Project 1 – Introduction to Smart Contract Development

COMP6452 Software Architecture for Blockchain Applications

1 Learning Outcomes

In this project, you will learn how to write a smart contract using Solidity and deploy it on the Ethereum blockchain. After completing the project, you will be able to:

- develop a simple smart contract using Solidity
- create and fund your account on the Ethereum Testnet
- deploy your contract to the Ethereum Testnet
- test your smart contract by issuing transactions
- programmatically interact with your smart contract

2 Introduction

Smart contracts are user-defined code deployed on and executed by nodes in a blockchain. In addition to executing a set of instructions, smart contracts can hold, manage, and transfer digitalised assets. They are deployed to a blockchain as transaction data. Execution of a smart contract function is triggered using a transaction issued by a user (or a system acting on behalf of the user) or another smart contract which in turn triggered by a user-issued transaction. Inputs to a smart contract function are provided through transactions and the current state of the blockchain. Due to the immutability, consistency, integrity, and transparency properties of blockchains, smart contract code is immutable and deterministic, as well its execution is trustworthy. While the "code is law" is synonymous with smart contracts, smart contracts are neither smart nor have legal binding as per the contract law.

While Bitcoin [1] supports an elementary form of smart contracts, it was Ethereum [2] that demonstrated the true power of smart contracts by developing a *Turing complete* language and a run-time environment to code and execute smart contracts. Smart contracts in Ethereum are deployed and executed as *bytecode*, i.e., binary code results from compiling code written in a high-level language. Bytecode runs on the Ethereum Virtual Machine (EVM) on each blockchain node [3]. Solidity [4] is the most popular smart contract language for Ethereum. As Solidity is ultimately compiled into Ethereum bytecode, it can also be used in other blockchain platforms that support the EVM such as Hyperledger Besu. Solidity is a high-level, object-oriented language that is syntactically is similar to JavaScript. It is statically typed and supports inheritance, libraries, and user-defined types.

Figure 1 shows the typical development cycle of a smart contract. Like any program, it starts with the requirement analysis and modelling. State diagrams, Unified Modeling Language (UML), and Business Process Model and Notation (BPMN) are typically used to model smart contracts. The smart contract code is then developed using a suitable tool ranging from Notepad to sophisticated IDEs. Various libraries and Software Development Kits (SDKs) may be used in the process to minimise potential errors and enhance productivity. Depending on the smart contract language, code may also need to be compiled, e.g., Solidity. As the smart contract code and the result of a transaction are immutable and transparent, the code must be free of bugs. Because transactions trigger smart contracts, we need to pay fees to execute smart contracts on a public blockchain. In Ethereum this fee is referred to as *gas*. Amount of gas needs to execute a smart contract depends on several factors such as computational and memory complexity of the code, volume of data it handles, and bandwidth requirements. Therefore, extensive testing and optimisation of a smart contract are essential to keep the cost low. The extent that you can test (e.g., unit testing), debug, and optimize your code depends on the chosen smart contract language and available tools. While Ethereum has a rich set of tools, in this lab, we will explore only a small subset of them. Most public blockchains also host a test/development network, referred to as the *testnet*, that is identical in functionality to the production network. Further, they usually provide fast finality and do not charge real transaction fees. It is highly recommended to test a smart contract on a testnet. Testnet can also be used to estimate transaction fees you may need to pay in the production network. Once you are confident that the code is ready to go to the production/public blockchain network, the next step is to deploy the code using a transaction. Once the code is successfully deployed, you will get an address (aka., identifier or handler) for future interactions with the smart contract. Finally, you can interact with the smart contract by issuing transactions with the smart contract address as the recipient. The smart contract will continue to remain active until its creator disables it or reaches a terminating state. Due to the immutability of blockchains, smart contract code will remain in the blockchain even though it is deactivated and cannot be executed.

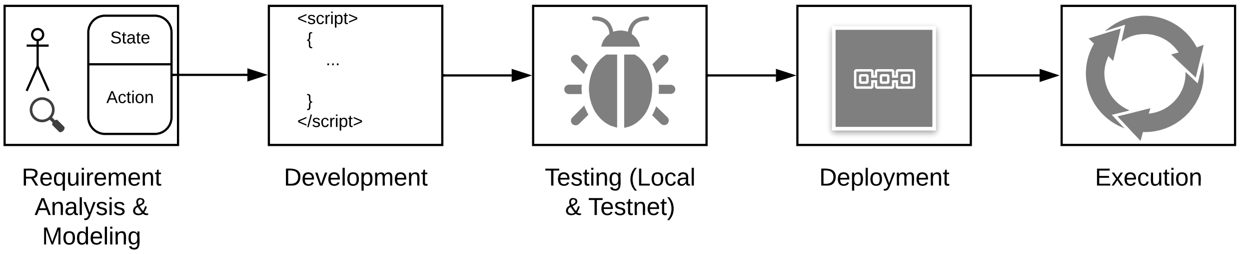

Figure 1: Smart contract development cycle.

The project has two parts. In part one (Section 3 to 8), you will develop, test, and deploy a given smart contract to the Ethereum testnet by following a set of steps. In part two (Section 9), you will update the smart contract to fix some of its functional weaknesses, and then deploy it on to the testnet.

3 Developing a Smart Contract

In this project, we will write a smart contract and deploy it to the public Ethereum Ropsten testnet. The motivation of our Decentralized Application (DApp) is to solve a million-Dollar question: *Where to have lunch?*

Basic requirements for our DApp are as follows:

1. Only the contract creator can create the list of venues v
2. The contract creator also sets the list of friends f to vote for v
3. The contract stop accepting votes when a quorum is met (e.g., number of votes $> f/2$) and declares the lunch venue

Following code shows a smart contract written in Solidity to decide the lunch venue based on votes. The first line indicates that the source code is released as non-open-source code. Third line tells that the code is written for Solidity and should not be used with a compiler earlier than version 0.8.0. Further, the $\wedge$ symbol says that the code is not designed to work on future compiler versions, e.g., 0.9.0 or higher. It could work on any version labelled as `0.8.xx`. These constraints are indicated using the `pragma` keyword, which is an instruction for compilers. As Solidity is rapidly evolving and smart contracts are immutable, it is desirable to indicate even a specific version such that all participants of a contract have a clear understanding of the behaviour of the smart contract. Between lines 8 and 16, we define two structures to keep track of the list of friends and votes. We keep track of individual votes to avoid non-repudiation. *address* is a special data type in Solidity that refers to a 160-bit address/account in Ethereum. In lines 18-19 and 26-27, we define several hash maps (aka., maps or hash tables) to keep track of the list of venues, friends, votes, and results. Compared to some of the other languages, Solidity cannot tell us how many keys are in a hash map or cannot directly iterate in a map. Thus, the number of entries are tracked separately (lines 20-22). Also, a hash map cannot be defined dynamically. `manager` is used to keep track of the creator of the smart contract. Selected lunch venue and voting state are stored in variables `votedVenue` and `voteOpen`, respectively. Also, note the permissions of these variables. The compiler will automatically generate getter functions for public variables.

