# Final Report

# Final Year Project 2017-2018, Department of Computer Science, The University of Hong Kong

Topic: Blockchain research and implementation (fyp17043)

Name: NG King Pui

UID: 3035178820

Date of Submission: 4/2018

**Abstract**

Blockchain is a hot topic in the technology world. Bitcoin is one of the typical examples of applications of blockchain. The principle behind blockchain is to chain up blocks of data and let everyone have a copy of the chain, a.k.a. distributed ledger system. The blocks of data are chained up with hash, which is mathematically-proven secure. Once a single block changes a bit, the whole chain looks different. When a new block is going to be appended, everyone has to validate the chain by comparing with their own copy. Unauthorized data entries are then revoked by the crowd. Also, the data are transparent to everyone as they all have a copy. With these two features, blockchain breaks the gap of trust, which allows blockchain to play an important role in many different areas, especially monetary transaction. Besides Bitcoin, smart contracts are another application of blockchain. Smart contracts will execute automatically once the condition set is met, preventing breach of contracts. However, there are still loopholes in smart contracts. It may result in millions of monetary losses. In this project, the security holes of smart contracts on Ethereum will be studied and a tool of checking smart contracts and detecting such holes is expected to be a final deliverable.

**Table of Content**


1.Background 5

2.Methodology 7

- 2.1 Ethereum 7
- 2.2 Solidity 9
- 2.3 ganache-cli 9
- 2.4 Web3.js 10
- 2.5 solc 11
- 2.5 Summary of checking tool 12

3. Works Accomplished 13

- 3.1 Research on security loopholes 13
  - 3.1.1 Wrong typecast 13
  - 3.1.2 Immutable bugs 14
  - 3.1.3 Unpredictable states and Reentrancy 14
  - 3.1.4 Uint Overflow/ Underflow 15
  - 3.1.5 Unclear Visibilities of Functions 15
  - 3.1.6 Forcing Ether to a Contract 16
  - 3.1.7 DoS By Calling to Unknown 16
  - 3.1.8 Short Address Attack 17
- 3.2 Development of the checking tool 18
  - 3.2.1 Launching a private chain 18
  - 3.2.2 Launching a prototype of web tool 18
  - 3.2.3 Checking Principles of different Vulnerabilities 20
    - 3.2.3.1 Checking Uint Overflow/ Underflow 20
    - 3.2.3.2 Checking Visibilities 20
    - 3.2.3.3 Checking Forcing Ether 21
    - 3.2.3.4 Checking DoS by Calling the Unknown 21
    - 3.2.3.5 Checking Short Address Attack 22
    - 3.2.3.6 Checking Reentrancy 22

4. Problems Encountered 23

- 4.1 Insufficient security loopholes 23
- 4.2 Computing resources of the tool 23
- 4.3 Deploying the tool to the Cloud 23

5. Future Works 24

5.1 Use Machine Learning to Read and Check the Contracts 24

5.2 UI Enhancement 25

6. Limitation 26

6.1 Uncovered security loopholes 26

6.2 Cross platform checking 26

6.3 User Input and Unpredicted Behaviors of Contracts

7. Conclusion 27

8. Reference 28

9. Table of Figures 29

## 1. Background

Blockchain is the backbone of Bitcoin, which is now gaining more and more attention in the world. The principle behind blockchain is to chain up blocks of data and let everyone have a copy of the chain, a.k.a. distributed ledger system. The blocks of data are chained up with hash, which is mathematically-proven secure and used in many other security measures. Every block will have its own unique hash value, which is generated based on its own data and the hash value of the last block. That is the reason why it is called blockchain, from chaining up blocks of data with hash. Therefore, if someone try to manipulate one block of data, e.g. change his own account balance, to fake others, his own copy of the chain will be different with others in terms of has value. Others can then tell his cop is not valid hence revoking that block. With the power of the crowd, it is difficult to create fake record in blockchain. Also as everyone has a copy, every transaction record is transparent to everyone. It then prevents under-the-table deals. With these 2 features, blockchain breaks the gap of trust. It can then be utilized in many areas, especially monetary transaction.

Smart contract is one of the typical examples of application of blockchain. Built on top of blockchain, smart contracts share the feature of security and transparency. Besides, smart contracts allow users to run their own script to make the contracts self-enforcing and self-executing. With the scripts, the application of smart contracts can be much wider across different industries. Apart from that, smart contracts reduce the cost and increase the efficiency significantly when compared with traditional contracts. When creating a traditional contract, legal consultancy is usually required to fit the interests of both sides and the regulations. This process will induce huge cost, in terms of both money and time. But with smart contracts, the consultancy cost can be eliminated. Additionally, based on the self-executing feature of smart contracts, the contract can become effective once the condition set is met.

For example, there is a contract between A and B, which is about A has to pay B $1 million to buy a house. With the traditional approach, A and B have to seek for relevant legal services in order to make the transaction. This induces a huge cost to both A and B. However, with smart contract, the script can check whether A has enough balance to pay B; and B really has the ownership of the house. Once both conditions above are met, the contract will

execute automatically. Also this contract is appended in a blockchain, which can be served as a proof of the fact that A owns the house after this transaction.

With such features, smart contract is expected to be more commonly used in the future. However, smart contract still has some flaws that may hinder its application. In this project, the focus will be on its security loopholes as security is the first concern of every digital transaction.

# 2. Methodology

In this section, the platform chosen Ethereum and the language of smart contracts Solidity are discussed. Moreover, the node.js modules ganache-cli, Web3.js and solc, which are the major modules of developing the checking tool, are discussed and their roles in the tool are explained.

