**YOU DON’T KNOW ETHEREUM**

**INTRODUCTION**

Vitalik Buterin conceptualized Ethereum in the year 2013.

The core idea proposed was the development of a Turing-complete language that allows the development of arbitrary programs (smart contracts) for blockchain and **Decentralized Applications** (**DApps**). This concept is in contrast to Bitcoin, where the scripting language is limited and only allows necessary operations.

WHAT IS ETHEREUM?

Ethereum is often described as “the world computer.”

Ethereum is a deterministic but practically unbounded state machine, consisting of a globally accessible singleton state and a virtual machine that applies changes to that state.

From a more practical perspective, Ethereum is an open source, globally decentralized computing infrastructure that executes programs called smart contracts. It uses a blockchain to synchronize and store the system’s state changes, along with a cryptocurrency called ether to meter and constrain execution resource costs.

The Ethereum platform enables developers to build powerful decentralized applications with built-in economic functions. While providing high availability, auditability, transparency, and neutrality, it also reduces or eliminates censorship and reduces certain counterparty risks.

THE ETHEREUM NETWORK

The Ethereum network is a peer-to-peer network where nodes participate in order to maintain the blockchain and contribute to the consensus mechanism. Networks can be divided into three types, based on the requirements and usage. These types are described in the following subsections.

1. The mainnet

The **mainnet** is the current live network of Ethereum. Its network ID is 1 and its chain ID is also 1. The network and chain IDs are used to identify the network. A block explorer that shows detailed information about blocks and other relevant metrics is available at <https://etherscan.io>. This can be used to explore the Ethereum blockchain.

1. Testnets

There is a number of testnets available for Ethereum testing. The aim of these test blockchains is to provide a testing environment for smart contracts and DApps before being deployed to the production live blockchain. Moreover, being test networks, they also allow experimentation and research. The main testnet is called Ropsten, which contains all the features of other smaller and special-purpose testnets that were created for specific releases. For example, other testnets include Kovan and Rinkeby, which were developed for testing Byzantium releases. The changes that were implemented on these smaller testnets have also been implemented in Ropsten. Now the Ropsten test network contains all properties of Kovan and Rinkeby.

## Private nets

As the name suggests, these are private networks that can be created by generating a new genesis block. This is usually the case in private blockchain networks, where a private group of entities start their blockchain network and use it as a permissioned or consortium blockchain.

COMPARING ETHEREUM TO BITCOIN

Both have much similarities.

Yet in many ways, both the purpose and construction of Ethereum are strikingly different from those of the open blockchains that preceded it, including Bitcoin.

Ethereum’s purpose is not primarily to be a digital currency payment network. While the digital currency ether is both integral to and necessary for the operation of Ethereum, ether is intended as a utility currency to pay for use of the Ethereum platform as the world computer.

Unlike Bitcoin, which has a very limited scripting language, Ethereum is designed to be a general-purpose programmable blockchain that runs a virtual machine capable of executing code of arbitrary and unbounded complexity. Where Bitcoin’s Script language is, intentionally, constrained to simple true/false evaluation of spending conditions, Ethereum’s language is Turing complete, meaning that Ethereum can straightforwardly function as a general-purpose computer.

STAGES OF ETHEREUM DEVELOPMENT

1. Frontier
2. Homestead
3. Metropolis
4. Serenity

THE GENERAL PURPOSE BLOCKCHAIN

Ethereum is a distributed state machine.

Ethereum has memory that stores both code and data, and it uses the Ethereum blockchain to track how this memory changes over time. Like a general-purpose stored-program computer, Ethereum can load code into its state machine and run that code, storing the resulting state changes in its blockchain. Two of the critical differences from most general-purpose computers are that Ethereum state changes are governed by the rules of consensus and the state is distributed globally.

COMPONENTS OF ETHEREUM BLOCKCHAIN

The Ethereum blockchain stack consists of various components. At the core, there is the Ethereum blockchain running on the peer-to-peer Ethereum network. Secondly, there's an Ethereum client (usually Geth) that runs on the nodes and connects to the peer-to-peer Ethereum network from where blockchain is downloaded and stored locally. It provides various functions, such as mining and account management.

1. P2P Network
2. Consensus rules
3. Transactions
4. The EVM or State machine
5. Data structures or Ether crypto/tokens
6. Consensus algorithm
7. Clients
8. Accounts
9. Keys and addresses
10. Smart Contracts

ETHEREUM AND TURING COMPLETENESS

As soon as you start reading about Ethereum, you will immediately encounter the term “Turing complete.” Ethereum, they say, unlike Bitcoin, is Turing complete. What exactly does that mean? The term refers to English mathematician Alan Turing, who is considered the father of computer science. Alan Turing further defined a system to be Turing complete if it can be used to simulate any Turing machine. Such a system is called a Universal Turing machine (UTM).

Ethereum’s ability to execute a stored program, in a state machine called the Ethereum Virtual Machine, while reading and writing data to memory makes it a Turing-complete system and therefore a UTM. Ethereum can compute any algorithm that can be computed by any Turing machine, given the limitations of finite memory.

Ethereum’s groundbreaking innovation is to combine the general-purpose computing architecture of a stored-program computer with a decentralized blockchain, thereby creating a distributed single-state (singleton) world computer. Ethereum programs run “everywhere,” yet produce a common state that is secured by the rules of consensus.

PROGRAMMING DAPPS

Ethereum started as a way to make a general-purpose blockchain that could be programmed for a variety of uses. But very quickly, Ethereum’s vision expanded to become a platform for programming DApps. DApps represent a broader perspective than smart contracts. A DApp is, at the very least, a smart contract and a web user interface. More broadly, a DApp is a web application that is built on top of open, decentralized, peer-to-peer infrastructure services.