In the constructor, we set the transaction/message sender (`msg.sender`) that deployed the smart contract as the `manager` of the contract. `addVenue` and `addFriend` functions are used to create a list of lunch venues and friends that can vote for a venue. These functions also return the number of lunch venues and friends added to the blockchain. The `memory` keyword is used to hold temporary variables. EVM provides two other areas to store data referred to as `storage` and `stack`. For example, all the variables between lines 18 and 28 are in the storage. `restricted` is a *function modifier*, which is used to create additional features or apply restrictions on a function. For example, `restricted` function (lines 106-109) indicates that only the `manager` can invoke this function. Therefore, *function modifier* can be used to enforce access control. If the condition is satisfied, the function body is placed on the line beneath `_;`. Similarly, `votingOpen` (lines 112-115) is used to enforce that votes are accepted only when `voteOpen` is `true`. `doVote` function is used for voting, as far as voting state is open and both the friend and venue are valid (lines 65-66). It further returns a Boolean value to indicates whether voting was successful. In lines 77, after the submission of each vote, we check whether the quorum is reached. If so, *finalResults* function is called to chose the most voted lunch venue. This function uses a hash map to track the vote count for each venue, and one with the highest number of votes is declared as the chosen lunch venue. Voting is also marked as no longer open by setting `voteOpen` to `false` (line 102).

```
1  // SPDX-License-Identifier: UNLICENSED
2
3  pragma solidity ^0.8.0;
4
5  /// @title Contract to agree on the lunch venue
6  contract LunchVenue{
7
8      struct Friend {
9          string name;
10         bool voted;
11     }
12
13     struct Vote {
14         address voterAddress;
15         uint venue;
16     }
17
18     mapping (uint => string) public venues;       //List of venues (venue no, name)
19     mapping(address => Friend) public friends;    //List of friends (address, Friend)
20     uint public numVenues = 0;
21     uint public numFriends = 0;
22     uint public numVotes = 0;
23     address public manager;                       //Manager of lunch venues
24     string public votedVenue = "";                //Where to have lunch
25
26     mapping (uint => Vote) private votes;         //List of votes (vote no, Vote)
27     mapping (uint => uint) private results;       //List of vote counts (venue no, no of
           votes)
28     bool voteOpen = true;                         //voting is open
29
30     //Creates a new lunch venue contract
31     constructor () {
32         manager = msg.sender;                     //Set contract creator as manager
33     }
34
35     /// @notice Add a new lunch venue
36     /// @dev To simplify the code duplication of venues is not checked
37     /// @param name Name of the venue
38     /// @return Number of lunch venues added so far
39     function addVenue(string memory name) public restricted returns (uint){
40         numVenues++;
41         venues[numVenues] = name;
42         return numVenues;
43     }
44
45     /// @notice Add a new friend who can vote on lunch venue
46     /// @dev To simplify the code duplication of friends is not checked
47     /// @param friendAddress Friend's account address
48     /// @param name Friend's name
49     /// @return Number of friends added so far
50     function addFriend(address friendAddress, string memory name) public restricted
           returns (uint){
51         Friend memory f;
52         f.name = name;
53         f.voted = false;
54         friends[friendAddress] = f;
```

```
        numFriends++;
        return numFriends;
    }

    /// @notice Vote for a lunch venue
    /// @dev To simplify the code multiple votes by a friend is not checked
    /// @param venue Venue number being voted
    /// @return validVote Is the vote valid? A valid vote should be from a registered friend and to a registered venue
    function doVote(uint venue) public votingOpen returns (bool validVote){
        validVote = false;                                //Is the vote valid?
        if (bytes(friends[msg.sender].name).length != 0) {  //Does friend exist?
            if (bytes(venues[venue]).length != 0) {         //Does venue exist?
                validVote = true;
                friends[msg.sender].voted = true;
                Vote memory v;
                v.voterAddress = msg.sender;
                v.venue = venue;
                numVotes++;
                votes[numVotes] = v;
            }
        }

        if (numVotes >= numFriends/2 + 1) { //Quorum is met
            finalResult();
        }
        return validVote;
    }

    /// @notice Determine winner venue
    /// @dev If top 2 venues have the same no of votes, final result depends on vote order
    function finalResult() private{
        uint highestVotes = 0;
        uint highestVenue = 0;

        for (uint i = 1; i <= numVotes; i++){   //For each vote
            uint voteCount = 1;
            if(results[votes[i].venue] > 0) { // Already start counting
                voteCount += results[votes[i].venue];
            }
            results[votes[i].venue] = voteCount;

            if (voteCount > highestVotes){ // New winner
                highestVotes = voteCount;
                highestVenue = votes[i].venue;
            }
        }
        votedVenue = venues[highestVenue];  //Chosen lunch venue
        voteOpen = false;                   //Voting is now closed
    }

    /// @notice Only manager can do
    modifier restricted() {
        require (msg.sender == manager, "Can only be executed by the manager");
        _;
    }

    /// @notice Only whenb voting is still open
    modifier votingOpen() {
        require(voteOpen == true, "Can vote only while voting is open.");
        _;
    }
}
```

Let us now create and compile this smart contract. For this, we will use Remix IDE, an online IDE for developing, testing, deploying, and administering smart contracts for Ethereum-like blockchains. Due to zero setup and simple user interface, it is a good learning platform for smart contract development.

Step 1. Using your favourite web browser, go to `https://remix.ethereum.org/`.

Step 2. Set the environment as `Solidity` and then select `New File` link. Enter `LunchVenue.sol` as the file name and click the `Ok` button. Remix stores smart contracts in `contracts` folder.

Step 3. Type the above smart contract in the editor. Better not cut and paste the above code from PDF, as it may introduce hidden characters preventing the contract from compiling.

