TURING COMPLETE BITCOIN SCRIPT

White Paper

CRAIG WRIGHT, NCHAIN

Electronic copy available at: https://ssrn.com/abstract=3160279

Contents

Electronic copy available at: https://ssrn.com/abstract=3160279

Executive Summary

Bitcoin scripts currently do not enable loops. Therefore, they are commonly considered to be not Turing Complete. This limits the types of algorithms the Bitcoin scripts can execute to linear or tree-like instructions.

This white paper describes the invention of an Oracle module that:

- Continuously monitors the state of Blockchain.
- Makes decisions based on the current state.
- Generates the next set of transactions to be written onto the Blockchain.

The above processes is executed continuously by the Oracle in parallel to the Blockchain network. It therefore implements looping constructs that makes the combined Oracle and Blockchain system Turing Complete.

As old blocks cannot be erased from the Blockchain, but new blocks can be added, the Oracle and the Blockchain will work as a non-erasable Turing Machine as described in Minsky et al. (1967).[1]

We will discuss the use of the Oracle in applications including:

- Distributed data storage
- Distributed computing
- The control of drones.

We will also describe how metadata storage and digital signature authorization on the Blockchain can be useful features for implementing these applications.

[1] Marvin Minsky (1967), Computation: Finite and Infinite Machines, Prentice-Hall, Inc. Englewood Cliffs, N.J.

Functional Specification

Digital entrepreneurs have started exploring both the use of the cryptographic security system Bitcoin is based on, and the data that can be stored on the Blockchain, to implement new systems. These include but are not limited to:

- Storing metadata
- Implementing digital tokens
- Establishing contracts that are signed with digital signatures.

One such area is Turing Complete, but it is widely thought that the Bitcoin scripting language is not Turing complete. For example, that it does not allow loops to occur. It is therefore not considered well suited for controlling automated tasks.

A main advantage of limiting the Bitcoin scripts to linear or tree-like decision tasks, is that this avoids infinite loops. These can be used as a means of launching a denial of service attack. As such, this limitation does mean Bitcoin scripts are often limited to linear tasks.

A solution that enables looping to be integrated with Bitcoin scripts, yet avoid the damaging effects of infinite loops would have many benefits. These include:

- Enabling the automation of complex Bitcoin-related transactions.
- Controlling the metadata stream that is recorded onto the Blockchain.

This paper will explore how the Blockchain can be coupled with a new system called the Oracle. This simulates loops outside of the typical Bitcoin script. We will explore some examples of the application of the Oracle system on automated tasks relating to distributed data storage, distributed computing and the control of drones. These applications will use the Blockchain for metadata storage, managing digital tokens and establishing contracts.

Technical Specification

In this section we look at the technical detail behind making the Blockchain Turing Complete. This includes:

- Making the Blockchain act as a non-erasable tape of the Turing Machine.
- The function of (and implementing) the Oracle.
- The Oracle acts as the instruction table of the Turing Machine.
- Managing the Oracle by a code registry.
- Transaction metadata relating to the Oracle's code and respawning the loop
- Using digital signatures to implement software updates to the Oracle
- A special Oracle implementation using an alternate Blockchain.

The Blockchain as the Turing Machine's Non-erasable Tape

We define the Blockchain as a non-erasable tape (see Figure 1) with the following definitions of the Turing machine:

1. Like a Turing Machine, the Blockchain acts as the tape. Each transaction represents a cell on the tape. This cell can contain symbols from a finite alphabet.
2. The tape head can read information from the blocks that have already written onto the Blockchain.
3. The tape head can write new blocks, containing many transactions, to the end of the Blockchain. However they cannot write onto blocks that already exist. As such, the Blockchain tape is non-erasable.
4. Metadata for each transaction can be stored as part of a multi-signature pay-to-script-hash transaction.

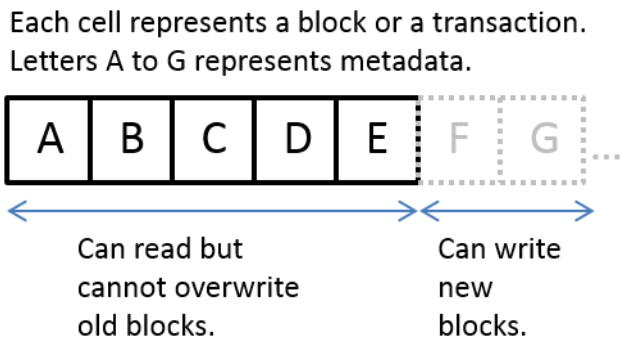


Figure 1: The Blockchain as a non-erasable tape for the Turing machine

Minsky et al. (1967) describes how a non-erasable tape can be used to implement a machine that is Turing complete, and is able to execute any algorithm that can also be executed on a Universal Turing machine. We will describe the invention of an Oracle function that uses the Blockchain as the non-erasable tape and implement this non-erasable Turing machine.

