VRFCoordintorV2 Mock with Oracle

Wayne Wignes

July 11, 2023

Contents

Introduction

This project was inspired by the Chainlinks VRF Coordinator Mock[2] which is a 'bare-bones' mock of the VRFCoordinatorV2.sol contract, the purpose of which is to act as a 'coordinator' between consumers who make requests for verifiable random values and the 'Oracle' which provides performs the mathematical/ computational steps to fulfill the request(s).

The applications of verifiable random values (VRF's) are numerous, e.g. applications that involve betting or which require the random selection of things/ people in the name of fairness. There are also a number of more subtle applications of VRF's in smart contract development (contracts which govern blockchain transactions) in which they can be used to add a critical or just additional layer of security for more common transactions in which, were an interested and capable hacker able to predict some value of interest, then they'd be able to manipulate transactions and divert funds into some unintended account.

The project takes the bare-bones VRF Coordinator Mock provided by Chainlink, implements it using *eth-brownie*, and scales it up appreciably – to the point it is essentially implementing the full `VRFCoordinatorV2.sol` contract. The highlight of the project however, is mocking of the behavior of the oracle contract and 'VRF Machinery' (see diagram on Chainlink's subscription method page). The script `VRFMachinery.py` listens for specified events (requests for random words in this case) which are emitted by the mock coordinator contract. This is accomplished using EventWatcher which is a threader function specific to the eth-brownie ecosystem. Once a request for random words is detected by `EventWatcher`, `VRFMachinery.py` will then respond by invoking the mock coordinator contract (referred to as `mock2` in deployment[1] and `COORDINATOR` in other scripts/ contracts) to fulfill the request using the function `fullfillRandomWords` as well as some other peripheral functions which will help emulate a transaction in its fullness, e.g. registering/ de-registering proving keys, updating the 'blockHashStore', calculating the cost of fulfillment for the user to see[2], and printing a list of pending/unfulfilled requests.

In the mock here developed the 'subscription method' of making requests is utilized.

For the sake of simplicity, a 'consuming' contract is not implemented as is done in the 'bare-bones' mock provided by Chainlink. However, a wrapper contract is deployed, and it is this contract which is called to execute the initial request for randomness. Aside from this, the only role assigned to the wrapper contract is to act as the blockHashStore which stores a mapping of block numbers to block hashes for each request made. To make the simulation more complete, a contract which mocks the oracle is also deployed, but the only role this contract will play is to withdraw funds from the coordinator contract once fulfillment of a request is complete as payment for the VRF Machinery's service (so the `VRFMachinery.py` script and the `Oracle.sol` contract can both be taken to be representative of Chainlink – the provider of randomness). Note that I do not claim that designating the wrapper contract to be the most realistic or complete, and there will be some other implementations in the mock for which the same can be said. The idea is instead to establish *some* functional mock of the VRF request/ fulfillment process so that the interested mathematician can come along and easily implement the necessarily elliptic curve cryptography[1] methods for providing verifiable randomness, thereby making the mock even more complete (likely to be a future project of mine[3]).

[1] I avoid calling it `mock2.sol` because there are two contracts – 'mock2' and 'mock3' [the V3 aggregator] – which are defined in the file `MOCK.sol`.

[2] though `fullfillRandomWords` calculates and returns the payment amount, it is not a view function

[3] I have intentionally avoided digging into elliptic curve cryptography because, knowing myself, once I go there, there will be

There are differences between the mock coordinator contract developed here and the actual `VRFCoordinatorV2.sol` contract, but I feel they are going to be relatively minor differences. Some of these differences have a functional purpose (and these will be explained) and some resulted from the fact that I developed the mock in a piecemeal fashion; I'd implement one function at a time, troubleshoot it, then add another – until I got all 800+ lines of it working. In fact, ironically, it did not occur to me that I could manipulate the `revert()` commands such that the revert message actually shows to users of eth-brownie which has a tendency to hide these messages to users. So, to make matters worse, I'd implement *part* of a function – one line at a time if necessary –, deploy the contracts, and test to see if a particular transaction reverted. It was a painful process, but a good learning experience!

