NATIONAL INSTITUTE OF TRANSPORT

Blockchain Technology Short Training

Introduction to Ethereum

Facilitator: Dr. Cleverence Kombe (PhD)

Introduction to Ethereum | Overview

- Ethereum Virtual Machine
- Ethereum Smart Contracts

Ethereum Blockchain Elements | The Ethereum Virtual Machine - EVM

- Ethereum Virtual Machine is the part of the Ethereum network that handles the deployment and execution of smart contracts.
- It has three addressable data components: an immutable program code ROM, a volatile memory, and permanent storage that is part of the Ethereum state.
- It is a global decentralized computer that contains millions of executable objects, each with its own permanent data store.

Ethereum Blockchain Elements | The Ethereum Virtual Machine - EVM

- The EVM is a simple stack-based execution machine that runs bytecode instructions to transform the system state from one state to another. The word size of the EVM is set to 256-bit.
- The stack size is limited to 1,024 elements and is based on the Last In, First Out (LIFO) queue.
- 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.

- The EVM is an entirely isolated and sandboxed runtime environment. The code that runs on the EVM does not have access to any external resources such as a network or filesystem. This results in increased security, deterministic execution, and allows untrusted code (code that can be run by anyone) to be executed on Ethereum blockchain.

- The EVM is a stack-based architecture. It is also a big-endian by design, and it uses 256-bit-wide words. This word size allows for Keccak 256-bit hash and ECC computations.

Ethereum Blockchain Elements | The Ethereum Virtual Machine - EVM

- There are three main types of storage available for contracts and the EVM:

1. **Memory:** The first type is called memory or volatile memory, which is a word-addressed byte array. When a contract finishes its code execution, the memory is cleared. It is akin to the concept of RAM. write operations to the memory can be of 8 or 256 bits, whereas read operations are limited to 256-bit words. Memory is unlimited but constrained by gas fee requirements.
2. **Storage:** The other type is called storage, which is a key-value store and is permanently persisted on the blockchain. Keys and values are each 256 bits wide. It is allocated to all accounts on the blockchain. As a security measure, storage is only accessible by its own respective CAs. It can be thought of as hard disk storage.

- There are three main types of storage available for contracts and the EVM:

    3. **Stack:** EVM is a stack-based machine, and thus performs all computations in a data area called the stack. All in-memory values are also stored in the stack. It has a maximum depth of 1024 elements and supports the word size of 256 bits.

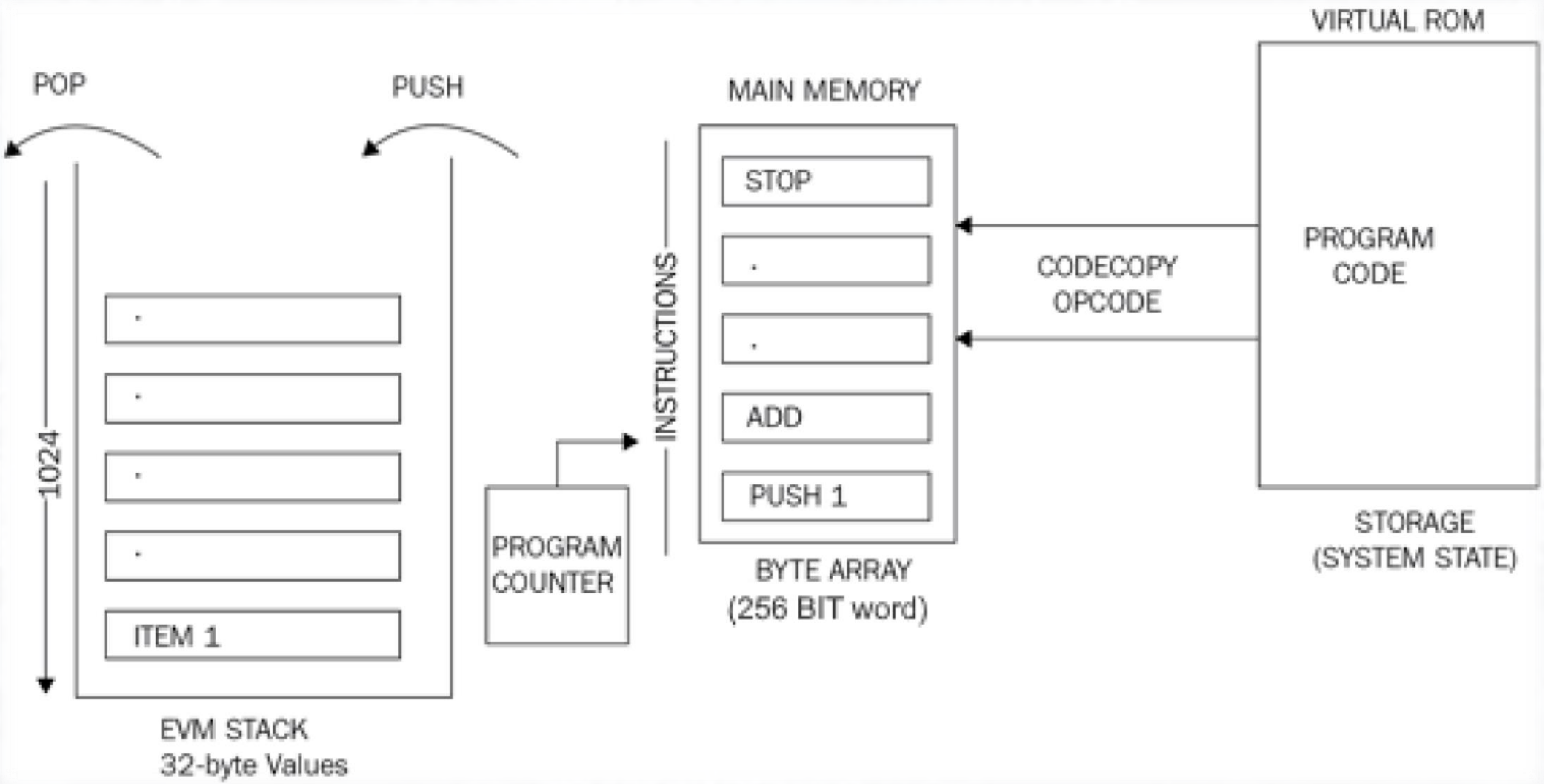
- The storage associated with the EVM is a word-addressable word array that is non-volatile and is maintained as part of the system state. Keys and values are 32 bytes in size and storage.