The Oracle's Function and Implementation

The function of the Oracle is to act as an agent that monitors the current state of the Blockchain. It then decides on the next set of transactions to be broadcasted to the Bitcoin network, and written to the Blockchain. It will run in parallel and simultaneously to the Bitcoin network and extend the function of Bitcoin script. This continuous monitoring implements the 'loop' control-flow constructs making the combined Oracle and Blockchain system Turing Complete.

This software will be open-sourced and runs peer to peer (P2P). For example, like the Onion Router (TOR) and Bitcoin network.

Note: The Oracle can also include innovations from other patents held by the company or intellectual property that we are yet to disclose.

The Oracle as the Turing Machine's Instruction Table

The Turing Machine will include two stacks:

- Data stack: This is represented by the Blockchain as described above.
- Control stack: This is represented by the Oracle function. It will store information relating to the repeat control-flow function.

The separation of the control track from the data stack will prevent infinite loops from occurring within the Bitcoin core. This in turn mitigates denial-of-service attacks on the Bitcoin system.

The Oracle manages and runs subroutines that are able to loop via any type of loop construct (e.g. FOR-NEXT; REPEAT UNTIL;). We describe the process using one example of the 'repeat' construct (see Figure 2). The user specifies the index (*i*) and the limit (*J*). These represent the current number of iteration starting from 0, and the total number of iterations of the repeat loop respectively.

For each iteration the:

1. Index increments by 1. For the exit condition, the iterations will stop when the index reaches the limit.
2. Code block containing an "if this then that" (IFTTT) control flow statement is executed.
3. A cryptographic hash of this subroutine is computed. This can be stored as Blockchain as part of the transaction. Since the hash is unique to each code, it will enable verification of which code has been used.

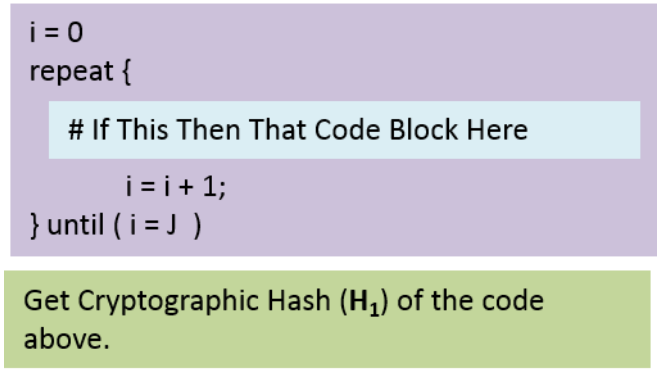


Figure 2: The Oracle subroutine implements the repeat 'loop'

Each code block will contain a "if this then that" (IFTTT) control flow statement (see Figure 3). This monitors the current state of the Blockchain for transactions matching the:

- Start or triggering condition (i.e. when a particular Bitcoin address reaches 10 BTC).
- Repeat condition (i.e. a metadata or hash associated with the previous iteration).
- Stop condition (i.e. last iteration of the loop).

Based on the current state, the IFTTT statement decides on the next transaction to make. This involves broadcasting the transaction onto the Bitcoin network, and writing the new transaction onto the Blockchain. This acts as a record that this iteration has been executed. Once the transaction has been written onto the Blockchain, the Oracle will find that the previous iteration has been executed and written onto the Blockchain, and executes the next iteration. The latter continues until the repeat loop exits when the index (*i)* reaches the limit (*J*) specified in the code block.

```
# If This Then That Code Block
If block state == 'D' then
   write 'E' onto Blockchain
Else if block state == 'E' then
   write 'F' onto Blockchain
Else if block state == 'F' then
   write 'G' onto Blockchain
END
```

Figure 3: IFTTT code block example

The 'If This' section of the IFTTT code block can monitor any conditions. This is similar to other programming languages (e.g. C, C++, Java) and not limited to information on the Blockchain. Some example conditions is listed below:

- Monitor the date and time (i.e. when a certain date and time is reached).
- Monitor the weather (i.e. when the temperature is below 10 ^{O}C and is raining).
- Monitor social media (i.e. when I've received a message from my friend).
- Monitor conditions of a contract or a trust (i.e. when company A buys company B).
- Monitor news and events (i.e. when soccer team A wins a match).
- Monitor information from the internet of things (i.e. a light bulb needs replacing).
- Monitor data from a mobile / wearable device (i.e. a Fitbit counts 10000 steps).
- Monitor results from cloud computing (i.e. computation completed and results received).
- Monitor remote data storage (i.e. file still exists remotely).