One of the first things the reader might notice upon comparing the mock developed here and the `VRFCoordinatorV2.sol` contract which is downloaded as part of the Chainlink framework – a package that includes numerous smart contracts which can be used for development purposes – is that I deleted the contextual comments which are contained in this specific version of `VRFCoordinatorV2.sol`. The user can always consult/ view these omitted comments by downloading this package and viewing this version of `VRFCoordinatorV2.sol`. In fact, downloading Chainlinks package will be a requisite to the implementation of this mock anyway.

Getting Started

If the user is not familiar with eth-brownie and the basics of smart-contract development I recommend starting with Patrick Collins video[3] on smart contract development.

Assuming the user is familiar with eth-brownie and has downloaded it, then the first thing to be done is to initialize a brownie project. Create a directory that you want to contain the project, navigate to it in VS Code (or whatever software you are using to run brownie) then run

```
brownie init
```

In VS Code `ctrl + b` will bring up the folders pane, in which the user should now see all the requisite folders for a brownie project. `scripts` and `contracts` are really the only two folders we need be concerned with, but there is also the `.env` file and `brownie-config.yaml` file. The `.env` file is used to store private keys to user accounts and node provider (e.g. Alchemy or Infura) project/ API keys which lets brownie know what account address and what node provider http address to use (respectively) when making transactions . `brownie-config.yaml` (or 'config' file) is used to specify the network and other settings for brownie, and this is where the addresses of our deployed contracts will be stored.

Copy and paste the contents of `scripts` and `contracts` folders in this github repo into the `scripts` and `contracts` folders of your newly created brownie project. Also copy and paste the `.env` file as well as the `brownie-config.yaml` (or config) file directly into your projects root directory (so it should appear below the aforementioned folders). The user will of course need to update the `.env` file with their own private key and project Id (Infura) or API key (Alchemy) for the node provider.

no stopping myself. At the moment of writing this time does not permit me to go there.

Note: upon running `deploy.py` the contract addresses will automatically be updated in the config file under the network of deployment. This occurs via the function `UpdateConfigAdresses` defined in the `helpful_scripts.py` script. `UpdateConfigAdresses` reads the config file then overwrites and replaces it with the newly deployed contract addresses, thereby saving the user from needing to update these addresses manually. So if all goes well the user should never have to touch the config file!. There is however, one possible exception to this which will be discussed shortly. The `.env` file will never be overwritten when deploying contracts.

Now download the aforementioned Chainlink framework package, or `node_modules` folder. From your projects root directory run

```
npm install @chainlink/contracts --save
```

The user should see the `node_modules` folder pop up in their projects root directory, and this will contain all the contracts which our mock contracts will need to inherit (or to declare instances of). If there are any problems with this step the user can simply delete the `node_modules` folder and reinstall it. Check that the specific version of the Chainlink framework package downloaded matches the value specified under `dependencies` and also under `remappings` in the config file (see the `package.json` file which should show up beneath your project folders after downloading). Also check the `package-lock.json` folder to see that the version of open-zeppelin (included in the Chainlink framework package) matches that specified in the config file. The line(s) of interest should look line,

```
    "node_modules/@openzeppelin/contracts-v0.7": {
      "name": "@openzeppelin/contracts",
      "version": "3.4.2",
        ...
```

**Note: For whatever reason, version `3.4.0` for open-zeppelin also seems to work. If issues persist, play around with this.* For more help see Chainlink framework or, if that link is broken, *GettingStarted* → *Resources* → *InstallFrameworks* on Chainlink's website.

It is recommended to first run the program on the development network, which in brownie is the ganache-cli by default. Because we are mocking an entire request/ receive network rather than the implementation of just one contract, a number of transactions need to execute in the deployment of contracts and when making a request. For example, in deployment there are five contracts to deploy, two of which need to be configured, three of which are funded[4], and a subscription which needs to be created, funded, and have two consumers (the wrapper and the owner [account holder]) added to it. As it is written the config file is set to run on two networks; development and the sepolia testnet. ganache-cli is a local network meaning it does not actually connect to any live testnet (in this sense it is a mock network). ganache-cli will execute transactions (e.g. deployments/ contract calls) with almost no delay whatsoever. However, in our case we actually *want* a delay because even on a mock network like ganache sometimes transactions are not processed (confirmed) in the order that we run them, and this is especially true when using a threader.

