COSC-373 Final Report: Bitcoin

Lee Jiaen, Hyery Yoo, and Alexander Lee

May 20, 2022

1 Introduction

Before Bitcoin, e-commerce on the internet relied heavily on trusted third parties to process electronic payments. For example, suppose Alice, a buyer, wants to purchase an item from Bob's e-commerce store. Alice would need to make an online payment that relies on a financial institution or some trusted third party to process the transaction. However, Alice may not trust such third parties and wish for a way where her online payment can be sent directly to Bob without going through a third party. While the previously mentioned payment system works for most transactions, it has a few significant problems with efficiency. Trusted third-party financial intuitions cannot avoid mediating disputes, making it impossible to have completely non-reversible transactions. Therefore, the need for trust increases, and "a certain percentage of fraud is accepted as unavoidable" [Nakamoto, 2008]. One could avoid these issues with the use of physical currency. However, like Alice, many people would prefer a decentralized version of electronic cash that allows any two willing parties to transact directly without a trusted third party. This is the motivation behind Bitcoin. This report describes how Bitcoin, an electronic payment system, is based on cryptographic proof instead of trust, allowing for direct transactions without trusted third parties or financial institutions.

2 Preliminaries

To understand Bitcoin, we first need to develop an understanding of its fundamental building-blocks: digital signatures and cryptographic hash functions.

2.1 Digital Signatures

Suppose for instance that Alice wants to send a message to Bob over a network. How can Bob ensure that the message he received was from Alice and not from a malicious actor like Eve? What Alice can do is append a *digital signature* to her message and send the message and signature to Bob, allowing Bob to verify that the message he received was indeed from Alice.

The main idea behind digital signatures is as follows. Both Alice and Bob each generate a *private key* and *public key* pair for themselves, where each key is an array of bits and private keys are kept secret (i.e., only Alice knows her private key and only Bob knows his private key). Producing a signature involves a function sign(msg, priKey), which takes Alice's message msg and private key priKey as inputs and outputs Alice's signature sig. Similarly, verifying the authenticity of the received message involves a function verify(msg, sig, pubKey), which takes the received message msg, the signature sig, and Alice's public key pubKey as inputs and outputs *true* if the signature was produced using Alice's message and private key, and *false* otherwise. We will not discuss how the sign() and verify() functions work in depth since such details are not the focus of this report.

Digital signatures have two required properties. First, the authenticity of Alice's signature generated from her message and private key can be verified easily using her corresponding public key. Secondly, it should be computationally infeasible for someone like Eve to generate a valid signature for Alice without knowing Alice's private key. That is, Eve has no better strategy than guessing and checking if random signatures are valid using Alice's message and public key. Assuming a 256 bit signature, the aforementioned brute force strategy would require Eve to check 2^{256} signatures in the worst case.

2.2 Cryptographic Hash Functions

A *cryptographic hash function* is a hash function that takes as input data of an arbitrary size, known as the *message*, and outputs a bit array of a fixed size, known as the *hash*. In the case of the SHA256 cryptographic hash function, the hash has a length of 256 bits.

Cryptographic hash functions ideally should have the following properties. First, computing the hash of a given message should be quick. Second, generating the message from only the hash should be computationally infeasible. Third, finding two different messages that map to the same hash under the function should also be infeasible. Fourth, slightly changing the message should change the hash so much such that the old and new hashes seem uncorrelated. Similar to digital signatures, the only way to find a message that produces a given hash is via a brute force approach of guessing and checking possible messages. Again, assuming a 256 bit hash, this strategy would require 2^{256} hashes to check in the worst case.

3 Bitcoin

The *Bitcoin network* is a randomly connected overlay network of a few thousand nodes, where nodes are controlled by various owners and perform the same operation. In other words, the network is a homogeneous network without central control. Each user of Bitcoin generates a private/public key pair, and a user is identified by their *address*, which is derived from their public key and

used to receive funds in Bitcoin. All nodes in the Bitcoin network work together to track each address' balance in bitcoins.

4 Transactions

Suppose we have a transaction in bitcoins, where Alice is the sender and Bob is the recipient. What is a transaction in this case?

A *transaction* is a data structure that describes the transfer of bitcoins from senders to recipients. The transaction consists of several inputs and outputs. Each *output* is a tuple composed of an amount in bitcoins and a spending condition requiring a valid signature associated with the private key of an address. Meanwhile, each *input* is a tuple consisting of a reference to a previously created output and a signature to the spending condition, proving that the transaction creator has the permission to spend the referenced output. An output has two states, *unspent* and *spent*, and can only be spent at most once. The sum of the unspent outputs associated with an address is the *address balance*, and the set of these unspent transaction outputs (UTXO) is the shared state of Bitcoin. That is, every node in the Bitcoin network holds a complete replica of the shared state. The local replicas of this shared state may temporarily diverge, but consistency is eventually re-established. For a given transaction, the inputs result in the referenced outputs spent (i.e., removed from the UTXO), and the new outputs are added to the UTXO. More specifically, inputs reference the output that is being spent by a (h, i) tuple, where h is the hash of the transaction that created the output, and i is the index of the output in that transaction.

