**Smart Contracts - Blockchain Independent Study**

Joseph Weller – Dr. Magana

1/18/2022

**Ethereum**

Ethereum’s white paper was originally published in January 2014 by one of its co-founders, Vitalik Buterin. For the first time, users could not only store transactions in the blocks of a blockchain, but they could now also store and run custom programs which follow “**when…then…**” statements on the blockchain. These programs would officially be called smart contracts, a term coined by Nick Szabo in 1998 ([Investopedia](https://www.investopedia.com/terms/s/smart-contracts.asp#:~:text=Smart%20contracts%20were%20first%20proposed,before%20the%20invention%20of%20bitcoin.)).

**Contracts**

Before going into more detail about these programs called smart contracts, it’s important to discuss the relevance of them being called ‘contracts’ in the first place. A contract, by definition, is a set of rules/clauses that multiple parties agree on which govern the relationship between said parties.

For example, a contract is written up between yourself and another party that states that they’ll own your house if he pays you a certain sum of cash, in legal terms, this is referred to as the offer part of a contract. Once you’ve come to an agreement on the details of the contract with the other party, it can be notarized and saved by a notary; this is called acceptance in legal terms. Next, **when** your neighbor signs the contract and hands you the cash amount agreed upon in the contract, both in the presence of the notary, **then** ownership of your house is transferred to them, which is, in legal terms, the consideration part of the contract.

**Smart Contracts**

If you think about it, programs, REST APIs in particular, have similar behavior. When creating them, the programmer creates rules (drafting of the contract / creating an API endpoint / offer) that once adhered to (the money is transferred / the endpoint is called / acceptance) result in an outcome (changing house ownership / state changes and returning of information / consideration).

These smart contract programs, similar to transactions, are immutably stored on the blockchain, which, when compared to the contract example, acts as the notarizing agent. The development these smart contracts opens an entirely new domain of computer science, which wasn’t previously attainable. Before the concept of these contracts existed, when an online user chose to agree to something online, they had to blindly trust the word of the developers who made said agreement. Now, the opportunity opens up to use a smart contract to facilitate these user facing agreements. This way, the user doesn’t have to place their trust in whoever created what they’re agreeing to, as they can verify for themselves that the settlement that they’re agreeing to is exactly what they’re expecting. This is because the code for these smart contracts, and the interactions surrounding them, is publicly available on the blockchain ([IBM](https://www.ibm.com/topics/smart-contracts)). Lastly, in addition to keeping agreements between parties trustless, the transparency of smart contracts can be leveraged for a variety of different purposes, such as supply chain management, digital identity, insurance claims, clinical trials, and the list goes on. ([101blockchains](https://101blockchains.com/smart-contract-use-cases/)).

To create a program such a smart contract, a programming language is needed. This language varies from blockchain to blockchain, with more modern blockchains even creating their own SDKs so that developers can program contracts using existing languages like Rust and Python. However, in the case of Ethereum, the language Solidity is used. Solidity is a Turing complete, object oriented, compiled programming language which has syntax which is similar to JavaScript and C++ ([Wiki](https://en.wikipedia.org/wiki/Solidity)). Once a program has been compiled is Solidity, it’s intended to be ran on the Ethereum Virtual Machine (EVM).

**Potential Vulnerabilities and Solutions**

As mentioned before, solidity is a Turing complete language. This essentially means that all possible logic is able to be captured within the language. For example, all general-purpose languages in wide use such as Java and Python are Turing complete. However, in order for any language to be considered Turing complete they must be able to provide the ability to create loops. However, these smart contracts run on every node on the blockchain, so if an infinite loop were to be created in one, it would significantly stall the performance of the blockchain network. This activity therefore needs to be prevented. Additionally, since the smart contract is running on the distributed network, more specifically on other people’s computers, it’s important to that these computers are not at risk of malicious programs/viruses being uploaded to them, as well as not being at risk of their personal files being read/removed.

The latter of these issues is solved by the aforementioned Ethereum Virtual Machine. Any computer that chooses to participate in the Ethereum network is provided with the EVM, which is responsible for completely encapsulating all Ethereum transactions and Smart Contract executions that take place on the machine. This means that if anything were to go wrong, it would be contained within the EVM and it could just be reset.

