Ethereum 101 1

Ethereum is “A Next-Generation Smart Contract and Decentralized Application Platform” (See here)

Ethereum 101 2

Ethereum is a blockchain with a built-in Turing-complete programming language, allowing anyone to write smart contracts and decentralized applications where they can create their own arbitrary rules for ownership, transaction formats and state transition functions. (See here)

Ethereum 101 3

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. It is often described as "the world computer.” (See here)

Ethereum 101 4

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. (See here)

Ethereum 101 5

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. (See here)

Ethereum 101 6

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. (See here)

Ethereum 101 7

The original blockchain, namely Bitcoin's blockchain, tracks the state of units of bitcoin and their ownership. You can think of Bitcoin as a distributed consensus state machine, where transactions cause a global state transition, altering the ownership of coins. The state transitions are constrained by the rules of consensus, allowing all participants to (eventually) converge on a common (consensus) state of the system, after several blocks are mined. Ethereum is also a distributed state machine. But instead of tracking only the state of currency ownership, Ethereum tracks the state transitions of a general-purpose data store, i.e., a store that can hold any data expressible as a key–value tuple. (See here)

Ethereum 101 8 (1/2)

Ethereum's core components (See here**):**

a. P2P network: Ethereum runs on the Ethereum main network, which is addressable on TCP port 30303, and runs a protocol called ĐΞVp2p.

b. Transactions: Ethereum transactions are network messages that include (among other things) a sender, recipient, value, and data payload.

Ethereum 101 8 (2/2)

c. State machine: Ethereum state transitions are processed by the Ethereum Virtual Machine (EVM), a stack-based virtual machine that executes bytecode (machine-language instructions). EVM programs, called "smart contracts," are written in high-level languages (e.g., Solidity or Vyper) and compiled to bytecode for execution on the EVM.

d. Data structures: Ethereum's state is stored locally on each node as a database (usually Google's LevelDB), which contains the transactions and system state in a serialized hashed data structure called a Merkle Patricia Tree.

Ethereum 101 9 (1/2)

Ethereum's core components (continued):

a. Consensus algorithm: Ethereum uses Bitcoin's consensus model, Nakamoto Consensus, which uses sequential single-signature blocks, weighted in importance by Proof-of-Work (PoW) to determine the longest chain and therefore the current state.

b. However, this is being transitioned to a Proof-of-Stake (PoS) algorithm in Ethereum 2.0.

Ethereum 101 9 (2/2)

c. Economic security: Ethereum currently uses a PoW algorithm called Ethash, but this is being transitioned to a PoS algorithm in Ethereum 2.0.

d. Clients: Ethereum has several interoperable implementations of the client software, the most prominent of which are Go-Ethereum (Geth) and OpenEthereum. The others are Erigon, Nethermind and Turbo-geth. OpenEthereum is being deprecated to transition to Erigon, which is the former Turbo-geth. (See here)

Ethereum 101 10

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. Turing-complete systems face the challenge of the halting problem i.e. given an arbitrary program and its input, it is not solvable to determine whether the program will eventually stop running. So Ethereum cannot predict if a smart contract will terminate, or how long it will run. Therefore, to constrain the resources used by a smart contract, Ethereum introduces a metering mechanism called gas. (See here)

As the EVM executes a smart contract, it carefully accounts for every instruction (computation, data access, etc.). Each instruction has a predetermined cost in units of gas. When a transaction triggers the execution of a smart contract, it must include an amount of gas that sets the upper limit of what can be consumed running the smart contract. The EVM will terminate execution if the amount of gas consumed by computation exceeds the gas available in the transaction. Gas is the mechanism Ethereum uses to allow Turing-complete computation while limiting the resources that any program can consume. (See here)

Ether needs to be sent along with a transaction and it needs to be explicitly earmarked for the purchase of gas, along with an acceptable gas price. Just like at the pump, the price of gas is not fixed. Gas is purchased for the transaction, the computation is executed, and any unused gas is refunded back to the sender of the transaction. (See here)

A Decentralized Application, abbreviated as ĐApp, is a web application that is built on top of open, decentralized, peer-to-peer infrastructure services and typically combines smart contracts with a web interface.

ĐApps represent a transition from “Web 2.0” where applications are centrally owned and managed to “Web 3.0” where applications are built on decentralised peer-to-peer protocols for compute (i.e. blockchain), storage and messaging.

Ethereum blockchain represents the decentralized compute part of Web 3.0. Swarm represents the decentralized storage and Whisper (now Waku) represents the decentralized messaging protocol.

Decentralization can be considered as three types (See here**):**

a. Architectural decentralization

b. Political decentralization

c. Logical decentralization

Ethereum's currency unit is called ether or "ETH." Ether is subdivided into smaller units and the smallest unit is named wei. 10**3 wei is 1 Babbage, 10**6 wei is 1 Lovelace, 10**9 wei is 1 Shannon and 10**18 wei is 1 Ether.