In brownie, launch a terminal and run,

[4] I probably could have gotten away with funding only one or two of the contracts, but the `LinkToken.sol` contract we are using (by inheriting it) gives something like 10^{27} test LINK token to play around with, so its not like we do not have enough to go around! Contracts are funded with 100 LINK which is an excessive amount – more than we'll be needing any time soon.

```
ganache-cli --deterministic --blockTime 2
```

The `--deterministic` flag ensures that our deployments to the local ganache network will remain active and accessible for as long as we keep this terminal open. The `--blockTime` flag actually sets the *number* of confirmations we wait for [each] transaction, but because the confirmation time is pre-determined in ganache (I'd guess its about ~ 1 second per confirmation) setting the number of confirmations amounts to the same as setting the time to confirmation. This is equivalent to the command `tx.wait(2)` when running on a live testnet.

***Note:** if running on sepolia or some other testnet, it is recommended the user add the command `tx.wait(2)` after any function call which alters the state of the contract (a transaction). The user can distinguish transactions which alter the state of a contract by the fact the last input argument for the called function will be a dictionary with the statement {`"from": account`} which lets brownie know what account to charge for the gas fees of executing the transaction.

Though the code is written with the presumption that it is being ran on the development network, when testing on sepolia I find that two block confirmations (only one for some transactions) is usually sufficient. In addition, the user will note `time.sleep([some number])` is used after some transactions. Just how necessary these statements (which simply cause the program to stall for some specified number of seconds) are when running a threader on a live testnet like sepolia I am unsure, but they definitely *are* necessary when running a threader on ganache. For example, in `request.py` there is the command `time.sleep(25)` which is by far the longest wait command in any of the scripts, but it ensures that the entirety of what transactions the threader contained in `VRFMachinery.py` needs to execute have time to do so before the initial request is confirmed.

Once ganache has been launched a new terminal needs to be initialized ('+' tab) and this is where we then run the command

```
brownie run scripts/deploy.py --network development
```

The above command initiates the deployment of our contracts. Once contracts have deployed, check the addresses which the running of `deploy.py` prints out at the very end indeed match the addresses in the config file under the network of deployment.

To initiate a request for randomness to the Oracle/ VRFMachinery run

```
brownie run scripts/request.py --network development
```

The printouts will indicate what steps the VRFMachinery is executing in order to fulfill the request.

About the Code

A Slight Modification Concerning the Public Key (`pk`):

The user will note that `fulfillRandomWords` accepts two arguments; `proof` and `rc`. The input `rc` needs to be in the structure of the `struct RequestCommitment` defined in the mock2 (coordinator) contract while

proof needs to be in the structure of the struct Proof which is defined in VRFI.sol file. VRFI.sol is a mock of the VRF.sol contract which can be located in the node_modules folder. The purpose of the VRF.sol contract is to verify the randomness given by the Oracle using the tools of elliptic curve cryptography – something that, for the moment at least, we are not going to do in this project (but in the future it is a worthy undertaking, and will likely be a future endeavor of mine). Here we mock the mathematical proof and response of the VRFMachinery by defining the following values,

```
    gamma  = [1,2]  ; pk = [2,5]
    proof = [keyHash, pk, gamma, preSeed]
    rc    = [\_blockNum, subId, callbackGasLimit, numWords, WRAPPER.address]
```

where the input parameters were either previously defined in the VRFMachinery.py file or they were obtained via detection of the request event which has been detected by the VRFMachinery. All we need to know about gamma and pk for constructing a crude mock of the proof for randomness is that they are two dimensional vectors with integer inputs; they can be any integers, and in a more realistic mock they'd likely change along with each request.