- The program code is stored in virtual read-only memory (virtual ROM) that is accessible using the CODECOPY instruction. The CODECOPY instruction copies the program code into the main memory. Initially, all storage and memory are set to zero in the EVM.

Ethereum Blockchain Elements | The Ethereum Virtual Machine - EVM

- The diagram in the next slide shows the design of the EVM where the virtual ROM stores the program code that is copied into the main memory using the CODECOPY instruction.
- The main memory is then read by the EVM by referring to the program counter and executes instructions step by step.
- The program counter and EVM stack are updated accordingly with each instruction execution

Ethereum Blockchain Elements | The Ethereum Virtual Machine - EVM



Ethereum Blockchain Elements | The Ethereum Virtual Machine - EVM

- The diagram from previous slide shows an EVM stack on the left side showing that elements are pushed and popped from the stack. It also shows that a program counter is maintained, and is incremented with instructions being read from the main memory.
- The main memory gets the program code from the virtual ROM/storage via the CODECOPY instruction.
- The main memory is then read by the EVM by referring to the program counter and executes instructions step by step.
- The program counter and EVM stack are updated accordingly with each instruction execution

Ethereum Blockchain Elements | The Ethereum Virtual Machine - EVM

- EVM optimization is an active area of research, and recent research has suggested that the EVM can be optimized and tuned to a very fine degree to achieve high performance.
- Research and development on Ethereum WebAssembly (ewasm)—an Ethereum-flavored iteration of WebAssembly—is already underway. WebAssembly (Wasm) was developed by Google, Mozilla, and Microsoft, and is now being designed as an open standard by the W3C community group.
- Wasm aims to be able to run machine code in the browser that will result in execution at native speed. (https://github.com/ewasm)
- Another intermediate language called YUL, which can compile to various backends such as the EVM and ewasm, is under development (https://solidity.readthedocs.io/en/latest/yul.html)

Execution environment

- There are some key elements that are required by the execution environment to execute the code. The key parameters are provided by the execution agent; for example, a transaction. These are listed as follows:
  - The system state.
  - The remaining gas for execution.
  - The address of the account that owns the executing code.
  - The address of the sender of the transaction. This is the originating address of this execution (it can be different from the sender).
  - The gas price of the transaction that initiated the execution.

Execution environment

- There are some key elements that are required by the execution environment to execute the code. The key parameters are provided by the execution agent; for example, a transaction. These are listed as follows:
  - Input data or transaction data depending on the type of executing agent. This is a byte array; in the case of a message call, if the execution agent is a transaction, then the transaction data is included as input data.
  - The address of the account that initiated the code execution or transaction sender. This is the address of the sender in case the code execution is initiated by a transaction; otherwise, it is the address of the account.
  - The value or transaction value. This is the amount in Wei. If the execution agent is a transaction, then it is the transaction value.

Execution environment

- There are some key elements that are required by the execution environment to execute the code. The key parameters are provided by the execution agent; for example, a transaction. These are listed as follows:
  - The code to be executed presented as a byte array that the iterator function picks up in each execution cycle.
  - The block header of the current block.
  - The number of message calls or contract creation transactions (CALLs, CREATEs or CREATE2s) currently in execution.
  - Permission to make modifications to the state.

