國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

Gcoin 智能合約實作

Gcoin Smart Contract Implementation

丁士宸

SHIH-CHEN TING

指導教授：廖世偉 博士

Advisor: Shih-Wei Liao, Ph.D.

中華民國 106 年 7 月

July, 2017

誌謝

感謝我的父母在背後無怨無悔的支持與付出，讓我得以無後顧之憂地追求學業，如果沒有你們，不會有今天的我。感謝廖世偉教授在研究所的指導與鼓勵，讓這篇論文得以完成。感謝實驗室的小柏、小邦、Wilson，你們不論在學業上還是生活中都是最好的同伴。感謝 Gcoin 社群的開發人員，柏宇及邦庭，提供這篇論文的最關鍵的核心技術以及支援。最後感謝陪我度過研究所這段奇妙旅程的霸主、恩冉、Kelly、上潁、鈞為、大柏 、土豪、Heman，謝謝你們。

摘要

「智能合約」一詞由 Nick Szabo 在 1994 年提出。智能合約是一種可以被自動化或人為執行的程式，程式內容即是雙方共同約定之合約事項。傳統人與人之間訂定之合約內容的執行，都需要人為相當程度地介入處理，而且往往曠日廢時；智能合約則可將透過正確的程式撰寫來將合約內容程式化，透過電腦來執行這些內容。智能合約可能的應用場景包含商品買賣、遺產分配、私人借貸等等。

合約的訂定通常與金錢八九不離十，所以合約執行的正確性及安全性就顯得十分重要。近年來，透明公開、公正且不具竄改性的區塊鏈技術越來越成熟，許多人開始將區塊鏈技術與智能合約結合，希望能夠移除人們對於合約自動化執行方面的疑慮。智能合約使用區塊鏈上不外乎兩個原因：一、所有的合約執行記錄都可查詢；二、合約狀態無法被輕易更改。而區塊鏈具有這兩項特性恰好成為一個良好的智能合約平台。

本論文旨在討論如何將建立智能合約功能於現在的 Gcoin 區塊鏈上，內容將包含智能合約的實作架構與細節。

Abstract

The concept of smart contract was first proposed by Nick Szabo in 1994. Smart contract is a program that can be executed manually or automatically, with contractual clauses written in the form of code inside the program. To enforce contractual clauses in a traditional contract made between parties are often time consuming; with program properly coded, the contractual clauses can be executed by a computer.

Contracts often deal with matters that involve money, so the correctness and security of smart contract become imperative. Blockchain with traits like transparent, just, tampering-proof makes it a great platform to implement smart contract architecture.

This thesis focus on the design and implementation of a smart contract architecture on Gcoin blockchain.

Table of Contents

Table of Figures

Table of Tables

Chapter 1 Introduction

Traditional contracts are often written in the form of paper or digital file, and the enforcement of the contracts are mostly overseed either by the envolving parties or trustworthy third party. The main problem with the traditional contracts is how inefficient and time-consuming while trying to enforce contractual clauses. It typically takes days if not years to properly deal with those events and only brings much hazzle to all the parties envolved.

A new type ot contract called “smart contract” is first proposed by Nick Szabo [1]. He suggested that with smart contract, which encodes traditional contractual clauses to program logic, enforcing the contractual clauses based on timed event or user actions will be much easier to every party.

An example would be a bet on the outcome of a sport between Alice and Bob. Say Alice and Bob had concented that each one of them will put a $5 wager into a third-party account they both trust, and once the outcome had been revealed, the winner takes it all. With smart contract, Alice and Bob only need to provide their accounts with sufficient money in the properly coded smart contract, then the outcome of the sport will be fetched by the smart contract and trigger an action that transfers the money form the third-party account to either Alice’s account or Bob’s.

How smart contract is implemented varies by different architechures they are built in or around. Lately, the trend of building smart contract around blockchain technology has been increasing. A well known blockchain technology, Ethereum, is an prominent example that has smart contract functionality built in it. In this thesis, we discuss another approach to integrate smart contract with an existing blockchain, Gcoin, without modifying any of its code.