The role that the parameter gamma plays as well as a more conceptual overview of *VRF.sol* will be discussed next. Here our focus will be on the *pk* or 'public key' value which we will see requires a custom modification to the mock.

Blockchain relies extensively on one-way cryptographic hashing algorithms (in solidity this is usually accomplished by keccack256(...)) which 'hashes' some input into a byte code that is A) unique to the input and B) not feasibly predicted or reproduced without knowing the original input which was hashed. So if I encode the number 1 using abi.encode(1) then hash it with the keccack256 the output of which will be,

```
    "0xd0679ef8aa6802866b61d0aef581f9f7b727af18a0585efdbca04990815c93ee"
```

and I then send this output to someone else, unless that person knows or can guess that the original input value which got encoded and hashed was 1, I am reasonably assured that they'll have little to no hope of deducing what the original input was simply by looking at the output hash value. However, because I know the hashing algorithm not only produces *unique* hash values for a given input, but furthermore that it provides *consistent* hash values (i.e. the same input will always yield the same output hashed value) soon as I tell the person I sent the hash to the original input which got hashed was 1 they'll be able to verify for themselves that this hash indeed represents the information which I originally encoded then hashed and sent to them. How they'd verify this is by simply encoding and hashing the number 1 for themselves and comparing it to the hash which I originally provided to them.

To understand what modification needs to me made to the mock with regard to the pk value it is helpful to highlight some key parts of a few different functions which deal with this value. Specifically, it is in the calculation of 'requestId' I wish to draw attention to. The calculations of this value are highlighted in blue.

```
==================== REQUEST ==============================
requestRandomWords(...){...
   (uint256 '_requestId', uint256 _preSeed) =
```

```
        computeRequestId(_keyHash,msg.sender,_subId,nonce);
-------------------------------------------------------------
computeRequestId {...
   uint256 preSeed = uint256(keccak256(abi.encode(keyHash, sender, subId,
                                                  nonce)));
   return (uint256(keccak256(abi.encode('keyHash, preSeed'))), preSeed);
   ...}

==================== FULLFILL ==============================
getRandomnessFromProof{...
   keyHash    = hashOfKey(proof.pk);
   'requestId = uint256(keccak256(abi.encode(keyHash, proof.seed)))';
   ...}

-------------------------------------------------------------
hashOfKey {
      return keccak256(abi.encode(publicKey));}
```

In the above `getRandomnessFromProof` is a function used within `fulfillRandomWords` to calculate the values of `keyHash, requestId, randomness` The parameter `randomness` will be discussed shortly. Note in the above how the parameters `keyHash` and `requestId` are not new – they were included in the original request which was sent to the oracle (everything past the '===FULLFILL===' line is post oracles response to the coordinator). In the fulfillment process these values are *reproduced* by the coordinator who seemingly opted to do so instead of simply storing them when the request is made then later retrieving them from storage during fulfillment. And from a couple of revert statements [not shown] we know these values must be the same as those involved in the request.

More specifically, it is the line `keyHash = hashOfKey(proof.pk)` which causes a problem for us from a mock perspective. Contrary to the name 'public key' the `pk` value is actually not available to the public (or at least it is not made readily available within Chainlinks documentation). What *is* given to us is the resulting value of `keyHash` (see supported networks in the chainlink docs) which tells the coordinator contract what 'gas lane' to use. Without knowing the `pk` value we obviously cannot *reproduce* its keyHash. We can however, alter our mock such that the value of `keyHash` is used directly in the fulfillment process rather than being reproduced/ re-calculated. This was acheived by altering the structure of the proof such that `keyHash`, and not just `pk` is included in the proof. This is achieved by altering the `struct Proof` declaration in our mock `VRFI.sol` then slightly altering the behavior of the `getRandomnessFromProof` to use this value instead of the one it calculates form the arbitrary value of `pk` which we previously fed into the proof.

