An Analysis of Solidity through ERC-20 Development

Archibald Latham, Lexie Rista, Chiedozie Anazia
CSC421: Final Project

Introduction:

Solidity is a high level object oriented programming language that was created for the purpose of smart contracts development on the Ethereum blockchain. The language is comparable to more popular languages such as Java and is developed and maintained by the Ethereum team, most notably, Gavin Wood. Ethereum is currently the second largest crypto currency by market capitalization in the world and is the largest blockchain that supports smart contracts. To understand smart contracts it is important to understand the concept of a blockchain. Typically when people refer to a blockchain they are referring to an immutable ledger that is constantly added to and verified by a decentralized network of computers. In the case of Ethereum, this network is referred to as the Ethereum Virtual Machine (EVM). Smart contacts are pieces of code that are run by the EVM and have their output recorded onto the blockchain. This is in contrast to other languages such a Python where the code is run in a local or centralized environment. To run a smart contract on the EVM, one must pay a fee of Ether typically referred to as gas. When people refer to Ethereum mining they are referring to the distribution of this gas to the nodes that construct the EVM.

Smart contracts offer exciting opportunities especially in the world of finance. As of now most smart contracts are used to facilitate the creation of assets or the transfer of assets. The benefits of using a smart contract opposed to a traditional contract is that smart contracts are trustless and do not require a centralized third party. In this paper, we will detail the design of Solidity and demonstrate how it can be used to construct an ERC-20 smart contract.

Language and Semantics:

Solidity is very similar to Java and is influenced by Python and C++. It is statically typed like Java, C, and C++ (Ethereum Revision). This means that the type of variable is initialized at compile-time rather than run-time. Therefore, it is required to specify each type of variable before compiling the code. All undefined variables will result in a compile error. Java, C, and C++ will throw the same error as Solidity with undeclared variables. In addition, single line and multi-line comments are declared as they are in Java with "//" and "/* */" (Figure 1). Finally, Solidity like Java requires a semicolon to note the end of a line.

Solidity's similar syntax to Python can be seen in the example of imports. Even though the syntax is the same, the ordering can differ. Syntax of Python imports are "import __ from __" or "from __ import __ as __". However, when importing in Solidity, declaration is "import __ as ___ from "filename"; " (Figure 2).

All Solidity documents begin with the line "pragma solidity" followed by the version number of Solidity being used and all Solidity documents are noted by the . sol extension.

```
import matplotlib.pyplot as plt
```

```
import "github.com/ethereum/dapp-bin/library/iterable_mapping.sol" as it_mapping;
```

Figure 1. Python import syntax vs Solidity import syntax

```
// This is a single-line comment.

/*
This is a
multi-line comment.
*/
```

Figure 2. Solidity comments (Ethereum Revision)

Unique Features:

Contracts: In Solidity, contracts are declared using the keyword “contract” and are comparable to classes in a language like Java. Contracts can only be called from a unique address through a transaction on the EVM. This address is noted as “msg.sender” When a contract is created it’s constructor function is called once using the keyword “constructor” This is a unique function that is typically used to intilalize variables. In addition, contracts have the property of inheritance which means that they can inherit the functionality of other contracts using the keyword “is”.

Function Visibility: In Solidity, functions are declared using the keyword “function” and can accept and return variables. Functions have four possible types of visibility that must be noted upon declaration and include external, public, internal, and private. External functions can be called from other contracts and from transactions on the EVM. Public functions can be called internally or through a message which does not require a transaction. Internal functions can only be called by the contract it belongs to or a contract that inherits that contract. Private functions can only be called by the contract in which they are declared.

```
contract SafeMath {
    function safeAdd(uint a, uint b) public pure returns (uint c) {
        c = a + b;
        require(c >= a);
    }
    function safeSub(uint a, uint b) public pure returns (uint c) {
        require(b <= a);
        c = a - b;
    }
    function safeMul(uint a, uint b) public pure returns (uint c) {
        c = a * b;
        require(a == 0 || c / a == b);
    }
    function safeDiv(uint a, uint b) public pure returns (uint c) {
        require(b > 0);
        c = a / b;
    }
}
```