Step 4. As seen in Figure 2, set the compiler options are as follows, which can be found under `Solidity compiler` menu option on the left:

- Compiler – `0.8.0+....` (Any commit option should work)
- Language – `Solidity`
- EVM Version – `compiler default`
- Compiler Configuration – Make sure `Hide warnings` is not ticked. Other two are optional

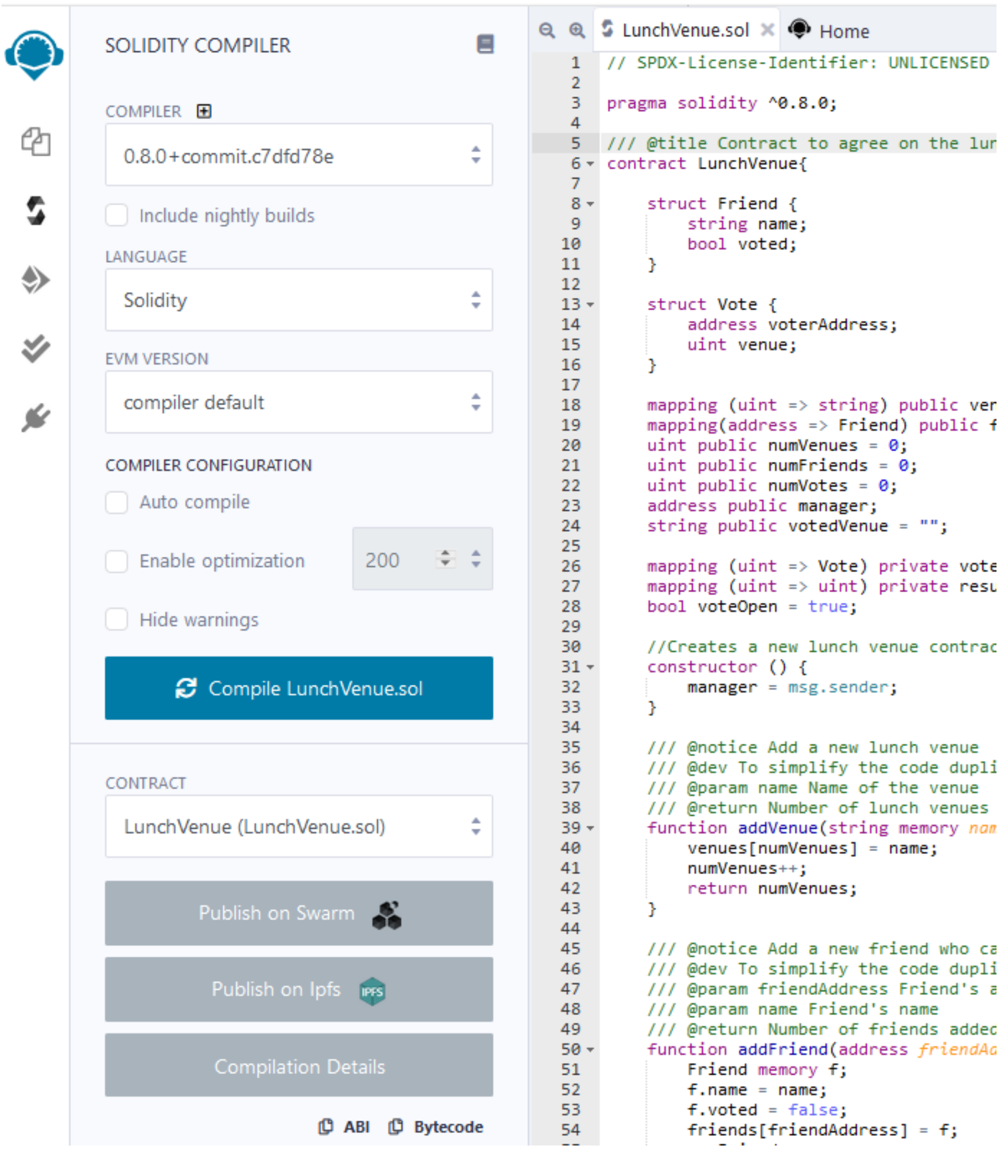

Figure 2: Compiler options.

Step 5. Then click on the `Compile LunchVenue.sol` button. Carefully fix any errors and warnings. While Remix stores your code on the browser storage, it is a good idea to link it to your Github account. You may also save a local copy.

Step 6. Once compiled, you can access the binary code by clicking on the `Bytecode` link, which will copy it to the clipboard. Paste it to any text editor to see the binary code and EVM instructions (opcode). Similarly, you can check the Application Binary Interface (ABI) of the smart contract using the `ABI` link. ABI is the interface specification to interact with a contract in the Ethereum ecosystem. Data is encoded as a schema that describes the set of functions, their parameters, and data formats. Also click on `Compilation Details` button for see more details about the contract and its functions.

4 Unit Testing

Next, we test our smart contract using unit testing features of Remix.

Step 7. Click on the Solidity unit testing option on the left (look for the icon with 2 checks).

If the icon is not visible, you have to activate it from Remix plugin manager. Click on the icon, and the load up the Solidity Unit Testing plugin by clicking on the `Activate` button.

Step 8. This will show up the unit testing dialogue box similar to Figure 3. Make sure `tests` is set as the `Test directory:`. Then click on the `Generate` button to generate a new sample Solidity test file. Usually, the name of the test file reflects our contract name and has a suffix of *_test*. This file contains the minimum code to run unit testing. Click on the `How to use...` button, and read through the *Unit Testing Plugin* web page and the next 2 web pages to get an idea about how to perform a unit test with Remix.

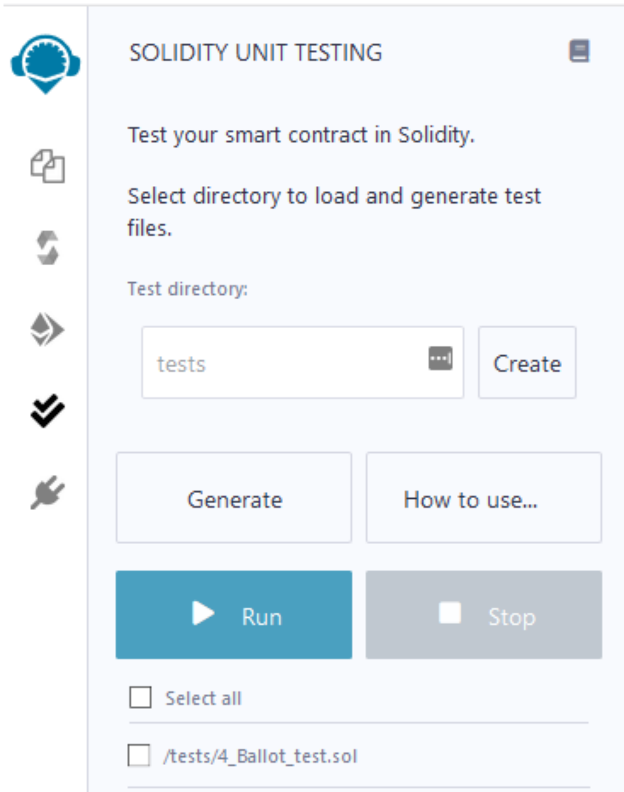

Figure 3: Unit test options.