A DApp is composed of at least:

* Smart contracts on a blockchain
* A web frontend user interface

In addition, many DApps include other decentralized components, such as:

* A decentralized (P2P) storage protocol and platform
* A decentralized (P2P) messaging protocol and platform

ETHEREUM BASICS

Ethereum’s blockchain is being run by a currency called ether (ETH). Ether is subdivided into smaller units, down to the smallest unit possible, which is named wei.

Ethereum is the system or protocol or network, ether is the cryptocurrency or currency for short.

CHOOSING AN ETHEREUM WALLET

An Ethereum wallet is your gateway to the Ethereum system. It holds your keys and can create and broadcast transactions on your behalf.

Examples of some good wallets include: metamask, jaxx, myethereumwallet etc.

HOW TO CREATE METAMASK WALLET

* 1. Open the Google Chrome browser and navigate to [*https://chrome.google.com/webstore/category/extensions*](https://chrome.google.com/webstore/category/extensions).
  2. Search for “MetaMask” and click on the logo of a fox.
  3. Once you confirm you are looking at the correct extension, click “Add to Chrome” to install it.

Once MetaMask is installed you should see a new icon (the head of a fox) in your browser’s toolbar. Click on it to get started. You will be asked to accept the terms and conditions and then to create your new Ethereum wallet by entering a password

Once you’ve set a password, MetaMask will generate a wallet for you and show you a mnemonic backup consisting of 12 English words. These words can be used in any compatible wallet to recover access to your funds should something happen to MetaMask or your computer. You do not need the password for this recovery; the 12 words are sufficient.

Congratulations! You have set up your first Ethereum wallet.

SWITCHING NETWOKS

As you can see on the MetaMask account page, you can choose between multiple Ethereum networks. By default, MetaMask will try to connect to the main network. The other choices are public testnets, any Ethereum node of your choice, or nodes running private blockchains on your own computer (localhost):

Main Ethereum Network

The main public Ethereum blockchain. Real ETH, real value, and real consequences.

Ropsten Test Network

Ethereum public test blockchain and network. ETH on this network has no value.

Kovan Test Network

Ethereum public test blockchain and network using the Aura consensus protocol with proof of authority (federated signing). ETH on this network has no value. The Kovan test network is supported by Parity only. Other Ethereum clients use the Clique consensus protocol, which was proposed later, for proof of authority–based verification.

Rinkeby Test Network

Ethereum public test blockchain and network, using the Clique consensus protocol with proof of authority (federated signing). ETH on this network has no value.

Localhost 8545

Connects to a node running on the same computer as the browser. The node can be part of any public blockchain (main or testnet), or a private testnet.

Custom RPC

Allows you to connect MetaMask to any node with a Geth-compatible Remote Procedure Call (RPC) interface. The node can be part of any public or private blockchain.

ACCOUNTS: EXTERNALLY OWNED ACCOUNTS (EOAs) and CONTRACT ACCOUNTS (CAs)

Accounts are one of the main building blocks of the Ethereum blockchain. They are defined by pairs of private and public keys. Accounts are used by users to interact with the blockchain via transactions. A transaction is digitally signed by an account before submitting it to the network via a node. Ethereum, being a transaction-driven state machine, the state is created or updated as a result of the interaction between accounts and transaction executions. All accounts have a state that, when combined together, represents the state of the Ethereum network. With every new block, the state of the Ethereum network is updated. Operations performed between and on the accounts represent state transitions.

The type of account you created in the MetaMask wallet is called an externally owned account (EOA).

Externally owned accounts are those that have a private key; having the private key means control over access to funds or contracts.

Now, you’re probably guessing there is another type of account. That other type of account is a *contract account*. A contract account has smart contract code, which a simple EOA can’t have. Furthermore, a contract account does not have a private key. Instead, it is owned (and controlled) by the logic of its smart contract code: the software program recorded on the Ethereum blockchain at the contract account’s creation and executed by the EVM.

Contracts have addresses, just like EOAs. Contracts can also send and receive ether, just like EOAs. However, when a transaction destination is a contract address, it causes that contract to *run* in the EVM, using the transaction, and the transaction’s data, as its input. In addition to ether, transactions can contain *data* indicating which specific function in the contract to run and what parameters to pass to that function. In this way, transactions can *call* functions within contracts.

Note that because a contract account does not have a private key, it cannot *initiate* a transaction. Only EOAs can initiate transactions, but contracts can *react* to transactions by calling other contracts, building complex execution paths. One typical use of this is an EOA sending a request transaction to a multisignature smart contract wallet to send some ETH on to another address. A typical DApp programming pattern is to have Contract A calling Contract B in order to maintain a shared state across users of Contract A.

The various properties of each type of account are as follows:

#### EOAs

* They have a state.
* They are associated with a human user, hence are also called user accounts.
* EOAs have an ether balance.
* They are capable of sending transactions.
* They have no associated code.
* They are controlled by private keys.
* EOAs cannot initiate a call message.
* Accounts contain a key-value store.
* EOAs can initiate transaction messages.

#### CAs

* They have a state.
* They are not intrinsically associated with any user or actor on the blockchain.
* CAs have an ether balance.
* They have associated code that is kept in memory/storage on the blockchain. They have access to storage.
* They can get triggered and execute code in response to a transaction or a message from other contracts. It is worth noting that due to the Turing-completeness property of the Ethereum blockchain, the code within CAs can be of any level of complexity. The code is executed by the EVM by each mining node on the Ethereum network. The EVM is discussed later in the chapter in the The Ethereum Virtual Machine (EVM) section.
* Also, CAs can maintain their permanent states and can call other contracts. It is envisaged that in the Serenity (Ethereum 2.0) release, the distinction between EOAs and CAs may be eliminated.
* CAs cannot start transaction messages.
* CAs can initiate a call message.
* CAs contain a key-value store.
* CAs' addresses are generated when they are deployed. This address of the contract is used to identify its location on the blockchain.

Accounts allow interaction with the blockchain via transactions.

ETHETEUM CLIENTS

An Ethereum client is a software application that implements the Ethereum specification and communicates over the peer-to-peer network with other Ethereum clients.

