2

TRANSACTIONS, UTXO MODEL

One of the most momentous characteristic of blockchain is **decentralization**. There is no third-party authority entrusted to store and mange users resources. Instead, a decentralized network of nodes share the same database with information about digital assets. Users have direct control over their properties, being able to transfer them to other users.

In a traditional payment system, as presented in fig 1, when Alice wants to buy a product from Bob, she first deposits some money to a Bank, the bank emits digital cash attesting Alice's account balance, then Alice transfers Bob some digital coins. Before Bob sends the requested product to Alice, he checks with the bank the validity of the digital cash sent by Alice.

There are some inconveniences that may occur during this paying procedure. Transactions may be delayed or refused; there are obvious security and availability issues, issues known for all systems with a single point of failure. Bank services may be down or vulnerable to attacks; users may be dissatisfied by lack of privacy; bank has full record of Alice's transactions etc.

Blind signatures In 1983 **[blindsig]**, David Chaum proposed a type of electronic money, that could be exchanged among users, without a third party being able to know all

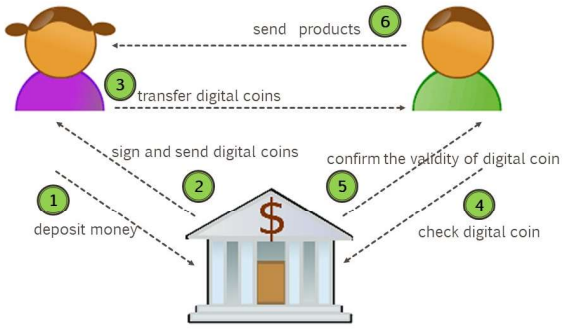

Figure 1: Traditional payment system

account balances or to track the owner of product that has been traded, the time of transactions or the amount of payments. This digital coin was not fully decentralized. There was still a central authority like a *bank* that signed checks, and verified signatures, attesting the validity of digital coins that has been emitted.

How did it work? Imagine a sealed write-through envelope (for instance a carbon-paper envelope). Banks receives from Alice a dollar and a token blinded by such an envelope, along with Alice's address. Bank signs the token through the envelope, without opening the envelope, and sends it back to Alice. The signature attests that the token can be indeed exchanged with real money. Alice can remove the envelope and send the unblinded signed token directly to Bob. Anyone can recognize the signature of the bank, so Bob can check the validity of the token. Later Bob can claim a dollar from the bank. Bank can't keep trace of Alice's transactions or tokens, while Alice is able to distinguish her tokens. What if Alice tries to buy two products using the

same token? To avoid double spending, bank may keep a list of token that have been claimed before.

Such a procedure may be also used for digital voting protocols. Instead of tokens representing money, Alice may send a blinded ballot slip. After receiving the ballot signed by the organizers, she post it again, without the blinding envelope and without sending her address. Organizers are unable to see Alice's vote, yet she is able to recognize her ballot so that she may be convinced that her vote was taken into account.

Blind signatures, RSA schema A simple blind signature schema is obtained using RSA. Recall that RSA involve a public key and a private key generated in the following way:

1. Choose p and q, two large prime numbers and calculate $n = pq$.
2. Calculate $\lambda(n) = \text{lcm}(p-1, q-1)$.
3. Choose e such that $\gcd(e,\lambda(n)) = 1$ and $1 < e < \lambda(n)$
4. Find d such that $ed \equiv 1 \pmod{\lambda(n)}$
5. Set public key (e, n). Set private key (d).
6. Set **coding** function $c(m) = m^e \pmod{n}$
7. Set **decoding** function $d(c) = c^d \pmod{n}$

The basic idea is to find three large positive integers e, d, and n, such that: knowing e and n (the public key), it can be computationally difficult to find d (the private key); anyone knowing the public key may send encoded messages that can be decoded only by someone knowing the private key, provided that $(m^e)^d = m \pmod{n}$.

Let's see how RSA encryption scheme is used for blind signatures. As stated by David Chaum, three functions form the blind signature cryptosystem:

1. a signing function s' known only by the signer, and the corresponding inverse s, such that $s(s'(x)) = x$ without revealing information bout x;

2. a commuting (or blinding) function c and its inverse c', known only by the provider, such that $c'(s'(c(x))) = s'(x)$, and c and c' does not leak any information about x. Informally, function c' recovers the signed message from the blinding envelope;

3. a redundancy checking predicate r, that checks for sufficient redundancy to make search for valid signatures impractical.

Given the public key (e, n), Alice generates m, the token that needs to be signed, and chooses a random blinding factor r such that $\gcd(r, n) = 1$. Then she sends to the signing authority m', where $m' \equiv mr^e \pmod{n}$.