All transactions are broadcasted to the Bitcoin network and processed by every node that receives them. Transactions can be in one of two states: *unconfirmed* or *confirmed*. Incoming transactions are unconfirmed and added to a pool of transactions referred to the *memory pool*. We will discuss how transactions are confirmed in section 7.

Returning to our example, for the transaction between Alice to Bob, the inputs are Alice's previous unspent outputs and Alice's digital signature, which was created using her private key. Meanwhile, the outputs of this transaction are the amount in Bitcoins for Bob and the spending condition requiring a valid signature associated with the private key of Bob's address. The transaction is then broadcasted to the Bitcoin network and processed by every node that receives it.

5 The Doublespending Problem

Now suppose there is a transaction from Alice to Charlie that attempts to spend the same output as the transaction from Alice to Bob. Observe that, for now, nothing prevents nodes in the Bitcoin network to locally accept different transactions that spend the same output, i.e., doublespend. More formally, *doublespending* is a situation in which multiple transactions attempt to spend

the same output. However, only one transaction can be valid because outputs can only be spent at most once. Doublespending occurs because the order in which transactions are processed may not be the same for all nodes, and the validity of transactions depends on the order in which the transactions arrive. If a node sees two conflicting transactions (i.e., transactions that spend the same output), the node only considers the first transaction it sees valid and the second invalid. The second transaction seen is invalid since it tries to spend an already spent output. As a result, different nodes in the network may accept different transactions, making their shared states inconsistent. If the problem of doublespending is not resolved, the shared states of the Bitcoin network diverge. Therefore, a conflict resolution mechanism is needed to decide which of the conflicting transactions should be accepted by every node to achieve eventual consistency in the network.

6 Processing a Transaction

Before we continue with discussing the solution to the doublespending problem, we first summarize how a node processes an incoming transaction (see Alg. 1). First, the node receives a transaction t (line 1). Then, for each input (h, i) in transaction t (where h is the hash of the transaction that created the referenced output, and i is the index of the output in that transaction), if (h, i) is not in the node's local set of unspent transaction outputs or the signature for the output is invalid (line 3), then transaction t is dropped and the node halts. Next, if the sum of the values of inputs is less than the sum of values of the new outputs (line 5), then transaction t is dropped and the node halts since one cannot spend more than they have. Then, for each input (h, i) in transaction t, (h, i) is removed from the node's local set of unspent transaction outputs (line 8). Finally, transaction t is appended to the node's local history of transactions and t is broadcasted to its neighbors in the Bitcoin network.

Algorithm 1: Process Transaction

```
1  Receive transaction t
2  foreach input (h, i) in t do
3  |  if output (h, i) not in local UTXO or signature invalid then
4  |  |  Drop t and stop
5  if sum of values of inputs < sum of values of new outputs then
6  |  Drop t and stop
7  foreach input (h, i) in t do
8  |  Remove (h, i) from local UTXO
9  Append t to local history
10 Forward t to neighbors in the Bitcoin network
```

7 Proof-of-Work

Bitcoin solves the doublespending problem using *Proof-of-Work* (PoW), a mechanism in which one party can prove to another that it has spent a specific amount of computational effort by presenting a nonce x (a bit-string) to the PoW function $F_d(c, x) \rightarrow \{true, false\}$, where d is the difficulty (a positive number) and c is the challenge (a bit-string). Another party can confirm this PoW by quickly checking that $F_d(c, x) = true$ for the given x. For instance, suppose that Alice and Bob both know a PoW function $F_d(c, x)$ with d and c given. Alice expends a certain amount of computation to find a x such that $F_d(c, x) = true$. Alice presents her solution x to Bob, and Bob checks the solution by computing $F_d(c, x)$ and seeing that the functions output is $true$. The PoW function must be defined such that verifying a solution x is fast. Through this process, Alice proves to Bob that she has expended the amount of computational work specified by difficulty d.

What does PoW look like in the context of Bitcoin? First, we must understand the notion of a block in Bitcoin. A *block* is a data structure that consists of a list of transactions, a reference to a previous block, and a nonce. The list of transactions are the transactions the node has accepted to its memory pool since the previous block. Thus, a block encodes incremental changes to the local state of the node. A node finds a block by finding a valid nonce x that satisfies the Bitcoin PoW function:

$$F_d(c, x) = \text{SHA256}(\text{SHA256}(c|x)) < \frac{2^{224}}{d}.$$

This process is called *mining*. Simply put, a node must find a value x that, when concatenated with a node-specific challenge c and hashed twice with the SHA256 hash function, returns a hash that starts with a fixed number of zeroes. SHA256 is a cryptographic hash function, so a valid nonce can only be found by iterating over all possible values of x, requiring significant computational work. Once a node finds a valid nonce for the block, the block is broadcasted to the node's neighbors. Transactions contained in a block are said to be *confirmed* by that block. The process for finding a block is summarized in Alg. 2.