Ethereum is defined by a formal specification called the ‘Yellow Paper’: <https://ethereum.github.io/yellowpaper/paper.pdf>

ETHEREUM NETWORKS

There exist a variety of Ethereum-based networks that largely conform to the formal specification defined in the Ethereum Yellow Paper, but which may or may not interoperate with each other.

Among these Ethereum-based networks are Ethereum, Ethereum Classic, Ella, Expanse, Ubiq, Musicoin, and many others. While mostly compatible at the protocol level, these networks often have features or attributes that require maintainers of Ethereum client software to make small changes in order to support each network. Because of this, not every version of Ethereum client software runs every Ethereum-based blockchain.

Currently, there are six main implementations of the Ethereum protocol, written in six different languages:

* Parity, written in Rust
* Geth, written in Go
* cpp-ethereum, written in C++
* pyethereum, written in Python
* Mantis, written in Scala
* Harmony, written in Java

CRYPTOGRAPHY

Cryptography means “secret writing” in Greek, but the study of cryptography encompasses more than just secret writing, which is referred to as encryption. Cryptography can, for example, also be used to prove knowledge of a secret without revealing that secret (e.g., with a digital signature), or to prove the authenticity of data (e.g., with digital fingerprints, also known as “hashes”).

KEYS AND ADDRESSES

As we saw earlier in the book, Ethereum has two different types of accounts: externally owned accounts (EOAs) and contracts. Ownership of ether by EOAs is established through digital private keys, Ethereum addresses, and digital signatures. The private keys are at the heart of all user interaction with Ethereum. In fact, account addresses are derived directly from private keys: a private key uniquely determines a single Ethereum address, also known as an account.

Ethereum addresses are unique identifiers that are derived from public keys or contracts using the Keccak-256 one-way hash function.

Keys and addresses are used in the Ethereum blockchain to represent ownership and transfer ether. The keys used are made up of pairs of private and public parts. The private key is generated randomly and is kept secret, whereas a public key is derived from the private key. Addresses are derived from public keys and are 20-byte codes used to identify accounts.

BLOCKS AND BLOCKCHAIN

Blocks are the main building structure of a blockchain. Ethereum blocks consist of various elements, which are described as follows:

* The block header
* The transactions list
* The list of headers of ommers or uncles

NB: An uncle block is a block that is the child of a parent but does not have any child block. Ommers or uncles are valid, but stale, blocks that are not part of the main chain but contribute to security of the chain. They also earn a reward for their participation but do not become part of the canonical truth.

The transaction list is simply a list of all transactions included in the block. Also, the list of headers of uncles is also included in the block.

**Block header**: Block headers are the most critical and detailed components of an Ethereum block. The header contains various elements, which are described in detail here:

* **Parent hash**: This is the Keccak 256-bit hash of the parent (previous) block's header.
* **Ommers hash**: This is the Keccak 256-bit hash of the list of ommers (or uncles) blocks included in the block.
* **The beneficiary**: The beneficiary field contains the 160-bit address of the recipient that will receive the mining reward once the block is successfully mined.
* **State root**: The state root field contains the Keccak 256-bit hash of the root node of the state tree. It is calculated once all transactions have been processed and finalized.
* **Transactions root**: The transaction root is the Keccak 256-bit hash of the root node of the transaction tree. The transaction tree represents the list of transactions included in the block.
* **Receipts root**: The receipts root is the Keccak 256-bit hash of the root node of the transaction receipt tree. This tree is composed of receipts of all transactions included in the block. Transaction receipts are generated after each transaction is processed and contain useful post-transaction information. More details on transaction receipts are provided in the next section.
* **Logs bloom**: The logs bloom is a bloom filter that is composed of the logger address and log topics from the log entry of each transaction receipt of the included transaction list in the block. Logging is explained in detail in the next section.
* **Difficulty**: The difficulty level of the current block.
* **Number**: The total number of all previous blocks; the genesis block is block zero.
* **Gas limit**: This field contains the value that represents the limit set on the gas consumption per block.
* **Gas used**: This field contains the total gas consumed by the transactions included in the block.
* **Timestamp**: The timestamp is the epoch Unix time of the time of block initialization.
* **Extra data**: The extra data field can be used to store arbitrary data related to the block. Only up to 32 bytes are allowed in this field.
* **Mixhash**: The mixhash field contains a 256-bit hash that, once combined with the nonce, is used to prove that adequate computational effort (**Proof of Work**, or **PoW**) has been spent in order to create this block.
* **Nonce**: Nonce is a 64-bit hash (a number) that is used to prove, in combination with the mixhash field, that adequate computational effort (PoW) has been spent in order to create this block.

THE GENESIS BLOCK

The genesis block is the first block in a blockchain network. It varies slightly from normal blocks due to the data it contains and the way it has been created.

BLOCK DIFFICULTY MECHANISM

Block difficulty is increased if the time between two blocks decreases, whereas it increases if the block time between two blocks decreases. This is required to maintain a roughly consistent block generation time

WALLETS

An Ethereum wallet is a software application that serves as the primary user interface to Ethereum. The wallet controls access to a user’s money, managing keys and addresses, tracking the balance, and creating and signing transactions. In addition, some Ethereum wallets can also interact with contracts, such as ERC20 tokens.

A common misconception about Ethereum is that Ethereum wallets contain ether or tokens. In fact, very strictly speaking, the wallet holds only keys. The ether or other tokens are recorded on the Ethereum blockchain. Users control the tokens on the network by signing transactions with the keys in their wallets.

NB: Ethereum wallets contain keys, not ether or tokens. Wallets are like keychains containing pairs of private and public keys. Users sign transactions with the private keys, thereby proving they own the ether. The ether is stored on the blockchain.

A wallet is a generic program that can store private keys and, based on the addresses stored within it, it can compute the existing balance of ether associated with the addresses by querying the blockchain. It can also be used to deploy smart contracts.