Execution environment

- The execution environment can be visualized as a tuple of ten elements, as follows:

Address of code owner
Sender address
Gas price
Input data
Initiator address
Value
Bytecode
Block header
Message call depth
Permission

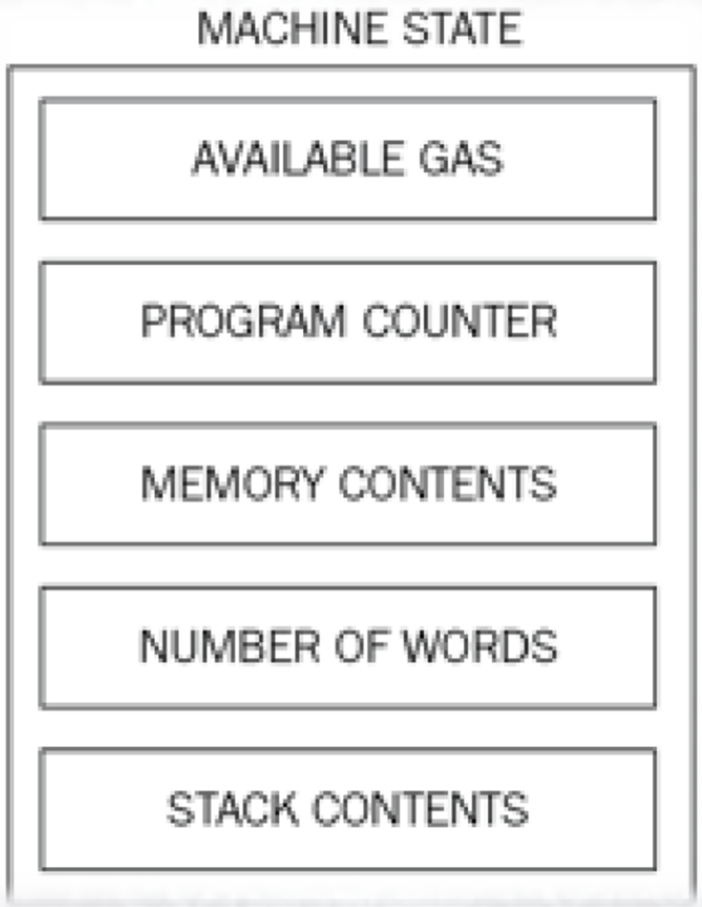
- The execution results in producing the resulting state, the gas remaining after the execution, the self-destruct or suicide set, log series, and any gas refunds.

The machine state

- The machine state is also maintained internally, and updated after each execution cycle of the EVM. An iterator function (detailed in the next slides) runs in the EVM, which outputs the results of a single cycle of the state machine.
- The machine state is a tuple that consists of the following elements:
  - Available gas.
  - The program counter, which is a positive integer of up to 256.
  - The contents of the memory (a series of zeroes of size 2^256).
  - The active number of words in memory (counting continuously from position 0).
  - The contents of the stack

The machine state

- Machine state can be viewed as a tuple, as shown in the following diagram:



The machine state

- The EVM is designed to handle exceptions and will halt (stop execution) if any of the following exceptions should occur:
  - Not having enough gas required for execution.
  - Invalid instructions.
  - Insufficient stack items.
  - Invalid destination of jump opcodes.
  - Invalid stack size (greater than 1,024)

The machine state

- The iterator function mentioned earlier performs various vital functions that are used to set the next state of the machine and eventually the world state. These functions include the following:
  - It fetches the next instruction from a byte array where the machine code is stored in the execution environment.
  - It adds/removes (PUSH/POP) items from the stack accordingly.
  - Gas is reduced according to the gas cost of the instructions/opcodes. It increments the Program Counter (PC).
- The EVM is also able to halt in normal conditions if STOP, SUICIDE, or RETURN opcodes are encountered during the execution cycle..

Ethereum Blockchain Elements | Ether cryptocurrency/tokens

- As an incentive to the miners, Ethereum rewards its own native currency called ether (abbreviated as ETH).
- 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.

Ethereum Blockchain Elements | Ether cryptocurrency/tokens

- The denomination table is as follows:

Unit	Alternative name	Wei value	Number of weis
Wei	Wei	1 Wei	1
KWei	Babbage	1e3 Wei	1,000
MWei	Lovelace	1e6 Wei	1,000,000
GWei	Shannon	1e9 Wei	1,000,000,000
microether	Szabo	1e12 Wei	1,000,000,000,000
milliether	Finney	1e15 Wei	1,000,000,000,000,000
ether	ether	1e18 Wei	1,000,000,000,000,000,000

Ethereum Blockchain Elements | Smart contracts and native contracts

- Smart contracts are computer programs that run on the Ethereum Virtual Machine (EVM) as part of the Ethereum network protocol. The term "contract" has no legal meaning in this context, and the programs are immutable and deterministic.
- Once deployed, the code of a smart contract cannot change, making them different from traditional software. To modify a smart contract, a new instance must be deployed.
- The outcome of executing 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.