The commutiong function is $c(x) = xr^e \pmod{n}$, where r is a random blinding factor. Notice that the commuting function does not reveal any information about the message m, since r^e is random.

The signing authority signs the message m' and sends back to Alice s', where $s' \equiv (m')^d \pmod{n}$.

The signing function is $s'(x) = (x)^d \pmod{n}$.

Alice can recover the signed token, removing the blinding factor, by applying the function $c'(x) = xr^{-1} \pmod{n}$.

Notice that

$c'(s'(c(x))) \equiv s'(x) \pmod{n}$ holds, since $c'(s'(c(x))) \equiv (xr^e)^d r^{-1} \equiv x^d r^{ed} r^{-1} \equiv s'(x) \pmod{n}$

Although blind signatures are an ingenious way to organize secure markets where privacy and anonymity are required, they still involve the intervention of a trusted third-party. Users can transact only tokens certified by the signing authority. What if Alice herself could authorize Bob to spend the money (digital tokens) she sends to him in further transactions ? With UTXO (**unspent transaction outputs**) model this is possible, provided each user posses a pair of public/private keys that he may use to sign transactions.

Bitcoin UTXO model provides pseudoanonymity. The identity of the owner of a pair of public/private keys is not made public. In fact, each individual may keep a set of public/private key pairs, each pair representing a digital wallet. Each user may have one or more wallets and may freely move their assets from one wallet to another. Alice may send her coins into Bob's wallet without the interference of a third party.

Each token of digital currency (UTXO) has an amount and is associated with a pair of cryptographic keys, in other words, each digital token is placed in a digital wallet. Only the owner possessing the wallet (the right private key) is enabled to spend its contents (the digital tokens associated with his public key). The database listing all available tokens contained in all wallets, namely the **set of all existing UTXOs**, is replicated by all nodes. Everyone has access to information about all digital coins, but coins can be used in transactions initiated only by the users that can prove their ownership. Notice that Bitcoin does not memorize users' balances, i.e. UTXO model does not memorize the sum of all unspent UTXOs owned by a wallet. Other cryptocurrencies use an account model. For example, on Ethereum account balances are part of the blockchain state and a transaction is

a state transition, changing both the balance of the sender and of the receiver.

UTXO transactions A transaction represents a transfer of money from a wallet to another, corresponding to a payment between two people. In Bitcoin, a transaction takes a list of inputs representing a specific value and produces a set of outputs, each output has a value and may be used in further transactions. Basically inputs are replaced by outputs. Each output can be spent only once (can be linked to only one input), this requirement is needed to prevent double-spending. The sum of the tokens consumed in a transaction, minus transactions fees, equals the sum of the values generated in the output, since money are not created in transaction, but are just being transferred from a wallet to another.

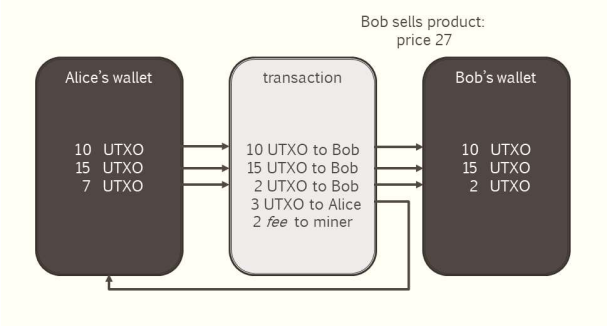

Figure 2: UTXO transaction

An example of transaction is presented in figure 2. Suppose Alice wants to buy o product from Bob. Bob sells the product at a price of 27. Alice has in her wallet three tokens valued 10, 15 and 7. These tokens are the result of previous transfers in Alice's wallet. Transaction consumes Alice's tokens and produces 4 new tokens. Bob receives three tokens valued 10, 15 and 2, while Alice receives back the change,

namely a token with the value of 3. Notice that the miner's fee is in this example 2.

Unspent Transaction Output (UTXO) set The UTXO set is the subset of Bitcoin transaction outputs that have not been spent at a given moment. Basically a transaction destroys some UTXOs from the UTXO set and produces new UTXOs. The ideas behind UTXO model to exchange used tokens with newly created ones originates in Hal Finney's *Reusable Proofs of Work* proposal that is considered the first attempt to create a digital coin. For more information see the following chapter. UTXO set provides all the information needed to validate transactions. **Full nodes** acting as validators store a complete copy of the UTXO set. The size of the UTXO set directly impacts storage capabilities and performances of full nodes. **Light nodes** use less storage space and *Simplified Payment Verification (SPV)* to verify transactions without downloading the entire blockchain, based only on block hashes. More on SPV and light nodes is presented in followings chapters.