Ethereum uses public key cryptography to create public–private key pairs (considered a "pair" because the public key is derived from the private key) which are not used for encryption but for digital signatures.

Ethereum uses Elliptic Curve Digital Signature Algorithm (ECDSA) for digital signatures (SECP-256k1 curve) which is based on Elliptic-curve cryptography (ECC), an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields. (See here)

An Ethereum private key is a 256-bit random number that uniquely determines a single Ethereum address also known as an account

An Ethereum public key is a point on an elliptic curve calculated from the private key using elliptic curve multiplication. One cannot calculate the private key from the public key.

Ethereum state is made up of objects called "accounts", with each account having a 20-byte address and state transitions being direct transfers of value and information between accounts. (See here)

Ethereum account contains four fields:

a. The nonce, a counter used to make sure each transaction can only be processed once

b. The account's current ether balance

c. The account's contract code, if present

d. The account's storage (empty by default)

Ethereum has two different types of accounts:

a. Externally Owned Accounts (EOAs) controlled by private keys

b. Contract Accounts controlled by their contract code

Ownership of ether by EOAs is established through private keys, Ethereum addresses, and digital signatures. Anyone with a private key has control of the corresponding EOA account and any ether it holds.

An EOA has no code, and one can send messages from an EOA by creating and signing a transaction

A contract account has code and associated storage and every time it receives a message its code activates, allowing it to read and write to internal storage and send other messages or create contracts in turn.

Smart contracts can be thought of as "autonomous agents" that live inside of the Ethereum execution environment, always executing a specific piece of code when "poked" by a message or transaction, and having direct control over their own ether balance and their own key/value store to keep track of persistent variables.

Ethereum uses Keccak-256 as its cryptographic hash function. Keccak-256 was the winning candidate for the SHA-3 competition held by NIST but is different from the finally adopted SHA-3 standard. (See here)

Ethereum address of an EOA account is the last 20 bytes (least significant bytes) of the Keccak-256 hash of the public key of the EOA’s key pair.

Transactions are signed messages originated by an externally owned account (EOA), transmitted by the Ethereum network, and recorded on the Ethereum blockchain. Only transactions can trigger a change of state. Ethereum is a transaction-based state machine. (See here)

Transaction properties:

a. Atomic: it is all or nothing i.e. cannot be divided or interrupted by other transactions

b. Serial: Transactions are processed sequentially one after the other without any overlapping by other transactions

c. Inclusion: Transaction inclusion is not guaranteed and depends on network congestion and gasPrice among other things. Miners determine inclusion.

d. Order: Transaction order is not guaranteed and depends on network congestion and gasPrice among other things. Miners determine order.

A transaction is a serialized binary message that contains the following components (See here**):**

a. nonce: A sequence number, issued by the originating EOA, used to prevent message replay

b. gasPrice: The amount of ether (in wei) that the originator is willing to pay for each unit of gas

c. gasLimit: The maximum amount of gas the originator is willing to pay for this transaction

d. recipient: The destination Ethereum address

e. value: The amount of ether (in wei) to send to the destination

f. data: The variable-length binary data payload

g. v,r,s: The three components of an ECDSA digital signature of the originating EOA

Nonce: A scalar value equal to the number of transactions sent from the EOA account or, in the case of Contract accounts, it is the number of contract-creations made by the account. (See here)

Gas price: The price a transaction originator is willing to pay in exchange for gas. The price is measured in wei per gas unit. The higher the gas price, the faster the transaction is likely to be confirmed on the blockchain. The suggested gas price depends on the demand for block space at the time of the transaction.

Gas limit: The maximum number of gas units the transaction originator is willing to pay in order to complete the transaction

Recipient: The 20-byte Ethereum address of the transaction's recipient which can be an EOA or a Contract account.

a. The Ethereum protocol does not validate recipient addresses in transactions. One can send a transaction to an address that has no corresponding private key or contract. Validation should be done at the user interface level.

b. Note that there is no *from* address in the transaction because the EOA's public key can be derived from the v,r,s components of the ECDSA signature and the transaction originator's address can be derived from this public key

Value: The value of ether sent to the transaction recipient. If the recipient is an EOA then that account's balance will be increased by this value. If the recipient is a contract address then the result depends on any data that is sent as part of this transaction. If there is no data, the recipient contract's *receive* or *fallback* function is called if they are present. Depending on the implementation of those functions, the ether value is added to the contract account's balance or an exception occurs and this ether remains with the originator's account.

Data: The information (typically) sent to a contract account indicating the contract's function to be called and the arguments to that function.

v,r,s: *r* and *s* are the two parts of the ECDSA signature produced by the transaction originator using the private key. *v* is the recovery identifier which is calculated as either one of 27 or 28, or as the chain ID (Ethereum mainnet chainID is 1) doubled plus 35 or 36. (See here)

A digital signature serves three purposes in Ethereum: 1) proves that the owner of the private key, who is by implication the owner of an Ethereum account, has authorized the spending of ether, or execution of a contract 2) guarantees non-repudiation: the proof of authorization is undeniable 3) proves that the transaction data has not been and cannot be modified by anyone after the transaction has been signed.