## 2.1 Ethereum

Ethereum is the chosen platform of development in this project. Ethereum is open-source blockchain platform. It is the most prominent platform for smart contracts (Buterin, 2013). It allows users to create their own scripts written in Solidity, a contract-oriented programming language for writing smart contracts. It is designed to create smart contracts on Ethereum. However, Solidity is also considered as one of the reasons why the implementation of smart contracts particularly prone to errors in Ethereum (Atezi, Bartoletti, Cimoli, 2017). In order to investigate these errors caused by Solidity, this is chosen as the language to use.

Ethereum is a blockchain platform. To create incentive for others to compute the result together, a token Ether is created to pay for those who have computed. For example, to validate a block of data about A buying a house from B, the transaction need to be validated by the crowd. To pay for the computing power, both A and B have to pay in Ether, or namely gas, for their effort. Currently one Ether token equals around US$300.

**Figure 1: Architecture of Ethereum**

Figure 1 shows the architecture of Ethereum. The header consists of many data which help identifying blocks. The PrevHash will help when chasing back the chain of data. This part makes Ethereum adopt the structure of blockchain, which make them share the same features mentioned in Background section. The state is the data stored in the block. These data can be balance or other data as shown in Fig.1.

## 2.2 Solidity

Solidity is a programming language designed for writing smart contracts. Fig. 2 shows how does a simple wallet object is created with Solidity. Owner contains the unique address of the owner of the wallet. It initializes the owner as who creates this object. Also the Pay function takes in the amount to send and the address of recipient. Before the owner actually pays, the function will first check does the owner call this function and if the owner has enough balance to pay that amount. If these conditions are all met, the function will deduct the amount from the owner and call recipient.send(amount) to increase the balance of recipient. It is only a simple example. More functions can be added if required.

```
1  contract AWallet{
2      address owner;
3      mapping (address => uint) public outflow;
4
5      function AWallet(){ owner = msg.sender; }
6
7      function pay(uint amount, address recipient) returns (bool){
8          if (msg.sender != owner ||  msg.value != 0) throw;
9          if (amount > this.balance) return false;
10         outflow[recipient] += amount;
11         if (!recipient.send(amount)) throw;
12         return true;
13     }
14 }
```

**Figure 2: Simple demo code of a wallet in Solidity (Atezi, Bartoletti, Cimoli, 2017)**

## 2.3 ganache-cli

Besides researching on the potential attacks on Ethereum smart contracts, another key deliverable of this project is a tool checking smart contracts whether the identified security loopholes exist in those contracts. In order to check those contracts, a private blockchain is needed. With a private blockchain, a isolated testing environment is created so that no real monetary transaction is involved. Additionally, all the factors in a private testing blockchain are under my control. In a public blockchain, due to the nature of decentralized system, it is impossible to control the behavior of every single user. Therefore, some extreme cases cannot be tested. On the other hand, virtual users can be created to simulate different scenario, including extreme cases. Due to these two reasons, a private blockchain is a desired testing environment of Ethereum smart contracts.

In order to create a private blockchain to simulate attacks on Ethereum smart contracts, the node.js module ganache-cli is used to create such a isolated environment. It is widely used in testing and communicating with the Ethereum private network. Its functionalities include managing different accounts, mining cryptocurrency Ether and executing smart contracts. In this project, the functionalities to be used are mainly managing different dummy accounts and executing smart contracts under testing.

In order to create a private chain without communicating with the main public Ethereum chain, all the nodes in the private chain should not be connected to the public chain and be discovered by other users. By default, 10 accounts are created and users can use them for testing.

## 2.4 Web3.js

With ganache-cli, an isolated testing environment is created. The next step is to deploy the smart contracts. Web3.js is the library to use. Web3.js is JavaScript API (application programming interface) which can compile smart contracts written Solidity and execute them in the private chain.

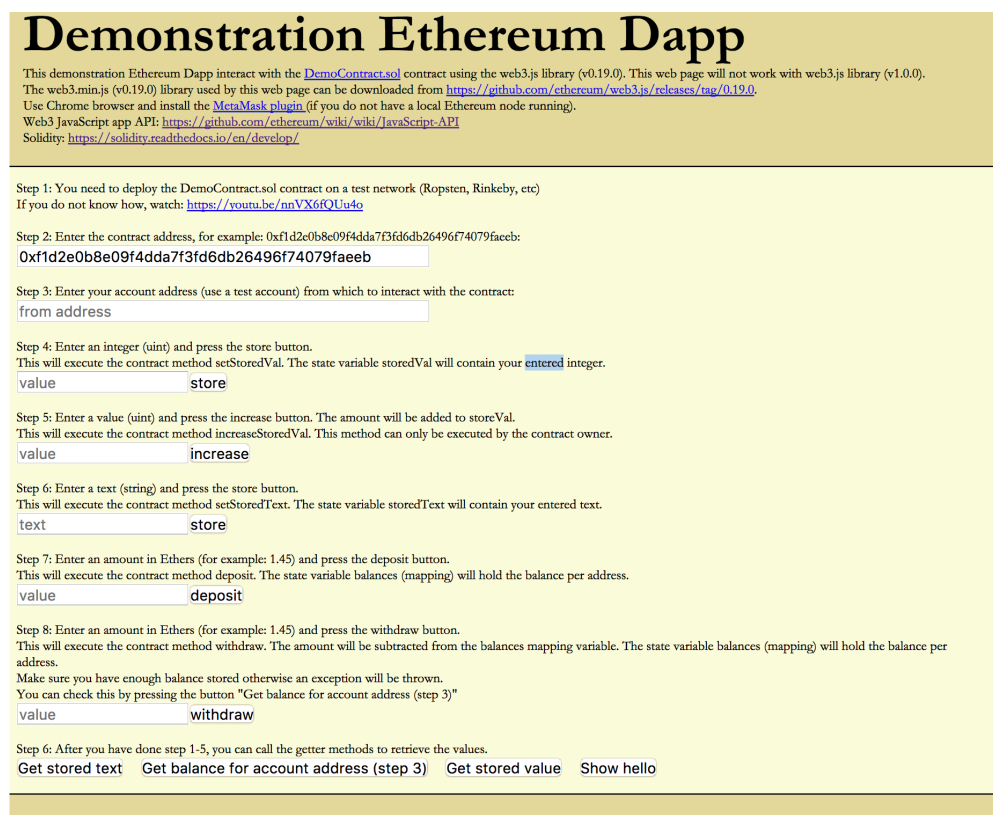



**Figure 3: A screenshot of web-based application in Web3.js by mobilefish.com**

Additionally, as it is a module in Node.js, another framework building web application, this tool can be launched as a web application. It makes the tool more user-friendly as no command input is needed to use tool. The entry barrier of using this tool is lowered by a simple web interface. The above figure is a screenshot of a web-application of Ethereum. It is easier to use compared with traditional command line tool.