Vera the Verifier and Reggie the Oracle

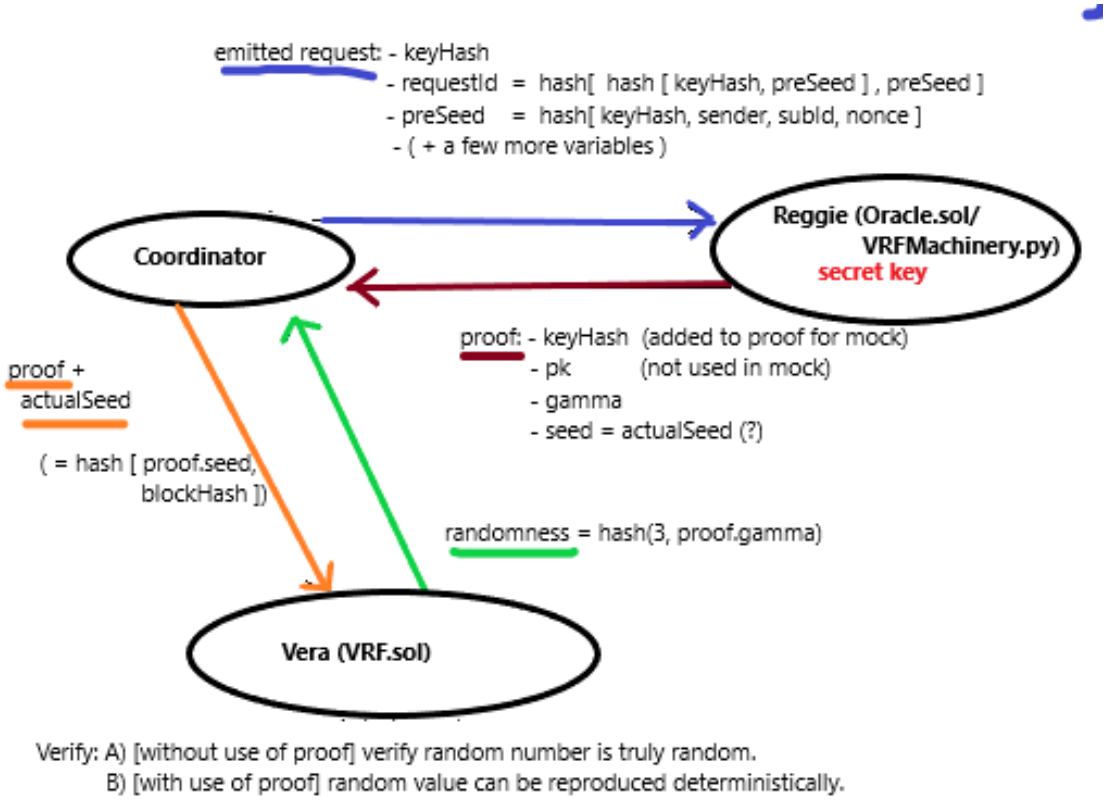
```

 * @dev Reggie the Random Oracle (not his real job) wants to provide randomness
 * @dev to Vera the verifier in such a way that Vera can be sure he's not
 * @dev making his output up to suit himself. Reggie provides Vera a public key
 * @dev to which he knows the secret key. Each time Vera provides a seed to
```

```
6  * @dev Reggie, he gives back a value which is computed completely
7  * @dev deterministically from the seed and the secret key.
8
9  * @dev Reggie provides a proof by which Vera can verify that the output was
10 * @dev correctly computed once Reggie tells it to her, but without that proof,
11 * @dev the output is computationally indistinguishable to her from a uniform
12 * @dev random sample from the output space.
```

The above comment is from `VRF.sol`. It is a slightly modified version of the same analogy found in a few of the chainlink contracts to do with the request process. We should note that `VRF.sol` is where all the elliptic curve cryptography comes into play, so we might take the above analogy as some high-level mathematicians attempt to make some esoteric process more conceptually digestible to the public. However, without the tools of elliptic curve cryptography one would be hard pressed to make overly detailed since of the above analogy.

A diagram will help clarify what the code actually does.