Chapter 2 Background

2.1 Blockchain

Blockchains are often known as digital ledger technology that hold transactional data made by the users in the peer-to-peer network. Bitcoin [2] is the most well-known blockchain technology as of writing. In fact, as the term “blockchain” suggested, blockchain can also be referred to as a data structure that chains multiple *blocks* together, with each block points to another block created prior to itself by the network. Each block is also another data structure with one or more *transactions* that mark the value transfer between the users.

When a user on the blockchain network decides to transfer some value to another user, he/she have to first broadcast a transaction to the network, then some nodes in the network will collect a set of received transactions into a single block. To create a block on the blockchain, a node in the blockchain network must solve a difficult hashing problem before the block can be broadcasted to the network and be accepted by other nodes. Once a block is successfully generated, it is appended to the last block that the network had universally accepted, thus forming a “blockchain”. After the block is appended, nodes in the network starts to collect transaction broadcasted to them and try to build another block. A transaction is said to be valid or confirmed if the transaction is included in a block that is accepted by the blockchain network.

2.2 Script

In Bitcoin, *scripts* [3] are a part of transaction data. Bitcoin implemented scripting system that are used to validate transaction data. Script is comprised of various *op codes* that are read by the scripting system from left to right. Op codes have different meanings to the scripting system, some represent constant data, some represent arithmetic or cryptography operations.

A transaction in the Bitcoin blockchain has two parts: input and output. The output part may contain a kind of script called "locking script" that locks the currency in it. Addresses are used in locking script denoting that it owns the currency locked in the script. To spend the currency locked in locking script, another transaction is created with input referencing the output script and provide another kind of script called "unlocking script" that unlocks the locking script. In order to prevent others from spending the currency locked in the locking script, the unlocking script should contain the signature produced from the private key of the address in the locking script. Locking scripts that lock currency in them that are not unlocked by other unlocking scripts are called *unspent transaction output* or UTXO.

2.3 Address

In Bitcoin, an address is where digital asset is stored. An address is the hashed output of a cryptography outcome.

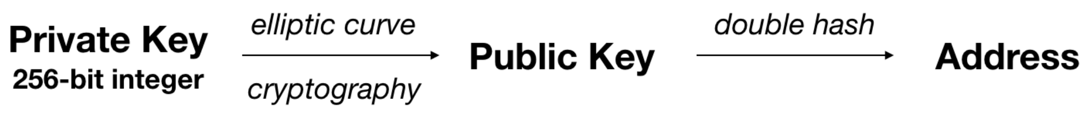

Figure 1: Private Key, Public Key and Address

To generate a new address, one simply choose a 256-bit random number as private key and apply elliptic curve cryptography to get a public key. Once a public key is generated, two hash function are used to get the address: first with SHA-256 then RIPEMD-160:

```
address = RIPEMD160(SHA256(public_key))
```

Address are often further encoded with Base58 [4] to represented as a human-readable string of characters and digits, so one can share this address with anyone who wants to transfer money to the owner of the address. Whenever the owner wishes to spend the money in the address, he/she will have to create a transaction with the address as sender and other addresses listed as recipients, then sign the transaction with the private key that was used to derive the address.

Since the address can be derived from public key and the public key from private (not true from the other way around), one must keep the private key absolutely safe to himself/herself so no one has the right to spend the currency in that address.

2.4 Multisignature Address

In Bitcoin, a multisignature address is a special kind of address that is derived by a script that combines multiple public keys. Multisignature address acts just like a normal address, it can receive money from any address and send money to any address. A multisignature address is derived from a script we called multisignature script that is often described in the M-of-N scheme. The M-of-N scheme indicates that the address is created from N public keys and requires M signatures from N of the private keys to sign the transaction before one can transfer the currency out from that multisignature address. The general form of a multisignature script looks like this:

```
M <public key 1> <public key 2> ... <public key N> N OP_CHECKMULTISIG
```

And a 1-of-3 multisignature script looks like this:

```
1 <public key A> <public key B> <public key C> 3 OP_CHECKMULTISIG
```

To get the multisignature address of the script, two hash functions have to be applied: first use SHA-256, then RIPEMD-160:

```
multisignature address = RIPEMD160(SHA256(script))
```

In order to spend the currency locked in a multisignature address, one should also provide an unlocking script to unlock it. One example of an unlocking script that satisfies the 1-of-3 multisignature script is:

```
OP_0 <signature B>
```

where the signature can be replaced with one of two other signatures from the private key corresponding to the public key listed in the locking script.