TYPES OF WALLETS

1. Deterministic Wallets
2. Hierarchical Deterministic Wallets
3. Non deterministic Wallets etc.

TRANSACTIONS

Transactions are signed messages originated by an externally owned account, transmitted by the Ethereum network, and recorded on the Ethereum blockchain. Contracts don’t run on their own. Ethereum doesn’t run autonomously. Everything starts with a transaction.

A transaction in Ethereum is a digitally signed data packet using a private key that contains the instructions that, when completed, either result in a message call or contract creation.

THE STRUCTURE OF A TRANSACTION

1. Nonce: A sequence number, issued by the originating EOA, used to prevent message replay. The nonce is a number that is incremented by one every time a transaction is sent by the sender. It must be equal to the number of transactions sent and is used as a unique identifier for the transaction. A nonce value can only be used once. This is used for replay protection on the network.
2. Gas price: The price of gas (in wei) the originator is willing to pay. The **gas price** field represents the amount of Wei required to execute the transaction. In other words, this is the amount of Wei you are willing to pay for this transaction. This is charged per unit of gas for all computation costs incurred as a result of the execution of this transaction.
3. Gas limit: The maximum amount of gas the originator is willing to buy for this transaction. The **gas limit** field contains the value that represents the maximum amount of gas that can be consumed to execute the transaction.
4. Recipient: The destination Ethereum address
5. Value: The amount of ether to send to the destination. Value represents the total number of Wei to be transferred to the recipient; in the case of a CA, this represents the balance that the contract will hold.
6. Data: The variable-length binary data payload. If the transaction is a message call, then the **Data** field is used instead of **init**, and represents the input data of the message call.
7. To: As the name suggests, the **To** field is a value that represents the address of the recipient of the transaction. This is a 20-byte value.
8. **Signature**: The signature is composed of three fields, namely **V**, **R**, and **S**. These values represent the digital signature (R, S) and some information that can be used to recover the public key (V).
9. **Init**: The **Init** field is used only in transactions that are intended to create contracts, that is, contract creation transactions.

A block is a data structure that contains batches of transactions. Transactions can be found in either transaction pools or blocks. In transaction pools, they wait for verification by a node, and in blocks, they are added after successful verification. When a mining node starts its operation of verifying blocks, it starts with the highest-paying transactions in the transaction pool and executes them one by one. When the gas limit is reached, or no more transactions are left to be processed in the transaction pool, the mining starts.

In this process, the block is repeatedly hashed until a valid nonce is found, such that once hashed with the block, it results in a value less than the difficulty target. Once the block is successfully mined, it will be broadcasted immediately to the network, claiming success, and will be verified and accepted by the network. This process is similar to Bitcoin's mining process.

THE TRANSACTION NONCE

The nonce is one of the most important and least understood components of a transaction.

nonce: A scalar value equal to the number of transactions sent from this address or, in the case of accounts with associated code, the number of contract-creations made by this account.

Strictly speaking, the nonce is an attribute of the originating address; that is, it only has meaning in the context of the sending address. However, the nonce is not stored explicitly as part of an account’s state on the blockchain. Instead, it is calculated dynamically, by counting the number of confirmed transactions that have originated from an address. However, with the nonce value included in the transaction data, every single transaction is unique, even when sending the same amount of ether to the same recipient address multiple times. Thus, by having the incrementing nonce as part of the transaction, it is simply not possible for anyone to “duplicate” a payment you have made. In practical terms, the nonce is an up-to-date count of the number of confirmed (i.e., on-chain) transactions that have originated from an account. This is a value that is incremented every time a transaction is sent from the address. In the case of CAs, it represents the number of contracts created by the account.

TRANSACTION GAS

Gas is the fuel of Ethereum. Gas is not ether — it’s a separate virtual currency with its own exchange rate against ether. Ethereum uses gas to control the amount of resources that a transaction can use, since it will be processed on thousands of computers around the world. The open-ended (Turing-complete) computation model requires some form of metering in order to avoid denial-of-service attacks or inadvertently resource-devouring transactions.

Gas is separate from ether in order to protect the system from the volatility that might arise along with rapid changes in the value of ether, and also as a way to manage the important and sensitive ratios between the costs of the various resources that gas pays for (namely, computation, memory, and storage).

The gasPrice field in a transaction allows the transaction originator to set the price they are willing to pay in exchange for gas. The price is measured in wei per gas unit. Wallets can adjust the gasPrice in transactions they originate to achieve faster confirmation of transactions. The higher the gasPrice, the faster the transaction is likely to be confirmed. Conversely, lower-priority transactions can carry a reduced price, resulting in slower confirmation. The minimum value that gasPrice can be set to is zero, which means a fee-free transaction. During periods of low demand for space in a block, such transactions might very well get mined. Gas is required to be paid for every operation performed on the Ethereum blockchain. This is a mechanism that ensures that infinite loops cannot cause the whole blockchain to stall due to the Turing-complete nature of the EVM. A transaction fee is charged as an amount of Ether and is taken from the account balance of the transaction originator.

A fee is paid for transactions to be included by miners for mining. If this fee is too low, the transaction may never be picked up; the more the fee, the higher are the chances that the transactions will be picked up by the miners for inclusion in the block. Conversely, if the transaction that has an appropriate fee paid is included in the block by miners but has too many complex operations to perform, it can result in an out-of-gas exception if the gas cost is not enough. In this case, the transaction will fail but will still be made part of the block, and the transaction originator will not get any refund.

Transaction costs can be estimated using the following formula:

*Total cost* = *gasUsed* \* *gasPrice*

# TRANSACTION VALUE AND DATA

The main “payload” of a transaction is contained in two fields: value and data. Transactions can have both value and data, only value, only data, or neither value nor data. All four combinations are valid.

