Mathematics Behind Blockchain

Edward Bickerton

January 2022

Contents

1 Introduction **2**

2 Cryptographic Hash Functions **3**
2.1 Introduction 3
2.2 SHA 4
2.3 Python example 4

3 Digital Signatures **5**
3.1 Introduction 5
3.2 RSA crypto-system 6
3.2.1 Encryption 6
3.2.2 Decryption 7
3.2.3 Signing messages 7
3.3 Python demonstration 7

4 Blockchain **9**
4.1 Introduction 9
4.2 Oldest Blockchain 9

5 Merkle Trees **11**
5.1 Introduction 11
5.2 Bitcoin's Merkle trees 11
5.3 Python demonstration 13
5.4 Full nodes vs. SPV nodes 14

6 Proof of Work **16**
6.1 Introduction 16
6.2 Hashcash 16
6.3 Python demonstration 17

7 Conclusion **20**

Chapter 1

Introduction

In this document I will go over some of the cryptographic ingredients which come together to form the loosely defined and rapidly changing space of blockchain technology. My personal focus is on Bitcoin (I use Bitcoin to refer to the decentralised network and bitcoin or BTC to refer to the asset/currency), mainly because in my opinion its purpose is simpler and thus easier to define compared to smart contract platforms such as Ethereum, only time will tell if these blockchains have a place in the economy. However, much of what is discussed applies to, not only blockchain technologies but also broader technologies in our internet age.

Chapter 2

Cryptographic Hash Functions

2.1 Introduction

Cryptographic hash functions are a fundamental building block in cryptography and are used extensively in Bitcoin. The fingerprint of a person is an apt analogy to the hash of a digital document in the following sense; the hash of a digital document:

- is characteristic of the document,
- is unique to the document,
- is small no matter it's size, and
- gives you no information about the document itself.

The cryptographic hash function takes data of arbitrary size as input, sometimes referred to as the message; and the output is a bit array of a fixed size referred to as the input's hash value or sometimes the message digest. Practically this can be thought of as a function from $\mathbb{N}$ to some finite subset of $\mathbb{N}$ (i.e. $\{1, \dots, 2^n\}$ where n is the length of the bit array). These functions are deterministic, meaning that the same message always gives the same hash value. They are one way functions, meaning, given a hash value it is computationally unfeasible to find a message which has this hash value and ideally the most efficient way of finding such a message would be a brute force attack i.e. guessing a random message and computing its hash value to compare it to the target value, repeating until successful.

2.2 SHA

SHA stands for secure hash algorithms and are a family of cryptographic hash functions published by the National Institute of Standards and Technology. The most commonly used function is SHA256, but this name is slightly ambiguous. There have been four generations of SHA, SHA-0 has an undisclosed significant flaw, SHA-1 and 2 were both designed by the NSA and SHA-3 (the most current version, also known as Keccak) was chosen in 2012 after a public competition. So the 256 in SHA256 simply refers to the length of its bit array output. The design of these functions are outside the scope of this project, but SHA-3 uses the sponge construction, meaning data is absorbed into the sponge and then the result is squeezed out.

2.3 Python example

The following Python code demonstrates that changing the input of a hash function, even only slightly, yields a completely different output. On a mac, if you open the terminal app and type *python* (python and hashlib should already be installed) you will be able to type and execute code.

```
>>> from hashlib import sha3_256

>>> def string_hash(string):
...     return sha3_256(string.encode()).hexdigest()
...

>>> string_hash("A specter is haunting the modern world, the specter of
    crypto anarchy.")

"dba25e3027b573af12e8337433c3ad102e884f95e9bb8d823b5492d80d5ce1c5"

>>> string_hash("a specter is haunting the modern world, the specter of
    crypto anarchy.")

"e31a3ea917466318369e5e979b4eaf02aa253b094cf153d0cf0b3ef7aafaae08"
```

The second string differs by only its first character and yet their hash values show no apparent correlation. Note that the hash outputs are given in hexadecimal, not the familiar decimal number system (using the sixteen characters: 0-9 and a-f). This is done so as to use less characters when displaying very large numbers.

Chapter 3

Digital Signatures

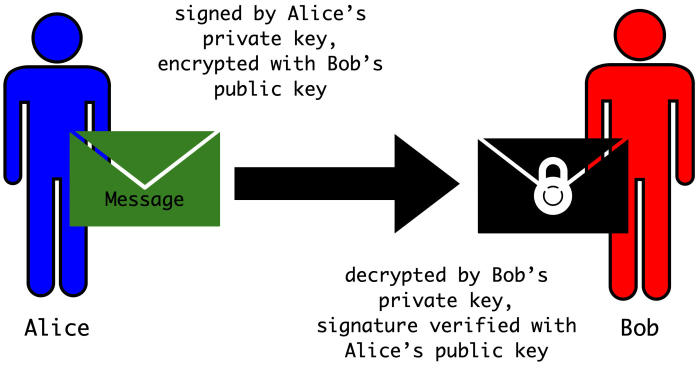
3.1 Introduction

Signatures in the traditional sense are less than perfect. Their purpose is to prove commitment by a specific person to a specific statement (or contract). However, at least for my signature, *anyone* who stares at it for long enough could quite easily reproduce it on *any* document they like. Also the traditional signature has no strong defence against the document being altered after it has been signed.

Digital signatures offer a much better solution to the same problem regular signatures address. Only the owner of a private key can produce the signature. It is much easier for an individual to defend a private key than it is for them to make their written signature sufficiently complex so as to make it non reproducible by anyone else. Digital signatures also have the added benefit that the signature depends on the document being signed (and not just the signer), thus the document cannot be altered after it has been signed while maintaining a valid signature.