To compile a Solidity smart contract, the command to use is:

```
newContract = new web3.eth.Contract(abi, accounts[0]);
```

abi is the Application Binary Interface returned from the compiled contract object from solc, which will be discussed in section 2.5. The second parameter is the address of the contract. If there is no error, the contract has to be deployed to the chain by:

```
newContract.deploy({data: bytecodes[abi_id], arguments: params})
.send({from: accounts[0], gas: 1000000}).then(function(newc){});
```

Here by calling deploy(), with its bytecode and parameters of constructor, a new promised contract instance is created and deployed on the blockchain.

In this project, Web3.js version 1.0 is used.

## 2.5 solc

solc is a node.js module that compiles smart contracts written in solidity in a node.js project. In checking tool, this module will compile the smart contracts and pass the compiled instance to Web3.js for further communication with the private chain.

To compile a smart contract, the code to use is:

```
contracts = solc.compile(input);
```

Here contracts is the compiled contract object returned and input is the contract itself in string form.

## 2.6 Summary of checking tool

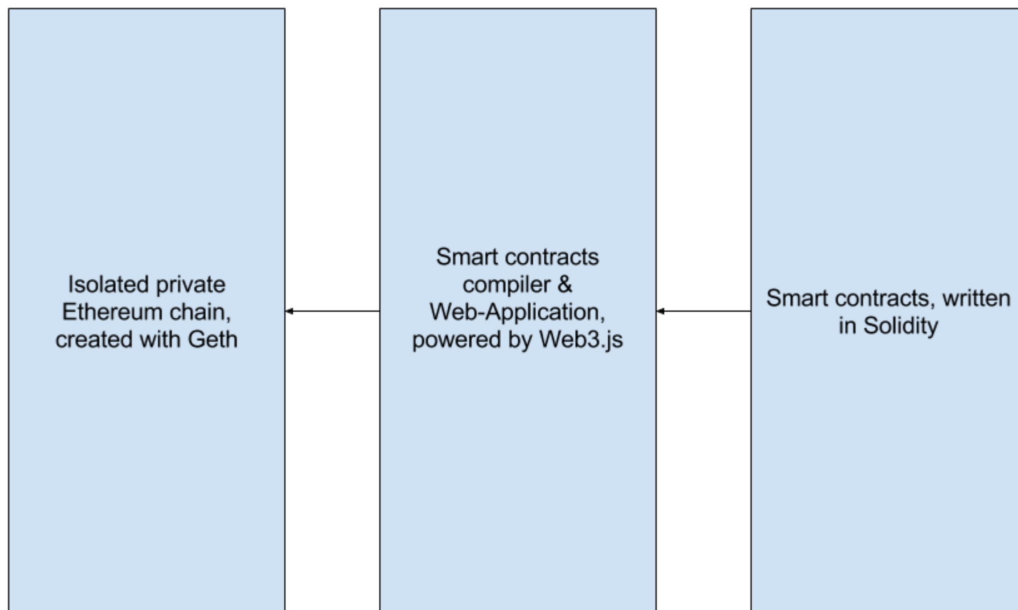



**Figure 4: Architecture and logic flow of the checking tool**

To conclude, the architecture of the checking tool is shown as above. The smart contracts in Solidity are the input of the system. They will be compiled with Web3.js and be executed on an isolated private chain created with ganache-cli. The identified security loopholes are then tested under that environment. The tool is launched as a web-based application powered by Web3.js and Node.js to make the tool more user-friendly.

# 3. Works Accomplished

## 3.1 Research on security loopholes

A major part of this project is to study smart contracts and its security loopholes. These includes the principle of blockchain and smart contracts, the skills to write smart contracts on Ethereum with Solidity and the existing and identified security loopholes of smart contracts. The principle of blockchain is mentioned in Background section and introduction of Ethereum is included in Methodology section. Therefore, this section will focus on the findings on the identified security loopholes of smart contracts.

In "A Survey on Attacks on Ethereum smart contracts, it provides a list of known vulnerability of Ethereum Smart Contracts. Additionally, different developers have posted some newly discovered vulnerabilities online, like "How to Secure Your Smart Contracts: 6 Solidity vulnerabilities and how to avoid them" by Georgios Konstantopoulos (2018).

### 3.1.1 Wrong Typecast

```
function sweepCommission(uint amount)  {
  owner.send(amount);
}
```

**Figure 5; A sample code of wrong typecast**

First, wrong typecast to variables is one of the vulnerabilities identified. Solidity compiler dons not check whether a function takes in a correct type of variable. A sample code is shown in Figure 5. The function sweepCommission takes in a parameter with typr uint, which stands for unsigned integer, like 20. If parameters with other types, like string or decimal number, error should be prompted to notify the developer. However in Solidity, no error will be returned and the developer cannot notice such an error as usually the other language compile will check the type of variables. So the developer may think that the contract is correctly executed. It could bring chaining effect if the number of parties involved is huge. The whole system may fall.