To make concept of PoW more concrete, suppose Alice is a node who puts in the computational effort to find a valid nonce x and broadcasts the block containing x to other nodes. Alice's PoW can be verified by another node Bob who sees Alice's block. Bob is given the values of c and x, so he can quickly compute the hash and verify that the result starts with the required number of zeroes.

8 Receive Block

A primary goal of the Bitcoin network is for all its nodes to agree on a single state of the network. To do this, the blocks build a tree with their reference to a previous block, where the tree's root is known as the *genesis block*. The

Algorithm 2: Find Block

Input: Nonce $x = 0$, challenge c, difficulty d, previous block b_{t-1}

1 **while** $F_d(c, x) = false$ **do**

2 $\quad x = x + 1$

3 Broadcast block $b_t = (memory\ pool, b_{t-1}, x)$

longest path from the genesis block to the leaf of the tree is called the *blockchain*. Eventual consistency is achieved by accepting the longest path in the tree (i.e., the blockchain) as the true state of the network. Branches are formed if a node receives two conflicting versions of a block. In this case, it works on extending the first block it receives and saves the other block as a branch. If the node later learns that the branch has become longer than the chain it was working on, it switches to extending the branch. Because every block requires PoW in order to be added to the chain, the longest chain in the network represents the greatest amount of collective computational effort, i.e., majority decision. All nodes eventually receive and accept the longest chain, so the network reaches consistency. More formally, Alg. 3 describes how a node updates its local state once it receive a block. We define the height h_b as the path length from the genesis block to block b. Namely, we denote the node's current head as block b_{max} with height h_{max}. First, the node receive block b (line 1). Then, the node connects block b in the tree as the child of its parent p at height $h_b = h_p + 1$. If the height h_b of block b is larger that the height h_{max} of the node's current head at block b_{max} (line 3), then the head block and height are updated appropriately, the node computes the UTXO for the path leading to the new head block b_{max}, and the memory pool is cleaned up.

Algorithm 3: Receive Block

Input: Node's current head block b_{max} at height h_{max}

1 Receive block b

2 Connect block b in the tree as child of its parent p at height $h_b = h_p + 1$

3 **if** $h_b > h_{max}$ **then**

4 $\quad h_{max} = h_b$

5 $\quad b_{max} = b$

6 $\quad$ Compute UTXO for the path leading to b_{max}

7 $\quad$ Cleanup memory pool

9 Network

To summarize, the Bitcoin network operates by each node in the network executing the same set of protocols. All transactions are broadcasted to the network, and every node works on collecting new transactions from the network into a

block and finding a valid nonce for the block. When a node finds the valid nonce, it broadcasts the block to the network, adding the block to the end of the chain in effect. When another node in the network learns about the new block, it checks that all transactions in the block are valid. If they are, it accepts the block and works on extending the chain by creating a new block with the hash of the accepted block.

10 Correctness

Let us now look at how the Bitcoin network solves the doublespending problem. Suppose Alice attempts to doublespend by creating two transactions containing the same output, and the two transactions are collected into two separate blocks, Block 1 and Block 2. If one of the blocks, Block 1, is broadcasted to the network and gets accepted by all nodes before Block 2 is broadcasted, Block 2 is rejected by all nodes because it contains an invalid transaction. Thus, Alice's doublespending attempt fails. In a more subtle case, suppose that Block 1 and Block 2 are broadcasted to the network (almost) simultaneously. Some of the nodes in the network receive Block 1 first, so they accept Block 1 and reject Block 2. The opposite is true for other nodes in the network. This creates two conflicting chains in the network with some of the nodes working on extending each of the chains. Suppose the majority of nodes accept Block 1. Then the sum of the computational effort put into Block 1's chain is greater than that for Block 2's chain, and Block 1's chain grows faster. Eventually, every node receives this longer chain, and Block 2's chain becomes rejected by all nodes, leaving only one of Alice's transactions in the network.

11 Conclusion

This report discusses the fundamental structure of Bitcoin and how it solves the doublespending problem. Other aspects of Bitcoin include incentivizing miners, reclaiming unnecessary disk space, and scripting. We refer the reader to learn more about these topics from the original Bitcoin white paper [Nakamoto, 2008] and lecture notes from ETH Zurich by Wattenhofer. Further work aims to increase the rate of transactions by establishing micropayment channels between two parties outside the blockchain network [Decker and Wattenhofer, 2015]. Moreover, the introduction of a decentralized system for financial transactions by Bitcoin led to a number of other blockchain-based cryptocurrencies such as Ethereum [Buterin, 2014].

References

Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform, 2014. URL `https://ethereum.org/en/whitepaper/`.

Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Symposium on Self-Stabilizing Systems*, pages 3–18. Springer, 2015.

Satoshi Nakamoto. A peer-to-peer electronic cash system, 2008. URL `https://bitcoin.org/bitcoin.pdf`.

Roger Wattenhofer. Chapter 20: Eventual consistency & bitcoin. In *Principles of Distributed Computing (Lecture Collection)*. 2015. URL `https://disco.ethz.ch/courses/podc_allstars/lecture/chapter20.pdf`.