Step 9. Update the `LunchVenue_test.sol` file in `tests` folder to include the following test code.

In line 5, we include the `remix_accounts.sol` file. This gives us access to a set of test accounts to emulate the behaviour of Ethereum accounts. In line 10, we inherit our `LunchVenue` contract to test its functionality. This is required as we want to emulate user behaviour and did not explicitly create getter functions to access public variables in our contract. Between lines 13-17 we create a set of variables for user accounts. Their values are assigned to test accounts between lines 22-26. `beforeAll` function runs before all the tests; hence, it can be used to set states needed for testing. We use `acc0` as the manager of our contract. `acc0` to `acc3` are used as friends who could vote for a lunch venue. As per `remix_accounts.sol`, account at zero index (i.e., `account-0`) is the default account; hence, our manager will be set to the default account. It is important to note that `account-0` is a label that can be used to set the transaction context, whereas `acc0` is the respective variable.

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity >=0.8.00 <0.9.0;
import "remix_tests.sol"; // this import is automatically injected by Remix.
import "remix_accounts.sol";
import "../contracts/LunchVenue.sol";

// File name has to end with '_test.sol', this file can contain more than one testSuite contracts
/// Inherit 'LunchVenue' contract
contract LunchVenueTest is LunchVenue {

    // Variables used to emulate different accounts
    address acc0;
    address acc1;
    address acc2;
    address acc3;
    address acc4;

    /// 'beforeAll' runs before all other tests
    /// More special functions are: 'beforeEach', 'beforeAll', 'afterEach' & 'afterAll'
    function beforeAll() public {
        acc0 = TestsAccounts.getAccount(0);     // Initiate account variables
        acc1 = TestsAccounts.getAccount(1);
        acc2 = TestsAccounts.getAccount(2);
        acc3 = TestsAccounts.getAccount(3);
        acc4 = TestsAccounts.getAccount(4);
    }

    /// Account at zero index (account-0) is default account, so manager will be set to acc0
    function managerTest() public {
        Assert.equal(manager, acc0, 'Manager should be acc0');
    }

    /// Add lunch venue as manager
    /// When msg.sender isn't specified, default account (i.e., account-0) is considered as the sender
    function setLunchVenue() public {
        Assert.equal(addVenue('Courtyard Cafe'), 1, 'Should be equal to 1');
        Assert.equal(addVenue('Uni Cafe'), 2, 'Should be equal to 2');
    }

    /// Try to add lunch venue as a user other than manager. This should fail
    /// #sender: account-1
    function setLunchVenueFailure() public {
        try this.addVenue('Atomic Cafe') returns (uint v) {
            Assert.ok(false, 'Method execution should fail');
        } catch Error(string memory reason) {
            // Compare failure reason, check if it is as expected
            Assert.equal(reason, 'Can only be executed by the manager', 'Failed with unexpected reason');
        } catch (bytes memory /*lowLevelData*/) {
            Assert.ok(false, 'Failed unexpected');
        }
    }

    /// Set friends as account-0
    /// #sender doesn't need to be specified explicitly for account-0
    function setFriend() public {
       Assert.equal(addFriend(acc0, 'Alice'), 1, 'Should be equal to 1');
       Assert.equal(addFriend(acc1, 'Bob'), 2, 'Should be equal to 2');
       Assert.equal(addFriend(acc2, 'Charlie'), 3, 'Should be equal to 3');
       Assert.equal(addFriend(acc3, 'Eve'), 4, 'Should be equal to 4');
    }

    /// Try adding friend as a user other than manager. This should fail
    /// #sender: account-2
```