### 3.1.2 Immutable bugs

Next, the bugs are immutable once it is on the blockchain. Based on the mechanism of blockchain, it is nearly impossible to change a single block of data once it is appended. That means once a bug is on the blockchain, it is difficult to remove it. Once the bugs or vulnerability stack up, the system may fall.

### 3.1.3 Unpredictable States and Reentrancy

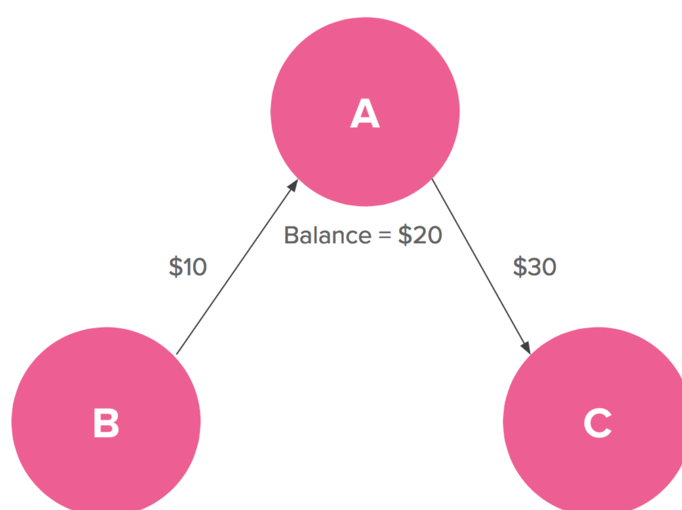



**Figure 6: An example of unpredictable state scenario**

Finally, unpredictable state is another problem to be considered. As mentioned in Methodology section, state is the data contained in a block, e.g. balance. However, sometimes the smart contracts cannot be executed immediately. An example scenario is shown in above figure. A has 20 dollars as balance. B sends 10 dollars to A and A pays C 30 dollars. These transactions will be valid if A first collects money from B followed by paying C. However, it is not guaranteed that A collects money first. The time taken for checking processes of two different transactions and network speed of different users are some potential factors affecting the order of executing those contracts.

One attack exploiting this vulnerability is reentrancy. Here we refer back to the case of figure 6. B has a variable storing all debts he owns, say debt[A] = 10. After A receives 10 dollars from B, debt[A] should be 0 instead of 10. Here the value of debt[A] is a checking mechanism of paying A. However, if B updates that value of debt[A] after sending 10 dollars

to A, A can then call for debt for twice and receive 20 dollars. Since .send() takes unknown time, debt[A] may not be updated when the second call arrives. Then the second call will send 10 dollars to A again, which is a loss to B.

To solve this problem, updating balance must be done before sending ether to others. Another approach is to use .transfer() instead of .send().

### 3.1.4 Uint Overflow/ Underflow

Uint is a data type commonly used in solidity. It stands for unsigned integers, ranging from 0 to 2^(256) -1, i.e. a 256 bit number. One typical use of this data type to represent account balance, which is crucial in monetary transactions. Although the range of values seems enough, overflow or underflow may occur. If we increment to 2^(256) – 1 by 1, the value will be out of bound and return to 0. Here overflow occurs. The same logic applies to subtracting 1 from 0. The value will be to 2^(256) – 1 and underflow occurs. This is like the Year 2000 problem where the numbers of bits is not enough.
To solve this, there is a solidity library SafeMath.sol from Zeppelin (Zeppelin, 2018) which throws errors to user if overflow or underflow occur.

### 3.1.5 Unclear Visibility of Functions

Like any other languages, solidity allows users to specify the visibility of functions and variables. There are 4 types of visibility level:

| Level | Scope |
|---|---|
| Public | All users can access |
| External | Only accessed by external parties, cannot be accessed by other functions in same contract |
| Internal | Only accessed by other functions in same contract, or child contracts inherited from it |
| Private | Only accessed by other functions in same contract |

**Figure 7: A table of different visibility levels of Solidity**

By the above levels the behavior of different contracts can be controlled. However, if no visibility level is specified, the level will be public by default. If the developer did not notice, hackers can call the functions that change key information of the contract, like the account

balances. This causes delegate calls which is not desired. To solve this, the visibility should be decided carefully and always specify one level to prevent setting the level by default.

### 3.1.6 Forcing Ether to a Contract

A contract can stores Ethers and it can be retrieved by this.balance in solidity. However, if it is used as a condition of executing some other lines, it may be bypassed by receiving Ether.

```
pragma solidity 0.4.18;

contract ForceEther {

  bool youWin = false;

  function onlyNonZeroBalance() {
      require(this.balance > 0);
      youWin = true;
  }
  // throw if any ether is received
  function() payable {
    revert();
  }
```

The above is a sample code of a contract. The fallback function, which will be called whenever others sends Ether to it, will always revert, i.e. throws. Therefore, the variable youWin will never be true as the initial balance is 0. However, the other contracts can call Selfdestrut, which will renders the contract useless and send all its fund to a target address. Moreover the fallback function of target contract is not called in this case. Therefore, when some other contracts call selfdestruct and have the target as the contract above, its balance will be greater that 0, hence youWin is then true. To avoid this loophole, this.balance should never be used as a checking condition as others can forcefully send Ether.