An example usage of multisignature address would be as follow: say a couple wants to have a shared address on the Bitcoin that they can only spend the currency in it when both of them agree to. To achieve this scheme, they can create a multisignature address with their own public keys, so when one of them wants to spend some currency in the address, it requires two signatures to unlock the currency stored in it.

2.5 OP_RETURN

OP_RETURN script is a special kind of script that can be put at the output part of a transaction, also known as data output. One can write any kind of data in the OP_RETURN script as a note or comment of the transaction. An OP_RETURN script looks like this:

```
OP_RETURN <data>
```

The blockchain does not interact with the data saved in the OP_RETURN script or parse it whatsoever. However, Bitcoin only allows 40 bytes of data in the OP_RETURN script.

2.6 Gcoin

Gcoin [5] is a blockchain developed by researchers from National Taiwan University; it is a fork from Bitcoin with additional features included. Since Gcoin is a fork of

Bitcoin, the address and script system are still compatible with Gcoin. Two major features that distinguished Gcoin from Bitcoin includes:

- Faster block generation
- Larger OP_RETURN script capacity

In Bitcoin, the average block generation time is 10 minutes, which means a transaction will have to wait up to a maximum of 10 minutes for it to be confirmed. Gcoin is modified so the average block generation time is narrowed down to 15 seconds. This enables Gcoin to have higher transaction throughput and lower transaction confirmation time.

Lastly, the maximum size of data that can be put into the OP_RETURN script is 40 bytes, whereas Gcoin can hold a maximum of 128 kilobytes.

Chapter 3 Overview

Unlike Ethereum, at the creation time of Gcoin, smart contract is not taken into consideration to be incorporated into a part of Gcoin. To integrate Gcoin with smart contract, we have to come up with a design that will work with Gcoin externally so we don't have to modify any code in Gcoin. The main goal of this design is to fully decouple the smart contract system from Gcoin, in hope that porting to Bitcoin will relatively be an easy task to do.

3.1 Smart Contract

Smart contract is a piece of program that upon *invocation* will execute some part of the code written in it. Before invocation, a contract has to be *deployed* first. Different smart contract design uses different approaches to implement this behavior. In our design, smart contract is written as a Java class packed with other classes in a jar file and executed by JVM. Deploying a contract is the process of constructing an instance of the contract class, while invoking a contract is to call a method in the class. The class members of the Java smart contract as a whole is said to be the *state* of the contract.

3.2 Contractor and Endorser server

In this thesis, we introduce two kinds of service provider: *contractor server* and *endorser server*. These two servers are the pivot parts of our design, for they are the backbone of the architecture.

Contractor server is the intermediate between Gcoin and the endorser servers. No matter if a user wishes to deploy or invoke a contract, he/she must reach out to a contractor server that will act as a delegate to help the user prepare the contract deployment or invocation. In the big picture, contractor server will first deploy the contract or invoke the contract to get the state of the contract. Once the state is generated, it gives multiple endorser servers the hashed state, trying to get their “endorsements” of the state. If enough endorsements are given, the contractor writes back the smart contract state to the Gcoin.

Endorser server is responsible for the endorsement of a contract. Each endorser will own a private key (just like those used on the blockchain) to “endorse” a contract. The corresponding public key of the private key is used to identified different endorser server. Endorser server actually runs the contract code with JVM and check if the hashed contract state is consistent with the hashed state provided by the contractor server. If the contract state is consistent, it gives to contractor server its signature, indicating that it “endorses” the contract state. On the other hand, if the contract state differs, endorser server should not endorse the contract and report a failure to the contractor server.

