UCSB CS 291D: Blockchains and Cryptocurrencies **Fall 2020**

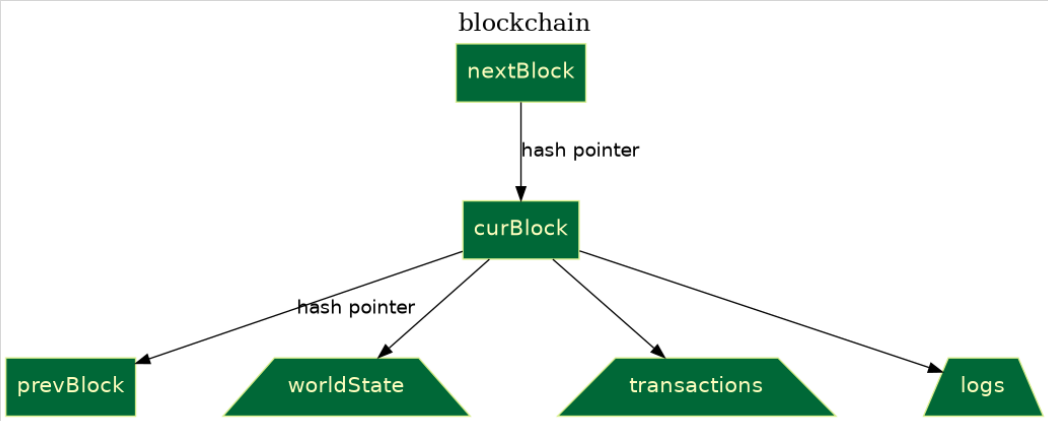
Lecture 7: EVM and Solidity Programming

Lecturer: Shumo Chu *Scribes: Harlan Kringen, Jiyu Zhang*

This lecture's notes walk through a high-level overview of the Ethereum Virtual Machine (EVM) and its associated programming language, Solidity.

7.1 Architecture and Design of the EVM

The design of the Ethereum blockchain resembles Bitcoin overall but departs in its handling of the users' account state. Whereas Bitcoin employs a UTXO approach, Ethereum uses a more traditional input/output ledger approach which is implemented by the blockchain. In this blockchain model a block has hash pointers to a world state, transactions, logs, as well as other transaction blocks. This may be roughly visualized in the following diagram.



7.1.1 Workflow

The Ethereum blockchain, like Bitcoin, has the ability to store information immutably, but it can also run general computations on its blockchain, in the form of smart contracts, which are simply small programs that can work with any information committed to the blockchain.

The Ethereum blockchain runs these programs on the Ethereum Virtual Machine (EVM), which is a stack machine similar to the engine that runs Bitcoin script. Smart contracts can be written in any language that compiles to EVM bytecode, however Solidity is the most common.

Every contract, like every transaction, is sent to all miners on Ethereum to be verified. The manner in which a contract is successfully verified is somewhat involved. First, the miners are rewarded with 2 Ether plus a transaction cost for every block. Second, there is an extra layer of payment that is mediated through the notion of "gas" that is unique to Ethereum. This will be explained below.

The EVM itself is not a powerful computer, having about as much computing power as a smartphone. Additionally, the EVM stack machine's stack depth is relatively shallow, capped at 1024 levels. This contributes

to its spartan computational profile.

7.2 Gas and Incentives

As mentioned in the introduction, the process of verifying a block for the blockchain requires the willful participation of other miners. The miners must spend computational resources to verify a given block, and so the reward for doing this must be clear and competitive with whatever else the miners could be using their computational power for.

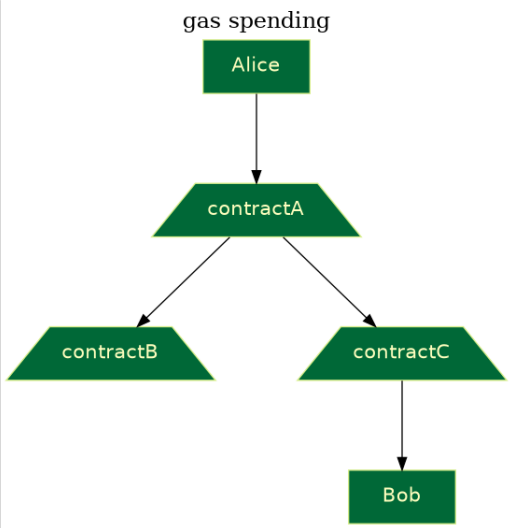
Ethereum creates another layer to the incentive system, namely gas. The concept of gas, which is decoupled into two subcomponents, gas price and gas limit, is an amount of Ether (typically denoted in Wei) that approximates the amount of computational power needed to run a contract. For a given contract, the EVM must spend some number of cycles executing and Ethereum attributes to each cycle a gas price.