### 3.1.7 DoS by Calling to the Unknown

This vulnerability can be illustrated by King of the Ether case.

```
pragma solidity ^0.4.18;
contract CallToTheUnknown {
  // Highest bidder becomes the Leader.
  // Vulnerable to DoS attack by an attacker contract which reverts all
transactions to it.

    address currentLeader;
    uint highestBid;

    function() payable {
        require(msg.value > highestBid);
```

```
        require(currentLeader.send(highestBid)); // Refund the old leader, if it
fails then revert
        currentLeader = msg.sender;
        highestBid = msg.value;
    }
}

contract Pwn {
  // call become leader
  function becomeLeader(address _address, uint bidAmount) {
    _address.call.value(bidAmount);
  }

  // reverts anytime it receives ether, thus cancelling out the change of the
leader
  function() payable {
    revert();
  }
}
```

In this case, the user sending most amount of Ether will be the King. When there is a new king, the amount paid by the previous king will be refunded. To attack this contract, the attack can create a new contract, i.e. contract Pwn above, having a fallback function which always revert. Then he will be the king first by sending enough Ether. When others send enough ether to be the new king, the refund cannot not be done as the attacker's contract always throw the refund. Then the attack will be the king forever as the king cannot updated, which is done after a successful refund. With such a simple contract, an effective DoS is launched. At this stage, there is nothing we can do to completely avoid this vulnerability. One recommendation to solidity is that an explicit type of error can be delivered to warn the developers.

### 3.1.8 Short Address Attack

This vulnerability is found ERC20 tokens, which is technical standard used for smart contracts on the Ethereum blockchain for implementing tokens, and published by the Golem team (2017). Besides send(), transfer(address, uint) is used to send ether to others. After calling it, the function will be encoded into a 68-byte long data. The structure of the data is:

| No. of Bytes | Data Descirption |
|---|---|
| 4 | Method ID |
| 32 | Destination address of 20 bytes, then filled with leading zeros |
| 32 | Value to transfer |

**Figure 8: A table of structure of full transaction of .transfer()**

Normally a address is 20-byte long. However, if an address ends with 0, like 0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa0000, the address will still be the same after removing trailing zeros. So by passing in a shortened address, the data will be shorted and the value to transfer will be larger. To avoid this vulnerability, the length of input must be check and throw if data with invalid length is passed in.

The above are some of the vulnerabilities of smart contracts. There are more in other sources not studied and remained not identified. The research and the tool will mainly focus on the holes identified above.

## 3.2 Developing the checking tool

Besides the research on attacks on Ethereum smart contracts, the research on how to develop a checking is necessary. The working principle of the checking tool is discussed in the Methodology part. Therefore in this section, only the progress of the development is reported.

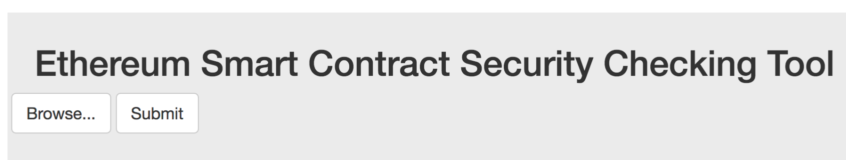
A private chain is set up in my own private machine by gnanche-cli. I can now perform simulated transactions under my control. However some parts, like some  scenarios of different attacks, are not able to be implemented. The reason will be discussed in later parts.

### 3.2.1 Launching a private chain

As mentioned above, the testing environment will be a private chain. A private chain is set up in my own machine using ganache-cli. After testing, simple Ethereum smart contracts can be executed in that chain. Additionally, as a private chain in a completely closed environment is used, all the user activities, like transactions, are all under control.

### 3.2.2 Launching a prototype of the web-based tool

Besides setting up the chain, a web-based tool is required to provide the best user experience.



Ethereum Smart Contract Security Checking Tool

Coursete

string _fName

uint256[] _age

Deploy and Run Security Test

Coursetro

string _fName

uint256 _age

Deploy and Run Security Test

**Figure 9: A screenshot of the web tool**

The above is a screen shot of the prototype of the web tool. It allows the users to upload the smart contract file with extension .sol. Then the user has to fill in valid parameters of constructor to deploy it, followed by being passed to the private chain and executed.

Ethereum Smart Contract Security Checking Tool

Possible Security Vulerabilities

Reentrancy

Possible Effect:

Checking mechanism of variables may be bypassed

Possible way(s) to fix:

- Update the variable before sending to others
- Use .transfer() instead .send()

Visibility

Possible Effect:

Hacker may claim unauthorized ownership of ypu contract

Possible way(s) to fix:

- Specify the visibility carefully. All functions are public by default

**Figure 10: A screenshot of security report generated by the web tool**

After different checking, a security report with recommendations is generated to users.

### 3.2.3 Checking Principles of different Vulnerabilities

Due to the nature of different vulnerabilities, only 2 of them can be simulated without user input. This tool aims at providing a simple checking to the smart contracts, and the users should not know much about smart contracts. If they know more, they can do other a more detailed simulation on other platform, like Remix which is online IDE of solidity. Moreover, due to the limitation of Web3.js and knowing the behavior of contracts with machines, it is difficult to simulate some scenarios. The limitation will be discussed in later sections. Therefore, some vulnerabilities will be checked by only looking for key words in the contracts, while some will be checked with simulation. The vulnerabilities checked are those mentioned in Section 3.1.

#### 3.2.2.1 Checking Uint Overflow/ Underflow

This can be tested without simulation. If the contract performs unit arithmetic without checking for overflow or underflow, we can conclude that this contract is vulnerable to unit overflow or underflow.

It can be done for searching for declaration of uint variables and arithmetic symbols, i.e. "+, -, *, /". To check whether overflow or underflow is detected, the common practice is to use require() to wrap up the arithmetic operation. Require() is used in solidity like an if-else statement in other languages. If the return value is true, the operations after require() are then executed. For instance, require(a – b < a) checks if a-b is smaller than a. If underflow occurs, a-b will be greater than a. Therefore this line can check for overflow or underflow. If the contract does not contain such lines, it may suffer from uint overflow or underflow.

#### 3.2.2.2 Checking Visibilities

It is difficult to simulate as we do not know the expected behavior of the methods. For example, it is acceptable to have a public getter function, while it is acceptable to have a public setter function of internal variables. Therefore, it is hard to detect. However, like what have been mentioned in section 3.1.5, the default visibility level is public. When solc compiles a contract, a warning of no visibility specified will be shown to user. Therefore this tool will propagate this warning to user and reminder them to specify the visibility, as public methods provides more room for attacking.