Ethereum Blockchain Elements | Smart contracts and native contracts

- Smart contracts operate with a limited execution context, meaning they can only access their own state, the context of the transaction that called them, and some information about the most recent blocks.
- The EVM runs as a local instance on every Ethereum node, but because all instances operate on the same initial state and produce the same final state, the system as a whole operates as a single "world computer."
- Smart contracts are typically written in high-level languages such as Solidity, then compiled to low-level bytecode that runs in the EVM.

Ethereum Blockchain Elements | Smart contracts and native contracts

- Contracts are deployed on the Ethereum platform using a special contract creation transaction, identified as such by being sent to the special contract creation address, 0x0.
- Each contract is identified by an Ethereum address derived from the contract creation transaction as a function of the originating account and nonce.
- The contract address can be used in a transaction as the recipient, sending funds to the contract or calling one of its functions.

Ethereum Blockchain Elements | Smart contracts and native contracts

- There are no keys associated with an account created for a new smart contract, and the contract creator doesn't receive any special privileges at the protocol level. Smart contract accounts own themselves.
- Contracts only run if they are called by a transaction, either directly or indirectly as part of a chain of contract calls. They do not run "on their own" or "in the background."
- Transactions are atomic, and all execution must terminate successfully for any changes in the global state to be recorded. Failed transactions are recorded as having been attempted but rolled back as if the transaction never ran.

Ethereum Blockchain Elements | Smart contracts and native contracts

- A contract's code cannot be changed, but it can be "deleted" by executing the SELFDESTRUCT opcode, removing the code and its internal state (storage) from its address and leaving a blank account.
- The SELFDESTRUCT opcode costs "negative gas," a gas refund, incentivizing the release of network client resources from the deletion of stored state. It does not remove the transaction history of the contract since the blockchain itself is immutable.
- The SELFDESTRUCT capability will only be available if the contract author programmed the smart contract to have that functionality. If the contract's code does not have a SELFDESTRUCT opcode, or it is inaccessible, the smart contract cannot be deleted.

Ethereum Blockchain Elements | Smart contracts and native contracts

- High-level languages such as Solidity allow developers to write smart contracts more easily and with fewer errors. They compile to EVM bytecode that can be executed on the Ethereum platform.
- Solidity is a statically typed, contract-oriented language with features such as inheritance, libraries, and user-defined types. It supports event-driven programming and allows for the creation of contracts with complex functionality.
- Other high-level languages such as Vyper and Bamboo are also available for writing smart contracts, each with their own strengths and weaknesses.

Ethereum Blockchain Elements | Smart contracts and native contracts

- Smart contracts can interact with each other on the Ethereum network, allowing for the creation of complex decentralized applications (dApps) with a variety of use cases.
- The decentralized nature of the Ethereum network means that smart contracts can be executed without the need for intermediaries or trusted third parties.
- The ability to create and deploy smart contracts quickly and easily has led to the growth of the decentralized finance (DeFi) ecosystem, enabling peer-to-peer lending, trading, and other financial services.

- A smart contract has the following properties:

1. **Automatically executable:** It is self-executable on a blockchain without requiring any intervention.
2. **Enforceable:** This means that all contract conditions are enforced automatically.
3. **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.

- A smart contract has the following properties:

4. **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.

5. **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.

6. **Semantically sound:** This means that they are complete and meaningful to both people and computers.

Ethereum Blockchain Elements | Smart contracts and native contracts

```
1  // SPDX-License-Identifier: MIT
2
3  pragma solidity ^0.8.0;
4
5  contract SimpleStorage {
6      uint storedData;
7
8      function set(uint x) public {
9          storedData = x;
10     }
11
12     function get() public view returns (uint) {
13         return storedData;
14     }
15 }
16
```

Ethereum Blockchain Elements | Smart contracts and native contracts

```
1  // SPDX-License-Identifier: MIT
2
3  pragma solidity ^0.8.0;
4
5  contract HelloWorld {
6      string public message;
7
8      constructor(string memory _message) {
9          message = _message;
10     }
11
12     function updateMessage(string memory _newMessage) public {
13         message = _newMessage;
14     }
15 }
16
```

Ethereum Blockchain Elements | Smart contracts and native contracts

```solidity
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract SimpleContract {
    address payable public party1;
    address payable public party2;
    uint public agreedValue;

    constructor(address payable _party1, address payable _party2, uint _agreedValue) {
        party1 = _party1;
        party2 = _party2;
        agreedValue = _agreedValue;
    }

    function confirmAgreement() public payable {
        require(msg.sender == party1 || msg.sender == party2, "Not authorized");
        require(msg.value == agreedValue, "Value must match agreed value");
        party1.transfer(msg.value);
        party2.transfer(msg.value);
    }
}
```