Beginning with the emitted request, the `preSeed` value is formed by hashing together `keyHash, sender, subId , nonce`. [somehow] the oracle uses this preSeed to generate a proof of randomness that is returned to the Coordinator who in turn sends it to Vera (`VRF.sol`) along with `actualSeed` value which is the original seed hashed

together with the blockHash value. We know the value of proof.seed = preSeed (Reggie returned to the Coordinator the same seed value as was original sent to him) because fulfillRandomWords computes requestId using this seed value. Why the blockHash is hashed into actualSeed I am unsure exactly, but it may be to provide an added layer of security (or randomness) to the process.

In the above analogy it is claimed that Reggie would also provide the random value itself, so it must be a part of the proof, no? Skipping ahead to the end output/ reply of Vera to the Coordinator, we see that randomness = hash[3,gamma] where the constant 3 is for whatever reason declared as VRF_RANDOM_OUTPUT_HASH_PREFIX in VRF.sol. gamma is declared as a two dimensional vector with integer inputs. As is suggested by the name 'elliptic curve cryptography' (and also from a glance at some of the functions contained in VRF.sol) a reasonable inference might be that gamma represents either the coordinates or the tangent of some point on a two dimensional curve. In either case we see that gamma provides the value of randomness; if you know this value, then you can reproduce the value of the random output (randomness) by hashing it with the number 3.

Vera has two tasks;

1. (without looking at the proof given to her) verify that the value of randomness = hash[3,gamma] "belongs to a uniform random sample". Noting that mathematicians tend to choose their words with purpose, strictly speaking from a statistics perspective, there are a few of things wrong with that last quoted phrase' 1) one does not test if a value belongs to a 'sample' so much as one tests to see if it belongs to a probability distribution 2) to do so requires a sample of input values to be tested – not just a single value (gamma) and 3) in the end what we'd get is a *probability* that the sample belongs to this distribution (a uniform random distribution), and as far as mathematical rigor goes this would be a far cry from a 'proof'. Ergo, we infer that it is more than a simple statistical test being performed; to understand what is happening would require the tools of elliptic curve cryptography.

somehow reproduce the random value (hash[3,gamma]) with use of the proof (gamma excluded?).

Another thing which is not immediately obvious [to those who lack a background in random number generation and/ or elliptic curve cryptography] in the above analogy of Vera the Verify and Reggie the Oracle is this; what role does actualSeed play in the generation of randomness and in its validation by Vera, and again why was the purpose of hashing the original seed value with the blockHash? What we do know is that "Each time Vera provides a seed to Reggie, he gives back a value which is computed completely deterministically from the seed and the secret key", yet it is not immediately apparent why the oracle would require some user-input seed value to produce *some* random number, and if so, what role the seed would play in validating its value and unpredictability. But clearly the seed value plays a critical role in creating the random value as well as proving and validating its randomness.

As I understand it, a seed will act as the initial starting point of the random number generation process, and this make intuitive sense; once you have some algorithm which *appears* to generate random outputs[5] it would then require some starting point (so imagine a simple random walk algorithm in which you must specify the random walkers initial position). And this might help explain why the Coordinator hashed the seed value before sending it to the oracle to begin with; it makes the initial starting point given to the Oracle itself an [almost/ apparently] random value. But its not random enough as anyone who knows or who can

[5] true randomness is, as far as we mortals yet know, not achievable.

guess the user inputs to the seed could simply hash these inputs and reproduce the `preSeed` value. And this all [perhaps] gives us a little more insight as to why the Coordinator, upon receiving the proof, further hashed the seed value with the `blockHash` to prodduce `actualSeed` as it provides an additional layer of randomness. But at the same time it (hashing the blockHash and seed value) makes the question of how Vera goes about using the resulting `actualSeed` value to validate the proof more curious; if the seed value which the oracle gave back to the Coordinator was not hashed together with the blockHash value (and it was on this seed value that the proof was generated) then how is it that the Coordinators manipulation before sending it to Vera does not interfere with her ability to validate the proof?.