Contract creation transactions are sent to a special destination address called the zero address i.e. 0x0. A contract creation transaction contains a data payload with the compiled bytecode to create the contract. An optional ether amount in the value field will create the new contract with a starting balance.

Transactions vs Messages:

a. A transaction is produced by an EOA where an external actor sends a signed data package which either: 1) triggers a message to another EOA where it leads to a transfer of value or 2) triggers a message to a contract account where it leads to the recipient contract account running its code

b. A message is either: 1) triggered by a transaction to another EOA or contract account or 2) triggered internally within the EVM by a contract account when it executes the CALL family of opcodes and leads to the recipient contract account running its code or value transfer to the recipient EOA

Transactions are grouped together into blocks. A blockchain contains a series of such blocks that are chained together.

Blocks: are batches of transactions with a hash of the previous block in the chain. This links blocks together (in a chain) because hashes are cryptographically derived from the block data. This prevents fraud, because one change in any block in history would invalidate all the following blocks as all subsequent hashes would change and everyone running the blockchain would notice. To preserve the transaction history, blocks are strictly ordered (every new block created contains a reference to its parent block), and transactions within blocks are strictly ordered as well. (See here)

Ethereum node/client: A node is a software application that implements the Ethereum specification and communicates over the peer-to-peer network with other Ethereum nodes. A client is a specific implementation of Ethereum node. The two most common client implementations are Geth and OpenEthereum. Ethereum transactions are sent to Ethereum nodes to be broadcast across the peer-to-peer network. (See here)

Miners: are entities running Ethereum nodes that validate and execute these transactions and combine them into blocks. The process of validating each block by having a miner provide a mathematical proof is known as a "proof of work." Miners are rewarded for blocks accepted into the blockchain with a block reward in ether (currently 2 ETH). A miner also gets fees which is the ether spent on gas by all the transactions included in the block.

Block gas limit is set by miners and refers to the cap on the total amount of gas expended by all transactions in the block, which ensures that blocks can't be arbitrarily large. Blocks therefore are not a fixed size in terms of the number of transactions because different transactions consume different amounts of gas. See here for historical block gas limits.

Blocks take time to propagate through the network and multiple miners are simultaneously producing valid blocks. This leads to the blockchain considering multiple blocks at the same level but ultimately choosing only one block at any level that creates the canonical blockchain. This choice is dictated by Ethereum's Greedy Heaviest Observed Subtree (GHOST) protocol which includes stale blocks up to seven levels in the calculation of the longest chain. Stale blocks are called uncles or ommers.

Consensus: Decentralized consensus in the context of Ethereum refers to the process of determining which miner's block should be appended next to the blockchain. This involves two key components of Proof-of-Work (PoW) and the Longest-chain Rule. Miners apply these rules to build on the canonical blockchain. This is referred to as "Nakamoto Consensus" and is adapted from Bitcoin.

State is a mapping between addresses and account states implemented as a modified Merkle Patricia tree or trie. A Merkle tree or trie is a type of binary tree composed of a set of nodes with:

a. Leaf nodes at the bottom of the tree that contain the underlying data

b. Intermediate nodes, where each node is the hash of its two child nodes

c. A single root node formed from the hash of its two child nodes representing the top of the tree

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

a. The algorithm is formally defined as $m = Hm \wedge n <= 2**256/Hd$ with $(m, n) = PoW(Hn', Hn, d)$ where Hn' is the new block's header but without the nonce and mix-hash components; Hn is the nonce of the header; d is a large data set needed to compute the mixHash and Hd is the new block's difficulty value

b. PoW is the proof-of-work function which evaluates to an array with the first item being the mixHash and the second item being a pseudorandom number cryptographically dependent on H and d.

Blocks contain block header, transactions and ommers' block headers. Block header contains (See here**):**

a. *parentHash*: The Keccak 256-bit hash of the parent block's header, in its entirety

b. *ommersHash*: The Keccak 256-bit hash of the ommers list portion of this block

c. *beneficiary*: The 160-bit address to which all fees collected from the successful mining of this block be transferred

d. *stateRoot*: The Keccak 256-bit hash of the root node of the state trie, after all transactions are executed and finalisations applied

e. *transactionsRoot*: The Keccak 256-bit hash of the root node of the trie structure populated with each transaction in the transactions list portion of the block

f. *receiptsRoot*: The Keccak 256-bit hash of the root node of the trie structure populated with the receipts of each transaction in the transactions list portion of the block

g. *logsBloom*: The Bloom filter composed from indexable information (logger address and log topics) contained in each log entry from the receipt of each transaction in the transactions list

h. *number*: A scalar value equal to the number of ancestor blocks. The genesis block has a number of zero;