#### 3.2.2.3 Checking Forcing Ether

This can be tested without simulation. As mentioned in section 3.1.6, the problem in this case is to use balance of the contract as a condition for executing some lines. To detect this, we can try to look for the key word “this.balance” in the contract. Furthermore, we look for any comparison of value with this.balance. This can be shown by key words “this.balance >”. Therefore, simulation is not needed in this case.

#### 3.2.2.4 Checking DoS by Calling to the Unknown

In this case, both text analysis and simulation will be used. For text analysis, the key words are “.send(” as it is triggered by calling .send(). So by looking for these key words, we may say there is a risk of being vulnerable to DoS by calling to unknowns.

To simulate this scenario, another contract is needed. The contract is:

```
pragma solidity 0.4.18;

contract TestCallToUnknown {

    test t;
  function TestCallToUnknown {
     t.method(params);
  }
  // throw if any ether is received
  function() payable {
    revert();
  }
}
```

This contract creates an instance of the contract under testing and calls the method that will send Ether. In the original contract, a new event is added after sending Ether. If the event is not caught, it means that the sending of Ether fails, and hence all the lines after that cannot b executed. By having this TestCallToUnknown contract, the attack can be simulated.

To get the names and parameters of the method, they can available in the ABI of the contract.

### 3.2.2.5 Checking Short Address Attack

In this case, both text analysis and simulation will be used. For text analysis, the key words are “msg.data.length” and “.transfer(”. By looking for “.transfer(”, we know whether the method .transfer(). If ”msg.data.length” does not appear in the contract, it means that the length of the message is not checked, hence my suffer from short address attack.

To simulate this, an account with address ends with 0 is needed. To ensure this, an error will be prompted if there is no such account.

No account with address ends with 0. Please restart the server by executing "node index.js", and restart the chain by executing "ganache-cli" in terminal.

**Figure 11: A screenshot of error of not having an account with address ends with 0**

With such an account, we can then explicitly make a new address with deleted trailing 0 and call the .transfer() function with that account as destination, and 1 as the value to transfer. If the value send is different, we can conclude that the contract may be vulnerable to short address attack.

### 3.2.2.5 Checking Reentrancy

As mentioned in section 3.1.3, the reason why reentrancy occurs is that the variable determining the condition is not updated when waiting for successful send. In order to detect it, the method must call .send(), and it must be followed by updating a variable. Since most likely the variable to update is a number or a Boolean variable, the arithmetic symbol like +, - , *, = will appear after .send(). Therefore by this text checking, we may detect such vulnerability existing in the contract.

## 4. Problems Encountered

There are several problems encountered in this project. This section will discuss those problems in both the research and development.

### 4.1 Insufficient security loopholes

Smart contract is still quite new to the society. The reliable research in this field is still not enough. The main reference I take in this project is "A Survey of Attacks on Ethereum Smart contracts", as well as "How to Secure Your Smart Contracts: 6 Solidity vulnerabilities and how to avoid them". However, there are some loopholes not covered. If some reports on the Internet are also considered, more research will be needed to verify the identified loophole. It is a pity that some unverified and potential security holes are not covered in this project.

### 4.2 Computing Resources of the tool

The checking tool requires an isolated environment for testing. However, the computing power and storage required to run a private chain is not negligible. Moreover, if a private chain is created for every test, the storage will be soon full. Therefore, a way to solve this problem is needed for making the tool scalable for future use. A possible approach is to reset the data on the chain when the test ends.

### 4.3 Deploying the tool to the cloud

To make the tool available to public, the ideal approach is to deploy the tool to cloud. However, as mentioned above in 4.2, the storage and computing power are major concerns. Since it is currently in development phase, the development and testing will be done in local machines to reduce development costs like running a machine on cloud. However, the tool can be migrated to cloud when conducting pressure test on the tool, which is still not implemented due to time constraint.

## 5. Future Works

At this stage, the research of the topic and a simple smart contract checking tool is done. However due to the scope of this project and the different limitation, there are still future works that can be done.

### 5.1 Use Machine Learning to Read and Check the Contracts

In the checking process, the major way to detect certain vulnerabilities is looking up key words. However, it is not accurate and efficient enough. Indeed, there are similar tools available in market.

