UCSB CS 291D: Blockchains and Cryptocurrencies **Fall 2020**

Lecture 2: Bitcoin Primer

Lecturer: Shumo Chu *Scribes: Shiyang Li, Yuxiao Zhang, and Ishtiyaque Ahmad*

2.1 Building Digital Currency

In this lecture, we will discuss how we can build a digital currency. Let us first introduce a naive approach to do that. Later, we will build on that to obtain a more robust solution. Finally, we will have a brief overview of the design and architecture of Bitcoin.

2.1.1 Naive solution 1: Alice Coin

Let Alice is a banker whom everyone trusts. Alice has the authority to create new coins. However, we do not want to trust anybody other than Alice. Therefore, whenever we get a coin, it must be verified to be generated by Alice. One way of doing that is using digital signature. When Alice creates a new coin, she assigns a serial number SN to the coin and signs it with her secret key: $\pi :=$ Sign(SK_{Alice}, create_coin(SN))

After creation, Alice can spend this coin. Suppose, Alice wants to send this coin to Bob. To complete this transfer, Alice needs to ensure two things: i) She needs to sign the recipient ii) She needs to include a hash pointer of the coin she wants to send to Bob. So, the transfer signature will look like following: $\pi_1 :=$ Sign(SK_{Alice}, pay_to("Bob",Hash(π)))

The recipient, Bob can pass the received coin afterwards. Suppose Bob wants to send the coin to Charlie. Then Bob needs to follow the same procedure as above: $\pi_2 :=$ Sign(SK_B, pay_to("Charlie",Hash(π_1)))

One major limitations of this solution is *double spending*, i.e. the same coin can be spent more than once. In this example, Bob can send the same coin to two different persons as π_2^1 and π_2^2, both containing a hash pointer to π_1. Since we do not trust Bob, there is no way to prevent this.

2.1.2 Naive solution 2: Bob Coin

The root cause of failure for *Alice Coin* is its lack of *a consistent global state*. It is possible to make two different transactions to different people using the same coin, because, each of the recipients does not know the existence of the other one. We need a sequential and consistent global state in order to prevent double spending. We will develop such a scheme in this solution.

For this case, let us assume that we only trust Bob and nobody else. Bob will be responsible to store the consistent global state. Whenever someone wants know the current global state, she needs to query Bob. The global consistent state stored by Bob needs to have to properties: i) it should be a sequential log of events that happened ii) Each event in the log should be verifiable (this may be desirable even though we trust Bob). These two properties can be achieved by using a blockchain. Bob Coin can create a consistent global state called blockchain to fix this issue.

The structure of a blockchain is depicted in Figure **??**. It has a genesis block, which is a randomly generated data and stored by everyone. In this way, anyone can trace back to the genesis block. All other blocks have two parts: i) *header* and ii) *data*. The header contains some metadata about the block and a hash pointer to

the previous node in the blockchain. Data contains a set of transactions that belong to the block. A client only needs to store two pieces of data in order to verify the integrity of the blockchain i) The genesis block ii) A hash pointer to the most recent block. Any tampering to the blockchain by anyone, even Bob, can be detected with the help of these two pieces of information.

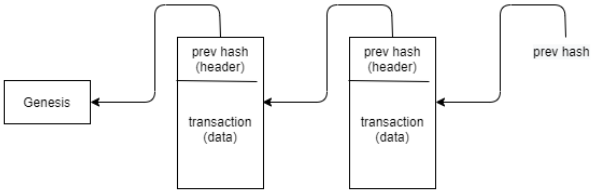

Figure 2.1: Structure of a blockchain (refer to Figure 1.11 on page 45 of [2])

2.1.3 Block Structure of Bitcoin

The header of a Bitcoin block contains several fields e.g. version, a hashpointer to the previous block header, timestamp of block creation, bits, nonce, and the root of a Merkle tree consisting of the transactions in the data block. The data block organizes all the transactions in a Merkle tree with each leaf of the tree containing a single transaction. The first transaction of a block is known as a *coinbase transaction*, which essentially creates new coins. The advantage of having the Merkle tree root in the header is, one can just keep a hash pointer to the previous block header. It is not required to hash the entire block, and thus saves computation. The structure of Bitcoin block is demonstrated in Figure **??**.