i. *difficulty*: A scalar value corresponding to the difficulty level of this block. This can be calculated from the previous block's difficulty level and the timestamp

j. gasLimit: A scalar value equal to the current limit of gas expenditure per block

k. *gasUsed*: A scalar value equal to the total gas used in transactions in this block

l. *timestamp*: A scalar value equal to the reasonable output of Unix's time() at this block's inception

m. *extraData*: An arbitrary byte array containing data relevant to this block. This must be 32 bytes or fewer

n. *mixHash*: A 256-bit hash which, combined with the nonce, proves that a sufficient amount of computation has been carried out on this block

o. *nonce*: A 64-bit value which, combined with the mixhash, proves that a sufficient amount of computation has been carried out on this block

stateRoot, *transactionsRoot* and *receiptsRoot* are 256-bit hashes of the root nodes of modified Merkle-Patricia trees. The leaves of stateRoot are key-value pairs of all Ethereum address-account pairs, where each respective account consists of:

a. 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

b. balance: A scalar value equal to the number of Wei owned by this address

c. storageRoot: A 256-bit hash of the root node of a modified Merkle-Patricia tree that encodes the storage contents of the account (a mapping between 256-bit integer values), encoded into the trie as a mapping from the Keccak 256-bit hash of the 256-bit integer keys to the RLP-encoded 256-bit integer values.

d. codeHash: The hash of the EVM code of this account—this is the code that gets executed should this address receive a message call; it is immutable and thus, unlike all other fields, cannot be changed after construction.

Transaction receipt is a tuple of four items comprising:

a. The cumulative gas used in the block containing the transaction receipt as of immediately after the transaction has happened

b. The set of logs created through execution of the transaction

c. The Bloom filter composed from information in those logs

d. The status code of the transaction

Gas refund and beneficiary: Any unused gas in a transaction (gasLimit minus gas used by the transaction) is refunded to the sender's account at the same gasPrice. Ether used to purchase gas used for the transaction is credited to the beneficiary address (specified in the block header), the address of an account typically under the control of the miner. This is the transaction "fees" paid to the miner.

EVM is a quasi Turing complete machine where the quasi qualification comes from the fact that the computation is intrinsically bounded through a parameter, gas, which limits the total amount of computation done. EVM is the runtime environment for smart contracts.

The code in Ethereum contracts is written in a low-level, stack-based bytecode language, referred to as "Ethereum virtual machine code" or "EVM code". The code consists of a series of bytes (hence called bytecode), where each byte represents an operation.

The EVM is a simple stack-based architecture consisting of the stack, volatile memory, non-volatile storage with a word size of 256-bit (chosen to facilitate the Keccak256 hash scheme and elliptic-curve computations) and Calldata.

Stack is made up of 1024 256-bit elements. EVM instructions can operate with the top 16 stack elements. Most EVM instructions operate with the stack (stack-based architecture) and there are also stack-specific operations e.g. PUSH, POP, SWAP, DUP etc.

Memory is a linear byte-array addressable at a byte-level and is volatile. All locations are well-defined initially as zero. This is accessed with MLOAD, MSTORE and MSTORE8 instructions.

Ethereum 101 62

Storage is a 256-bit to 256-bit key-value store. Unlike memory, which is volatile, storage is non-volatile and is maintained as part of the system state. All locations are well-defined initially as zero. This is accessed with SLOAD/SSTORE instructions.

Ethereum 101 63

Calldata is a read-only byte-addressable space where the data parameter of a transaction or call is held. This is accessed with CALLDATASIZE / CALLDATALOAD / CALLDATACOPY instructions.

Ethereum 101 64

EVM does not follow the standard von Neumann architecture. Rather than storing program code in generally accessible memory or storage, it is stored separately in a virtual ROM accessible only through a specialized instruction.

Ethereum 101 65

EVM uses big-endian ordering where the most significant byte of a word is stored at the smallest memory address and the least significant byte at the largest

EVM instruction set can be classified into 11 categories:

a. Stop and Arithmetic Operations

b. Comparison & Bitwise Logic Operations

c. SHA3

d. Environmental Information

e. Block Information

f. Stack, Memory, Storage and Flow Operations

g. Push Operations

h. Duplication Operations

i. Exchange Operations

j. Logging Operations

k. System Operations

Stop and Arithmetic Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):

a. 0x00 STOP 0 0 Halts execution

b. 0x01 ADD 2 1 Addition operation

c. 0x02 MUL 2 1 Multiplication operation

d. 0x03 SUB 2 1 Subtraction operation

e. 0x04 DIV 2 1 Integer division operation

f. 0x05 SDIV 2 1 Signed integer division operation (truncated)

g. 0x06 MOD 2 1 Modulo remainder operation

h. 0x07 SMOD 2 1 Signed modulo remainder operation

i. 0x08 ADDMOD 3 1 Modulo addition operation

j. 0x09 MULMOD 3 1 Modulo multiplication operation

k. 0x0a EXP 2 1 Exponential operation

l. 0x0b SIGNEXTEND 2 1 Extend length of two's complement signed integer

Comparison & Bitwise Logic Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):