Without further insight to elliptic curve cryptography I think I've hit the end of any reasonable speculation of which I am at the moment capable of offering. I will close by including some comments in `VRF.sol` which give clarity to the relation between the seed value and the random number generation process/ proof.

```
1  * @dev Full uniqueness: For any seed and valid VRF public key, there is
2  * @dev   exactly one VRF output which can be proved to come from that seed, in
3  * @dev   the sense that the proof will pass verifyVRFProof.
4
5  * @dev Full collision resistance: It's cryptographically infeasible to find
6  * @dev   two seeds with same VRF output from a fixed, valid VRF key
7
8  * @dev Full pseudorandomness: Absent the proofs that the VRF outputs are
9  * @dev   derived from a given seed, the outputs are computationally
10 * @dev   indistinguishable from randomness.
```

Additional Notes

- what is likely the first potential error which those who are new to brownie are likely to encounter is that the commands in the 'Getting Started' section caused an error. The commands printed in this section *are* correct (at least at the time of writing this) as they appear here, but for whatever reason copying and pasting these commands from this pdf (compiled with Miktex) causes an extra space after '-' or '' characters. Make sure to delete these extra spaces.
- as previously mentioned in a footnote, there is not `mock2.sol` (coordinator) *file*. There is instead a `Mock.sol` file which contains both the `mock2` and `mock3` (V3 aggregator, or 'pricefeed' contract which gives us the most up to Link to Eth (or Weth) conversion rates).
- One or two of the contracts (e.g. the wrapper) declares the mock coordinator as an instance in a way which requires a function to be invoked *after* deployment in order for the deployed address of the coordinator contract to be set. This is neccessary because the coordinator declares the wrapper as an instance, but if there is a way for two contracts to simultaneously inherit one another (via declaring an instance of the other) then I am unaware of it. In lieu of such a method we must deploy the wrapper, then deploy the coordinator with the wrappers address included in its constructor, then go back and update the wrapper contract with the coordinators address.

- one major discrepancy between the mock and the real `VRFCoordinatorV2.sol` contract which the user may note is the absence of the `nonReentrant` modifier in a number of functions. There is no functional purpose to this ommission; as noted in the introduction the mock was constructed in a piecemeal fashion, and I was learning as I went. `nonReentrant` is used to prevent reentry attacks by malicious actors. But constructing the most secure or even realistic contract is not the focus of this project, so it did not seem important to me to go back and include `nonReentrant`.
- As mentioned in the introduction both the `VRFMachinery.py` script and `Oracle.sol` contract can be considered to be representative of Chainlink. `Oracle.py` simply gives us a contract to payments to while `VRFMachinery.py`, as the name implies, is intended to mock the proprietary(?) software which Chainlink uses to detect and respond to requests. To my knowledge we cannot know what this 'machinery' *actually* looks like. So far as I can tell, a simple python script seems to suffice for the purpose of mocking this machinery.
- Chainlink contracts are imported using the command `import "node_modules/@chainlink/contracts/src/..."` which is in contrast to the more common method of importing these contracts with `import "@chainlink/contracts/src/.` I am unsure why I ran into errors when attempting to import these contracts without including the `node_modules` statement at the beginning. If the user has issues with this, try both methods.
- `onTokenTransfer` is not utilized. Though I played around with it, it seems to have no purpose in the subscription method. I believe it is only used in the direct funding method of making requests.
- The user might wonder what the purpose of the `LinkCoordinator.sol` contract is. This has to do with the previous point; in the `onTokenTranser` function defined in the real `VRFCoordinator.sol` contract there is a revert statement,

```
    if (msg.sender != address(LINK)) {
      revert OnlyCallableFromLink();
    }
```

where `LINK` represents the declared instance of `LinkTokenInterface.sol`. Compare this to the equivalent statement found in the the mock Coordinator developed here,

```
      if (msg.sender != linkCoordinator) {
        revert ("only callable from LINK");
      }
```

So I declared an instance of what is essentially an entirely new contract which acts as the 'coordinator' for the Link contract. Its purpose is to execute the `onTokenTransfer` function.

But why not just have the mock `LinkToken.sol` contract do this as is intended in the `onTokenTransfer` function?