Figure 3: SafeMath contract with public functions

Events: In Solidity, events are a unique data type similar to a function that is declared using the keyword “event” and can contain multiple variables. One can call an event using the keyword “emit” and passing it relevant variables. When called through a transaction all of the data passed to the event will be recorded on the blockchain with that transaction. This is an important tool for logging data about transactions on the blockchain.

```
contract ClientReceipt {
    event Deposit(
        address indexed _from,
        bytes32 indexed _id,
        uint _value
    );

    function deposit(bytes32 _id) public payable {
        // Events are emitted using `emit`, followed by
        // the name of the event and the arguments
        // (if any) in parentheses. Any such invocation
        // (even deeply nested) can be detected from
        // the JavaScript API by filtering for `Deposit`.
        emit Deposit(msg.sender, _id, msg.value);
    }
}
```

Figure 4: Example event

Address: To interact with the EVM one must have a unique Ethereum address. Typically personal addresses are referred to as wallets and are used to store and transfer cryptocurrency. In Solidity, there are two kinds of address types: address and address payable. Both types of addresses are represented as a unique 20 byte value beginning with 0x, but address payable is distinct from address as it is able to hold and transfer funds. Depending on the type of conversion (implicit or explicit) address to address payable is not allowed, in this case it would be implicit. Contracts deployed to the network are also identified by a unique address. Users are able to

query the balances of any address with the use of “balance” and transfer Ether to another address by calling the function transfer() (Figure 5).

```
address x = 0x123;
address myAddress = this;
if (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);
```

Figure 5. Using a query of address to transfer Ether (Ethereum Revision)

Mappings: Solidity’s mapping function serves as a hashtable or dictionary, essentially mapping a key to a value type. In Solidity, mapping is used to link unique addresses to its associated value type. Declaring a mapping type is shown in Figure 6. However, only a data’s keccak256 is stored in the hash in order to look up the value, not key data. Note that “keccak256” is a cryptographic function used to compute Ethereum-SHA-3 (Keccak-256) hash (Ethereum Revision) and has additional functionality generating pseudo random numbers within a contract.

```
mapping(_KeyType => _ValueType)
```

Figure 6. Declaration of mapping type

Require: In Solidity, Require is a “convenience function” that will check for conditions, throw exceptions or check invariants (BitDegree). This function is similar to Assert and Revert, which all are very important when handling errors because they are considered “state-reverting exceptions” (Ethereum Revision). Assert and Require are used to check conditions before moving forward. If the condition lacks compliance, an exception is thrown. While these functions both handle errors, they differ when they are used. Assert is usually used for internal checking, while Require ensures that all inputs, state variables, and return values are addressed to external contracts. Require can check the amount of ether that is being exchanged. If it is not

enough (meaning it doesn't comply with the require or assert statement), the transaction is denied. Revert can also be used to tell the EVM to "revert all changes made to the state" (Ethereum Revision) if the condition is not met. Users know that there is a bug in their code if it reaches a "failing assert statement." Contracts with these bugs must be fixed.

```
pragma solidity ^0.4.22;

contract Sharer {
    function sendHalf(address addr) public payable returns (uint balance) {
        require(msg.value % 2 == 0, "Even value required.");
        uint balanceBeforeTransfer = this.balance;
        addr.transfer(msg.value / 2);
        // Since transfer throws an exception on failure and
        // cannot call back here, there should be no way for us to
        // still have half of the money.
        assert(this.balance == balanceBeforeTransfer - msg.value / 2);
        return this.balance;
    }
}
```

Figure 7. Implementation of Require and Assert in contract called Sharer (Ethereum Revision)

Design Pattern Safeguards:

The prominence of the Ethereum blockchain have led to several security issues. In order to mitigate damage or to defend against possible attacks, Solidity has implemented better security design patterns to assure safe transport of instructions. Checks-And-Effects-Interactions are a design pattern preventing reentracy attacks by making sure the address call is the last step of code in order to avoid any type of external interaction. The unwanted interaction between functions is usually the root cause of this, but limiting the code execution by minimizing the transfer of funds(ether) protects the contract from malicious operations (Wöhrer and Zdun 5).