In the next section I will go through the RSA crypto-system. However, Bitcoin actually uses the elliptic curve digital signature algorithm. In the case of Bitcoin, digital signatures are used to ensure that only the owner of some bitcoin can write and broadcast a valid transaction spending that bitcoin, which will be accepted by the network and (hopefully) included into a block.

3.2 RSA crypto-system



The RSA crypto-system was publicly described by Ron Rivest, Adi Shamir and Leonard Adleman in 1977 (GCHQ had developed an equivalent system in 1973). I use the word crypto-system because it allows for encryption and decryption as well as producing digital signatures. Each user has both a private and public key, naturally the private key is kept secret and the public key is made public. The private key is used to sign and decrypt messages, while the public key is used to verify the validity of signatures and to encrypt messages. Both keys are tuples as such:

- Private Key: (p, q) where p and q are large primes.
- Public Key: (N, e) where $N = p \times q$ and $e = 65537$.

As you can see, the public key can be produced from the private key but the reverse is not true since there is no efficient method for finding the factors of N. It's worth noting that p and q should be so large (say 512 bits) that it is implausible to guess them in the same way an attacker might guess a weak password.

3.2.1 Encryption

Encryption is done by the sender of a message using the recipients public key. First the message needs to be converted to an integer m, there are many ways of doing this for example replacing A with 01, B with 02 and so on (note: m must satisfy $0 \leq m < N$). The ciphertext c, (the encrypted message) is then produced by calculating:

$$c = m^e \ (mod \ N) \tag{3.1}$$

The system depends on there being no efficient method for calculating the discrete logarithm, i.e. there is no known way of finding m from c, e and N.

3.2.2 Decryption

Once the recipient has received the encrypted message they can use their private key to decrypt it. Let ϕ be Euler's totient function, that is the number of positive integers up to a given integer n that are relatively prime to n. We have that:

$$\phi(N) = (p-1)(q-1) = N - p - q + 1 \tag{3.2}$$

and that $\phi(N)$ is very hard to compute without knowing p and q. To decrypt c we must find the multiplicative inverse of e modulo $\phi(N)$. That is d such that, for some integer k, $e \times d = k \times \phi(N) + 1$ which can be found using the extended Euclidean algorithm, then we have that:

$$c^d = (m^e)^d = m^{e\times d} = m^{k\times\phi(N)+1} = (m^k)^{\phi(N)} \times m = m \ (mod\ N) \tag{3.3}$$

since by Euler's totient theorem $(m^k)^{\phi(N)} = 1 \ (mod\ N)$. Once m is found it can then be converted back into the original message.

3.2.3 Signing messages

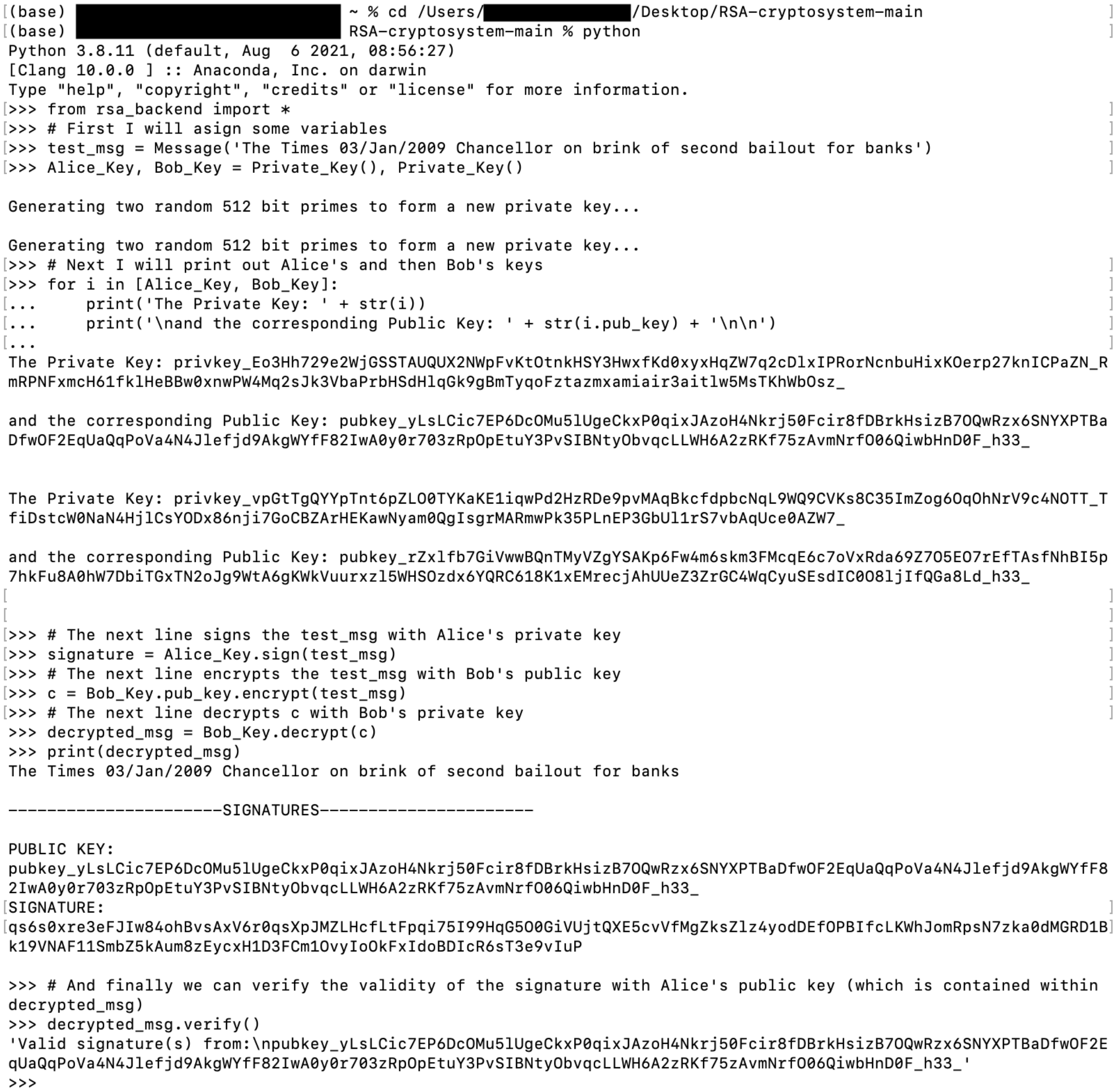
Anyone with a private key can use it to sign any message (or digital document) they choose, to verify the validity of the signature all that is required is the corresponding public key and the original message. So long as the private key is kept private and is known only by the intended user(s), the digital signatures produced by the private key can be trusted. Let h be the hash of the message as an integer, then the signature s is given by $s = h^d \ (mod\ N)$. Where d is as before and thus can only be calculated by the owner of the private key. The validity of the signature can then be verified by calculating $s^e \ (mod\ N)$ which will give h since, again, by Euler's totient theorem:

$$s^e = (h^d)^e = h^{e\times d} = h^{k\times\phi(N)+1} = (h^k)^{\phi(N)} \times h = h \ (mod\ N), \tag{3.4}$$

where k is some integer.

3.3 Python demonstration

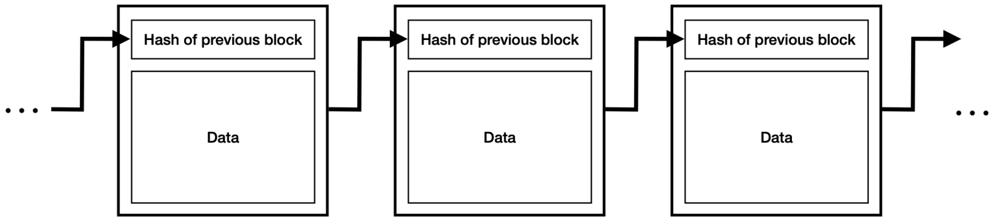
The python files used below can be found on my GitHub account Satsuma-LN in the RSA-cryptosystem repository. There is even a graphical user interface to go with rsa_backend.py.



```
(base) [redacted] ~ % cd /Users/[redacted]/Desktop/RSA-cryptosystem-main
(base) [redacted] RSA-cryptosystem-main % python
Python 3.8.11 (default, Aug  6 2021, 08:56:27)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from rsa_backend import *
>>> # First I will asign some variables
>>> test_msg = Message('The Times 03/Jan/2009 Chancellor on brink of second bailout for banks')
>>> Alice_Key, Bob_Key = Private_Key(), Private_Key()

Generating two random 512 bit primes to form a new private key...

Generating two random 512 bit primes to form a new private key...
>>> # Next I will print out Alice's and then Bob's keys
>>> for i in [Alice_Key, Bob_Key]:
...     print('The Private Key: ' + str(i))
...     print('\nand the corresponding Public Key: ' + str(i.pub_key) + '\n\n')
...
The Private Key: privkey_Eo3Hh729e2WjGSSTAUQUX2NWpFvKtOtnkHSY3HwxfKd0xyxHqZW7q2cDlxIPRorNcnbuHixKOerp27knICPaZN_R
mRPNFxmcH61fklHeBBw0xnwPW4Mq2sJk3VbaPrbHSdHlqGk9gBmTyqoFztazmxamiair3aitlw5MsTKhWbOsz_

and the corresponding Public Key: pubkey_yLsLCic7EP6DcOMu5lUgeCkxP0qixJAzoH4Nkrj50Fcir8fDBrkHsizB7OQwRzx6SNYXPTBa
DfwOF2EqUaQqPoVa4N4Jlefjd9AkgWYfF82IwA0y0r703zRpOpEtuY3PvSIBNtyObvqcLLWH6A2zRKf75zAvmNrfO06QiwbHnD0F_h33_


The Private Key: privkey_vpGtTgQYYpTnt6pZLO0TYKaKE1iqwPd2HzRDe9pvMAqBkcfdpbcNqL9WQ9CVKs8C35ImZog6OqOhNrV9c4NOTT_T
fiDstcW0NaN4HjlCsYODx86nji7GoCBZArHEKawNyam0QgIsgrMARmwPk35PLnEP3GbUl1rS7vbAqUce0AZW7_

and the corresponding Public Key: pubkey_rZxlfb7GiVwwBQnTMyVZgYSAKp6Fw4m6skm3FMcqE6c7oVxRda69Z7O5EO7rEfTAsfNhBI5p
7hkFu8A0hW7DbiTGxTN2oJg9WtA6gKWkVuurxzl5WHSOzdx6YQRC618K1xEMrecjAhUUeZ3ZrGC4WqCyuSEsdIC0O8ljIfQGa8Ld_h33_


>>> # The next line signs the test_msg with Alice's private key
>>> signature = Alice_Key.sign(test_msg)
>>> # The next line encrypts the test_msg with Bob's public key
>>> c = Bob_Key.pub_key.encrypt(test_msg)
>>> # The next line decrypts c with Bob's private key
>>> decrypted_msg = Bob_Key.decrypt(c)
>>> print(decrypted_msg)
The Times 03/Jan/2009 Chancellor on brink of second bailout for banks

----------------------SIGNATURES----------------------

PUBLIC KEY:
pubkey_yLsLCic7EP6DcOMu5lUgeCkxP0qixJAzoH4Nkrj50Fcir8fDBrkHsizB7OQwRzx6SNYXPTBaDfwOF2EqUaQqPoVa4N4Jlefjd9AkgWYfF8
2IwA0y0r703zRpOpEtuY3PvSIBNtyObvqcLLWH6A2zRKf75zAvmNrfO06QiwbHnD0F_h33_
SIGNATURE:
qs6s0xre3eFJIw84ohBvsAxV6r0qsXpJMZLHcfLtFpqi75I99HqG5O0GiVUjtQXE5cvVfMgZksZlz4yodDEfOPBIfcLKWhJomRpsN7zka0dMGRD1B
k19VNAF11SmbZ5kAum8zEycxH1D3FCm1OvyIoOkFxIdoBDIcR6sT3e9vIuP

>>> # And finally we can verify the validity of the signature with Alice's public key (which is contained within
decrypted_msg)
>>> decrypted_msg.verify()
'Valid signature(s) from:\npubkey_yLsLCic7EP6DcOMu5lUgeCkxP0qixJAzoH4Nkrj50Fcir8fDBrkHsizB7OQwRzx6SNYXPTBaDfwOF2E
qUaQqPoVa4N4Jlefjd9AkgWYfF82IwA0y0r703zRpOpEtuY3PvSIBNtyObvqcLLWH6A2zRKf75zAvmNrfO06QiwbHnD0F_h33_'
>>>
```