A transaction with only value is a payment. A transaction with only data is an invocation. A transaction with both value and data is both a payment and an invocation. A transaction with neither value nor data — well that’s probably just a waste of gas! But it is still possible.

Transactions are the starting point of every activity in the Ethereum system. Transactions are the “inputs” that cause the Ethereum Virtual Machine to evaluate contracts, update balances, and more generally modify the state of the Ethereum blockchain.

NODES AND MINERS

The Ethereum network contains different nodes. Some nodes act only as wallets, some are light clients, and a few are full clients running the full blockchain.

NB: Mining is the process by which new blocks are selected via a consensus mechanism and added to the blockchain.

As a result of the mining operation, currency (ether) is awarded to the nodes that perform mining operations. These mining nodes are known as **miners**. Miners are paid in ether as an incentive for them to validate and verify blocks made up of transactions. The mining process helps secure the network by verifying computations.

At a theoretical level, a miner node performs the following functions:

* It listens for the transactions broadcasted on the Ethereum network and determines the transactions to be processed.
* It determines stale ommer blocks and includes them in the blockchain.
* It updates the account balance with the reward earned from successfully mining the block.
* Finally, a valid state is computed, and the block is finalized, which defines the result of all state transitions.
* The current method of mining is based on PoW, which is similar to that of Bitcoin. When a block is deemed valid, it has to satisfy not only the general consistency requirements, but it must also contain the PoW for a given difficulty.
* The PoW algorithm is due to be replaced by the PoS algorithm with the release of **Serenity**. There is no set date for the release of Serenity, as this will be the final version of Ethereum. Considerable research work has been carried out to build the PoS algorithm, which is suitable for the Ethereum network. An algorithm named **Casper** has been developed that will replace the existing PoW algorithm in Ethereum.

CONSENSUS

Consensus algorithms are the mechanism used to reconcile security and decentralization.

CONSENSUS VIA PROOF OF WORK

The creator of the original blockchain, Bitcoin, invented a *consensus algorithm* called *proof of work* (PoW). Arguably, PoW is the most important invention underpinning Bitcoin. The colloquial term for PoW is “mining,” which creates a misunderstanding about the primary purpose of consensus. Often people assume that the purpose of mining is the creation of new currency, since the purpose of real-world mining is the extraction of precious metals or other resources. Rather, the real purpose of mining (and all other consensus models) is to *secure the blockchain*, while keeping control over the system decentralized and diffused across as many participants as possible. The reward of newly minted currency is an incentive to those who contribute to the security of the system: a means to an end. In that sense, the reward is the means and decentralized security is the end. In PoW consensus there is also a corresponding “punishment,” which is the cost of energy required to participate in mining. If participants do not follow the rules and earn the reward, they risk the funds they have already spent on electricity to mine. Thus, PoW consensus is a careful balance of risk and reward that drives participants to behave honestly out of self-interest.

Ethereum is currently a PoW blockchain, in that it uses a PoW algorithm with the same basic incentive system for the same basic goal: securing the blockchain while decentralizing control. Ethereum’s PoW algorithm is slightly different than Bitcoin’s and is called *Ethash*.

CONSENSUS VIA PROOF OF STAKE

Ethereum is still using proof of work, but the ongoing research toward a proof-of-stake alternative is nearing completion. Ethereum’s planned PoS algorithm is called Casper.

In general, a PoS algorithm works as follows. The blockchain keeps track of a set of validators, and anyone who holds the blockchain’s base cryptocurrency (in Ethereum’s case, ether) can become a validator by sending a special type of transaction that locks up their ether into a deposit. The validators take turns proposing and voting on the next valid block, and the weight of each validator’s vote depends on the size of its deposit (i.e., stake). Importantly, a validator risks losing their deposit if the block they staked it on is rejected by the majority of validators. Conversely, validators earn a small reward, proportional to their deposited stake, for every block that is accepted by the majority. Thus, PoS forces validators to act honestly and follow the consensus rules, by a system of reward and punishment. The major difference between PoS and PoW is that the punishment in PoS is intrinsic to the blockchain (e.g., loss of staked ether), whereas in PoW the punishment is extrinsic (e.g., loss of funds spent on electricity).

SMART CONTRACTS AND SOLIDITY

As we discussed in earlier, there are two different types of accounts in Ethereum: externally owned accounts (EOAs) and contract accounts. EOAs are controlled by users, often via software such as a wallet application that is external to the Ethereum platform. In contrast, contract accounts are controlled by program code (also commonly referred to as “smart contracts”) that is executed by the Ethereum Virtual Machine. In short, EOAs are simple accounts without any associated code or data storage, whereas contract accounts have both associated code and data storage. EOAs are controlled by transactions created and cryptographically signed with a private key in the “real world” external to and independent of the protocol, whereas contract accounts do not have private keys and so “control themselves” in the predetermined way prescribed by their smart contract code. Both types of accounts are identified by an Ethereum address.

WHAT IS A SMART CONTRACT?

DEFINITION BY NICK SZABO: "A smart contract is an electronic transaction protocol that executes the terms of a contract. The general objectives are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries. Related economic goals include lowering fraud loss, arbitrations and enforcement costs, and other transaction costs."

A smart contract is a secure and unstoppable computer program (code snippet) representing an agreement that is automatically executable and enforceable.

Dissecting this definition reveals that a smart contract is, fundamentally, a computer program that is written in a language that a computer or target machine can understand.

In this book, we use the term “smart contracts” to refer to immutable computer programs that run deterministically in the context of an Ethereum Virtual Machine as part of the Ethereum network protocol — i.e., on the decentralized Ethereum world computer.

Explanation of terms:

Computer programs

Smart contracts are simply computer programs. The word “contract” has no legal meaning in this context.

Immutable

Once deployed, the code of a smart contract cannot change. Unlike with traditional software, the only way to modify a smart contract is to deploy a new instance.

Deterministic

The outcome of the execution of a smart contract is the same for everyone who runs it, given the context of the transaction that initiated its execution and the state of the Ethereum blockchain at the moment of execution.