The 'Then That' section of the IFTTT code block can execute a number of actions. The action is not limited to a transaction on the Blockchain, but a transaction containing metadata related to the action will need to be written on the Blockchain. The metadata can be of any form specified by the Oracle. However a useful way is to store the hash of a hyperlink to a file containing more data relating to the action. A list of example actions is listed below.

- Bitcoin transactions (i.e. send Bitcoins to a particular address).
- Social media (i.e. send a message to a friend).
- Trading (i.e. sell X shares).
- Internet of things (i.e. switch off a light bulb).
- Commerce (i.e. purchase an item online).
- Online services (i.e. pay a monthly fee or pay for services requested using Bitcoin).

The Oracle's control stack can be implemented in a number of ways that are specific to the needs of each user. For example, the repeat loop of the control stack can be based on any Turing Complete language. One possible choice of language is the Forth style stack-based language. This will keep the control stack consistent in programming style with the Bitcoin scripts. The Forth technical specifications will describe details on how repeat loops can be implemented in a Forth style language. [2]

The Bitcoin Script's Alternate Stack as a Data Storage Space

The Bitcoin script contains commands, also called op codes, which enable the users to move data onto an alternative stack, also called the alt stack. [3] The op codes are:

- OP_TOALTSTACK which moves data from the top of the main stack onto the top of the alt stack.
- OP_FROMALTSTACK which moves data from the top of the alt stack to the top of the main stack (Figure 4).

[2] Leo Brodie (1987) Starting Forth

[3] See https://en.bitcoin.it/wiki/Script#Stack for further details.

This enables data from intermediate steps of the calculations to be stored in the alt stack, similar to the 'memory' function which allows data to be stored on the calculator. The alt stack will be used for configuring bitcoin scripts to solve small computation tasks and returning the results in the computation.

Word	Opcode	Hex	Input	Output	Description
OP_TOALTSTACK	107	0x6b	x1	(alt)x1	Puts the input onto the top of the alt stack. Removes it from the main stack.
OP_FROMALTSTACK	108	0x6c	(alt)x1	x1	Puts the input onto the top of the main stack. Removes it from the alt stack.

Figure 4: The bitcoin commands that allow users to move data in and out of the alternative stack.

Managing the Oracle by a Code Registry

The Oracle also manages a registry of all the codes that it owns and runs. This registry is structured like a lookup table or dictionary that maps a specific key to a specific value (see Figure 5). The key and value pair is represented by the hash of the code block (H_1) and the IPv6 address of where the code is stored respectively. To retrieve the code block using the key H_1, the lookup table is used to retrieve the associated value (this is the location where the code is stored) and retrieves the source code accordingly.

The implementation of the code registry can vary. For example, the lookup table can be implemented using a locally managed list, or a P2P distributed hash table. The source code can be stored locally, remotely, or using a decentralized file storage system.

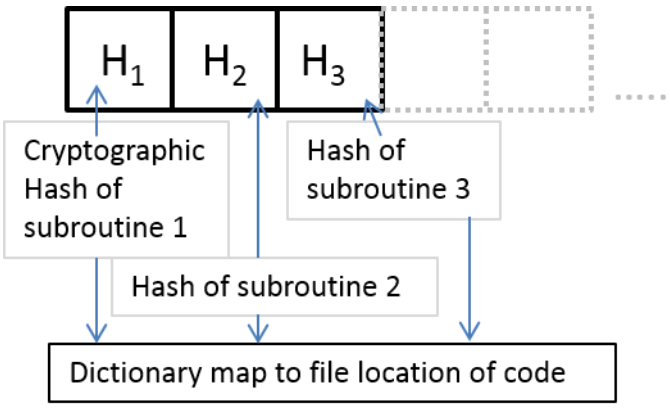


Figure 5: Oracle's code registry

Note: This section will be expanded in a future patent White Paper.

Transaction Metadata of the Oracle's Code and Respawning the Loop

All information required to respawn the Oracle's loop at a particular iteration, is stored as metadata in the transaction recorded on the Blockchain (see Figure 6 and Figure 7).