Chapter 4

Blockchain



4.1 Introduction

Put simply a blockchain is an append only ledger (or database) in which new data is added in blocks. Each new block contains the hash of the previous block, and so blockchains are resistant to retroactive modification since changing data in any of the previous blocks would alter its hash value, rendering all blocks made after said block invalid. I recommend visiting https://demoblockchain.org/blockchain for an interactive demonstration of blockchain. Crucially, the integrity of a blockchain depends heavily on the integrity of the hash function it uses.

4.2 Oldest Blockchain

The term blockchain gets misused, often it gets used to describe the whole cryptocurrency (or distributed computing) space. Interestingly the longest running blockchain does not belong to a decentralised protocol such as Bitcoin, but to a centralised company founded by Stuart Haber and W. Scott Stornetta in 1994 known as Surety. The company provides digital tamper evident seals for digital documents in a trust minimising way. Interestingly, part of their solution depends on making a hash value both widely available and widely witnessed, which they achieve by publishing it in The New York Times which is archived all over the world. Anyone (even non-customers) can independently verify the integrity of a document with one of their seals because this hash depends on every single document Surety has sealed up to that point. Importantly this is done without the users having to reveal their documents, they simply provide Surety with the hash of the document they wish to seal.

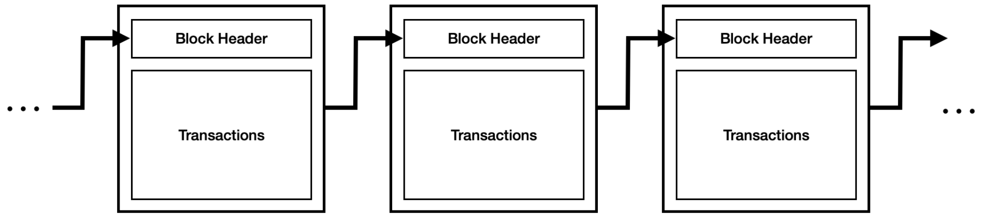
Chapter 5

Merkle Trees

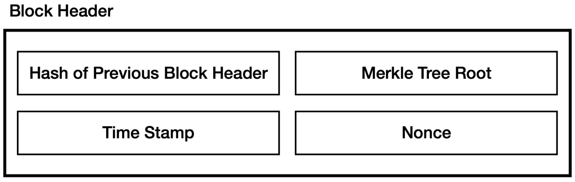
5.1 Introduction

Merkle trees offer a method for efficiently verifying the integrity of data even when its source is untrusted, given that the Merkle tree root is obtained from a trusted source. Patented by Ralph Merkle in 1979 (the inventor of cryptographic hashing) merkle trees depend heavily on the integrity of the hash function it uses. In the following sections I will look at Merkle trees from the context of how they are used in Bitcoin.