Emergency Stop/Circuit Breakers are a simple way to stop contract code execution once a malicious agent is detected. For example, if a bug is detected during the transfer of ether, then all the operations are halted in order to protect against a loss of any remaining assets. The emergency stop disables the function within the contract (Wöhrer and Zdun 5).

ERC-20:

One of the most notable use cases of smart contracts written using Solidity is the creation of tokens. In the world of cryptocurrency, coins are assets that are associated with a unique blockchain such Bitcoin, Ethereum, or Polkadot, while tokens are assets that are built off of an existing blockchain. For a smart contract to be considered a token on the Ethereum network it must meet what is called the ERC-20 standard. Other standards such as ERC-721 or ERC-1155 are also used to classify non-fungible tokens (NFTs) and these standards have even been extended to other blockchains such as Binance Smart Chain which uses the BEP-20 standard. In fact, a large majority of the cryptocurrencies that currently exist are ERC-20 tokens or some extension of the ERC-20 standard. The first notable feature of the ERC-20 standard is SafeMath which is a basic contract that a token must inherit in order to provide security against integer overflow attacks. The contract includes basic functions for addition, subtraction, multiplication, and division which must be used across the inheriting contract. The next notable features that an ERC-20 token must inherit are the Owned and ERC20Interface contracts. These contracts allow any address on the Ethereum network to have a balance of a given token associated with it. This balance is stored and verified on the blockchain. In addition this balance must be transferable which means that it can be sent to another address. This is done by subtracting a number from the balance of the sending address and adding that number to the balance of the receiving address using SafeMath. These are just the basic functions that an ERC-20 token must be capable of but many tokens will have further unique features such as burning which allows a portion of token to be destroyed during transfer as to create a deflationary asset.

CSC-421 Token:

In this project we used Solidity to create and deploy an ERC-20 compliant smart contract on the Ropsten Test Network. The Ropsten Network is a network that models the Ethereum Network but is used solely for the testing of smart contracts. The Ropsten Network operates on Ropsten Ethereum which has no value and can be received for free from a Ropsten faucet. To create our token we implemented all of the basic features outlined above and in addition gave our token a name, a ticker symbol, and a max supply of 100000000 tokens. The smart contract associated with the token was deployed to the Ropsten Network using Remix, a browser based IDE and the total supply was initialized to a personal wallet (Remix IDE). To demonstrate the use of this token on the network we used the total supply and 10 test Ether to create a liquidity pool on Uniswap, a decentralized exchange (DEX). Decentralized exchanges are unique to crypto currency markets and differ from centralized exchanges (CEX) such as Binance or NASDAQ. DEXs work by the creation of liquidity pools which are composed of tokens and some underlying asset, in this case Ether. Anyone is able to add liquidity to an existing pool provided they can supply both assets in an equal amount of value. If someone wants to buy or sell a token they can trade off of the liquidity in these pools. The fees that a trader pays are distributed to the liquidity providers according to their share of the pool. These DEXs are run entirely off of smart contracts and shares of liquidity pools are even allotted as ERC-20 tokens. Anyone is able to go to Uniswap on the Ropsten Network and trade test Ether for our token (Uniswap). The contract address of our token is:

0x4af1CD46b87Cea2Bf5f7c890Ae8ded4871e109CE0x4af1CD46b87Cea2Bf5f7c890Ae8ded4871e109CE

This contract can be viewed in depth on Etherscan, a popular tool used to explore the Ethereum Network. Here the transactions that occurred to create the contract and provide liquidity to Uniswap can be viewed as they were in fact recorded on the blockchain (Figure 8).