Ethereum Blockchain Elements | Smart contracts and native contracts

```
1  // SPDX-License-Identifier: MIT
2
3  pragma solidity ^0.8.0;
4
5  contract BankAccount {
6      address public owner;
7      uint256 public balance;
8
9      constructor() {
10         owner = msg.sender;
11     }
12
13     function deposit() public payable {
14         balance += msg.value;
15     }
16
```

```
16
17      function withdraw(uint256 amount) public {
18          require(msg.sender == owner, "Only the owner can withdraw funds");
19          require(balance >= amount, "Insufficient funds");
20
21          balance -= amount;
22          payable(msg.sender).transfer(amount);
23      }
24  }
25
```

Ethereum Blockchain Elements | Smart contracts and native contracts

```
1  // SPDX-License-Identifier: MIT
2
3  pragma solidity ^0.8.0;
4
5  contract VotingSystem {
6      // Create an array to hold the candidates
7      string[] public candidates;
8
9      // Create a mapping to keep track of the vote count for each candidate
10     mapping(string => uint256) public votes;
11
12     // Function to add a new candidate to the array
13     function addCandidate(string memory newCandidate) public {
14         candidates.push(newCandidate);
15     }
16
```

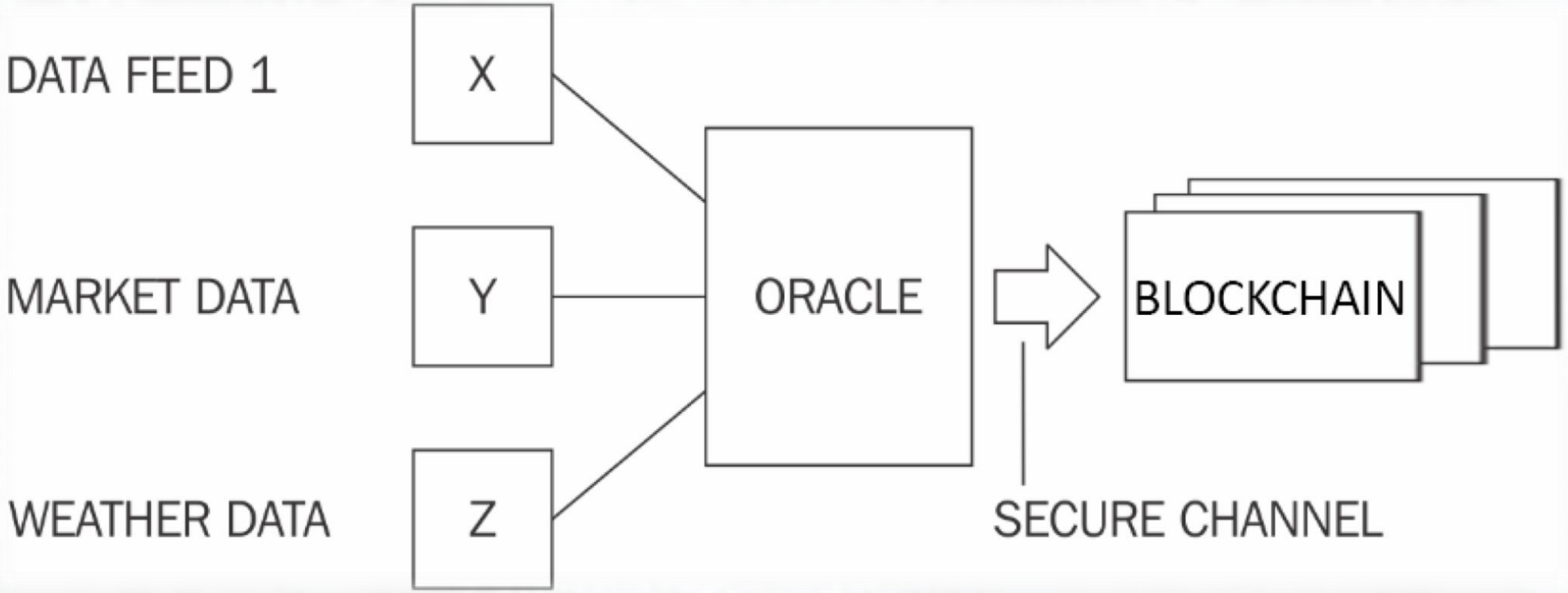
```
16
17      // Function to cast a vote for a candidate
18      function castVote(string memory candidate) public {
19          require(bytes(candidate).length > 0, "Candidate name cannot be empty.");
20          require(validateCandidate(candidate), "Candidate does not exist.");
21          votes[candidate] += 1;
22      }
23
24      // Function to check if a candidate exists in the array
25      function validateCandidate(string memory candidate) public view returns (bool) {
26          for (uint256 i = 0; i < candidates.length; i++) {
27              if (keccak256(bytes(candidates[i])) == keccak256(bytes(candidate))) {
28                  return true;
29              }
30          }
31          return false;
32      }
33  }
34
```

Oracles

- An oracle is an off-chain source of information that provides the required information to the smart contracts on the blockchain.
- 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.