a. 0x10 LT 2 1 Less-than comparison

b. 0x11 GT 2 1 Greater-than comparison

c. 0x12 SLT 2 1 Signed less-than comparison

d. 0x13 SGT 2 1 Signed greater-than comparison

e. 0x14 EQ 2 1 Equality comparison

f. 0x15 ISZERO 1 1 Simple not operator

g. 0x16 AND 2 1 Bitwise AND operation

h. 0x17 OR 2 1 Bitwise OR operation

i. 0x18 XOR 2 1 Bitwise XOR operation

j. 0x19 NOT 1 1 Bitwise NOT operation

k. 0x1a BYTE 2 1 Retrieve single byte from word

l. 0x1b SHL 2 1 Left shift operation

m. 0x1c SHR 2 1 Logical right shift operation

n. 0x1d SAR 2 1 Arithmetic (signed) right shift operation

SHA3 (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):

a. 0x20 SHA3 2 1 Compute Keccak-256 hash

Environmental Information (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):

a. 0x30 ADDRESS 0 1 Get address of currently executing account

b. 0x31 BALANCE 1 1 Get balance of the given account

c. 0x32 ORIGIN 0 1 Get execution origination address

d. 0x33 CALLER 0 1 Get caller address

e. 0x34 CALLVALUE 0 1 Get deposited value by the instruction/transaction responsible for this execution

f. 0x35 CALLDATALOAD 1 1 Get input data of current environment

g. 0x36 CALLDATASIZE 0 1 Get size of input data in current environment

h. 0x37 CALLDATACOPY 3 0 Copy input data in current environment to memory

i. 0x38 CODESIZE 0 1 Get size of code running in current environment

j. 0x39 CODECOPY 3 0 Copy code running in current environment to memory

k. 0x3a GASPRICE 0 1 Get price of gas in current environment

l. 0x3b EXTCODESIZE 1 1 Get size of an account's code

m. 0x3c EXTCODECOPY 4 0 Copy an account's code to memory

n. 0x3d RETURNDATASIZE 0 1 Get size of output data from the previous call from the current environment

o. 0x3e RETURNDATACOPY 3 0 Copy output data from the previous call to memory

p. 0x3f EXTCODEHASH 1 1 Get hash of an account's code

Block Information (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):

a. 0x40 BLOCKHASH 1 1 Get the hash of one of the 256 most recent complete blocks

b. 0x41 COINBASE 0 1 Get the block's beneficiary address

c. 0x42 TIMESTAMP 0 1 Get the block's timestamp

d. 0x43 NUMBER 0 1 Get the block's number

e. 0x44 DIFFICULTY 0 1 Get the block's difficulty

f. 0x45 GASLIMIT 0 1 Get the block's gas limit

Stack, Memory, Storage and Flow Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):

a. 0x50 POP 1 0 Remove item from stack

b. 0x51 MLOAD 1 1 Load word from memory

c. 0x52 MSTORE 2 0 Save word to memory

d. 0x53 MSTORE8 2 0 Save byte to memory

e. 0x54 SLOAD 1 1 Load word from storage

f. 0x55 SSTORE 2 0 Save word to storage

g. 0x56 JUMP 1 0 Alter the program counter

h. 0x57 JUMPI 2 0 Conditionally alter the program counter

i. 0x58 PC 0 1 Get the value of the program counter prior to the increment corresponding to this instruction

j. 0x59 MSIZE 0 1 Get the size of active memory in bytes

k. 0x5a GAS 0 1 Get the amount of available gas, including the corresponding reduction for the cost of this instruction

l. 0x5b JUMPDEST 0 0 Mark a valid destination for jumps. This operation has no effect on machine state during execution.

Push Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):

a. 0x60 PUSH1 0 1 Place 1 byte item on stack

b. 0x61 PUSH2 0 1 Place 2-byte item on stack

c. PUSH3, PUSH4, PUSH5…PUSH31 place 3, 4, 5..31 byte items on stack respectively

d. 0x7f PUSH32 0 1 Place 32-byte (full word) item on stack

Duplication Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):

a. 0x80 DUP1 1 2 Duplicate 1st stack item

b. DUP2, DUP3..DUP15 duplicate 2nd, 3rd..15th stack item respectively

c. 0x8f DUP16 16 17 Duplicate 16th stack item

Exchange Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):

a. 0x90 SWAP1 2 2 Exchange 1st and 2nd stack items

b. 0x91 SWAP2 3 3 Exchange 1st and 3rd stack items

c. SWAP3, SWAP4..SWAP15 exchange 1st and 4th..15th stack items respectively

d. 0x9f SWAP16 17 17 Exchange 1st and 17th stack items

Logging Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):