As a decoupled notion, gas price is a user-defined primitive that reflects the user's expectation of how much computational power will be required to run the contract. The gas limit is a ceiling on how much the gas price may be, which is simply another check on users abusing the stack machine's (and miners') computational power. The total cost of a transaction is $gasPrice * gasLimit$, and the fee paid to the miner is the total amount of gas needed to run the contract multiplied by the gasPrice and any leftover gas is returned to the user.

If the gas price for a contract is larger than the amount of gas needed to run the contract, the difference will be refunded to the user. The gas itself then pays the miners who worked to verify the block. If the gas price is insufficient, this will trigger a runtime error and the contract itself will not be added to the blockchain, although gas spent on the contract will still go to the miners involved.

There are of course problems inherent in the concept of gas. Because it is dynamically computed, a user can only make an educated guess at how much gas a contract will require. Even worse, if a contract invokes other contracts, the problem of estimating gas usage is compounded in a non-obvious way. We can see this in the below diagram.

Ultimately gas serves a dual purpose. In the first case gas adds another mechanism for miners and users to leverage in order to commit their blocks to the blockchain, by, for instance, setting a competitive and attractive gas price. In the second case, gas is a convenient way to ensure the EVM is not swamped with unnecessary work and forces smart contracts to be slim and lightweight.



7.3 Solidity

The Solidity language is a front-end to the EVM, offering a programming language with java-like syntax that compiles to EVM bytecode. Solidity enables programming with two fundamental notions, accounts and transactions. Accounts cover human-owned wallets as well as generic smart contracts. Transactions are any communication between accounts.

7.3.1 Memory

There are two types of memory offered to the user, persistent and volatile. Persistent memory keeps track of data across multiple contract invocations, and is relatively expensive, partially because every miner on the blockchain must keep track of it as well. Volatile memory is a freshly cleared amount of memory allotted to each contract at the beginning of each message call.

7.3.2 Language Features

The Solidity language offers a feature called "require" which acts like an assert as in C/C++. There is additionally a reserved function "emit" which directly writes to the blockchain. Moreover, there are several global variables and functions that a contract might call out to, including "gasLeft", "SHA256" and "keccak256".

There are also several function visibilities. These are keywords that constrain who may invoke a given function.

1. external: functions marked "external" may only be called from outside the contract
2. public: anyone may invoke these functions
3. private: such functions may only be used from within the defining contract

4. internal: may be used by the defining contract as well as derived contracts
5. view: these functions may only read storage
6. pure: these functions do not touch storage whatsoever

7.3.3 Code Reuse

There are two unique ways in which contracts can be reused, revolving around an import mechanism, similar in spirit to the import statement in Python or Haskell. The first style is termed "inheritance", and involves using other contracts as pre-compiled bytecode directly in a contract. This is commonly done with the SafeMath contract, which provides exception-free commonly used mathematical functions. The second style uses other contracts as libraries, which are not pre-compiled.

7.3.4 Types

Solidity contains several base types, including uint256, byte32, bool, and address literals that are used to hold hashes but may not be used in arithmetic.

There are also reference types, including structs, arrays, strings, as well as sort of constrained hash table called a "mapping".

7.4 ERC20 Tokens

Solidity contracts have a strong tendency to describe financial instruments and related constructions. To this end, there was an Ethereum Improvement Proposal that created a standard template for fungible tokens. The template describes ERC20 tokens and provides a standard API for transfering and spending such tokens. It is a very commonly reused contract in the Ethereum ecosystem.

7.4.1 ABI

Fundamentally, the fact that Solidity offers multiple programming features that encourage writing safe, minimal code is irrelevant if the core bits that are being exchanged in these complicated transactions cannot be agreed upon. This fact put the concept of ABI or Application Binary Interface at the forefront of Solidity's design. The ABI is a lower level description of functions (than say, an API), which explains how functions will exactly behave at the byte level.

References

[Sol] "Solidity, v0.7.4". [Online]. Available: https://solidity.readthedocs.io/en/v0.7.4/index.html". Accessed on: Nov. 2, 2020.