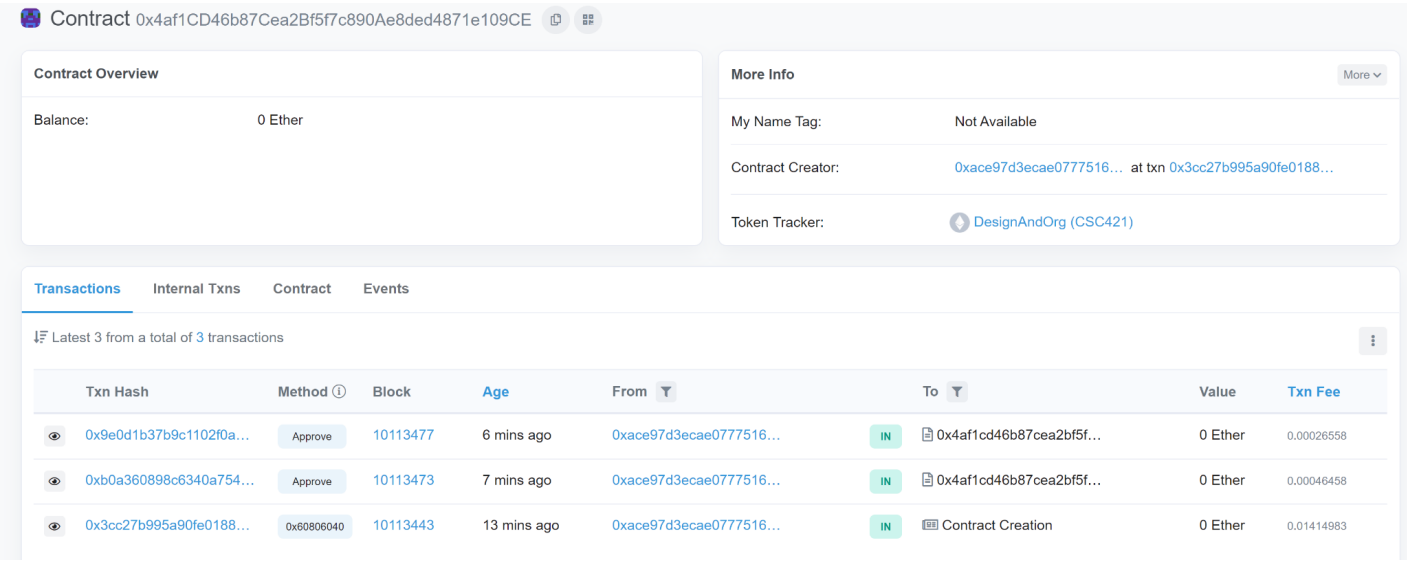

Figure 8: Etherscan contract query

Resources:

BitDegree. “Learn to Use Solidity Require and Master Solidity Assert.” *BitDegree*, BitDegree, 8 Dec. 2017, www.bitdegree.org/learn/solidity-require.

Ethereum Revision. *Solidity*, 2021, docs.soliditylang.org/en/v0.8.4/.

Etherscan. *DesignAndOrg*, 2021, https://ropsten.etherscan.io/address/0x4af1cd46b87cea2bf5f7c890ae8ded4871e109ce

Remix IDE. 2021, https://remix.ethereum.org/

Uniswap. 2021, https://app.uniswap.org/#/swap

Vahiwe. “Vahiwe/Building-Your-Own-ECR20-Token.” *GitHub*, 2019, www.github.com/vahiwe/Building-your-own-ECR20-Token

Woher and Zdun. “Smart Contracts: Security Patterns in Ethereum Ecosystem and Solidity” http://eprints-dev5.cs.univie.ac.at/5433/7/sanerws18iwbosemain-id1-p-380f58e-35576-preprint.pdf