Chainstate is a LevelDB database that implement persistent key-value storage for the UTXOs set with metadata about the transactions they are generated from.

Coinbase transcations Miners aggregate transactions into blocks that are copied by all nodes and receive transactions fees plus the value produced by a special kind of transaction, namely **coinbase transactions**. Coinbase transactions are the only kind of transactions that mint new bitcoin. Such a transaction has no inputs and is added as the first transaction in the block. The output of a coinbase transaction represents the value a miner is rewarded with, for adding a block of transaction into the blockchain. Chainstate is the set of all unspent transactions as they appear up to the last block added in the blockchain. Technically the information in

chainstate is redundant as it can be reconstructed from the date stored in the blocks but this would be extremely slow and inefficient.

Transaction structure Now let's get into more details about transactions structure and verification. [**bitctransaction**]. For simplification purposes we will not cover in this section the elements reserved for *SegWit* upgrade. As depicted in figure 3 a transaction is comprise of a list of inputs (unspent UTXOs), a list of outputs and a lock_time.

Field	Description	Size
Version	currently 1	4 bytes
In-counter	Number of inputs	1 – 9 bytes
List of inputs	List of inputs	<in-counter>-many inputs
Out-counter	Number of outputs	1 – 9 bytes
List of outputs	List of outputs	<out-counter>-many outputs
lock_time	block height or timestamp when transaction is final	4 bytes

Figure 3: Transaction structure

To prevent double-spending a transaction is considered final after receiving enough confirmations. Recall that transactions are gathered into blocks. Blocks are linked in a chain confirmed by all nodes. When a new block is added it will store the hash of the previous block in the chain. A transaction receives a confirmation each time a new subsequent block is published to the network. When the transaction is first added into the blockchain it is said that it is mined at a depth of 1 block. For each block added after it the depth is increased by 1. By default, a transaction is considered unconfirmed until 6 block depth, but merchants may choose their own policy considering that the probability of reverting a transaction decreases with each confirmation (each subsequent block). If someone want's to make sure a transaction

will not be confirmed until a specific timestamp or block number he may provide **lock_time** field.

A transaction input is a reference to a previous transaction output. In figure 4 (see also 2) transaction "c84be..." has three inputs from three previous transactions ("ac4be...", "589e0...", "14e9f..."). Each input links to a preceding transaction output specifying the id of the transaction and the index of the output. For instance, the first input of transaction "c84be..." refers to the second output (index 1) of transaction "ac4be", the third input refers to the first output (index 0) of transaction "14e9f...". Notice that the sum of the values of transaction inputs add up to the sum of the transactions outputs, minus transaction fees. Any input value that is not transformed into an output is implicitly considered transaction fee and may be inserted into the coinbase transaction of the block it is mined in, and thus may be given to the miner. In this example the fee is 2.

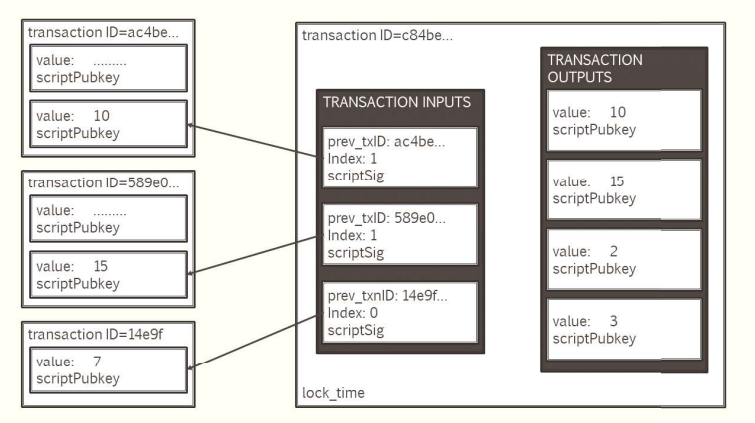

Figure 4: Transaction inputs/outputs

In what follows we will explain how we can be sure that when Bob initiate a transaction that links to outputs

previously sent into his wallet by Alice, he is indeed the right owner of those outputs, and how we differentiate the outputs attributed to Bob (output index 0, 1 and 2) from the output representing Alice's change (value 3).

Addresses A wallet is a collection of public/private keypairs. To generate such a pair begin with a private key, i.e a random number of size 256-bit. Then the corresponding public key is obtain through an Elliptic Curve Digital Signature Algorithm (ECDSA) [**bitcaddresses**]. Finally we get a Bitcoin address by hashing the public portion of the public/private ECDSA keypair (160-bit hash) and convert the result into a base58 identifier of 26-35 alphanumeric characters.