Oracles

- 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.



Oracles

- A list of some of the common use cases of oracles is shown here:

Type of data	Examples	Use case
Market data	Live price feeds of financial instruments. Exchange rates, performance, pricing, and historic data of commodities, indices, equities, bonds, and currencies.	DApps related to financial services, for example, decentralized exchanges and **decentralized finance** (**DeFi**)
Political events	Election results	Prediction markets
Travel information	Flight schedules and delays	Insurance DApps
Weather information	Flooding, temperature, and rain data	Insurance DApps
Sports	Results of football, cricket, and rugby matches	Prediction markets
Telemetry	Hardware IoT devices, sensor data, vehicle location, and vehicle tracker data	Insurance DApps Vehicle fleet management DApps

Oracles

- 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.

Oracles

- The standard mechanics of how oracles work is presented here:

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.

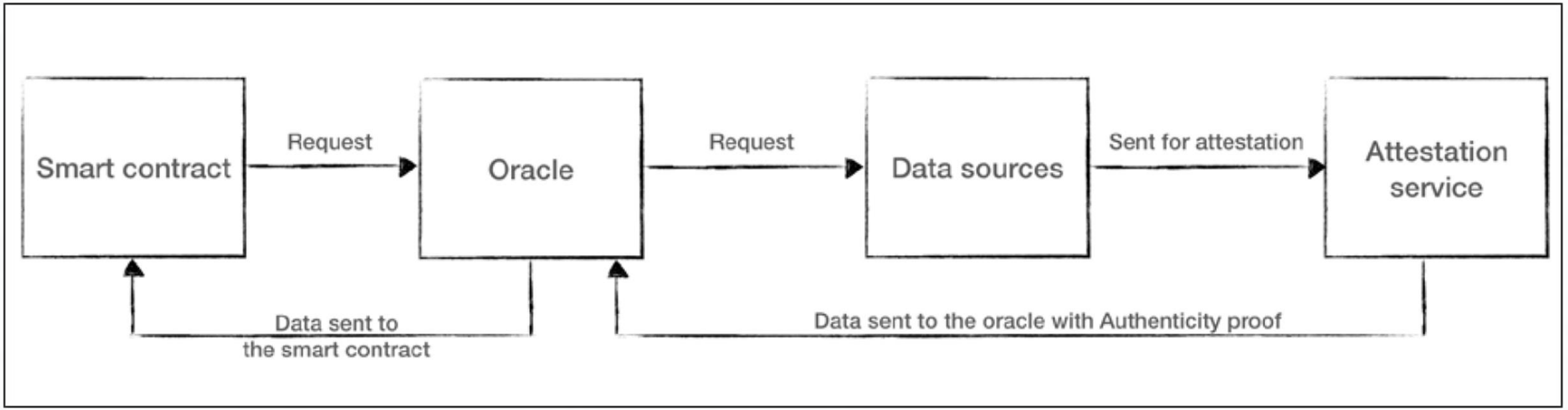
5. The requested data with its proof of authenticity can be optionally saved on a decentralized storage system such as Swarm or IPFS and can be used by the smart contract/blockchain for verification. This is especially useful when the proofs of authenticity are of a large size and sending them to the requesting smart contracts (storing them on the chain) is not feasible.

Oracles

- The standard mechanics of how oracles work is presented here:

6. Finally, the data, with the proof of validity, is sent to the smart contract.

- This process can be visualized in the following diagram:



Oracles

- The preceding diagram shows the generic data flow of a data request from a smart contract to the oracle.
- The oracle then requests the data from the data source, which is then sent to the attestation service for notarization.
- The data is sent to the oracle with proof of authenticity.
- Finally, the data is sent to the smart contract with cryptographic proof (authenticity proof) that the data is valid.
- Due to security requirements, oracles should also be capable of digitally signing or digitally attesting the data to prove that the data is authentic. This proof is called **proof of validity** or **proof of authenticity**.

Oracles

- Smart contracts subscribe to oracles. Smart contracts can either pull data from oracles, or oracles can push data to smart contracts.
- It is also necessary that oracles should not be able to manipulate the data they provide and must be able to provide factual data.
- Even though oracles are trusted (due to the associated proof of authenticity of data), it may still be possible that, in some cases, the data is incorrect due to manipulation or a fault in the system.
- Therefore, oracles must not be able to modify the data. This validation can be provided by using various cryptographic proofing schemes.