3.3 Deploy Transaction and Invoke Transaction

To implement smart contract on Gcoin, we define two special kinds of transaction.

- Deploy transaction
- Invoke transaction

When a user wants to deploy a contract to Gcoin, the user first chooses a set of endorsers he/she trusts. Contractor server will ask these endorsers for their public keys then create a unique multisignature address. This unique multisignature address will be used to associate and identify the contract on the Gcoin. After a multisignature address is generated, a deploy transaction is created by the contractor server. This transaction contains the multisignature and the contract jar file in its output. Once this transaction is broadcasted to Gcoin network, the contract is said to be “deployed”.

When a user wants to execute a method in the deployed smart contract, the user must provide the method name and the arguments to the method, to the contractor server. The contractor server will execute the method with the provided arguments to get the new state of the contract. An invoke transaction is created with the invocation info and hash of the new state listed in the output section of the transaction. This transaction is later passed to the set of endorser servers chosen at the deployment time to see if they endorse the state transition. If an endorser had verified that the invocation is successful and the new state hash is consistency with the state hash provided by the contractor server, it signs the transaction with its private key.

Chapter 4 Smart Contract

In our implementation, smart contracts are written in Java. Every contract in our architecture is a Java class with public methods to be “invoked”.

4.1 Contract State

Since contract is a Java class, it most probably has some class members defined. Class members can be primitive types like *int*, *float*, *boolean* or simply other java classes. For a contract written in java, we say its state to be the values and states of all class members. As an example, if a contract class has an *int* variable n with value 0, when it’s value is set to 1, then we say that that state of the contract has changed.

How to preserve the state of the contract (or the class) is an important issue, for the state has to be kept and recovered for further invocation.

4.2 State Serialization

To preserve the state of the smart contract written in Java class, the class has to be serializable so we can write the serialized object to an output stream. A class should implement *java.io.serializable* interface before it can be serialized or deserialized; the serialized state is a collection of bytes that can be deserialized to Java object.

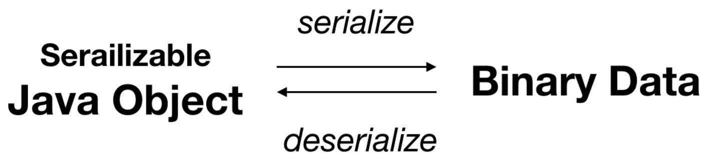

Figure 2: Java serialization, deserialization

Once the contract state has been serialized, we apply SHA-256 hashing algorithm to the bytes of the serialized state to get the *state hash*. A slight modification to the serialized state will generate a different state hash, thus we can use the state hash to check if two states are identical. In our implementation, the contract states are saved as files with their stash hash converted to hex format being their filename.

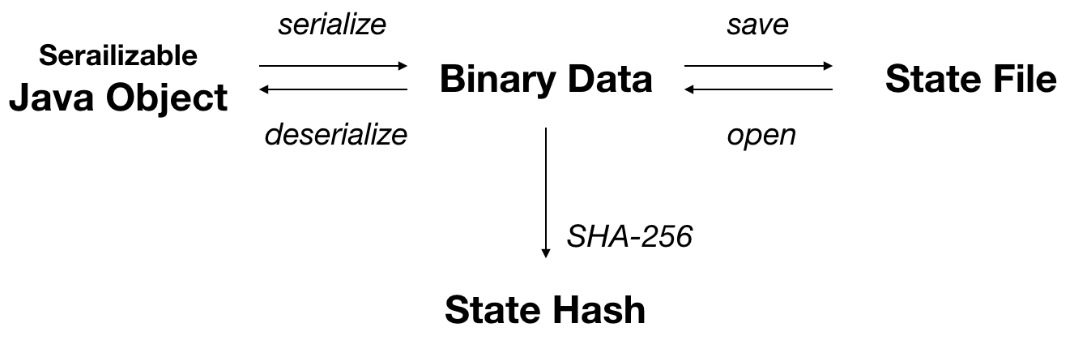

Figure 3: State File and State Hash

The process and the format of the serialization is not a part of this thesis and can be referred to the Java language specification [6].