```
65      function setFriendFailure() public {
66          try this.addFriend(acc4, 'Daniels') returns (uint f) {
67              Assert.ok(false, 'Method execution should fail');
68          } catch Error(string memory reason) {
69              // Compare failure reason, check if it is as expected
70              Assert.equal(reason, 'Can only be executed by the manager', 'Failed with
                    unexpected reason');
71          } catch (bytes memory /*lowLevelData*/) {
72              Assert.ok(false, 'Failed unexpected');
73          }
74      }
75
76      /// Vote as Bob (acc1)
77      /// #sender: account-1
78      function vote() public {
79          Assert.ok(doVote(2), "Voting result should be true");
80      }
81
82      /// Vote as Charlie
83      /// #sender: account-2
84      function vote2() public {
85          Assert.ok(doVote(1), "Voting result should be true");
86      }
87
88    /// Try voting as a user not in the friends list. This should fail
89      /// #sender: account-4
90      function voteFailure() public {
91          Assert.equal(doVote(1), false, "Voting result should be false");
92      }
93
94      /// Vote as Eve
95      /// #sender: account-3
96      function vote3() public {
97          Assert.ok(doVote(2), "Voting result should be true");
98      }
99
100     /// Verify lunch venue is set correctly
101     function lunchVenueTest() public {
102         Assert.equal(votedVenue, 'Uni Cafe', 'Selected venue should be Uni Cafe');
103     }
104
105     /// Verify voting is now closed
106     function voteOpenTest() public {
107         Assert.equal(voteOpen, false, 'Voting should be closed');
108     }
109
110     /// Verify voting after vote closed. This should fail
111     /// #sender: account-2
112     function voteAfterClosedFailure() public {
113         try this.doVote(1) returns (bool validVote) {
114             Assert.ok(false, 'Method Execution Should Fail');
115         } catch Error(string memory reason) {
116             // Compare failure reason, check if it is as expected
117             Assert.equal(reason, 'Can vote only while voting is open.', 'Failed with
                    unexpected reason');
118         } catch (bytes memory /*lowLevelData*/) {
119             Assert.ok(false, 'Failed unexpectedly');
120         }
121     }
122 }
```

`managerTest` test case (lines 30-32) validates whether the default account is set as the manager of the contract. `setLunchVenue` test case adds 2 lunch venues where we expect the smart contract to return the number of available venues for each addition. In `setLunchVenueFailure` test case (lines 43-52), we try to add another lunch venue. `#sender : account-1` in line 42 indicates that we call the function while setting `account-1` as the `msg.sender` of the transaction used to invoke the function. This test case should fail, as only the manager is allowed to add a lunch venue. However, to prevent our unit test from failing, we leverage `try-catch` keywords. Similarly, `setFriend` and `setFriendFailure` test cases try to add list of friends that could vote for a venue. In `vote` and `vote2` test cases, we vote for a venue as Bob and Charlie, respectively. Because `account-4` is not in the friends list, in `voteFailure` test case (lines 90-92) `doVote` should return `false`. Next, we vote as Eve. As the minimum number of votes is received, the smart contract should select *Uni Cafe* as the highest voted venue and disable further voting. `lunchVenueTest` and `voteOpenTest` test cases verify this behaviour. Finally, between lines 112 and 121, we make sure no more votes can be cast once the venue is decided.

Step 10. To run our test cases, click on the `Run` button as seen on Figure 3. You should see an output similar to Figure 4. All other tests should be successful.

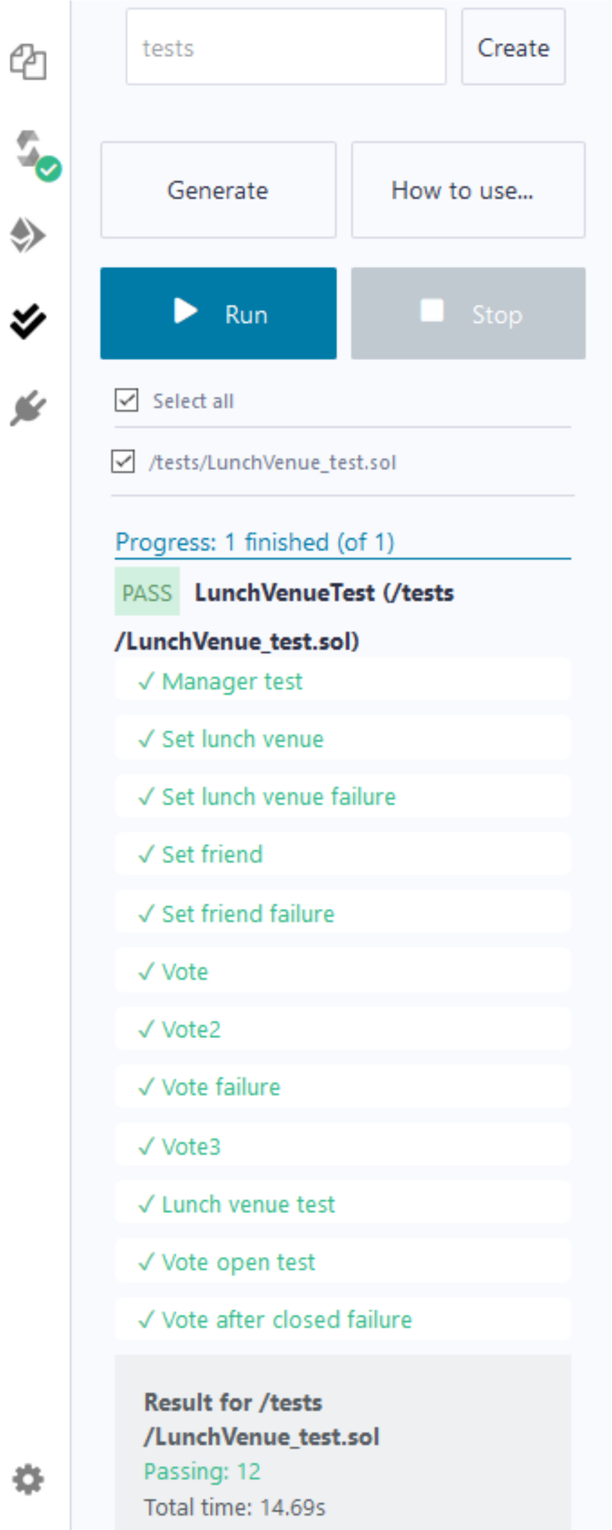

Figure 4: Unit test results.

Next, we need to deploy our smart contract to the Ethereum Ropsten testnet. While Rinkeby and Kovan are other testnet options, we will use Ropsten, as it is the closest to the current production environment. Nevertheless, we still need to create an account and fund our account with test Ether to deploy smart contracts and issue transactions. These steps are discussed in the next section.

5 Creating and Funding an Account

We use a digital wallet to create and issue Ethereum transactions. In this project, we use MetaMask, which is a browser-based, easier to use, and secure way to connect to blockchain-based applications. Once the account is created, we will fund it using an already deployed Faucet smart contract on the Ropsten network. We also use Etherscan – an explorer or search engine for Ethereum data – to check the details of transactions.