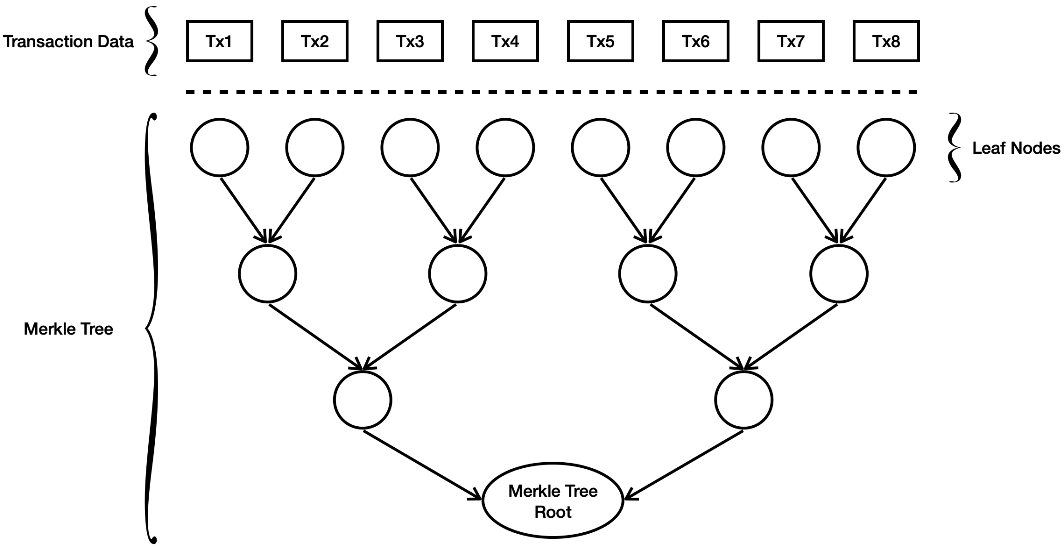
5.2 Bitcoin's Merkle trees



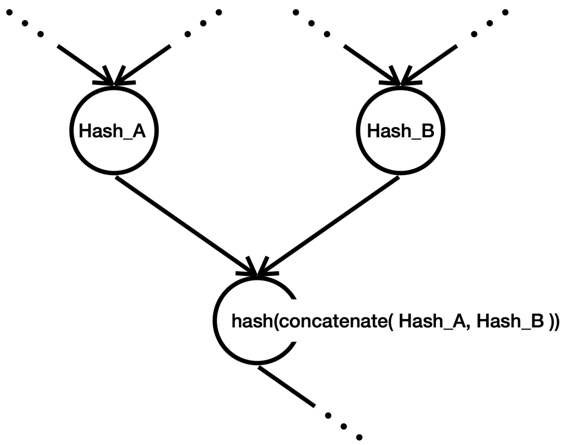
The diagram above is a slightly altered version of the diagram from the Blockchain chapter. It is somewhat misleading because, in the case of Bitcoin the transaction data (the data which is used to determine its users' balances) is not actually stored in the blockchain itself. This may sound alarming, but Merkle trees allow for an efficient method for creating a *fingerprint* of all of the transactions in the form of a Merkle tree root which *is* stored in the blockchain.



The diagram above shows what is contained within Bitcoin's block headers. Since, a hash of only the previous block header is included in Bitcoin's block headers, and not a hash of the *entire* previous block, it is perhaps more precise to refer to the series of block headers as Bitcoin's blockchain, leaving out the transaction data.

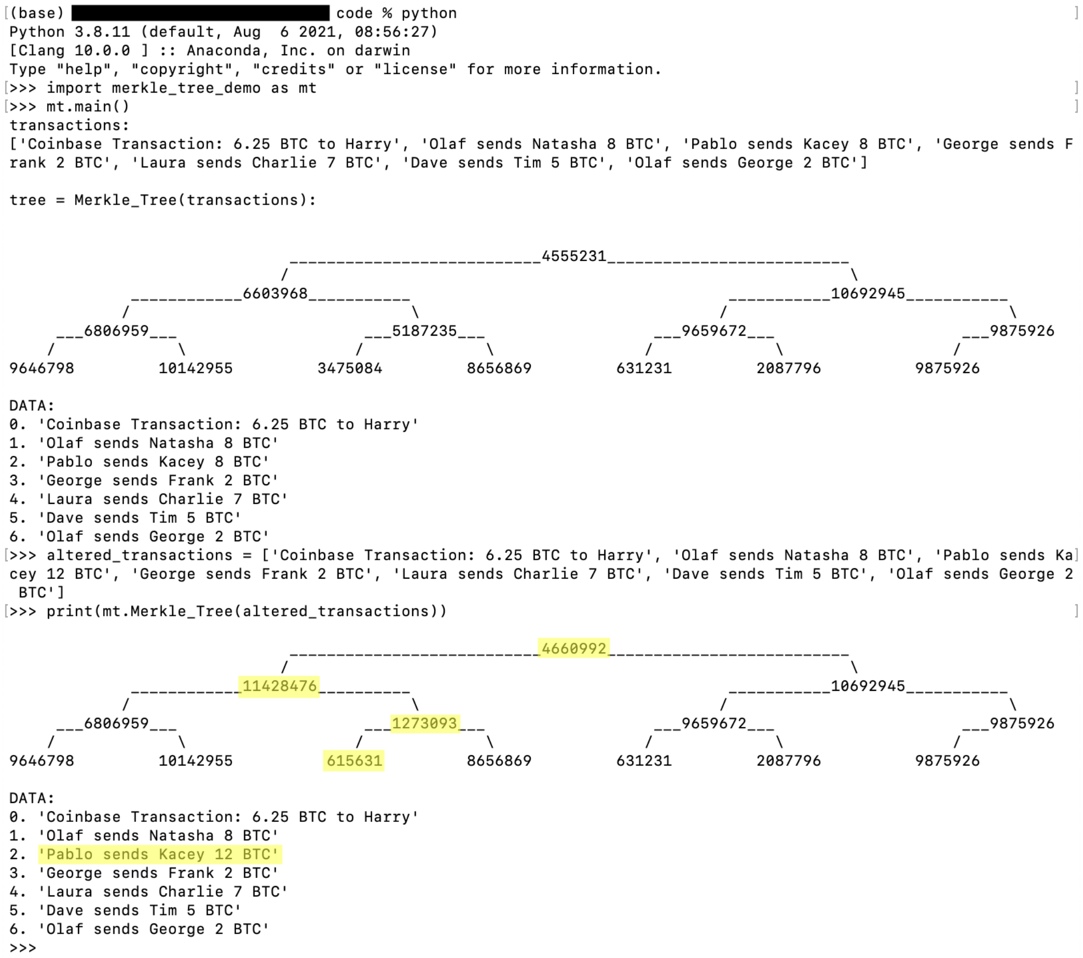


Above is an illustration of how a Merkle tree root is constructed for eight transactions (depending on how busy the Bitcoin network is, a block may contain a few thousand transactions). The tree is a binary tree, meaning each node has at most two child nodes. The leaf nodes are formed by taking the hash of their corresponding transaction. Then the parent node is constructed by taking the hash of the concatenation of the two child nodes' hashes.