Chapter 5 Contractor

In Chapter 3, we have briefly discussed the role a contractor plays in the smart contract architecture we proposed. In this chapter, we will delve into the detail implementation of the contractor.

The main idea of a contractor is to facilitates the process of creating and invoking contracts for the user. Without the help of a contractor, a user is forced to interact with the Gcoin directly, which could be an overwhelming thing to do. For one, the user must possess the basic knowledge of the blockchain to interact with it effectively. There are two operations on a contract: deployment and invocation. Contract deployment and invocation are considered successful only when the Gcoin network have the transaction recorded on the blockchain, reflecting these two operations.

5.1 Deploy Contract

In our implementation, when a user give contractor a jar file containing smart contract code to a contractor, the contractor will go through a five-step process to deploy the contract. First, we have to acquire public keys from user-selected endorsers to generate a unique multisignature address that is used to identify the contract. Second, deploy the contract to get the state file and state hash. Third, build a Gcoin transaction containing contract deployment information. Fourth, send the transaction built in step two to the endorsers. Lastly, broadcast the transaction to Gcoin network.

5.1.1 Generate multisignature address

Generating a unique multisignature address might seem a trivial thing to do, yet there is a limited multisignature addresses for any N public keys. An M of N multisignature address is formed by combining N public keys in any order in the script. For example, there are only two standard 1-of-2 multisignature script that consist of two public keys shown below:

```
1 <public key A> <public key B> 2 OP_CHECKMULTISIG
```

```
1 <public key B> <public key A> 2 OP_CHECKMULTISIG
```

To generate different multisignature addresses for different contract, we append the hash of the contract deployment state and an OP_DROP op code at the end to get a non-standard multisignature script. OP_DROP basically tells the scripting system to drop the stash hash, so the behavior of the script is not changed. The modified multisignature scrip looks like:

```
1 <public key A> <public key B> 2 OP_CHECKMULTISIG <contract hash> OP_DROP
```

Recall that multisignature address is the hashed result of multisignature script, with the state hash appended to the end of the script, we can generate different multisignature addresses for different contract.

5.1.2 Deploy contract

To deploy a contract, a user has to provide the jar file containing Java classes and the name of the contract class. Contractor uses the contract class name to construct an instance of the contract using the no-argument constructor. When deploying, only the constructor is called, then the instance is serialized immediately afterward. The state hash is also calculated for next step.

5.1.3 Deploy transaction

A Gcoin transaction is then created by the contractor. The only input should be an UTXO record from any other transaction output. This indicates the address owner of the

UTXO is the contract creator. In the output section, two parts are listed: multisignature address generated from the first step and an OP_RETURN script.

The multisignature address is used as the identifier of smart contract, it will be associated with the smart contract put in the second part of the output. To subsequently invoke the contract, this multisignature address output should be used as another transaction's input.

The OP_RETURN script contains serialized key-value pairs which are smart contract code in jar file format, the name of the main contract class and the initial state hash. The state hash should be generated by the contractor and must be included to show what the state a contract should have at deployment time. Smart contract state hash is generated by loading the contract code then serialize it without executing any part of the contract code.

Table 1: OP_RETURN data (deploy)

Key	***Value***
contract code	jar file
entry class	entry class name
state hash	state hash at deployment time

5.1.4 Send deployment information to endorsers

After creating the transaction, this transaction should be sent to all the endorsers consist in the multisignature address. When an endorser receives the transaction, it has to deploy the contract just like contractor did and check if the state hash calculated by the endorser is identical to the state hash provided by the contractor in the transaction. Only when all the endorsers have agreed that the contract state is indeed the same, the deployment is considered valid.

5.1.5 Broadcast transaction

The final step of deploying transaction is to put the transaction containing the deployment on the Gcoin network. Once the transaction has been broadcasted and successfully accepted in the blockchain, the deployment of the contract is completed.

