ACCESS Coin

F. Guevara Vasquez and C. Hohenegger

July 9, 2015

1 Bitcoin

Bitcoin was created by a person or group of persons going by the name of Satoshi Nakamoto, sometime in 2009 as a decentralized, resilient electronic payment system that instead of relying on trust between different parties, relies on a proof-of-work system and ensures that transactions are not tampered with cryptography (we will see exactly how this is done!). All the details are in the original paper `https://bitcoin.org/bitcoin.pdf`. The security features built into Bitcoin make the system extremely time consuming to game and people are using this virtual currency to buy real things. Overstock.com and Expedia.com accept Bitcoin payments already. As of June 9, 2015, one Bitcoin is worth about \$271 (and the value surpassed \$1000 in January 2014!), making all the Bitcoins in existence worth about $\$3.9 \times 10^9$. The daily transaction volume (as estimated by `http://blockchain.info`) is in the order of 2.5×10^5 Bitcoins/day or about $\$70 \times 10^6$/day. This value is dwarfed by the transaction volume in conventional institutions (the NASDAQ for example trades about $\$60 \times 10^9$/day). So Bitcoin and other cryptocurrencies that work on roughly the same principles (and there are many of them: Litecoin, Nxt, Darkcoin, Dogecoin,... see `http://www.coinmarketcap.com`) are economically significant!

So why are people giving value to Bitcoins? We could ask the same question about conventional money. The value we give to the pieces of paper that we carry in our wallets comes from our trust in the central authority that printed the note. We trust it symbolizes certain economic output and thus we use it for payments and accept it too as payment. The best way to understand why many trust Bitcoin is to write our own version of a cryptocurrency **Access Coin**, which is somewhat simpler than Bitcoin but also relies on a proof-of-work system and cryptography to make sure transactions are legitimate.

2 Transactions

2.1 The parties

Let's start with the usual suspects Alice, Bob, Charlie, ... Each party creates a public/private key pair and publishes their public key somewhere. If the parties choose to use the RSA cryptographic scheme, this means that Alice, Bob, Charlie, ... publish pubkeyA, pubkeyB, pubkeyC, ... with a trusted party and that each public key is, for instance,

```
pubkeyA = PublicKey(N = ..., # encrypt mod for Alice
                    e = ...) # encrypt exp for Alice
```

Bitcoin uses a different cryptographic system based on Elliptic Curves. It helps to see the actual Python definition for a PublicKey in module `coins`.

2.2 A transaction

Now lets say that Alice wants to buy an espresso from Bob for the price of 15 access coins. This is encoded in a transaction. We can think of a transaction as a check that lists all the "accounts" we are taking funds from (the inputs) and all the parties we are sending the money to (the outputs):

```
tx = Transaction(inputs  = ..., # list of input
                 outputs = ...) # list of outputs
```

So a transaction can have several inputs and ouputs? Why so complicated? The reason is that in Access Coin (and Bitcoin) whenever an input appears in a transaction, **all the funds in the input are spent**. One of the things that matter when making a transaction is a coin conservation law: all the funds in the inputs must match all the funds in the outputs. So a transaction does not create or consume coins, it just moves them from place to place. (this is simpler than in Bitcoin, where any excess inputs are automatically considered transaction fees).

2.3 A Transaction output

This is a reference to who we want to send the money to and the amount:

```
txout = TxOut(pubkey = ..., # public key of destination
              amount = ...) # funds assigned to dest.
```

2.4 A Transaction input

So what is an input? This is a reference to the output of an earlier transaction. If that output has not yet been spent, it provides proof that Alice owns those coins. We reference previous transactions by using a **hash**, which is an (almost) **unique identifier** that is generated using all the data from the previous transaction (more on this in section 2.5). A hash has the property that if anything changes in the transaction, then the hash will (with very high probability) change. A Transaction input looks as follows:

```
txin = TxIn(pubkey = ..., # public key of origin
            hash_prev = ..., # hash of prev tx
            output_num= ...) # output in prev tx to spend
```

It includes an integer `output_num` that references to a particular output in a previous transaction, and thus a particular amount as well. The public key is also included so that the sender of the funds can cryptographically sign the transaction, i.e. show (by using his private key) that the sender agrees with the transaction.

In Access Coin we simplify things and assume that all inputs in a transaction belong to the same person (i.e. have the same public key) so that the transaction needs to be signed with only one key. In Bitcoin, signing is more complicated than this because a transaction could have inputs with different public keys, so everyone needs to sign to agree to the transaction.

2.5 Hash functions

A hash is a one way function that takes some arbitrary data and gives an integer in some predetermined range. A good hash function distributes all possible inputs as uniformly as possible over its outputs. So if the hash values are different, it is very likely that the inputs are different as well. Here is an example. Let p is a prime number and e an integer with $\gcd(e, p-1) = 1$,

$$f(x) = x^e \mod p.$$

We know that f is invertible (why?) so it maps all integers $0, \ldots, p-1$ to all integers $0, \ldots, p-1$. No one stops us from taking for x an arbitrary integer (not necessarily $< p$). So the function f maps all integers to the integers $0, \ldots, p-1$, and does so uniformly (since if $x \equiv y \mod p \Rightarrow x^e \equiv y^e \mod p$, so all numbers that are congruent $\mod p$ have the same hash value). Here is an illustration in Python of the power function as a hash function, it shows that two nearly identical texts have radically different hash values.

```
>>> import primes
>>> p = 19469025781 # randomly generated
>>> e = 3
>>> primes.egcd(p,e)
(1, -6489675260, 1)
>>> x = primes.davis_enc("This is some text")
>>> print pow(x,e,p)
15533248356
>>> x = primes.davis_enc("This is s0me text")
>>> print pow(x,e,p)
12155720627
```