Step 11. Visit `https://metamask.io/` and install the browser extension. For this lab, it will be easier to work with the browser extension than the mobile app.

Step 12. Once installed, click `Get Started` button. If this is the first time you are using MetaMask, you need to create a new Wallet. Hence, click `Create a Wallet` button. Enter a new password, accept the policy, and click `Create`. This will generate a 12-word `Secret Backup Phrase` (aka., mnemonic) to recover your wallet in case of an error. Save the mnemonic on a secure location. You are required to confirm the `Secret Backup Phrase` too. This creates a new account with associated public and private key pairs. Your 160-bit address is derived from the public key and the private key is used to sign transactions. Your address will appear as a long hexadecimal string with a prefix of `0X`. Click *Copy to clipboard* to see your address (you may have to move the mouse pointer to where it says `Account 1`). You can also get your address as a QR code or even update your account name and export your private key using the `Details` button. Notice that your account balance is `0 ETH`.

Step 13. Next, let us fund our account with some test Ether. Go to Ropsten Faucet webpage at `https://faucet.ropsten.be/`. Copy and paste your address from MetaMask into *Enter your testnet account address* textbox. Then click on `Send me test Ether` button. This creates a new transaction where the transaction ID appears at the bottom of the webpage (a long hexadecimal string with prefix `0x`).

MataMask also provides a faucet contract which can be accessed from `https://faucet.metamask.io/`. You may alternatively use MataMask faucet.

Step 14. Click on the transaction ID link which will take you to `https://etherscan.io`. Here you can see details of the transaction such as Transaction Hash, Status, From, To, and Value (see Fig. 5). For a few seconds, transaction `Status` may appear as `Pending`. Once the transaction is included in a block, `Status` changes to `Success` and additional details such as Block, Timestamp, and Transaction Fee will appear. Use the `Click to see More` link see further details of the transaction.

Step 15. Go back to MetaMask. Select `Ropsten Test Network` from the dropdown list on the top of MetaMask. You will now see 1 ETH as your balance.

Now that we have a sufficient amount of Ether to deploy and issues transactions to our smart contract, go back to the Remix IDE.

6 Deploying the Smart Contract

First, we will try to deploy the smart contract on Remix JavaScript VM to ensure that our contract can be deployed without much of a problem. This will also give us an idea about the transaction fees. Ethereum defines the transaction fee as follows:

$$transaction\,fee = gas\,limit \times gas\,price \tag{1}$$

`Gas limit` defines the maximum amount of gas we are willing to pay to deploy or execute our smart contract. This should be determined based on the computational and memory complexity of the code, volume of data it handles, and bandwidth requirements. If the set `gas limit` is too low, the smart contract could terminate abruptly as it runs out of gas. Whereas if it is too high, errors such as infinite loops could consume all our Ether. Hence, it is a good practice to estimate the `gas limit` and set a bit

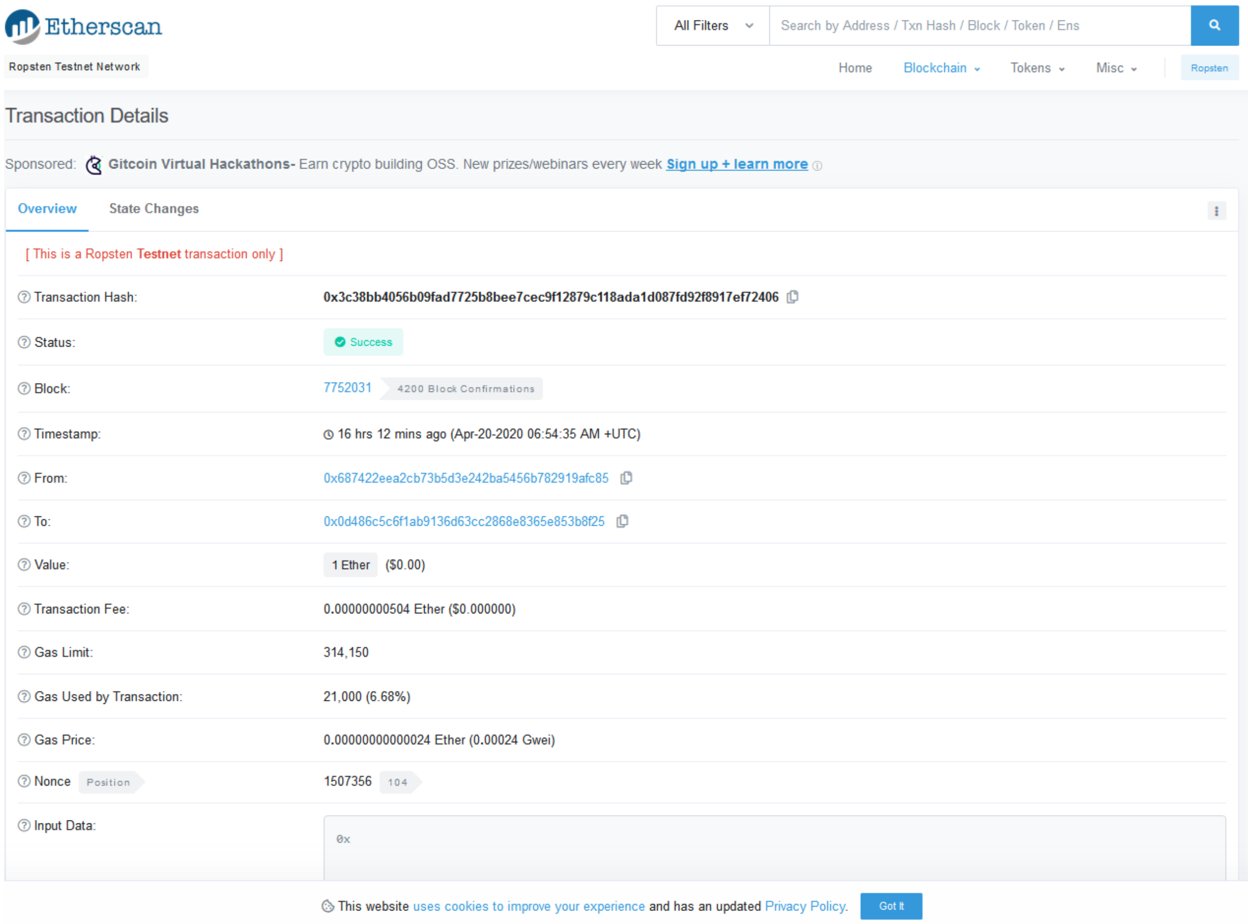

Figure 5: Transaction details.

higher value to accommodate any changes during the execution (it is difficult to estimate exact gas limit as cost of execution depends on the state on the blockchain). The *gas price* determines how much we are willing to pay for a unit of gas. When a relatively higher *gas price* is offered, the time taken to include the transaction in a block typically reduces. Most blockchain explorers such as Etherscan provide statistics on market demand for *gas price*. It is essential to consider such statistics when using the Ethereum production network to achieve a good balance between transaction latency and cost. MetaMask can do this for you.

Step 16. Select `Deploy & run transactions` menu option on the left. Then set the options as follows (see Figure 6):

- Environment – `JavaScript VM`
- Account – Pick one of the accounts with some Ether
- Gas Limit – `3000000` (use the default)
- Value – `0` (we are not transferring any Ether to the smart contract)
- Contract – `LunchVenue`