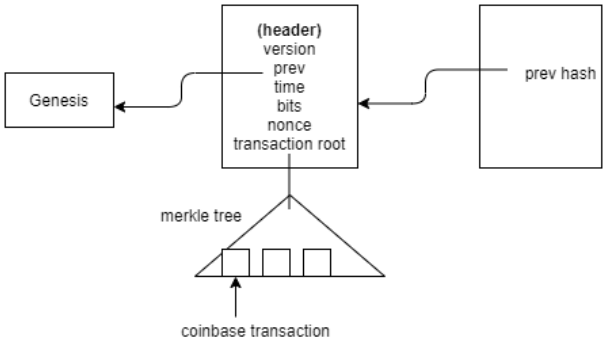

Figure 2.2: Block structure of Bitcoin

Below we introduce and discuss some key concepts related to Bitcoin.

- **UTXO:** Unspent Transaction Output. The UTXO model states that, a coin cannot be transferred, subdivided or combined. Whenever a coin needs to be sent to someone, the old coin is destroyed and a new coin is generated. For example, suppose Alice has 11 Bitcoins and she wants to send 9 Bitcoins to Bob. To perform this transaction, Alice first needs to destroy her 11 coins, send 9 coins to Bob, and send 2 coins to herself.
- **Structure of Transaction:** A transaction consists of an id, the type of the transaction, the set of input coins, and the set of output coins. The input coins are represented by the transaction id and

index in which it was generated. The output coins contain the recipient's address and number of coins to be sent to each recipient. Finally, a transaction must contain the signature of the owner of the consumed coins. The structure of a transaction is depicted in Figure **??**. A transaction is valid if the following three conditions hold:

1. Consumed coins are valid, i.e. it is owned by the signer and has not been spent yet.
2. Total value out = Total value in
3. Signed by the owner of consumed coins

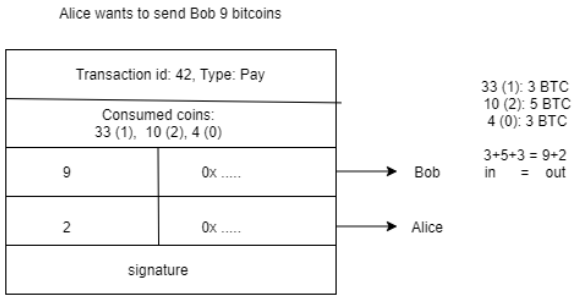

Figure 2.3: Structure of a transaction (refer to Figure 1.13 on page 47 of [2])

- **Bitcoin Script** A bitcoin script is a non-Turing complete script programming language that runs on a stack machine. It is designed to run in a very short period of time. Bitcoin script supports many programming language constructs and cryptographic primitives. A comprehensive list of the operations supported by Bitcoin script is available in [1].

- **Pay to Public Key Hash(P2PKH):** This policy actually defines how addresses are generated in Bitcoin. In order to send any coin, it must be sent to the recipient's address. In Bitcoin, the SHA256 hash of the recipient's public key acts as the address. Let us again consider the example where Alice has 11 Bitcoins and she wants to send 9 Bitcoins to Bob. The transaction takes place in the following steps:

  1. Bob generates his public key, secret key pair: $(\mathrm{PK}_B, \mathrm{SK}_B) := \mathrm{Generate}(1^\lambda)$
  2. Bob generates his address by taking hash of his public key: $\mathrm{Addr}_B := \mathrm{Hash}(\mathrm{PK}_B)$
  3. Bob sends Addr_B to Alice
  4. Alice creates the transaction

  After the transaction is complete, Bob can spend his UTXO referring to the transaction.

The following two design choices make the distributed ledger tamper-proof and more secure:

1. Alice specifies the address of Bob ($\mathrm{Hash}(\mathrm{PK}_B)$) in the transaction rather than Bob's actual public key.

2. No adversary can change the recipient address (Addr_B) to steal the fund, because the transaction is signed by Alice. Any such tampering will be detectable using bitcoin script.

References

[1] "Script - Bitcoin wiki" url: *https://en.bitcoin.it/wiki/Script*

[2] Narayanan, Arvind and Bonneau, Joseph and Felten, Edward and Miller, Andrew and Goldfeder, Steven, "Bitcoin and cryptocurrency technologies: a comprehensive introduction", 2016, Princeton University Press