5.3 Python demonstration

In the following, the hash function I use is based off of SHA256, only converted to decimal and taking only the first eight digits. You would not want to use this in any serious setting due to the increased chance of a hash collision. Both the blockchain_project.py and merkle_tree_demo.py files from my GitHub account Satsuma-LN in the Mathematics-behind-blockchain-project repository and the binarytree python module are needed to go through the demonstration.



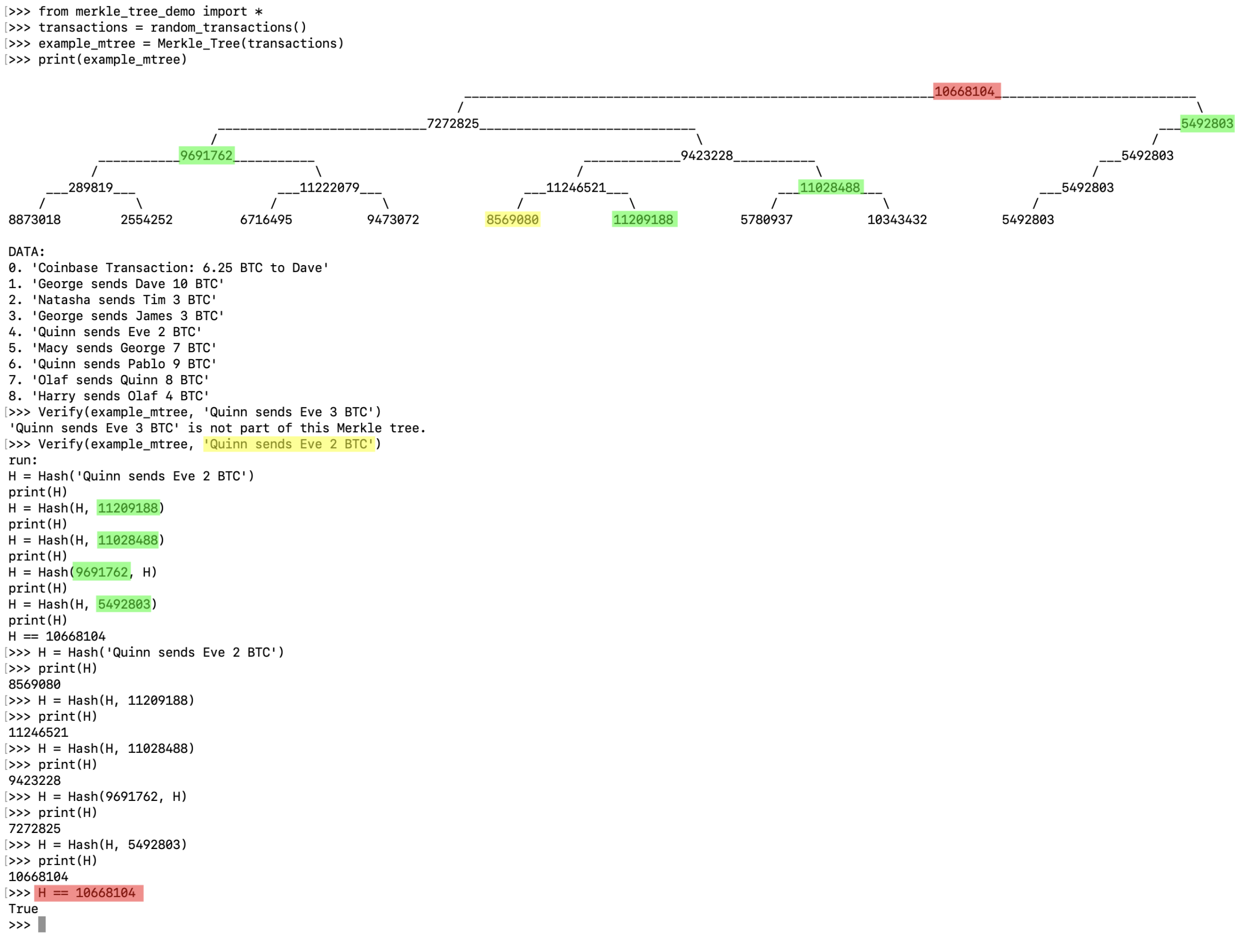
```
[(base) [redacted] code % python                                                                                 ]
Python 3.8.11 (default, Aug  6 2021, 08:56:27)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> import merkle_tree_demo as mt                                                                               ]
[>>> mt.main()                                                                                                   ]
transactions:
['Coinbase Transaction: 6.25 BTC to Harry', 'Olaf sends Natasha 8 BTC', 'Pablo sends Kacey 8 BTC', 'George sends F
rank 2 BTC', 'Laura sends Charlie 7 BTC', 'Dave sends Tim 5 BTC', 'Olaf sends George 2 BTC']

tree = Merkle_Tree(transactions):


                              ___________________________4555231___________________________
                             /                                                             \
             ____________6603968___________                                  ___________10692945___________
            /                              \                                /                              \
     ___6806959___                    ___5187235___                  ___9659672___                    ___9875926
    /             \                  /              \                /             \                  /
9646798         10142955         3475084         8656869         631231         2087796          9875926

DATA:
0. 'Coinbase Transaction: 6.25 BTC to Harry'
1. 'Olaf sends Natasha 8 BTC'
2. 'Pablo sends Kacey 8 BTC'
3. 'George sends Frank 2 BTC'
4. 'Laura sends Charlie 7 BTC'
5. 'Dave sends Tim 5 BTC'
6. 'Olaf sends George 2 BTC'
[>>> altered_transactions = ['Coinbase Transaction: 6.25 BTC to Harry', 'Olaf sends Natasha 8 BTC', 'Pablo sends Ka]
cey 12 BTC', 'George sends Frank 2 BTC', 'Laura sends Charlie 7 BTC', 'Dave sends Tim 5 BTC', 'Olaf sends George 2
 BTC']
[>>> print(mt.Merkle_Tree(altered_transactions))                                                                 ]

                              ___________________________4660992___________________________
                             /                                                             \
             ____________11428476__________                                  ___________10692945___________
            /                              \                                /                              \
     ___6806959___                    ___1273093___                  ___9659672___                    ___9875926
    /             \                  /              \                /             \                  /
9646798         10142955         615631          8656869         631231         2087796          9875926

DATA:
0. 'Coinbase Transaction: 6.25 BTC to Harry'
1. 'Olaf sends Natasha 8 BTC'
2. 'Pablo sends Kacey 12 BTC'
3. 'George sends Frank 2 BTC'
4. 'Laura sends Charlie 7 BTC'
5. 'Dave sends Tim 5 BTC'
6. 'Olaf sends George 2 BTC'
>>>
```