For the former issue of preventing loops/heavy calculations from frequently occurring across the network, the concept of Gas was created. The concept of gas can be simplified to the idea that for every computation that’s executed through a Smart Contract on the blockchain, the Smart Contract must pay a fee to the network. This fee is equal to the computational effort required to execute the specific operations queried in the Smart Contract on the blockchain network.

Briefly demo [this table](https://docs.google.com/spreadsheets/d/1n6mRqkBz3iWcOlRem_mO09GtSKEKrAsfO7Frgx18pNU/edit#gid=0) of Gas fees per operation. Mention that these gas fees incentivize developers of smart contracts to write efficient code, so they can not only save money, but also not throttle the entire network. Additionally, mention that these fees are distributed to the network of computers who are running these Smart Contracts.

**Use cases (dApps)**

In order to use a smart contract, interfaces must be developed to interact with them. By combining a smart contract, acting as a back end, and a web application, acting as a front end, a dApp (decentralized application) is created.

Example: <https://opensea.io/>

* Connect metamask ETH wallet
* Navigate to the explore tab and then click on an NFT collection
* Explain that the data being displayed for the general stats of the collection are populated through reading the state of the dApps smart contract that was created to return said information
* Click on one of the NFT’s in the collection
* Explain how adding a bid for the NFT is done by interacting with a smart contract, by clicking on the bid button, you’re sending a query to a smart contract that binds the amount of Ethereum tokens that you’re willing to spend on the NFT
  + If you’re outbid your Ethereum will be refunded, however, if you’re in the lead when the auction timer expires, your wallet will be sent the NFT and your Ethereum will be sent to the NFT’s previous owner
* Show how the functionality is completely trustless as both the interface and the smart contract are visible to the public
  + [Frontend interface code](https://github.com/ProjectOpenSea)
  + [Smart Contract code](https://etherscan.io/address/0x7be8076f4ea4a4ad08075c2508e481d6c946d12b#code)

**ERC Standards**

While it is technically possible to write the Smart Contracts running these dApps from scratch, that would raise the bar to enter this space as a developer way higher than necessary. In addition to this, if each Smart Contract had their own custom functions and unique data types, the interoperability of Smart Contracts would be next to nonexistent.

To combat these issues, ERC (Ethereum Request for Comments) were created. These are rules that Ethereum based Smart Contracts can choose to comply with if they want to be part of the large interoperable ecosystems of other Ethereum dApps. Each standard can be thought of as an OOP interface, describing the inputs and outputs of various methods required to interact with other Smart Contracts that use the same standards.

Before jumping into the next examples, it’s important to discern between what a fungible token is and what a non-fungible token is. The best way to distinguish these two types of tokens is by looking at an analogy. The US dollar is considered to be fungible, because if you were to trade one $20 bill for someone else’s $20 bill, no value would be lost or gained by either party. However, an example of non-fungible objects would be art. If you were to trade a recreation of a Picasso painting you made that actual Picasso piece, a large gain of value would occur on your part, and not so much for the other party trading you the art.

For the first example, ERC-20 is the fungible token standard, which is meant for you creating your own token on top of the Ethereum blockchain. Some of the methods in this standard’s interface include: balanceOf(account), transfer(account, amount), totalSupply(), etc. It should be noted that methods can be created in addition to the methods required by the ERC-20 standard, these additional methods are what make each of these tokens unique from each other. Some popular tokens using the ERC-20 standard include Chainlink, Shiba Inu, WBTC, Loopring, BAT, etc.

A different example would be the ERC-721 standard which is meant for non-fungible tokens (NFTs). These are tokens which are completely unique to each other that can represent a wide range of things from digital artwork in the metaverse to real estate in the real world. The ERC-721 standard includes methods such as: safeTransferFrom(address, address), ownerOf(tokenID), etc. Examples of these NFTs were seen in the previous OpenSea demonstration and can also be seen using in [this dApp](https://www.lofty.ai/deals) which is powered by the Algorand blockchain.