What I ran into was this; if we want the (declared instance of) `LINK` contract to act as the sender of `onTokenTransfer` we have to include this function in whatever contract which the declared instance of `LINK` represents. However, as is in line with the mock method given by Chainlink, we rely on the `src/v0.4` version of the `LinkToken.sol` which uses the `0.4.11` compiler. Though I cannot recall precisely what error messages I was receiving, when I attempted to declare an instance of the coordinator

contract in this version of LinkToken.sol I either A) got compiler errors (0.4.11 compiler did not jive with the 0.8.4 compilers used in the other contracts) or B) ran into other compatibility issues to do with src-v0.4 contracts which LinkToken.sol inherited.

I recall coming to the conclusion that I'd be forced to reach into the node_modules folder itself and modify these contracts to make things compatable with my mock – a thing I did *not* want to do else it would require users who want to deploy this mock for themselves have to go through the same steps.

There is no LinkToken.sol in the node_modules/src/v0.8 folder. So, as is in line with Chainlinks 'bare-bones' mock contract, we instead rely on the the v0.4 version which inherits ERC677Token.sol. But in the v0.4 version of ERC677Token.sol there is no onTokenTransfer function. now compare this to ERC677ReceiverInterface.sol in the v0.8 folder – there the onTokenTransfer is but, as previously stated, there is no LinkToken contract – just an interface!

What means I came up with to circumnavigate this difficulty was by implementing the LinkCoordinator.sol contract which declares an instance of the coordinator mock while inheriting the v0.8 version of ERC677ReceiverInterface.sol.

BUT as soon as I got this far I realized onTokenTransfer is only used for the direct funding method – it would do this project as it is written no good. I did however, leave the contract developed in the contracts folder as well as in the deployment file (the coordinator mock has this address in its constructor). For those interested in the mocking the direct funding method, I figure this all (the discussion if not the contract) might serve as a good start for them.

Solidity is user-friendly in that it does not care much for indentations. The effect of indentations is simply to allow the user to collapse/expand sections of code. The user will notice that the sections of the coordinator contract have been organized into sections labeled as \\[LABEL]. In the section entitled \\MISC PERSONAL FUNCS the user will find a number of my own contrived functions which are either used in the .py script or at some point or another I found useful for troubleshooting.

- launchBlockHash was previously alluded to. It is used in the deploy.py script to set the blockHashStore to the wrapper.sol contract. Should the user desire, this function can also be used to change what contract acts as the blockHashStore. Again, the fact that the wrapper was set to the blckHashStore is likely not a very realistic choice. The real VRFCoordinatorV2.sol contract requires that the blockHashStore address be input to its constructor, and in Chainlinks mock contract it is instead the wrapper which is input here, so it is not without reason that I made this choice. In either case it was for me preferable to further convoluting the code by introducing another contract whose only purpose is just to map the blockNumbers to blockHash values.

- withdrawBal is used to get the withdrawable Tokens assigned to the Oracle in the coordinator contract. This function is used in VRFMachinery.py as an indirect way of calculating the cost of/ payment required for fulfilling requests. Though fullfillRandomWords calculates the payment amount directly, it is not a view function so the user will not be able to see this return value. An alternative would be to use calculatePaymentAmount as is done in fullfillRandomWords, but this would require preparing the inputs within VRFMachinery.py, and this would only add complexity to the code[6]. Instead, withdrawBal is used to ascertain the amount of withdrawable tokens available to the oracle before and

[6] truth be told I was just too tired/ out of time to worry about this.

after fulfillment. Subtracting these before and after values allows for an indirect means of calculating the payment cost which the oracle then withdraws from the coordinator contract as payment for its service.

References

[1] Certicom Corp., *Standards for Efficient Cryptography. SEC 1: Elliptic Curve Cryptography.* Certicom Research, 2009. Available https://www.secg.org/sec1-v2.pdf

[2] Chainlink, *Local testing using a Mock contract.* Chainlink, available https://docs.chain.link/vrf/v2/subscription/examples/test-locally

[3] Patrick Collins, *Solidity, Blockchain, and Smart Contract Course – Beginner to Expert Python Tutorial.* available https://www.youtube.com/watch?v=M576WGiDBdQ