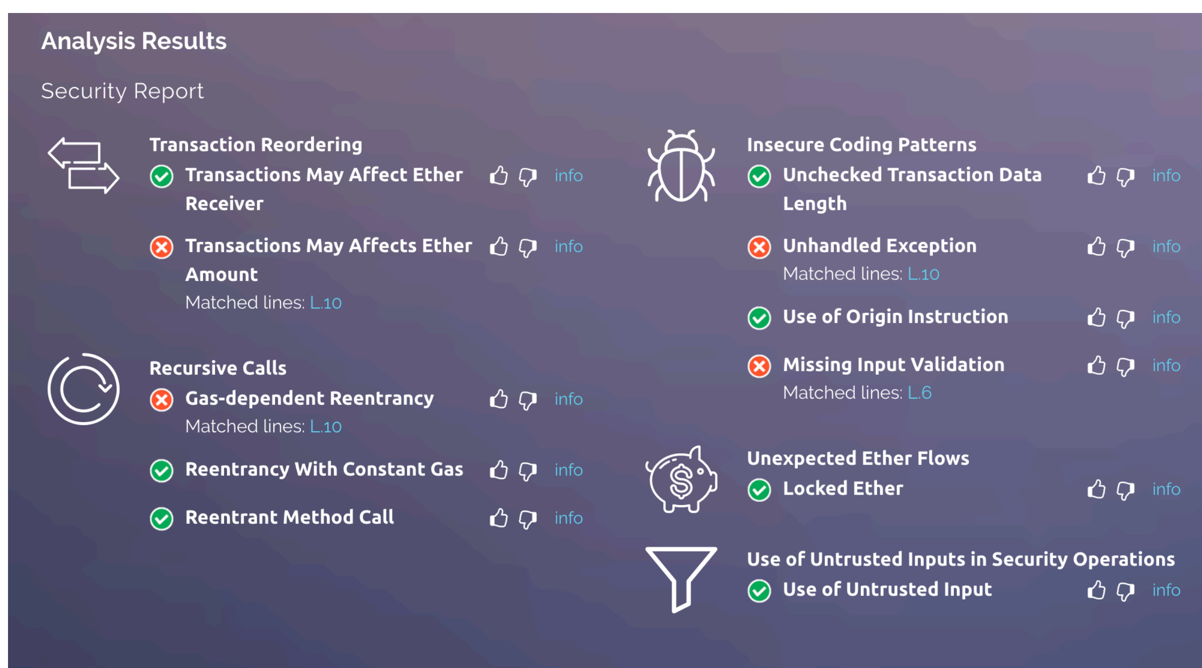



**Figure 12: A screenshot of security report of Securify**

The above screenshot is from another smart contract security auditing service provider Securify. The first screenshot is a security report from free tier service and the next one is the list of auditing service. Here we can observe that the security will ask for feedback from the user, asking if their judgement is correct. It is then guessed that the backbone of their free and automated service is machine learning, which needs much feedback for training.

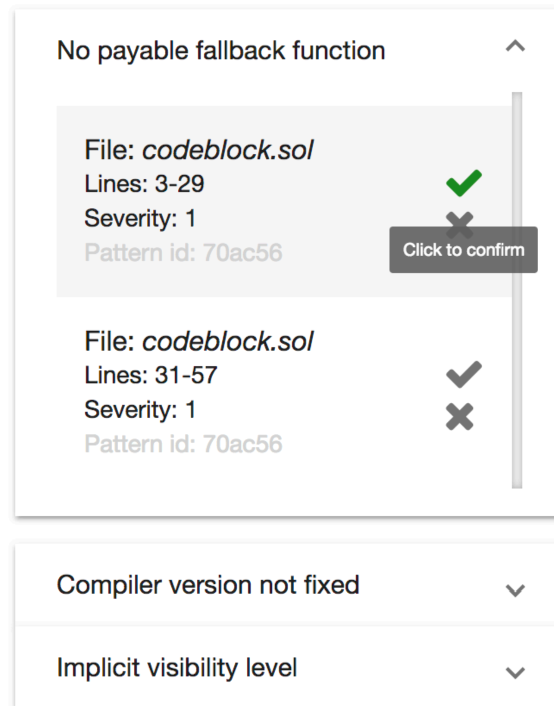



**Figure 13: A screenshot of security report of SmartCheck**

The above is the security report from SmartCheck, another tool available in market. The "click to confirm" is another sign of using machine learning to check to code. Here it is observed that machine learning is commonly used for automated checking of code. In the future the checking mechanism can be implemented with the help of machine learning, rather than naïve text check in current approach. However, due to time constraint and insufficient of data, i.e. sample contracts, this is not implemented in this project. However, this is a new topic with great potential to work on.

## 5.2 UI Enhancement

In this project, the styling library used is Bootstrap. However, the design is still to plain and may affect the user experience. Also this tool only allows user to upload the file directly, where other tools in the market allows direct pasting the code, like in Remix and SmartCheck. Therefore adding a text field for pasting the code may be another area to work on.

# 6. Limitation

During the research of the project, there are some challenges and limitations of the project.

## 6.1 Uncovered security loopholes

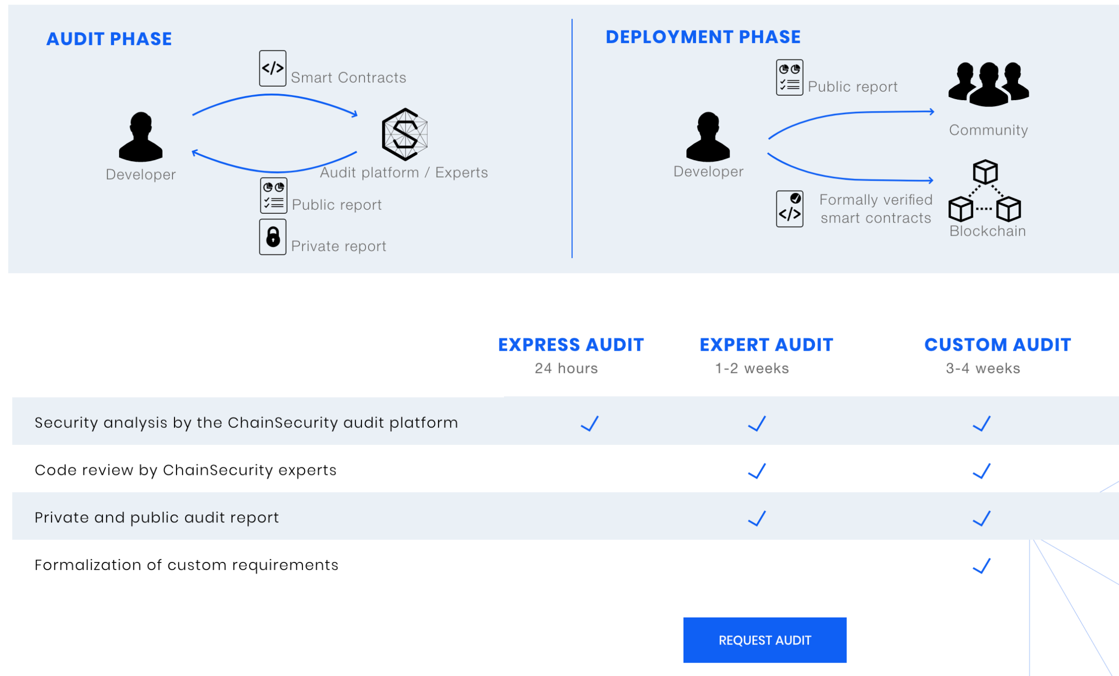
First, there may be other security loopholes of smart contracts that are not included in this tool. Although there are various identified attacks on smart contracts being studied in this project, it is foreseeable that there will be new attacks in the future. As a checking tool of smart contracts, it should identify as many security holes as possible to prevent false-positive situation. Therefore in the future this tool needs regular update to deal with new security loopholes. However, due to time constraint, the development work will take up most of the time. So not much research on new attacks can be done.