EVM context

Smart contracts operate with a very limited execution context. They can access their own state, the context of the transaction that called them, and some information about the most recent blocks.

Decentralized world computer

The EVM runs as a local instance on every Ethereum node, but because all instances of the EVM operate on the same initial state and produce the same final state, the system as a whole operates as a single “world computer.”

In summary, a smart contract has the following properties:

* **Automatically executable**: It is self-executable on a blockchain without requiring any intervention.
* **Enforceable**: This means that all contract conditions are enforced automatically.
* **Secure**: This means that smart contracts are tamper-proof (or tamper-resistant) and run with security guarantees. The underlying blockchain usually provides these security guarantees; however, the smart contract programing language and the smart contract code themselves must be correct, valid, and verified.
* **Deterministic**: The deterministic feature ensures that smart contracts always produce the same output for a specific input. Even though it can be considered to be part of the secure property, defining it here separately ensures that the deterministic property is considered one of the important properties.
* **Semantically sound**: This means that they are complete and meaningful to both people and computers.
* **Unstoppable**: This means that adversaries or unfavorable conditions cannot negatively affect the execution of a smart contract. When the smart contracts execute, they complete their performance deterministically in a finite amount of time.

LIFE CYCLE OF A SMART CONTRACT

Smart contracts are typically written in a high-level language, such as Solidity. But in order to run, they must be compiled to the low-level bytecode that runs in the EVM. Once compiled, they are deployed on the Ethereum platform using a special contract creation transaction.

Each contract is identified by an Ethereum address, which is derived from the contract creation transaction as a function of the originating account and nonce. The Ethereum address of a contract can be used in a transaction as the recipient, sending funds to the contract or calling one of the contract’s functions. Note that, unlike with EOAs, there are no keys associated with an account created for a new smart contract. As the contract creator, you don’t get any special privileges at the protocol level (although you can explicitly code them into the smart contract). You certainly don’t receive the private key for the contract account, which in fact does not exist — we can say that smart contract accounts own themselves.

Importantly, contracts *only run if they are called by a transaction*. All smart contracts in Ethereum are executed, ultimately, because of a transaction initiated from an EOA. A contract can call another contract that can call another contract, and so on, but the first contract in such a chain of execution will always have been called by a transaction from an EOA. Contracts never run “on their own” or “in the background.” Contracts effectively lie dormant until a transaction triggers execution, either directly or indirectly as part of a chain of contract calls. It is also worth noting that smart contracts are not executed “in parallel” in any sense — the Ethereum world computer can be considered to be a single-threaded machine.

ORACLES

Oracles are an essential component of the smart contract and blockchain ecosystem. The limitation with smart contracts is that they cannot access external data because blockchains are closed systems without any direct access to the real world. This external data might be required to control the execution of some business logic in the smart contract; for example, the stock price of a security product that is required by the contract to release dividend payments. In such situations, oracles can be used to provide external data to smart contracts.

An oracle can be defined as an interface that delivers data from an external source to smart contracts. Oracles are trusted entities that use a secure channel to transfer off-chain data to a smart contract.

The standard mechanics of how oracles work is presented here:

1. A smart contract sends a request for data to an oracle.
2. The request is executed and the required data is requested from the source. There are various methods of requesting data from the source. These methods usually involve invoking APIs provided by the data provider, calling a web service, reading from a database (for example, in enterprise integration use cases where the required data may exist on a local enterprise legacy system), or requesting data from another blockchain. Sources can be any external off-chain data provider on the internet or in an internal enterprise network.
3. The data is sent to a notary to generate cryptographic proof (usually a digital signature) of the requested data to prove its validity (authenticity). Usually, TLSNotary is used for this purpose (<https://tlsnotary.org>). Other techniques include **Android proofs**, **Ledger proofs**, and **trusted hardware-assisted proofs**, which we will explain shortly.
4. The data with the proof of validity is sent to the oracle.
5. The requested data with its proof of authenticity can be optionally saved on a decentralized storage system such as Swarm or IPFS and can be used by the smart contract/blockchain for verification. This is especially useful when the proofs of authenticity are of a large size and sending them to the requesting smart contracts (storing them on the chain) is not feasible.
6. Finally, the data, with the proof of validity, is sent to the smart contract.

Smart contracts can either pull data from oracles, or oracles can push data to smart contracts.

TYPES OF BLOCKCHAIN ORACLES

### Inbound oracles

This class represents oracles that receive incoming data from external services, and feed it into the smart contract. We will shortly discuss software, hardware, and several other types of inbound oracle.

#### Software oracles

These oracles are responsible for acquiring information from online services on the Internet. This type of oracle is usually used to source data such as weather information, financial data (stock prices, for example), travel information and other types of data from third-party providers. The data source can also be an internal enterprise system, which may provide some enterprise-specific data. These types of oracle can also be called standard or simple oracles.

#### Hardware oracles

This type of oracle is used to source data from hardware sources such as IoT devices or sensors. This is useful in use cases such as insurance-related smart contracts where telemetry sensors provide certain information, for example, vehicle speed and location. This information can be fed into the smart contract dealing with insurance claims and payouts to decide whether to accept a claim or not. Based on the information received from the source hardware sensors, the smart contract can decide whether to accept or reject the claim.

These oracles are useful in any situation where real-world data from physical devices is required. However, this approach requires a mechanism in which hardware devices are tamper-proof or tamper-resistant. This level of security can be achieved by providing cryptographic evidence (non-repudiation and integrity) of IoT device's data and an anti-tampering mechanism on the IoT device, which renders the device useless in case of tampering attempts.

#### Computation oracles

These oracles allow computing-intensive calculations to be performed off-chain. As blockchain is not suitable for performing compute-intensive operations, a blockchain (that is, a smart contract on a blockchain) can request computations to be performed on off-chain high-performance computing infrastructure and get the verified results back via an oracle. The use of oracle, in this case, provides data integrity and authenticity guarantees.

