmap to indicate the set of signers. The size of the bitmap field is the length of the validator set, for 1 million validators we would use 1 million bits, or 0.125Mb. With these assumptions we can save 99.6% in data transmission.

7.3 RANDAO

The RANDAO (Random Number and Distribution Accumulation Oracle) is a critical part of Ethereum's consensus layer infrastructure and provides the system with trustless randomness. Core operations such as choosing which validator gets to be the proposer and which validators get to attest a slot need to be random for the sake of security[36].

The RANDAO is a random value which is updated during each slot of an epoch. During the first slot, the first proposer determines the RANDAO value Subsequently, during each slot, that slot's proposer mixes a new value with the RANDAO value from the previous slot to generate a new RANDAO value.

Since controlling the RANDAO value can allow a validator to exert some control over the network, we have to restrict the values which can be proposed by validators, as well as the way these values are mixed with the current RANDAO value. This can be achieved through the use of public-private key signatures as follows.

Each validator possesses a public-private BLS key pair which it uses to sign messages at the consensus layer of the network. When it becomes a proposer, the validator computes a value called *randao reveal* by signing the current epoch with its private key. The corresponding public key is registered with the Ethereum network and is known by other validators on the network.

This scheme has three consequences. Firstly, the validator is not able to control the value of the randao reveal. Secondly the other validators know that the validator didn't just make up the value, because the signed message can be verified using the public key. Thirdly the other validators cannot know the value being generated in advance as this would require knowledge of the proposer's private key.

Once the randao reveal value has been computed by the proposer, it is hashed to compress it down to 256 bits. The resulting value is mixed with the existing RANDAO value using XOR.

On concern with RANDAO is that the final proposer in an epoch has some control over the final value. This is because the proposer can compute the upcoming RANDAO value in advance, and then decide whether it prefers the new RANDAO value or the previous RANDAO value. If it prefers the new RANDAO value, it will propose a block to change the RANDAO value. However, if it does not it will not propose a block. The result is that the previous block becomes the last block of the epoch, and its RANDAO value becomes the last RANDAO value of the epoch.

The problem is that the RANDAO value of epoch N is used to determine proposers and

[36] An example of an attack which can be carried out against a predictable system is to mount a DoS attack to knock out specific validators and take control of a committee

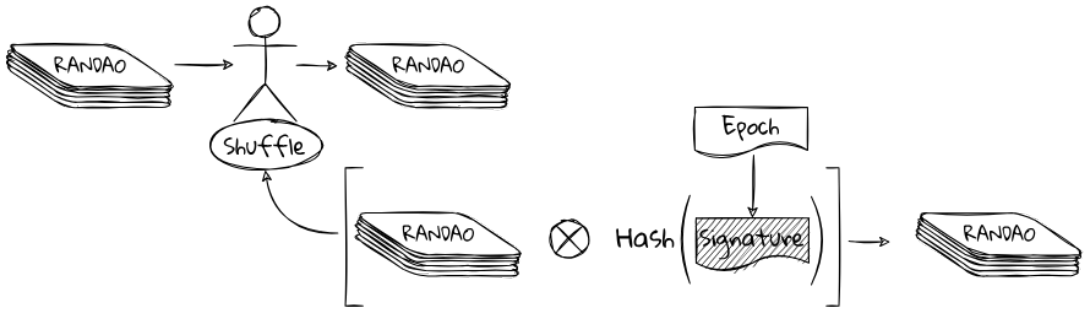


Figure 13: Combining Randao

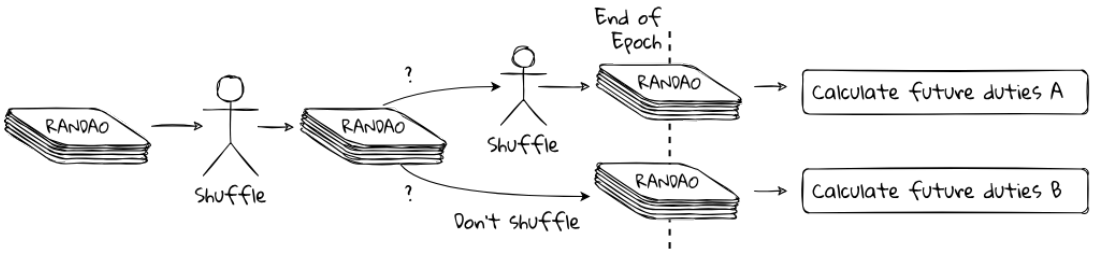


Figure 14: RANDAO biasing

validators for each slot in epoch $N + 2$ [37]. Being chosen as a proposer has an economic advantage in terms of rewards and allows the censorship of transactions. In addition if an attacker is able to control multiple slots (perhaps even by mounting a DoS attack on intervening proposers), the attacker can also affect the outcome of lottery contracts (since a slot's RANDAO value can also be used by some smart contracts as a source of on-chain randomness). This is a disadvantage of RANDAO which one needs to be aware of.

[37] This also means that an attacker cannot mainpulate proposers and validators for epoch $N + 2$.