a. 0xa0 LOG0 2 0 Append log record with no topics

b. 0xa1 LOG1 3 0 Append log record with one topic

c. 0xa2 LOG2 4 0 Append log record with two topics

d. 0xa3 LOG3 5 0 Append log record with three topics

e. 0xa4 LOG4 6 0 Append log record with four topics

System Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):

a. 0xf0 CREATE 3 1 Create a new account with associated code

b. 0xf1 CALL 7 1 Message-call into an account

c. 0xf2 CALLCODE 7 1 Message-call into this account with an alternative account's code

d. 0xf3 RETURN 2 0 Halt execution returning output dat

e. 0xf4 DELEGATECALL 6 1 Message-call into this account with an alternative account's code, but persisting the current values for sender and value

f. 0xf5 CREATE2 4 1 Create a new account with associated code

g. 0xfa STATICCALL 6 1 Static message-call into an account

h. 0xfd REVERT 2 0 Halt execution reverting state changes but returning data and remaining gas

i. 0xfe INVALID ∅ ∅ Designated invalid instruction

j. 0xff SELFDESTRUCT 1 0 Halt execution and register account for later deletion

Gas costs for different instructions are different depending on their computational/storage load on the client. Examples are:

a. STOP, INVALID and REVERT are 0 gas

b. Most arithmetic, logic and stack operations are 3-5 gas

c. CALL*, BALANCE and EXT* are 2600 gas

d. MLOAD/MSTORE/MSTORE8 are 3 gas

e. SLOAD is 2100 gas and SSTORE is 20,000 gas to set a storage slot from 0 to non-0 and 5,000 gas otherwise

f. CREATE is 32000 gas and SELFDESTRUCT is 5000 gas

A transaction reverts for different exceptional conditions such as running out of gas, invalid instructions etc. in which case all state changes made so far are discarded and the original state of the account is restored as it was before this transaction executed.

A transaction with a contract address destination has the contract's function target and the required arguments in the data field of the transaction. These are encoded according to the Application Binary Interface (ABI):

Application Binary Interface (ABI): The Contract Application Binary Interface (ABI) is the standard way to interact with contracts in the Ethereum ecosystem, both from outside the blockchain and for contract-to-contract interaction.

a. Interface functions of a contract are strongly typed, known at compilation time and static.

b. Contracts will have the interface definitions of any contracts they call available at compile-time.

Function Selector: The first four bytes of the call data for a function call specifies the function to be called.

a. It is the first (left, high-order in big-endian) four bytes of the Keccak-256 hash of the signature of the function.

b. The signature is defined as the canonical expression of the basic prototype without data location specifier, i.e. the function name with the parenthesised list of parameter types. Parameter types are split by a single comma - no spaces are used.

c. Function Arguments: The encoded arguments follow the function selector from the fifth byte onwards.

Block explorers: are portals that allow anyone to see real-time data on blocks, transactions, accounts, contract interactions etc. A popular Ethereum block explorer is etherscan.io.

Mainnet: Short for "main network," this is the main public Ethereum blockchain. There are other Ethereum “testnets” where protocol or smart contract developers test their protocol upgrades or contracts. While mainnet uses real ETH, testnets use test ETH that can be obtained from faucets. The popular testnets are:

a. Görli: A proof-of-authority (a small number of nodes are allowed to validate transactions and create blocks) testnet that works across clients

b. Kovan: A proof-of-authority testnet for those running OpenEthereum clients

c. Rinkeby: A proof-of-authority testnet for those running Geth client

d. Ropsten: A proof-of-work testnet. This means it's the best representation of mainnet Ethereum

Ethereum 101 85 (1/2)

Ethereum Improvement Proposals (EIPs) describe standards for the Ethereum platform, including core protocol specifications, client APIs, and contract standards. Standards Track EIPs are separated into a number of types: (See here)

a. Core: Improvements requiring a consensus fork as well as changes that are not necessarily consensus critical but may be relevant to “core dev” discussions

b. Networking: Includes improvements around devp2p and Light Ethereum Subprotocol, as well as proposed improvements to network protocol specifications of whisper and swarm

c. ERC: Application-level standards and conventions, including contract standards such as token standards (ERC-20), name registries, URI schemes, library/package formats, and wallet formats

Ethereum 101 85 (2/2)

d. Interface: Includes improvements around client API/RPC specifications and standards, and also certain language-level standards like method names and contract ABIs. The label “interface” aligns with the interfaces repo and discussion should primarily occur in that repository before an EIP is submitted to the EIPs repository

e. Meta: Describes a process surrounding Ethereum or proposes a change to (or an event in) a process

f. Informational: Describes a Ethereum design issue, or provides general guidelines or information to the Ethereum community, but does not propose a new feature

Ethereum 101 86

Eth2 or Ethereum 2.0: refers to a set of interconnected upgrades that will make Ethereum more scalable, more secure, and more sustainable (See here)