5.2 Invoke contract

When a transaction containing smart contract deployment information has been accepted by the Gcoin, the smart contract is ready for invocation at any time. To invoke a contract, contract user must provide the contract's multisignature address, the method name and the method arguments to the contractor. After receiving invocation information, contractor finds the smart contract corresponding to the multisignature address, then it goes through four steps to invoke the contract. First, call the method with the provided arguments and get the new smart contract state. Second, build a Gcoin transaction containing contract invocation information. Third, send this transaction to the endorsers chosen at the deployment time. Lastly, broadcast the transaction to Gcoin network.

5.2.1 Invoke contract

A deployed contract on the Gcoin is packed as jar file with one class designated as the entry class. The entry class of the smart contract can have multiple methods, and only those methods can be invoked. In order to determine which method to be called, the contract user must provide the name of the method. In Java, overloading method is possible, which means that multiple methods can have identical name as long as their method signatures are different, so the invoker should also provide correct arguments to be passed to the method.

When the contractor received all the information from the contract user, it loads the contract code to an instance of JVM by using class loader and actually calls the method. The method can possibly change the value or state of class members, thus changing the state of the contract. No matter if the state of the contract has changed, the final state after calling the method is serialized and hashed to get the state hash.

5.2.2 Invoke transaction

Just like deploying contract, contractor also prepares a transaction with invocation information. The transaction contains only one input that links back to the multisignature output of a contract deployment transaction or another contract invocation transaction. Two outputs are put into the transaction: the multisignature address of the contract and an OP_RETURN script containing the invocation information. The invocation information is serialized key-value pairs contains 4 values: the method to be invoked, the arguments for the method, previous state hash that this invocation depends on and the state hash after invocation.

Table 2: OP_RETURN data (invoke)

Key	***Value***
invoke method	the name of method to be invoked
method arguments	a list of arguments
previous state hash	hash of state this invocation depends on
state hash	state hash after invocation

5.2.3 Ask for endorsement

It is not enough if only the contractor be the only party that executes the contract and says the contract is in the correct state. The input of the transaction is locked in the multisignature address that only when enough endorsers have signed the transaction can the input be used. The transaction with invocation information is later sent to every endorser, then the endorser executes the contract accordingly. When endorser completes the contract invocation, it compares the resulting state hash to the state hash provided by the contractor, then the endorser returns its signature to the contractor if two state hashes match.

5.2.4 Broadcast transaction

The invocation transaction input contains a multisignature script that requires enough signatures from the endorsers before the transaction can be broadcasted to the Gcoin network and be accepted. Once the transaction is accepted by the blockchain, its state is fixed as the OP_RETURN shows.

Further invocation can use the output of this transaction to create a new one. Since each contract invocation has to use the previous transaction output as the input of new transaction, a chain of "invocation" is formed, thus ensures the invocation order is maintained.

Chapter 6 Endorser

Endorsers are mainly responsible for ensuring the contract state consistency with contractor. In reality, endorsers should be an unbiased third party who should only give transaction signature when the invoke transaction is valid. When deploying contract, contractor also asks endorsers for their public keys to generate a multisignature address for the contract.

6.1 Generating Public Key

Gcoin use the same elliptic curve cryptography as Bitcoin to generate public key from a 256-bit private key. There are many libraries implemented the Bitcoin elliptic curve cryptography that can be used to generate public keys. However, the implementation of the cryptography is out of the scope of this thesis. The only requirement is that the endorser should keep the private absolutely safe, so no imposture can use the private key to sign the transactions.

6.2 Deploy and Invoke Contract

Either deploying or invoking contract, the endorser has to receive an unsigned transaction from a contractor. Both deployment and invocation information are recorded in the OP_RETURN part of the transaction output, and the information has to be extracted before contract can be deployed or invoked.

6.2.1 Deploying contract

When asked to deploy a transaction, endorser is given a transaction with deployment information mentioned in Chapter 5.1. The information contains contract jar file, the name of entry class and the state hash of the deployed contract. The main job of endorser is to actually deploy the contract again and check if the deployed contract stash hash is identical to the stash hash in the deployment information.