A Bitcoin address represents a possible destination for a bitcoin payment. The output of a transaction uses such an address to nominate the recipient of bitcoins, providing a challenge to users that want to claim a specific output, in the form of a locking script, **scriptPubKey**. Only the nominated recipient should be able the unlock the output providing in the input the unlocking script **scriptSig**.

Bitcoin scripts (for example scriptPubKey and scriptSig) are written in a stack-based execution language. Operations can push data onto the stack or pop one or more parameters, process them and push back in the stack the result of the processing [**bitcascript**]. Some operations are listed in figure 5 With the limited purpose of validating transactions this language is not Turing-complete. Moreover, to increase network security, a test was added in order to accept transactions if scriptPubKey and scriptSig match a small set of patterns. The test is called IsStandard() [**bitctransactiondev**] and the transactions which pass this test are called standard transactions.

In Bitcoin, there are two types of standard transaction, P2PKH Pay to Public Key Hash and P2SH Pay to Script Hash (more than 99% of UTXO set match one of those two templates [**utxoanalysis**]). Previous to this types of transaction a simpler version was used for coinbase transactions, P2PK Pay to Public Key.

OPCODE	Description	Input	Output
OP_HASH160	The input is hashed twice: first with SHA-256 and then with RIPEMD-160	x	hash
OP_CHECKSIG	The entire transaction's outputs, inputs, and script are hashed. The signature used by OP_CHECKSIG must be a valid signature for this hash and public key. If it is, 1 is returned, 0 otherwise.	sig pubkey	boolean
OP_DUP	Duplicates the top stack item.	x	x x
OP_EQUAL	Returns 1 if the inputs are exactly equal, 0 otherwise.	x1 x2	boolean
OP_EQUALVERIFY	Same as OP_EQUAL, but runs OP_VERIFY afterward.	x1 x2	Nothing / *fail*

Figure 5: Bitcoin script operation codes

P2PKH Pay to Public Key Hash transaction. P2PkH is the most used type of transaction and also the default type in a Bitcoin client.

The validation of a P2PKH works as follows. The sender (Alice) provides the recipient's address (Bob's address) <pubKeyHash> in scritPubKey (see white cells in fig 6). The locking script contains along with the recipient's address four operation codes.

scriptPubKey: OP_DUP, OP_HASH160, <pubKeyHash> OP_EQUALVERIFY, OP_CHECKSIG

If the receiver Bob wants to use the output received from Alice he provides his public key and his signature (marked with grey in 6) in the unlocking script.

scriptSig: signature public_Key

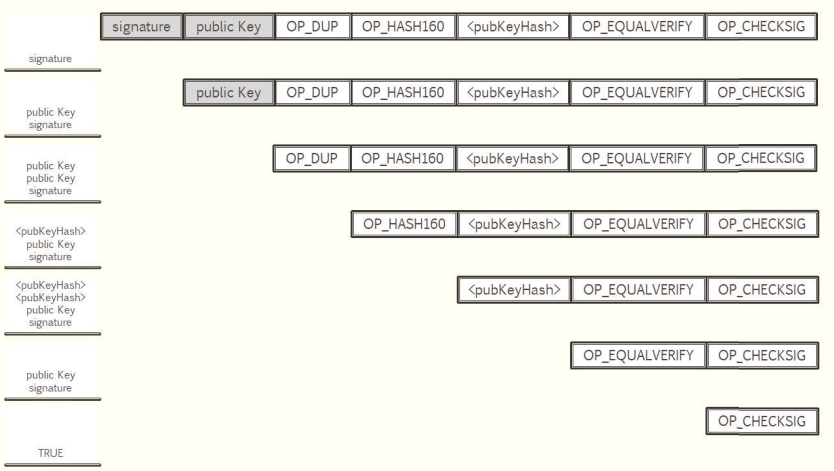

Figure 6: Bitcoin script operation codes

Together the instructions from scriptSig and scriptPubKey form the script that verifies that the public key provided by Bob does hash to the hash designated by the sender Alice in scriptPubKey and also checks the signature of the recipient against the public key. Thus, gluing the instructions from the unlocking script with the one in scriptPubKey we get a procedure to check is Bob is authorize to use the transaction output sent by Alice. The checking process begins with an empty stack. The the instructions from <scriptSig , scriptPubKey> operate in order from left to right. After the first two instructions, the signature and the public key of the user claiming the right to spend the output (in this case Bob) are pushed onto the stack. OP_DUP duplicates the top element of the stack. OP_HASH160 hashes the public key found on top of the stack, i.e the public key of the . The resulted hash must match the address provided in the

locking script. OP_EQUALVERIFY checks the equality of the two top-most elements of the stack and removes them. After this step signature and public Key remain on the stack and OP_CHECKSIG check the signature against the public key, returning TRUE if the signature is valid.