## 6.2 Cross platform checking

Next, this tool can only detect security holes within the same blockchain. On Ethereum, all the transaction is done with the currency Ether, a cryptocurrency produced in Ethereum computation. However, to make smart contracts practical, the smart contracts must get data from other sources to validate a contract. Like in the example of A buying a house from B, the contract must see if B has the ownership of the house from other source like the government if the data required is not on the same blockchain. So API (application programming interface) must be used for data exchange. However, communication of different APIs or the API itself may create security holes which cannot be checked with this tool.

## 6.3 User Input and Unpredicted Behaviors of Contracts

The checking tool in this project relies much of naïve text checking, which is expected to be not accurate enough. However, the behavior of different methods and the purpose of the contract should be known in order to perform attack simulation. This then relies on user input. However, the most effective and accurate way to audit a smart contract is still reviewing by developers.




| | EXPRESS AUDIT<br>24 hours | EXPERT AUDIT<br>1-2 weeks | CUSTOM AUDIT<br>3-4 weeks |
|---|---|---|---|
| Security analysis by the ChainSecurity audit platform | ✓ | ✓ | ✓ |
| Code review by ChainSecurity experts | | ✓ | ✓ |
| Private and public audit report | | ✓ | ✓ |
| Formalization of custom requirements | | | ✓ |

**Figure 14: A screenshot of smart contract auditing services list of Securify**

The above is the auditing service tier list of Securify. It is clearly stated that the contract is audited by experts and the time taken may be a month long in higher tier. Therefore, only relying on machines for checking and auditing is not good enough. The input from experts and developers is needed to make an accurate enough checking.

## 7. Conclusion

With the rise of blockchain and the convenience brought by smart contracts, the coverage of smart contracts is expected to grow rapidly. That is the reason why security loopholes of smart contracts are concerned. In this project, the loopholes are studied and a tool checking the vulnerability of smart contracts from those security holes will be developed as the final deliverable. Although this project is a year-long project, there are still rooms of improvement, like introducing machine learning to validate the contract and adding more security vulnerabilities to test.

## 8. Reference


1. Ardit Dika; Ethereum Smart Contracts: Security Vulnerabilities and Security Tools, Norwegian University of Science and Technology (2017)
2. Atzei, N, Bartoletti, M, and Cimol, T.i: A Survey of Attacks on Ethereum Smart contracts, Universita degli Studi di Cagliari, Cagliari, Italy (2017)
3. Buterin, V.: Ethereum: a next generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper (2013)
4. ConsenSys Diligence; Ethereum Smart Contract Security Best Practices, https://consensys.github.io/smart-contract-best-practices/ (2018)
5. Creating a Private Chain/Testnet, https://souptacular.gitbooks.io/ethereum-tutorials-and-tips-by-hudson/content/private-chain.html (2017)
6. Ganache-cli – Truffle, https://github.com/trufflesuite/ganache-cli (2018)
7. Georgios Konstantopoulos ; How to Secure Your Smart Contracts: 6 Solidity vulnerabilities and how to avoid them, https://medium.com/loom-network/how-to-secure-your-smart-contracts-6-solidity-vulnerabilities-and-how-to-avoid-them-part-1-c33048d4d17d, https://medium.com/loom-network/how-to-secure-your-smart-contracts-6-solidity-vulnerabilities-and-how-to-avoid-them-part-2-730db0aa4834 (2018)
8. Pawel Bylica from Golem; How to Find $10M Just by Reading the Blockchain, https://blog.golemproject.net/how-to-find-10m-by-just-reading-blockchain-6ae9d39fcd95 (2017)
9. Javascript API – Ethereum Wiki, https://github.com/ethereum/wiki/wiki/JavaScript-API#web3ethcontract (2017)
10. Mobilefish.com, Demonstration Ethereum Dapp, https://www.mobilefish.com/download/ethereum/DemoDapp.html (2017)
11. Peter Vessenes; More Ethereum Attacks: Race-To-Empty is the Real Deal, https://vessenes.com/more-ethereum-attacks-race-to-empty-is-the-real-deal/ (2016)
12. Securify, https://securify.ch/ (2018)
13. SmartCheck, https://tool.smartdec.net/ (2018)
14. Zeppelin, SafeMath.sol on Github, https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/math/SafeMath.sol (2018)

## 9. Table of Figures

Figure 1: Architecture of Ethereum 7
Figure 2: Simple demo code of a wallet in Solidity (Atezi, Bartoletti, Cimoli, 2017) 8
Figure 3: A screenshot of web-based application in Web3.js by mobilefish.com 10
Figure 4: Architecture and logic flow of the checking tool 11
Figure 5; A sample code of wrong typecast 12
Figure 6: An example of unpredictable state scenario 13
Figure 7: A table of different visibility levels of Solidity 15
Figure 8: A table of structure of full transaction of .transfer() 17
Figure 9: A screenshot of the web tool 19
Figure 10: A screenshot of security report generated by the web tool 19
Figure 11: A screenshot of error of not having an account with address ends with 0 22
Figure 12: A screenshot of security report of Securify 24
Figure 13: A screenshot of security report of SmartCheck 25
Figure 14: A screenshot of smart contract auditing services list of Securify 27