To get the deployment state of the contract, we first have to load the entry class by using *java.net.URLClassLoader*. *URLClassLoader* is used to load the entry class from the jar file and instantiate an object of the entry class. The entry class instance is checked if it has implemented the *java.io.Serializable* interface. If the instance is serializable, it is serialized right after the instantiation; otherwise, if the instance is not serializable, the deployment process will abort. In our implementation, we destine the serialization to a file on the filesystem with the state hash as file name. The state hash is derived by applying SHA-256 to the serialized data. Endorser should keep this file as long as possible, for next contract invocation depends on this state. Finally, the state hash and the one in the deployment information are compared to check if the state of the contract matches the state that contractor had deployed before.

6.2.2 Invoking contract

When asked to deploy a transaction, endorser is also given a transaction but this time with invocation information mentioned in Chapter 5.2. To invoke the contract, the endorser has to restore previous contract state before it can continue the invocation. State files are saved in the filesystem with state hash as their filename. Endorser can use the previous state hash in the invocation information to locate the file then deserialize it to get the instance of contract class. The method name and the arguments is then extracted to execute on the instance. After the execution, endorser should also keep the newly generated state file after invoking the contract, for next contract invocation may depend on this state. No matter if the contract state had been changed after executing the method, its state is serialized back to state file and its hash is also compared with the state hash in the invocation information that contractor created.

6.3 Signing Transaction

There are many libraries out the implementing Bitcoin transaction signing procedure, which is out of the scope of this thesis. Any library that can sign a Bitcoin transaction is compatible with Gcoin so using any of those libraries is suffice. The only two requirements for the endorser are (1) it has to sign the transaction with the exact private key that was used to generate the public key given to the contractor at contract deployment time (2) it signs the transaction only if the contract state hash that the endorser produced matches with the state hash from the contractor.

Chapter 7 Conclusion and Future Work

7.1 Conclusion

In this thesis, we have implemented a smart contract architecture on Gcoin blockchain without modifying any source code of Gcoin. By introducing contractor server and endorser server, we make Gcoin blockchain as the carrier of smart contract states. The state once recorded on the blockchain are open for anyone with access to Gcoin to verify. We exploit the fact that, in Gcoin, each transaction input depends on a transaction output, so we are able to put contract deployment and invocation info into OP_RETURN script along with a multisignature address locking script into transaction output, and use the multisignature address to enforce the smart contract state dependency.

7.2 Future Work

7.2.1 Halting problem

Halting problem is the problem to decide if a program will run forever or finish running. Halting problem has been proved unsolvable by Alan Turing. Both to contractor and endorser, smart contract is arbitrary code written in Java, so they cannot know in advance if the contract execution will stop at some point or run forever. If a malicious user writes an infinite loop in the smart contract code, the contract will run forever, which could waste a lot of computation power on the computer. The simplest and effective workaround is to set a limit on the execution time, yet it is not an elegant solution. What if a contract takes longer time to execute but it will eventually terminate? The halting problem is an interesting topic worth looking into, but unfortunately it is not the main focus in this thesis.

7.2.2 Contract state sharing

In our current implementation, contract states are stored in filesystem as files. When invoking a contract, the state of the contract has to be deserialized from the state file, which means that if the state file is missing or corrupted, contract invocation become impossible. A contract state file sharing mechanism can be implemented so if an endorser needs a certain state file, it can ask other endorsers for the state file.

Bibliography

[1] N. Szabo, "Formalizing and Securing Relationships on Public Networks," 1997.

[2] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.

[3] "Script," [Online]. Available: https://en.bitcoin.it/wiki/Script. [Accessed 22 Jul 2017].

[4] A. M. Antonopoulos, "Base58 and Base58Check Encoding," in *Mastering Bitcoin*, 1st Edition ed., O'Reilly Media, 2014, p. 72.

[5] Gcoin Community, "Gcoin white paper," 11 Jul 2016. [Online]. Available: https://github.com/OpenNetworking/gcoin-community/wiki/Gcoin-white-paper-English. [Accessed 22 Jul 2017].

[6] Oracle, "Java Object Serialization Specification," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html. [Accessed 24 Jul 2017].