An example of such an oracle is Truebit (<https://truebit.io>). It allows a smart contract to submit computation tasks to oracles, which are eventually completed by miners in return for an incentive.

#### Aggregation based oracles

In this scenario, a single value is sourced from many different feeds. As an example, this single value can be the price of a financial instrument, and it can be risky to rely upon only one feed. To mitigate this problem, multiple data providers can be used where all of these feeds are inspected, and finally, the price value that is reported by most of the feeds can be picked up. The assumption here is that if the majority of the sources reports the same price value, then it is likely to be correct. The collation mechanism depends on the use case: sometimes it's merely an average of multiple values, sometimes a median is taken of all the values, and sometimes it is the maximum value. Regardless of the aggregation mechanism, the essential requirement here is to get the value that is valid and authentic, which eventually feeds into the system.

An excellent example of price feed oracles is MakerDAO (<https://makerdao.com/en/>) price feed oracle (<https://developer.makerdao.com/feeds/>), which collates price data from multiple external price feed sources and provides a median ETHUSD price to MakerDAO.

#### Crowd wisdom driven oracles

This is another way that the blockchain oracle problem can be addressed where a single source is not trusted. Instead, multiple public sources are used to deduce the most appropriate data eventually. In other words, it solves the problem where a single source of data may not be trustworthy or accurate as expected. If there is only one source of data, it can be unreliable and risky to rely on entirely. It may turn malicious or become genuinely faulty.

In this case, to ensure the credibility of data provided by third-party sources for oracles, the data is sourced from multiple sources. These sources can be users of the system or even members of the general public who have access to and have knowledge of some data, for example, a political event or a sporting event where members of the public know the results and can provide the required data. Similarly, this data can be sourced from multiple different news websites. This data can then be aggregated, and if a sufficiently high number of the same information is received from multiple sources, then there is an increased likelihood that the data is correct and can be trusted.

#### Decentralized oracles

Another type of oracles, which primarily emerged due to the decentralization requirements, is called **decentralized** oracles. Remember that in all types of oracles discussed so far, there are some trust requirements to be placed in a trusted third party. As blockchain platforms such as Bitcoin and Ethereum are fully decentralized, it is expected that oracle services should also be decentralized. This way, we can address the Blockchain Oracle Problem.

This type of oracle can be built based on a distributed mechanism. It can also be envisaged that the oracles can find themselves source data from another blockchain, which is driven by distributed consensus, thus ensuring the authenticity of data. For example, one institution running their private blockchain can publish their data feed via an oracle that can then be consumed by other blockchains.

A decentralized oracle essentially allows off-chain information to be transferred to a blockchain without relying on a trusted third party.

Augur (visit <https://www.augur.net/whitepaper.pdf> for Jack Peterson et al.'s essay,A Decentralized Oracle and Prediction Market Platform) is a prime example of such type of oracles. The Augur white paper is also available here: <https://arxiv.org/abs/1501.01042>.

The core idea behind Augur's oracle is that of crowd wisdom-supported oracles, in which the information about an event is acquired from multiple sources and aggregated into the most likely outcome. The sources in case of Augur are financially motivated reporters who are rewarded for correct reporting and penalized for incorrect reporting.

#### Smart oracles

An idea of smart oracle has also been proposed by **Ripple labs** (**codius**). Its original whitepaper is available at <https://github.com/codius/codius-wiki/wiki/White-Paper#from-oracles-to-smart-oracles>. Smart oracles are entities just like oracles, but with the added capability of executing contract code. Smart oracles proposed by Codius run using Google Native Client, which is a sandboxed environment for running untrusted x86 native code.

### Outbound oracles

This type, also called **reverse oracles**, are used to send data out from the blockchain smart contracts to the outside world. There are two possible scenarios here; one is where the source blockchain is a producer of some data such as blockchain metrics, which are needed for some other blockchain. The actual data somehow needs to be sent out to another blockchain smart contract. The other scenario is that an external hardware device needs to perform some physical activity in response to a transaction on-chain. However, note that this type of scenario does not necessarily need an oracle, because the external hardware device can be sent a signal as a result of the smart contract event.

On the other hand, it can be argued that if the hardware device is running on an external blockchain, then to get data from the source chain to the target chain, undoubtedly, will need some security guarantees that oracle infrastructure can provide. Another situation is where we need to integrate legacy enterprise systems with the blockchain. In that case, the outbound oracle would be able to provide blockchain data to the existing legacy systems. An example scenario is the settlement of a trade done on a blockchain that needs to be reported to the legacy settlement and backend reporting systems.

THE DAO

The **Decentralized Autonomous Organization** (**DAO**), started in April 2016, was a smart contract written to provide a platform for investment. Due to a bug, called the **reentrancy bug**, in the code, it was hacked in June 2016. An equivalent of approximately 3.6 million ether (roughly 50 million US dollars) was siphoned out of the DAO into another account.

Even though the term hacked is used here, it was not really hacked. The smart contract did what it was asked to do but due to the vulnerabilities in the smart contracts, the attacker was able to exploit it. It can be seen as an unintentional behavior (a bug) that programmers of the DAO did not foresee. This incident resulted in a hard fork on the Ethereum blockchain, which was introduced to recover from the attack.

The DAO attack exploited a vulnerability (reentrancy bug) in the DAO code where it was possible to withdraw tokens from the DAO smart contract repeatedly before giving the DAO contract a chance to update its internal state to indicate that how many DAO tokens have been withdrawn. The attacker was able to withdraw DAOs. However, before the smart contract could update its state, the attacker withdrew the tokens again. This process was repeated many times, but eventually, only a single withdrawal was logged by the smart contract, and the contract also lost record of any repeated withdrawals.

PROGRAMMING LANGUAGES FOR BUILDING SMART CONTRACTS IN ETHEREUM

1. LLL

A functional (declarative) programming language, with Lisp-like syntax. It was the first high-level language for Ethereum smart contracts but is rarely used today.