Above, in the second Merkle tree you can see that changing transaction two from ‘Pablo sends Kacey 8 BTC’ too ‘Pablo sends Kacey 12 BTC’ is enough to change the Merkle tree root from 4555231 to 4660992.

You may be thinking, if a fingerprint of the transaction data is all that’s required, why not just concatenate all of the transaction data and then include its hash in the block header. The next section attempts to answer this question.

5.4 Full nodes vs. SPV nodes

A Bitcoin full node keeps a copy of every transaction since Bitcoin’s inception in 2009. This takes up about 400 GB of hard drive space and to buy the necessary hardware costs two to three hundred pounds (see projects such as getumbrel or raspiblitz). For many people this is infeasible, whether because of internet access limitations or because their primary computer is a smart phone. SPV (simple payment verification) nodes on the other hand keep a copy of only the block headers which is light weight enough to be run on a smart phone; and because of the nature of the proof of work system Bitcoin uses, the user has a copy of all of the Merkle tree roots from a trusted source. However, SPV nodes come with a trade off. The user is still dependent on a full node (which can be run by an untrusted entity), they rely on other users of the network to ensure its integrity and there are some negative privacy implications of using an SPV node instead of a full node.



```
[>>> from merkle_tree_demo import *
[>>> transactions = random_transactions()
[>>> example_mtree = Merkle_Tree(transactions)
[>>> print(example_mtree)

                                                             _______________________________________________________________10668104__________________________
                                                             /                                                                                                  \
                            ____________________________7272825_____________________________                                                               ___5492803
                           /                                                                 \                                                            /
            ___________9691762___________                                    _____________9423228___________                                       ___5492803
           /                              \                                  /                               \                                    /
     ___289819___                   ___11222079___                   ___11246521___                    ___11028488___                      ___5492803
    /            \                 /               \                 /              \                  /              \                   /
8873018        2554252         6716495          9473072         8569080          11209188         5780937          10343432           5492803

DATA:
0. 'Coinbase Transaction: 6.25 BTC to Dave'
1. 'George sends Dave 10 BTC'
2. 'Natasha sends Tim 3 BTC'
3. 'George sends James 3 BTC'
4. 'Quinn sends Eve 2 BTC'
5. 'Macy sends George 7 BTC'
6. 'Quinn sends Pablo 9 BTC'
7. 'Olaf sends Quinn 8 BTC'
8. 'Harry sends Olaf 4 BTC'
[>>> Verify(example_mtree, 'Quinn sends Eve 3 BTC')
'Quinn sends Eve 3 BTC' is not part of this Merkle tree.
[>>> Verify(example_mtree, 'Quinn sends Eve 2 BTC')
run:
H = Hash('Quinn sends Eve 2 BTC')
print(H)
H = Hash(H, 11209188)
print(H)
H = Hash(H, 11028488)
print(H)
H = Hash(9691762, H)
print(H)
H = Hash(H, 5492803)
print(H)
H == 10668104
[>>> H = Hash('Quinn sends Eve 2 BTC')
[>>> print(H)
8569080
[>>> H = Hash(H, 11209188)
[>>> print(H)
11246521
[>>> H = Hash(H, 11028488)
[>>> print(H)
9423228
[>>> H = Hash(9691762, H)
[>>> print(H)
7272825
[>>> H = Hash(H, 5492803)
[>>> print(H)
10668104
[>>> H == 10668104
True
>>> █
```

Above is a demonstration of how, say Eve can verify that Quinn has sent her two bitcoins, while Eve only has access to the Merkle tree root from a trusted source. With the hash values highlighted in green (and supplied by a full node) and the transaction data (highlighted in yellow), Eve can reconstruct the Merkle tree root and then compare it to the value on her SPV node. If the two values agree, Eve can be sure that the transaction has been included in the block.

Chapter 6

Proof of Work

6.1 Introduction

This section deals with what is arguably the most fascinating part of Bitcoin. It binds time and energy and thus value to the digital world. Proof of work offers a way for someone to verifiably prove to anyone that a certain amount computational work has been done. The key features of a Proof of Work system are:

- the work (or computation) should be hard but not infeasible; and
- it should be easy to verify that work has been done.

In the next section I will go through Hashcash, which is a system proposed by Adam Back in 1997 to reduce email spam by incurring on the sender a certain computational cost. Bitcoin and other cryptocurrencies have adopted the same proof of work system used in Hashcash. However, in the case of Bitcoin proof of work is used to ensure the integrity of a peer to peer distributed ledger by mandating a huge amount of resources be required to update this ledger.