Step 17. Then click on the `Deploy` button to deploy the smart contract. You can see the transaction details and other status information, including any errors at the bottom of Remix. Especially note values such as `status`, `contract address`, `transaction cost`, and `execution cost`. In the next section, we interact with our contract.

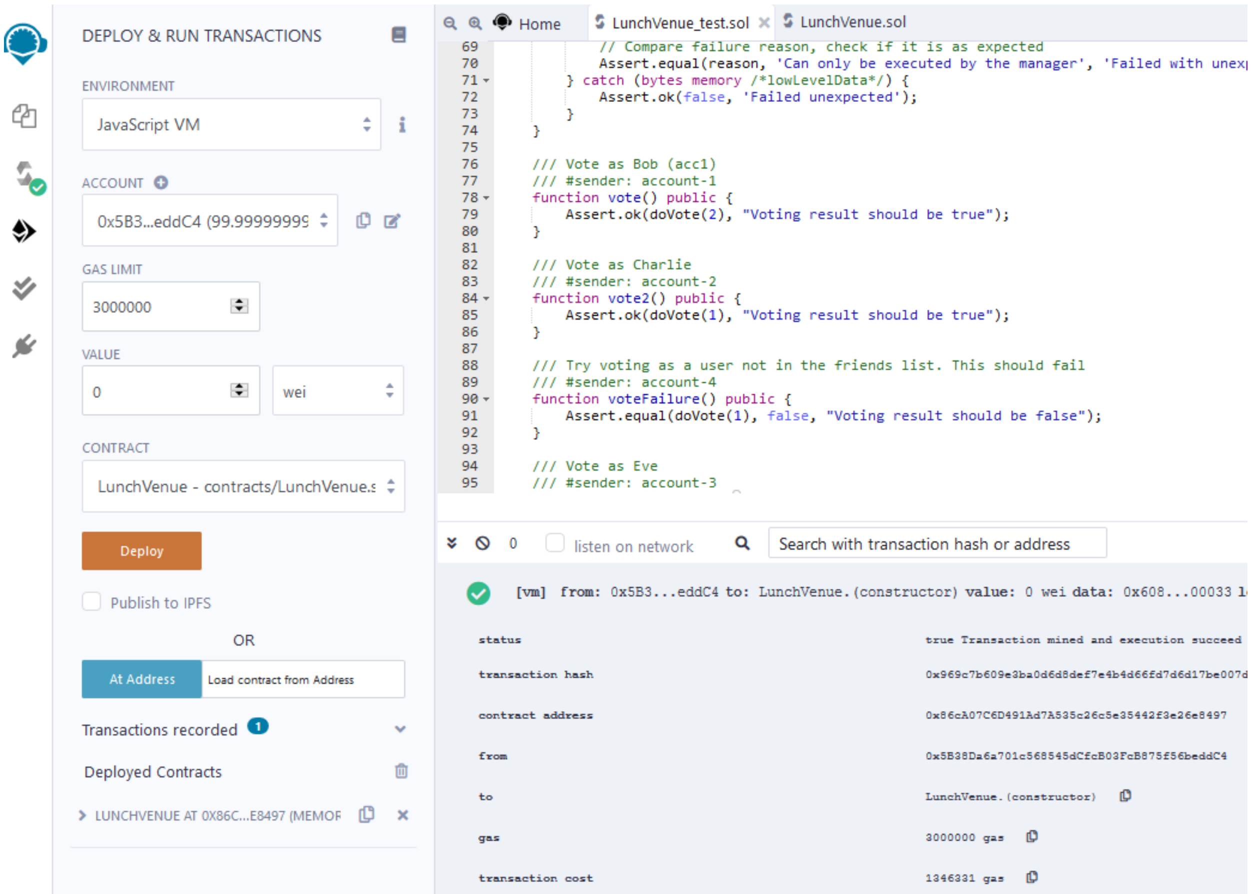

Figure 6: Deployment settings.

7 Manually Issuing Transactions

Step 18. As seen in Figure 7, we can interact with the deployed contract using the functions that appear under `Deployed Contracts`. For example, by clicking on the `manager` button, we can see the address of the manager is already set to the address of the account we used to deploy the smart contract.

To add yourself as a voter/friend, fill the `friendAddress:` textbox with your address and *name:* with your name. Then click the `addFriend` button. Getter function provided by `numFriends` button can be used to verify that a friend is added to the contract. We can also recall details of a friend by providing the address to the `friends` getter function. Similarly, check the `addVenue` function. Further, test the contract before deploying into the Ropsten testnet using the next step. Also, note the differences in gas consumption by different functions.

Step 19. Change the `Environment` to `Injected Web3`. The first time, this will popup MetaMask to connect with Remix. Make sure your address is set as the `Account`. MetaMask will popup again asking you to confirm the transaction to deploy the contract. You will see that MetaMask has already set a transaction fee. In case it is set to 0, change the value by clicking on `Edit` link. This will list some suggested gas prices based on the expected time to include a transaction in a block. Then click on the *Confirm* button.

Step 20. On the Remix console (usually at the bottom of Remix UI), you can find a link to Etherscan with a transaction ID. Click on the link and wait till the transaction is included in a block. If `Status` is marked as `Success`, your smart contract is successfully added to the test network. Carefully go through the transaction parameters. Note down the `To` address, which is the address of our contract. If you lose it, it could be impossible to access the contract again. If the `Success` is marked as `Failed`, check the error messages on Remix console. Do the needful to fix the error and attempt to redeploy the contract.

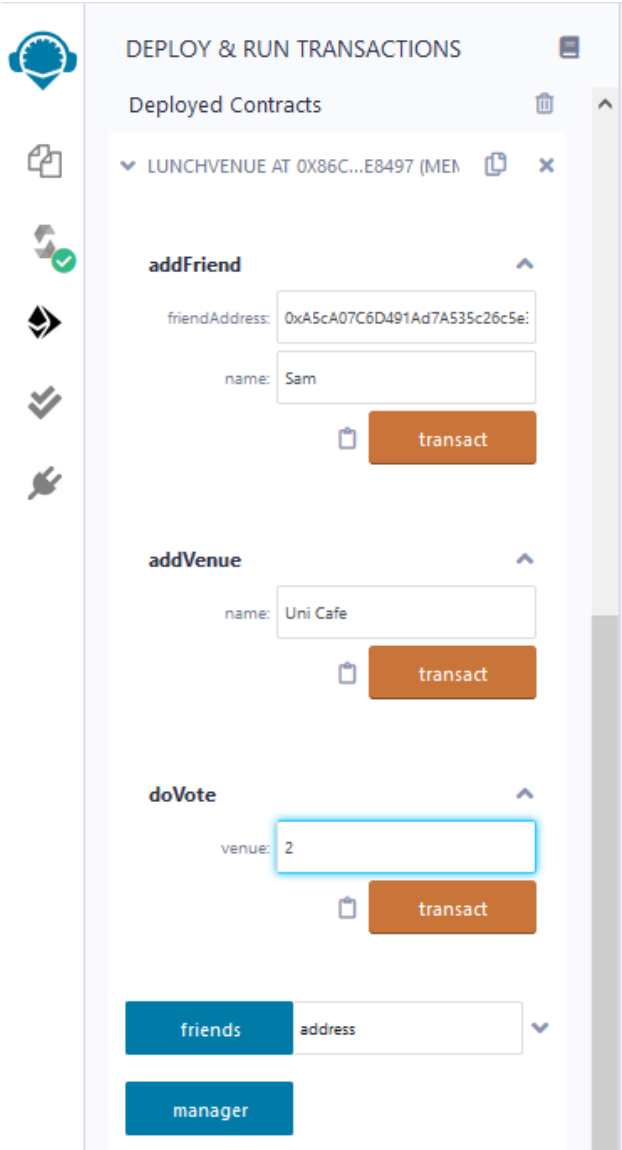

Figure 7: Interacting with the contract.

Step 21. Let us now interact with the smart contract on Ropsten testnet and validate its full functionality. Enter the contract address to the textbox near the `At Address` button. Once you click the button, similar to Figure 7, Remix will populate the UI with a list of buttons and textboxes to interact with our contract.

Step 22. Populate relevant textboxes with the following values (addition of each of these values will trigger a transaction which you need to submit via MetaMask) and submit a separate transaction for each value:

- Venues – Add at least four cafes at/near UNSW
- Friends – Add at least addresses and names of five valid Ethereum addresses. You may use your own address, addresses of other students, and/or create multiple accounts (remember each account needs to be funded before issuing transactions).

Also, verify that the `numVenues` and `numFriends` reflect the number of entries you added. Moreover, `numVotes` should be zero and `votedVenue` should be empty. Further verify that you get the same data as you entered using `venues` and `friends` getter functions.

Step 23. Using a subset of the friends' accounts, vote for a venue (only a user who holds a private key can issue a transaction to call `doVote` function). Once the quorum is reached, the contract will select the more preferred lunch venue. The selected venue can be found by calling the `votedVenue` getter function. Try to issue another vote and see what happens to the transaction. Congratulations, you have deployed and tested your first smart contract.

8 Programmatically Issuing Transactions

Rather than using Remix and MetaMask to interact with our smart contract, a custom program can be developed. While you could connect to the Ethereum blockchain using several programming abstractions, Ethereum Web3.js [5] library is the most popular option. Web3.js is a collection of JavaScript libraries that allows you to interact with a local or remote Ethereum node, using a HTTP or IPC connection. It provides an extensive set of functions to compile and deploy a smart contract; sign and issue transactions; check the state of blocks, transactions, and data; and call functions on a smart contract. Web3.js usually used within a Node.js application. While it is not essential to develop a Node.JS application to interact with our LunchVenue smart contract, it is highly recommended that you follow the tutorials at [6, 7, 8] as this could be useful in Project 2.