Oracles

- The following are different mechanisms to produce cryptographic proof of data authenticity:
  - **Software and network-assisted proofs:** These types of proofs make use of software, network protocols, or a combination of both to provide validity proofs. One of the prime examples of such proofs is TLSNotary, which is a technology developed to be primarily used in the PageSigner project (https://tlsnotary.org/pagesigner.html) to provide web page notarization. This mechanism can also be used to provide the required security services to oracles.
  - **TLS-N based mechanism:** TLS-N is a TLS extension that provides secure non-repudiation guarantees. This protocol allows you to create privacy preserving and non-interactive proofs of the content of a TLS session.

Oracles

- **Hardware device-assisted proofs**: These proofs rely on some hardware elements to provide proof of authenticity. They require specific hardware to work. Different mechanisms come under this category as follow:
  - **Android proof:** This proof relies on Android's SafetyNet software attestation and hardware attestation to create a provably secure and auditable device. SafetyNet validates that a genuine Android application is being executed on a secure, safe, and untampered hardware device. Hardware attestation validates that the device has the latest version of the OS, which helps to prevent any exploits that existed due to vulnerabilities in the previous versions of the OS. This secure device is then used to fetch data from third-party sources, ensuring tamper-proof HTTPS connections. The very use of a provably secure device provides the guarantee and confidence (that is, a proof of authenticity) that the data is authentic.

Oracles

- **Hardware device-assisted proofs**: These proofs rely on some hardware elements to provide proof of authenticity. They require specific hardware to work. Different mechanisms come under this category as follow:
  - **Ledger proof:** Ledger proof relies on the hardware cryptocurrency wallets built by the ledger company (https://www.ledger.com). Two hardware wallets, Ledger Nano S and Ledger Blue, can be used for these proofs. The primary purpose of these devices is as secure hardware cryptocurrency wallets. However, due to the security and flexibility provided by these devices, they also allow developers to build other applications for this hardware. These devices run a particular OS called Blockchain Open Ledger Operating System (BOLOS), which, via several kernel-level APIs, allows device and code attestation to provide a provably secure environment.

Oracles

- **Hardware device-assisted proofs**: These proofs rely on some hardware elements to provide proof of authenticity. They require specific hardware to work. Different mechanisms come under this category as follow:
  - **Ledger proof:** The secure environment provided by the device can also prove that the applications that may have been developed by oracle service providers and are running on the device are valid, authentic, and are indeed executing on the Trusted Execution Environment (TEE) of the ledger device. This environment, supported by both code and device attestation, provides an environment that allows you to run a third-party application in a secure and verifiable ledger environment to provide proof of data authenticity. Currently, this service is used by Provable, an oracle service, to provide untampered random numbers to smart contracts.

Oracles

- **Hardware device-assisted proofs**: These proofs rely on some hardware elements to provide proof of authenticity. They require specific hardware to work. Different mechanisms come under this category as follow:
  - **Trusted hardware-assisted proofs:** This type of proof makes use of trusted hardware, such as TEEs. A prime example of such a hardware device is Intel SGX. A general approach that is used in this scenario is to rely on the security guarantees of a secure and trusted execution provided by the secure element or enclave of the TEE device.

  A prime example of a trusted hardware-assisted proof is Town Crier (https://www.town-crier.org), which provides an authenticated data feed for smart contracts. It uses Intel SGX to provide a security guarantee that the requested data has come from an existing trustworthy resource.

Types of blockchain oracles

- There are various types of blockchain oracles, ranging from simple software oracles to complex hardware assisted and decentralized oracles. They categorized into inbound oracles and outbound oracles.
- **Inbound oracles**: This class represents oracles that receive incoming data from external services, and feed it into the smart contract. They can be 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.

Types of blockchain oracles

- There are various types of blockchain oracles, ranging from simple software oracles to complex hardware assisted and decentralized oracles. They categorized into inbound oracles and outbound oracles.
- **Inbound oracles**: This class represents oracles that receive incoming data from external services, and feed it into the smart contract. They can be 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.

Types of blockchain oracles

- **Inbound oracles**: This class represents oracles that receive incoming data from external services, and feed it into the smart contract. They can be software, hardware, and several other types of inbound oracle:
  - **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.

Types of blockchain oracles

- **Inbound oracles**: This class represents oracles that receive incoming data from external services, and feed it into the smart contract. They can be software, hardware, and several other types of inbound oracle:
  - **Hardware oracles:** 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.

Types of blockchain oracles

- **Inbound oracles**: This class represents oracles that receive incoming data from external services, and feed it into the smart contract. They can be software, hardware, and several other types of inbound oracle:
  - **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.

Types of blockchain oracles

- **Inbound oracles**: This class represents oracles that receive incoming data from external services, and feed it into the smart contract. They can be software, hardware, and several other types of inbound oracle:
  - **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.

Types of blockchain oracles

- **Inbound oracles**: This class represents oracles that receive incoming data from external services, and feed it into the smart contract. They can be software, hardware, and several other types of inbound oracle:
  - **Aggregation based oracles:** 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.