Field	Subfield	Bytes	Value	Comments
Metadata-CodeHash	CodeHash	20		RIPEMD-160(SHA256(the actual code file addressed by CodePointer)).
	Index	12		Integer index *i* denoting which iteration of the loop is at.
Metadata-CodePointer	CodeType	4	0x00000001	Indicates it is an Oracle code that performs loops.
	CodePointer	16		IPv6 address of the actual code file location.
	Padding	12	0x00000…	Spare bytes.

Figure 6: Metadata associated with the Oracle's code block

Field	Subfield	Bytes	Value	Comments
Metadata-OutputHash	OutputHash	20		RIPEMD-160(SHA256 (the output metadata file addressed by OutputPointer)).
	Padding	12	0x00000…	Spare bytes
Metadata-OutputPointer	OutputPointer	16		IPv6 address of the metadata relating to the output of the code.
	Padding	16	0x00000…	Spare bytes.

Figure 7: Metadata associated with the output at a particular iteration

The metadata itself will be stored as part of a multi-signature pay-to-script-hash script (P2SH) in the transaction. See Figure 8 for the script's format.

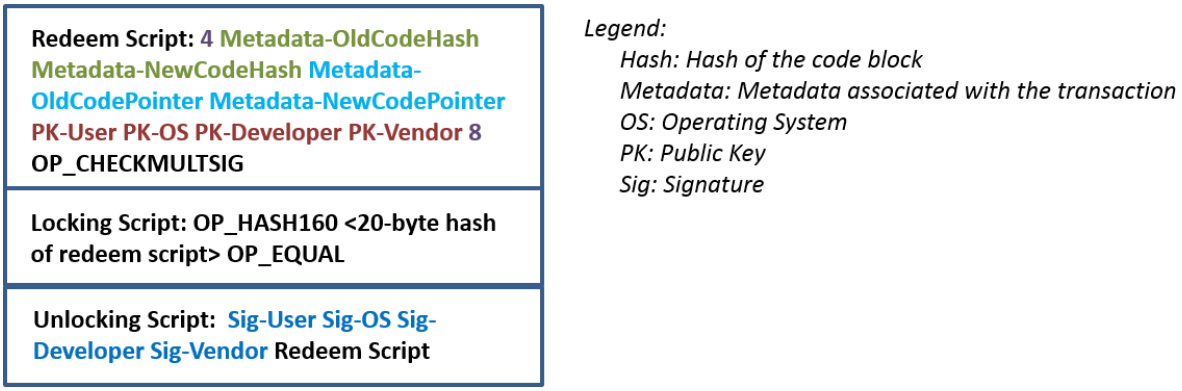


Figure 8: Transaction Script and Metadata

The metadata recorded with the transaction also gives the ability to record an audit trail of how the code has been executed in the past. The following steps provide details on how the Oracle respawns a repeat loop code block at the i^{th} iteration.

1. The Oracle will monitor the Blockchain for transactions that contain hashes of the code block that matches entries in the code registry.
2. The Oracle finds a transaction that contains the corresponding hash (H_1).
3. The Oracle reads the 'Metadata-CodeHash', get the CodeHash field to get H_1 and uses it to retrieve the code (C_1). If RIPEMD-160(SHA256(C_1)) equals H_1, the code has not been changed and we can proceed to the next step.
4. The Oracle reads the 'Metadata-CodeHash' which stores the index *I*, and respawn the code at the i^{th} iteration.
5. The signature of the User is included in the P2SH command to verify the origin of the metadata.
6. The Oracle reads the 'Metadata-OutputHash' and 'Metadata-OutputPointer' (see Figure 6) to retrieve the output of the previous steps, if these data are required for this iteration of the loop.

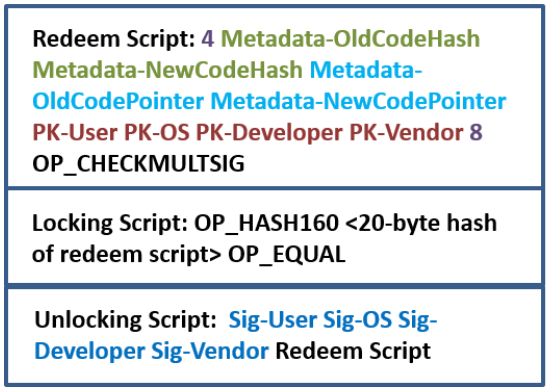
Multiple signatures are required to unlock the transaction (e.g. the User, the Operating System, the Software Developer and the Software Vendor). This provides a digital rights management (DRM) system for managing the rights to operate the codes by all parties involved in the P2SH transaction.