1. Serpent

A procedural (imperative) programming language with a syntax similar to Python. Can also be used to write functional (declarative) code, though it is not entirely free of side effects.

1. Solidity

A procedural (imperative) programming language with a syntax similar to JavaScript, C++, or Java. The most popular and frequently used language for Ethereum smart contracts.

1. Vyper

A more recently developed language, similar to Serpent and again with Python-like syntax. Intended to get closer to a pure-functional Python-like language than Serpent, but not to replace Serpent.

1. Bamboo

A newly developed language, influenced by Erlang, with explicit state transitions and without iterative flows (loops). Intended to reduce side effects and increase auditability. Very new and yet to be widely adopted.

As you can see, there are many languages to choose from. However, of all of these Solidity is by far the most popular, to the point of being the *de facto* high-level language of Ethereum and even other EVM-like blockchains.

NB: Because it’s not a programming course we won’t learn the Solidity programming language into details.

GAS IN DETAILS

Gas is a resource constraining the maximum amount of computation that Ethereum will allow a transaction to consume. If the gas limit is exceeded during computation, the following series of events occurs:

* An “out of gas” exception is thrown.
* The state of the contract prior to execution is restored (reverted).
* All ether used to pay for the gas is taken as a transaction fee; it is not refunded.

Because gas is paid by the user who initiates the transaction, users are discouraged from calling functions that have a high gas cost. It is thus in the programmer’s best interest to minimize the gas cost of a contract’s functions. To this end, there are certain practices that are recommended when constructing smart contracts, so as to minimize the gas cost of a function call.

TOKENS

It is commonly used to refer to privately issued special-purpose coin-like items of insignificant intrinsic value, such as transportation tokens, laundry tokens, and arcade game tokens.

HOW TOKENS ARE USED

1. Currency

A token can serve as a form of currency, with a value determined through private trade.

1. Resource

A token can represent a resource earned or produced in a sharing economy or resource-sharing environment; for example, a storage or CPU token representing resources that can be shared over a network.

1. Asset

A token can represent ownership of an intrinsic or extrinsic, tangible or intangible asset; for example, gold, real estate, a car, oil, energy, MMOG items, etc.

1. Access

A token can represent access rights and grant access to a digital or physical property, such as a discussion forum, an exclusive website, a hotel room, or a rental car.

1. Equity

A token can represent shareholder equity in a digital organization (e.g., a DAO) or legal entity (e.g., a corporation).

1. Voting

A token can represent voting rights in a digital or legal system.

1. Collectible

A token can represent a digital collectible (e.g., CryptoPunks) or physical collectible (e.g., a painting).

Identity

A token can represent a digital identity (e.g., avatar) or legal identity (e.g., national ID).

1. Attestation

A token can represent a certification or attestation of fact by some authority or by a decentralized reputation system (e.g., marriage record, birth certificate, college degree).

1. Utility

A token can be used to access or pay for a service.

TOKENS AND FUNGIBILITY

Tokens are fungible when we can substitute any single unit of the token for another without any difference in its value or function.

Non-fungible tokens are tokens that each represent a unique tangible or intangible item and therefore are not interchangeable.

DECENTRALIZED APPLICATIONS (DApps)

A DApp is an application that is mostly or entirely decentralized.

THE ETHERUM VIRTUAL MACHINE

The EVM is the part of Ethereum that handles smart contract deployment and execution. Simple value transfer transactions from one EOA to another don’t need to involve it, practically speaking, but everything else will involve a state update computed by the EVM. At a high level, the EVM running on the Ethereum blockchain can be thought of as a global decentralized computer containing millions of executable objects, each with its own permanent data store.

The EVM is a simple stack-based execution machine that runs bytecode instructions to transform the system state from one state to another. The EVM is a Turing-complete machine but is limited by the amount of gas that is required to run any instruction. This means that infinite loops that can result in denial-of-service attacks are not possible due to gas requirements. The EVM also supports exception handling should exceptions occur, such as not having enough gas or providing invalid instructions, in which case the machine would immediately halt and return the error to the executing agent.

ETHASH: ETHEREUM’S PROOF OF WORK ALGORITHM

Ethash is the Ethereum PoW algorithm. Ethash is the name of the PoW algorithm used in Ethereum.

Similar to Bitcoin, the core idea behind mining is to find a nonce (a random arbitrary number), which, once concatenated with the block header and hashed, results in a number that is lower than the current network difficulty level. The current reward scheme is 2 ETH for successfully finding a valid nonce. In addition to receiving 2 ether, the successful miner also receives the cost of the gas consumed within the block and an additional reward for including stale blocks (uncles) in the block.

Mining can be performed by simply joining the Ethereum network and running an appropriate client. The key requirement is that the node should be fully synched with the main network before mining can start.

CASPER: ETHEREUM’S PROOF OF STAKE ALGORITHM

Casper is the proposed name for Ethereum’s PoS consensus algorithm. It is still under active research and development and is not implemented on the Ethereum blockchain at the time of publication of this book.

ETHER CRYPTOCURRENCY/TOKENS (ETC and ETH)

As an incentive to the miners, Ethereum rewards its own native currency called **ether** (abbreviated as **ETH**). After the **Decentralized Autonomous Organization** (**DAO**) hack described in Chapter 10, Smart Contracts, a hard fork was proposed in order to mitigate the issue; therefore, there are now two Ethereum blockchains: one is called Ethereum Classic, and its currency is represented by ETC, whereas the hard-forked version is ETH, which continues to grow and on which active development is being carried out. Ether is minted by miners as a currency reward for the computational effort they spend to secure the network by verifying transactions and blocks. Ether is used within the Ethereum blockchain to pay for the execution of contracts on the EVM. Ether is used to purchase gas as crypto fuel, which is required to perform computation on the Ethereum blockchain.