Ethereum 101 87

Immutable code: Once a contract's code is deployed, it becomes immutable (with exceptions noted below). Standard software development practices that rely on being able to fix bugs and add new features to deployed code do not apply here. This represents a significant security challenge for smart contract development. There are three exceptions:

a. The modified contract can be deployed at a new address (and old state carried over) but all interacting entities should be notified/enabled to interact with the updated contract at the new address. This is typically considered impractical.

b. The modified contract can be deployed as a new implementation in a proxy pattern where the proxy points to the modified contract after the update. This is the most commonly used approach to update/add functionality.

c. CREATE2 opcode allows updating in place using init_code

Web3: is a permissionless, trust-minimized and censorship-resistant network for transfer of value and information.

a. The popular approach to realise Web3 is to build it over a foundation of peer-to-peer network of nodes for compute, communication and storage.

b. In the Ethereum ecosystem, this is a combination of the Ethereum blockchain, Waku (previously Whisper) and Swarm respectively.

c. Privacy and anonymity are big motivating factors in Web3.

d. Most of the foundational security design principles and development practices from Web2 still apply to Web3. But Web3 security is indeed a paradigm shift along many frontiers.

Languages: Web2 programming languages such as JavaScript, Go, Rust and Nim are used extensively in Web3. But the entire domain of smart contracts is new and specific to Web3. Languages such as Solidity and Vyper were created exclusively for Web3.

On-chain vs Off-chain: Smart contracts are “on-chain” Web3 components and they interact with “off-chain” components that are very similar to Web2 software. So the major differences in security perspectives between Web3 and Web2 mostly narrow down to security considerations of smart contracts vis-a-vis Web2 software.

Open-source & Transparent: Given the emphasis on trust-minimization, Web3 software, especially smart contracts, are expected to be open-source by default.

a. The deployed bytecode is also expected to be source code verified (on a service such as Etherscan). Security by obscurity with proprietary code is not part of Web3's ethos.

b. All interactions with smart contracts are recorded on the blockchain as transactions. This includes the transactions’ senders, data and outcome. Having complete visibility into the entire history of transactions and state transitions is akin to having a publicly accessible audit log of a system since inception.

c. Furthermore, transactions that are still “in flight” and are yet to be confirmed on the blockchain are also publicly visible in pending transaction queues (i.e. mempools) and lend to front-running attacks.

Ethereum 101 92

Unstoppable & Immutable: Web3 applications, popularly known as Decentralized Applications (ĐApps), are expected to be unstoppable and immutable because they run on a decentralized blockchain network.

a. There should not be any one entity that can unilaterally decide to stop a running ĐApp or make changes to it. Transactions and data on the blockchain are guaranteed to be immutable unless a majority of the network decides otherwise.

b. Smart contracts, in general, are expected (by users) to not have kill switches controlled by deployers. They are also expected to not be arbitrarily upgradeable. Both these stem from the Web3 goal of trust-minimization, i.e. lack of need to trust potentially malicious ĐApp developers. However, this makes fixing security vulnerabilities in deployed code and responding to exploits very challenging.

Ethereum 101 93 (1/2)

Pseudonymous Teams & DAOs: Perhaps inspired by Bitcoin's Satoshi Nakamoto, there is a trend among some project teams in Web3 to be pseudonymous and known only by their online handles.

a. One reason for this could be to avoid any potential legal implications in future, given the regulatory uncertainty in this space. This makes it harder to associate any social reputation as it pertains to perceived security trustworthiness of the product or the processes behind its development. It also makes it tricky to hold anyone legally/socially liable or accountable.

b. "Trust software not wetware" (i.e. people) is the mantra here. While this may be an extreme view, there are still social processes around rollout and governance of projects which affect security posture.

Ethereum 101 93 (2/2)

c. To minimise the role and influence of a few privileged individuals in the lifecycle of projects, there is an increasing trend towards governance by token-holding community members — a Decentralized Autonomous Organization (DAO) of pseudonymous token-holding blockchain addresses making voting-based decisions on project treasury spending and protocol changes. While this reduces centralized points of wetware failure, it potentially slows down decision-making on security-critical aspects and may even lead to project forks.

Ethereum 101 94

New Architecture, Language & Toolchains: Ethereum has a new virtual machine (EVM) architecture which is a stack-based machine with 256-bit words and associated gas semantics.

a. Solidity language continues to dominate smart contracts without much real competition (except Vyper perhaps).

b. The associated toolchains which include development environments (e.g. Truffle, Brownie, Hardhat), libraries (e.g. OpenZeppelin), security tools (e.g. Slither, MythX, Securify) and wallets (e.g. Metamask) are maturing but still playing catch up to the exponential growth of the space.

Byzantine Threat Model: The Web3 threat model is based on byzantine faults dealing with arbitrary malicious behavior and governed by mechanism design.

a. Given the aspirational absence of trusted intermediaries, everyone and everything is meant to be untrusted by default. Participants in this model include developers, miners/validators, infrastructure providers and users, all of whom could potentially be adversaries.

b. This is a fundamentally different threat model from that of Web2 where there are generalized notions of trusted insiders with authorized access to resources/assets that have to be protected against untrusted outsiders (and malicious insiders). Web3 is the ultimate zero-trust scenario.