Use Digital Signatures to Implement Updates / Patches to the Oracle's Code

Software updates and patches for codes that reside in the Oracle are securely authorized using a multi-signature P2SH transaction (see Figure 8). The multi-signature transaction records metadata of the old and new codes as shown in Figure 5. This makes a record of the changeover of the old code to the new code, thereby providing an audit trail of the software update. The Oracle will need to store all hashes of the old and new blocks of source codes. The hash of the new and old source codes can be used these to verify the integrity of the code files.

Note: Multiple signatures are required to unlock the transaction (e.g. the User, the Operating System, the Software Developer and the Software Vendor). This provides a DRM system for managing software updates and patches for codes that are used by the Oracle.

Unlike most software, which does not allow software to be updated while it is running, an advantage of the Oracle system is that software updates can occur in the middle of executing a loop. The information captured on the Blockchain (see Figure 8 and Figure 9) can be used to update to the new code in the middle of a loop, and start the next iteration step using the output metadata from the previous iteration from the old code.



Legend:
Hash: Hash of the code block
Metadata: Metadata associated with the transaction
OS: Operating System
PK: Public Key
Sig: Signature

Figure 9: Oracle software patching verification and audit trail

Note: This section will be expanded in a future patent White Paper.

Applications and Case Studies

The current Bitcoin scripting language does not allow loops to take place. This prevents using Bitcoin payments to trigger continuous and automated actions without external interference. However with the Oracle continuously monitoring information on the Blockchain, it allows automated actions based on up-to-date information on the Blockchain. This section illustrates how the Oracle's control stack can be used to automate processes involving an automated and online vote counting bot.

Controlling a Vote Bot

The vote counting bot is designed to facilitate fair and pseudo-anonymous voting, with the Blockchain recording an audit trail of the vote counting process. The vote counting bot is automated using the Oracle's control stack and repeat loop (see Figure 10). The following scenario illustrates how it works.

We have 100 voters. If 57 unique "Yes" votes are received before 1st January 2016, payments will be released to the Chair, Jason. The voting process is divided into two parts:

- Token distribution
- Counting

For the token distribution, 100 voting tokens are distributed to each authorized voter. Each token is represented by a Bitcoin public key and private key pair. This is distributed to each voter using a secret exchange protocol. Each Bitcoin public key and address will be loaded with a small amount of Bitcoin representing one vote. The bot will keep the list of public key associated with each authorized token and make this list public before voting begins. To ensure voting cannot be rigged and that voting is anonymised, the list of private keys and the mapping between the voter's identity and their token will be destroyed.

For the counting, the Oracle will run a repeat loop. The list of addresses will be kept in the bitcoin script, and transferred to the alternate stack for storage of data. Once an address has been counted, it will be removed from the alternate stack and no longer added to the next transaction. The repeat loop will stop when the list of address becomes empty.

Instead of using the integer index *i* to keep track of where the loop is currently at, the vote bot Oracle will use it to store the intermediate value of the vote count. This ensures the intermediate value of vote count is stored in the blockchain. This provides an audit trail, and shows that the vote counting process if fair.

If the amount of unique "Yes" votes received reaches 57, we will pay the agreed amount of Bitcoins to Jason's account. The cryptographic hash of the vote counting script, and the IPv6 address of where this script is stored, are released to the public. This means that the public will have enough information to perform a recount, and ensure the vote counting process is fair and correct.

 # Voting Counting Bot Example Repeat Loop
 # The conditions of the vote
We have 100 people and if we have 57 yes votes by 1st January 2016, we will release payments to Jason.

 # Part 1. Run the send vote tokens bot

i. Generate 100 Bitcoin public keys and fill each with a small amount of Bitcoins.
ii. Each address represents an authorised token which must be used to cast a vote.
iii. Securely send the token and the corresponding private key to each person entitled to vote.
iv. The bot will keep the list public key associated with each authorized token, and make this list public before the vote.
v. To ensure voting cannot be rigged, destroy the list of private keys.
vi. To ensure voting is anonymous, destroy any association of token with voter's identity to anonymise them.

 # Part 2. Run the vote count bot

```
Array_of_tokens = [list of 100 authorized tokens];
Yes_count = 0;
repeat {
        this_token = pop from Array_of_tokens;
        if ( signature received from this_token and
            current date < 1st January 2016)
                Increment Yes_count;
        }
} until (Array_of_tokens is empty)

if ( Yes_count >= 57 ) {
                Pay to Jason's Account.
}
```

Figure 10: The vote counting bot's repeat loop in pseudocode

Additional References

- Code Money Have Fun: LOOPTY DO I . LOOP ; blog post