There are many ways of defining a hash function. For Access Coin we use the MD5 algorithm `http://en.wikipedia.org/wiki/MD5`, since it is standard and more efficient than taking powers of large integers. With MD5 the hash values are a 128-bit number, i.e. a string of 128 zeros or ones (base 2). We will use an hexadecimal (or hex) representation of the number. So instead of counting 0,1 (base 2), or 0,1,2,3,4,5,6,7,8,9 (base 10), we count 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f (base 16). A 128-bit number fits in $128/8 = 16$ bytes and each byte can be represented with two hex digits (why?). So the whole hash fits in exactly 32 characters that are hex digits. Here is an illustration in Python that a little change in the input gives a radically different hash.

```
>>> import hashlib
>>> print hashlib.md5("This is some text").hexdigest()
97214f63224bc1e9cc4da377aadce7c7
>>> print hashlib.md5("This is s0me text").hexdigest()
39bc766bcf0f5d2d19b8edbcd5f40715
```

2.6 Signing a transaction

Let's say Alice wants to pay Bob 5 coins. Alice includes with the transaction a **signature**, which is the hash of the transaction, encrypted with Alice's private key.

For Bob, Charlie and other parties, it is easy to check that Alice authorized the transaction: they simply encrypt the signature with pubkeyA (which is public) and compare it to the hash of the transaction. If they are the same, then Alice did authorize the transaction, if not, something wrong happened!

If Evil tries to tamper with the transaction (say changing the output to benefit Evil, or changing the amounts), then they will not be able to generate a valid signature because tampering with the transaction will change the hash, and they do not know Alice's private key to sign the hash appropriately!

This effectively ensures that Alice and only Alice can unlock the funds in her favor. Of course, if Alice loses her private key she will not have access to these funds (because she will not be able to sign any transaction spending them without her private key). And if Evil can somehow know what Alice's private key is (by e.g. sneaking into Alice's computer to steal it) then they will be able to spend all of Alice's money!

3 Blocks and the Blockchain

3.1 Keeping a ledger

This system of signing transactions works if there is a central authority that checks that all transactions are signed correctly. For Bitcoin, there is no such thing. Many computers work on verifying transactions (are they signed properly?, do they have enough funds? etc...). Several transactions are lumped together in blocks, which look as follows:

```
b = Block(hash_prev = ...,     # hash of prev block
          transactions = ..., # list of tx
          nonce = ...,
          difficulty = ...)
```

The history of transactions is stored as a chain of blocks (which acts as a **ledger** recording all past transactions), where each block in the chain references the hash of the previous block in the chain. In this way, it is always possible to check that this history of transactions (or block chain) has not been tampered with. To do this it suffices to recompute the hash of all the blocks in the block chain and check whether the previous block hash stored in a block does match the one we re-computed.

3.2 Verification and mining

For a node, verifying the validity of transactions in a block does not take a long time. So we could have an evil (or buggy?) node adding blocks to the blockchain without first checking them or with fraudulent transactions. Other nodes will certainly spot the errors, but how do we know who to trust? This is where the **proof-of-work** concept comes in. Verifying a block is made into a problem that takes a long time by adding a nonce, which is just an integer. To verify a block, the nodes have to find a nonce so that the hash of the block starts with a certain number of zeroes. The more zeroes are asked, the harder the problem gets (this is why each block carries a difficulty). This is a hard problem because the only (known) way is brute force, i.e. in pseudo-code

```
while block hash does not start with difficulty zeroes:
   pick another nonce
```

Because this process takes a lot of computing power, it is in the interest of the nodes to submit blocks where all the transactions are legit, otherwise all their hard work will go to waste! Also because this is onerous, the node that first finds a nonce gets a reward (which in Access Coin is fixed to 10 coins / block). This is the only way of making new coins, and this is why verifying blocks is called **mining**.

If a node claims to have mined a block, the other nodes in the network can verify this easily as it is just a matter of hashing the block and checking whether the hash starts with a pre-agreed upon number of zeroes. So the news of a new block being found will be broadcast to other nodes and they will add it to their blockchain. In Bitcoin there is a predetermined way of setting the difficulty so that verifying a block takes the network about 10 minutes. Also there may be two nodes claiming valid blocks, which would fork the blockchain. This is dealt with by choosing the longest branch, because it corresponds to the "consensus" of the other nodes in the network.

3.3 Simplifications

3.3.1 Mining

To simplify things in Access Coin we only have one computer (the instructor's) verifying the transactions in each block. Any group can request a block from the server that already has valid transactions in it and work on mining it (finding a nonce that matches the difficulty). The first group to submit their block to the server will get a reward! Just as with Bitcoins, mining is a race, so you may end up mining a block for nothing because a group got there first. In this case, you do need to request a new block from the server to mine and try again!

Because this is a race and any group getting there first gets a reward, groups may get more technologically advanced of mining than others (maybe running different processes or in different computers?). For Bitcoin (and other cryptocurrencies) there are many hardware based solutions for mining, and all they do is compute hashes, but very very efficiently.

3.3.2 Transactions

In Access Coins, setting up a transaction is simplified since we can ask the server which transactions are in favor of a certain group and how many coins that group has to spend, this is done by generating a list of (transaction hash, output number, amount) related to a public key. This list is called a wallet and can also be viewed on the web server.

Any new signed transaction is sent to the server where it is stored in a list of pending transactions (which can be accessed from the web). This list of pending transactions is what is sent to the clients when they request a new block. The new blocks come already with the reward transaction built-in.