9 Extending the Smart Contract

While our smart contract works, it has a couple of issues. For example:

1. A friend can vote more than once. While this was handy in testing our contract, it is undesirable as one can monopolize the selected venue. Ideally, we should record a vote for a given address only once
2. While the contract is stuck at `doVote` function, other functions can still be called. Also, once the voting starts, new venues and friends can be added, making the whole process chaotic. In a typical voting process, voters are clear about who would vote and what to vote on before voting starts to prevent any disputes. Hence, a good vote process should have well-defined create, vote open, and end phases
3. If the quorum is not reached by the lunchtime, no consensus will be reached on the lunch venue. Hence, the contract needs a timeout. However, the wallclock time on a blockchain is not accurate due to clock skew. Hence, the timeout needs to be defined as a block number
4. There is no way to disable the contract once it is deployed. Even the manager cannot do anything to stop the contract in case the team lunch has to be cancelled
5. Gas consumption is not optimized. More simple data structures could help to reduce the transaction cost

Step 24. Update the LunchVenue smart contract to satisfy at least four of the above-listed weakness. Save it as a separate `.sol` file. Do not modify function definitions unless essential. Also, clearly mention which weakness you address and how do you address them. These could be added as comments to your code. You may need to look into more Solidity functions to resolve some of the issues.

Step 25. Create a new unit test file for the updated contract. In addition to test cases, we needed to test the previous contract, add at least 4 other test cases to validate the new functionality. Proceed with Step 22 for updated contract too. That way, we can check your work (code and transactions), in addition to us deploying a new instance of your contract.

10 Project Submission

You are required to submit the following as a single .zip or .tar.gz file:

Deliverable	**Points (15 in total)**
Source code of original LunchVenue.sol	1
Source code of original LunchVenue_test.sol	2
Source code of updated .sol file to fix at least 4 weaknesses	6
Source code of updated _test.sol test file	4
2 addresses of above smart contracts deployed on Ropsten as addresses.txt	2

These need to be submitted by the deadline given on the course Moodle page.

- 0.5 marks will be given to those who submit by the deadline
- 2 marks will be deducted per day for a submission after the deadline
- **Plagiarism checker will be used to analyze the submitted code and answer for open question (changing the name of state variables will not help). The UNSW has an ongoing commitment to fostering a culture of learning informed by academic integrity. All the UNSW staff and students have a responsibility to adhere to this principle of academic integrity. Plagiarism undermines academic integrity and is not tolerated at the UNSW.**

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[2] V. Buterin. (2014) A next-generation smart contract and decentralized application platform. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[3] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger: Byzantium version," *Ethereum project yellow paper*, Mar. 2019. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[4] Ethereum. (2021) Solidity. [Online]. Available: https://solidity.readthedocs.io/

[5] ——. (2021) web3.js – ethereum javascript api. [Online]. Available: https://web3js.readthedocs.io/

[6] G. McCubbin. (2020, Apr.) Intro to web3.js · ethereum blockchain developer crash course. [Online]. Available: https://www.dappuniversity.com/articles/web3-js-intro

[7] G. Simon. (2017, Oct.) Interacting with a smart contract through web3.js (tutorial). [Online]. Available: https://coursetro.com/posts/code/99/Interacting-with-a-Smart-Contract-through-Web3.js-(Tutorial)

[8] A. M. Antonopoulos and G. Wood, *Mastering Ethereum*. O'Reilly Media, 2018. [Online]. Available: https://github.com/ethereumbook/ethereumbook