Keys & Tokens: While "crypto" may indeed mean cryptocurrencies to some non-technical observers, it factually refers to cryptography which is a fundamental bedrock of Web3. As much as we unknowingly use cryptography in the Web2 world, Web3 is taking it to the masses. Cryptographic keys are first-class members of the Web3 world.

a. Without the presence of Web2 trusted intermediaries who can otherwise reset passwords or restore accounts/assets from their centralized databases, Web3 ideologically pushes the onus of managing keys (and the assets they control) to end users in their wallets. Loss of private keys (or seed phrases) is irreversible and many assets have been lost to such incidents. This is a significant mindset shift from the Web2 world where passwords have become far too common, security pundits are tired of bemoaning the use of commonly reused simple passwords, password databases continue to be dumped and password-killing technologies continue to evade us. Web2 passwords here symbolize the role of trusted centralized intermediaries that Web3 is seeking to replace.

b. Web2 security breaches targeting financial assets (i.e. excluding ransomware and botnets for DDoS) typically involve stealing of financial or personal data which is then sold on the dark web and used for monetary gain. This is getting much harder because of various checks and measures (both technical and regulatory) being put in place (at centralized intermediaries) to reduce such cybersecurity incidents and prevent anomalous asset transfers. When such unauthorised asset transfers do happen, the involved intermediaries may even cooperate to reverse such transactions and make good.

c. The notion of assets in Web3 is fundamentally different. Cryptoassets are borderless digital tokens whose accounting ledger is managed by consensus on the blockchain and ownership is determined by access to corresponding cryptographic keys. If someone gets access to your private keys controlling cryptoassets, they can transfer those assets to blockchain addresses controlled by their keys. In a perfectly decentralized world, no intermediary (e.g. centralized exchange) should exist that can reverse such a loss — transactions are immutable. Because there are limited response options, preventive security measures become more critical in the Web3 space.

Composability by Design: Permissionless innovation and censorship-resistance are core aspirational goals of Web3.

a. There are numerous stories of Web2 companies that initially enticed developers to build on their platforms only to shut them out later when they were perceived as a competitive threat.

b. Web3 applications, especially smart contracts, are open by design and can be accessed permissionlessly by end users and other smart contracts alike.

c. This makes characterizing Web3 vulnerabilities and exploit scenarios very challenging without deep knowledge of all interacting components, constraints and configurations.

d. This composability lends itself to applications that can be layered on top of others like legos, which is great if everything holds up and new lego toys are reliably built on others. However, this unconstrained composability introduces unexpected cross-systemic dependencies that may trigger invalid assumptions across components (likely built by different teams with different constraints in mind) and expose attack surfaces or modes previously unconsidered.

Compressed Timescales: It feels like innovation in the Web3 space moves at warp speed. Aspects of transparent-development and composability-by-design are strong catalysts to accelerating permissionless and borderless participation which is further incentivized by Internet-native cryptoeconomic tokens — a perfect storm.

a. This shrinks innovation timescales by orders of magnitude where new waves of experiments happen over weeks or months instead of the years it typically takes within the walled gardens of Web2. It may seem like the only moat here is the speed of execution.

b. This compressed timescale has a tangible impact on security considerations during design, development and deployment. Corners are cut and shortcuts taken to ride new waves of hype. The end result is a poorly tested system that holds millions of dollars worth of tokens but is vulnerable to exploits.

Test-in-Prod: A combination of compressed timescale, unrestricted composability, byzantine threat model and challenges of replicating full state for predicting failure modes of interacting components built with rapidly evolving experimental software/tools in many ways forces realistic testing to happen only in production, i.e. on the “mainnet”. This implies that complex technical and cryptoeconomic exploits may only be discoverable upon production deployment.

Audit-as-a-Silver-Bullet: Secure Software Development Lifecycle (SSDLC) processes for Web2 products have evolved over several decades to a point where they are expected to meet some minimum requirements of a combination of internal validation, external assessments (e.g. product/process audits, penetration testing) and certifications depending on the value of managed assets, anticipated risk, threat model and the market domain of products (e.g. financial sector has stricter regulatory compliance requirements).

Web3 projects seem to increasingly rely on external audits as a stamp of security approval. This is typically justified by the lack of sufficient in-house security expertise. While the optics of this approach seems to falsely convince speculators, this approach is untenable for several reasons:

a. Audits currently are very expensive because demand is much greater than supply for top-rated audit teams that have the experience and reputation to analyze complex projects

b. Audits are typically commissioned once at the end of project development just before production release

c. Upgrades to projects go unaudited for commercial or logistical reasons

d. The expectation (from the project team and users) is that audits are a panacea for all vulnerabilities and that the project is “bug-free” after a short audit (typically few weeks)