Types of blockchain oracles

- **Inbound oracles**: This class represents oracles that receive incoming data from external services, and feed it into the smart contract. They can be software, hardware, and several other types of inbound oracle:
  - **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.

Types of blockchain oracles

- **Inbound oracles**: This class represents oracles that receive incoming data from external services, and feed it into the smart contract. They can be software, hardware, and several other types of inbound oracle:
  - **Crowd wisdom driven oracles:** 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.

Types of blockchain oracles

- **Inbound oracles**: This class represents oracles that receive incoming data from external services, and feed it into the smart contract. They can be software, hardware, and several other types of inbound oracle:
  - **Decentralized oracles:** 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.

Types of blockchain oracles

- **Inbound oracles**: This class represents oracles that receive incoming data from external services, and feed it into the smart contract. They can be software, hardware, and several other types of inbound oracle:
  - **Decentralized oracles:** A decentralized oracle essentially allows off-chain information to be transferred to a blockchain without relying on a trusted third party. Augur is a prime example of such type of oracles. 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.

Types of blockchain oracles

- **Inbound oracles**: This class represents oracles that receive incoming data from external services, and feed it into the smart contract. They can be software, hardware, and several other types of inbound oracle:
  - **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-oraclesto-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.

Types of blockchain oracles

- **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.

Types of blockchain oracles

- **Outbound oracles**:

  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.

Native contracts

- There are nine precompiled contracts or native contracts in the Ethereum Istanbul:

1. **The elliptic curve public key recovery function:** ECDSARECOVER (the ECDSA recovery function) is available at address 0x1. It is denoted as ECREC and requires 3,000 gas in fees for execution. If the signature is invalid, then no output is returned by this function. Public key recovery is a standard mechanism by which the public key can be derived from the private key in ECC.

   The ECDSA recovery function is shown as follows:

   ECDSARECOVER (H, V, R, S) = Public Key

   It takes four inputs: H, which is a 32-byte hash of the message to be signed, and V, R, and S, which represent the ECDSA signature with the recovery ID and produce a 64-byte public key.

Native contracts

- There are nine precompiled contracts or native contracts in the Ethereum Istanbul:

2. **The SHA-256-bit hash function:** The SHA-256-bit hash function is a precompiled contract that is available at address 0x2 and produces a SHA256 hash of the input. The gas requirement for SHA-256 (SHA256) depends on the input data size. The output is a 32-byte value.

3. **The RIPEMD-160-bit hash function:** The RIPEMD-160-bit hash function is used to provide a RIPEMD 160-bit hash and is available at address 0x3. The output of this function is a 20-byte value. The gas requirement, similar to SHA-256, is dependent on the amount of input data.

Native contracts

- There are nine precompiled contracts or native contracts in the Ethereum Istanbul:

4. **The identity/datacopy function:** The identity function is available at address 0x4 and is denoted by the ID. It simply defines output as input; in other words, whatever input is given to the ID function, it will output the same value. The gas requirement is calculated by a simple formula: 15 + 3 [Id/32], where Id is the input data. This means that at a high level, the gas requirement is dependent on the size of the input data, albeit with some calculation performed, as shown in the preceding equation.

5. **Big mod exponentiation function:** This function implements a native big integer exponential modular operation. This functionality allows for RSA signature verification and other cryptographic operations. This is available at address 0x05.

Native contracts

- There are nine precompiled contracts or native contracts in the Ethereum Istanbul:

6. **Elliptic curve point addition function:** This is the implementation of the elliptic curve point addition function. This contract is available at address 0x06.

7. **Elliptic curve scalar multiplication:** This is the implementation of the same elliptic curve point multiplication function. Both elliptic curve addition and doubling functions allow for ZKSNARKS and the implementation of other cryptographic constructs. This contract is available at 0x07.

8. **Elliptic curve pairing:** The elliptic curve pairing functionality allows for performing elliptic curve pairing (bilinear maps) operations, which enables zk-SNARKS verification. This contract is available at address 0x08.

Native contracts

- There are nine precompiled contracts or native contracts in the Ethereum Istanbul:

    6. **Blake2 compression function 'F':** This precompiled contract allows the BLAKE2b hash function and other related variants to run on the EVM. This improves interoperability with Zcash and other Equihash-based PoW chains. This precompiled contract is implemented at address 0x09. The EIP is available at https://eips.ethereum.org/EIPS/eip-152.

- Ethereum Berlin update introduced new pre-compiled contracts for BLS12-381 curve operations along with other updates.

- EIP is available here at https://eips.ethereum.org/EIPS/eip-2070.

Introduction to Ethereum | Summary

►In this session, we discussed:

- Ethereum Blockchain Elements
  - Ethereum Virtual Machine
    - Stack-based architecture
    - EVM optimization
    - Execution environment
    - The machine state
  - Ethereum Smart Contracts
    - Smart contract properties
    - Oracles
    - Native contracts


THANK
YOU