6.2 Hashcash

From its name it is not surprising that Hashcash relies heavily on the cryptographic hash function, however there is no token or currency associated to the system that a user can own or trade. The concept of Hashcash is quite elegant, the sender would find and add a cryptographic nonce (number once) to their message such that the hash of the message satisfies certain properties. The receiver can then simply filter out the emails whose hash does not satisfy these properties. The idea being that scammers whose business model relies on sending spam emails to thousands of recipients cheaply would not be able to afford the processing power to find a valid proof of work for each email. However legitimate senders of email would not mind waiting a couple of seconds while their computer found a valid proof of work before sending their message.

To determine what these properties of the hash should be it will be useful to think of a hash function as such:

$$hash : \mathbb{N} \longrightarrow \{1, 2, 3, \ldots, \Psi\} \subset \mathbb{N},$$

where Ψ is astronomically large, and $\forall m, n \in \mathbb{N}$, such that $m \neq n$, $|m - n|$ tells you no information about $|hash(m) - hash(n)|$. Clearly this function is not injective because the domain is larger than the codomain but if we assume our hash function is *ideal* then we should have that for $m, n \in \mathbb{N}$ chosen randomly the probability that $hash(m) = hash(n)$ is equal to the probability of two randomly chosen elements from the codomain being equal, namely

$$\frac{1}{\Psi}.$$

This probability is so small that it is highly improbable for anyone to find $m \in \mathbb{N}$ such that $hash(m) = hash(n)$ when $m \neq n$. For example, in the cryptographic hash function SHA256 (which is widely used in Bitcoin) we have $\Psi = 2^{256} \approx 10^{77}$. For scale, estimates for the number of atoms in the universe range from 10^{78} to 10^{82}. However, if we let $D \in \{1, 2, 3, \ldots, \Psi\}$ then the probability that a randomly chosen $m \in \mathbb{N}$ satisfies $hash(m) \leq D$ is

$$\sum_{i=1}^{D} \mathbb{P}(hash(m) = i) = \sum_{i=1}^{D} \frac{1}{\Psi} = \frac{D}{\Psi} \ ,$$

thus giving us a way of varying the *difficulty* of finding such an m. We can increase the difficulty by *decreasing* D and decrease the difficulty by *increasing* D.

Hopefully you have some ideas about how the hash function could be used to construct a Proof of Work system, in the next section I'll try and demonstrate how Hashcash works using some Python code.

6.3 Python demonstration

In the following implementation I measure *difficulty* by the number of leading zeros a message's hash is required to have to be considered legitimate and not spam. It's worth noting that the computer finds a nonce which produces a valid proof of work, by producing a random number, appending it to the email, calculating its hash and checking if it has at least the desired number of leading zeros. If it doesn't it will repeat this process again until it finds a valid nonce. To run through the demonstration you'll need the blockchain_project.py file (found on my GitHub account Satsuma-LN in the Mathematics-behind-blockchain-project repository).

```
>>> import blockchain_project as bp

>>> bp.Hashcash_Message.find_appropriat_difficulty()

A difficulty of 5 should require at least 2 seconds of work from this
   computer.

>>> recipient = "Timothy C.May"

>>> message = "Privacy is the power to selectively reveal oneself to
   the world."

>>> test_hashcash_msg = bp.Hashcash_Message(
...           recipient,
...           message,
...           difficulty=5,
...           verbose=True)

It took 302999 attempts to find a valid nonce.

>>> print(test_hashcash_msg)

Recipient: Timothy C.May
Nonce:
   98958446389545255982712332171491307028228641815471035144663979947670924565459
Message:
Privacy is the power to selectively reveal oneself to the world.

>>> nonce =
   "98958446389545255982712332171491307028228641815471035144663979947670924565459"

>>> bp.Hashcash_Message.verify_proof_of_work(recipient, nonce, message)

Valid proof of work with difficulty 5, the hash is:
00000b8d1b2cf5af6f8e796ee1d16ddd1f5a5791d0b43e36a65ec658940a0aef
True
```

Requiring more leading zeros has the same effect as decreasing D from the previous section, thus increasing the difficulty and hence time taken to find a valid nonce. I recommend playing around with the last function call, if you change any one of the three inputs (recipient, nonce or message) you should find that the hash no longer has enough leading zeros, and thus would be considered spam by the email filter.

```
>>> bp.Hashcash_Message.verify_proof_of_work(
...         "Eric Hughes",
...         nonce,
...         message)

WARNING: No work done. This could be spam. The hash is:
ad19d4c4b266e2f67e3333d6584c2a0c7c551aa88b3da2fe834968069dcc15ce
False

>>> bp.Hashcash_Message.verify_proof_of_work(
...         recipient,
...         str(int(nonce) + 1),
...         message)

WARNING: No work done. This could be spam. The hash is:
4ee7b79977214ec1e50fafd30fdcf470a30b99e23edf760ca717917f8b1feecc
False

>>> bp.Hashcash_Message.verify_proof_of_work(
...         recipient,
...         nonce,
...         "Buy this really good lawn mower!")

WARNING: No work done. This could be spam. The hash is:
ba18e2b44cc393acd93888d25e81aebbff94055831224159730a255ed65310aa
False
```

Chapter 7

Conclusion

Bitcoin is a vast topic, touching on a diverse set of fields; computer science, mathematics, economics, anthropology and I'm sure there are more. There is an internet meme amongst Bitcoiners that Bitcoin is a rabbit hole whose bottom has not been found yet. I hope that this document sparks some interest in the reader and complements some of the educational content already out there. While writing this I found Andreas M.Antonopoulos' book 'Mastering Bitcoin' especially helpful.