Blockchain and Smart Contracts Downloaded from www.worldscientific.com
by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

Chapter 2

Bitcoin Mining and Python Programming Demonstration

2.1 Getting Started

Decentralised as the bitcoin network is, the bitcoin mining process may be difficult to understand for many. Basically, the underlying blockchain technology is a distributed public ledger where bitcoin transaction data are recorded. Each block in the bitcoin blockchain contains transaction data and blocks of data are chained together using cryptography (see Figure 2.1).

Transactions are grouped into a block that will be added to the end of the current blockchain. The process is called mining and the node that does it is known as the miner. As a reward, miners get certain amount of bitcoins (currently 12.5 BTC) for every block successfully mined and they can also collect the transaction fees of the transactions included in the block. How to determine when a block is successfully mined, or the consensus rule, in the bitcoin network is the Proof of Work (PoW) algorithm. The bitcoin blockchain architecture is summarised in Figure 2.2.

This chapter aims to provide a demonstration of mining in Python and a better understanding of the mining process of bitcoins. Do note that this is a simplified version and may differ from the real-life bitcoin mining as we hope to provide you with some interactive and programming experiences. When explaining the code line by line, we will go through the bitcoin mining process step by step and illustrate the meaning of each step as well.

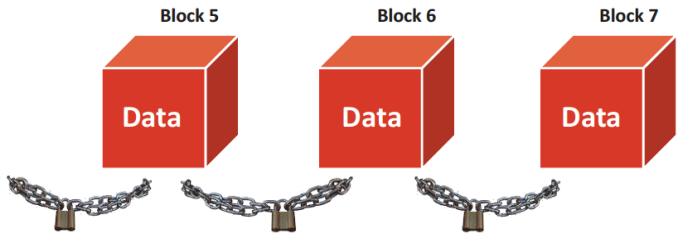

Figure 2.1: Blockchain structure.

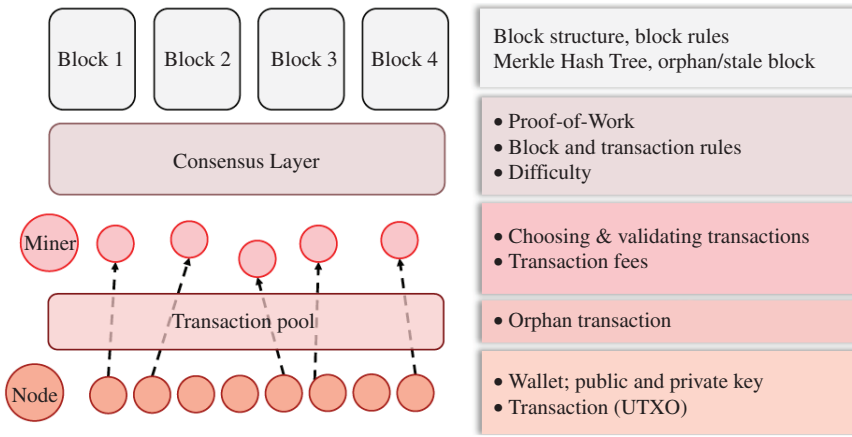

Figure 2.2: Summary of blockchain architecture.

2.2 Introduction of Python Programming

There are two ways to insert comments in Python: (1) to start with "#" and whatever follows the hashtag in that line will become comments or notes; (2) to start a line with """ and end it with """ so that whatever is in between will become comments for notes (see Figure 2.3). The former is usually used for shorter comments such as one line or a few words at the end of a line, while the latter is usually used for longer comments such as several lines. These comments or notes will not be run as codes.

In Python, there are many functions, codes or libraries readily available for users to use, instead of writing one from scratch. Such codes or scripts can be saved in a file named module. In order

Blockchain and Smart Contracts Downloaded from www.worldscientific.com by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

```
"""
Python code for bitcoin mining demonstration
"""

import pandas as pd
```

Figure 2.3: Importing modules in Python.

to use an existing module or a set of libraries or modules (also referred to as packages) in your Python IDE (Integrated Development Environment) like PyCharm, you need to import the library first. Importing a library means loading it into the memory and then it is there for us to work with. "pandas" is an important module in Python, especially for dealing with financial data. It stands for "Python Data Analysis Library", which is derived from the term "panel data". It allows us to see the data as a table format or "series"/"data frame" type in Python. The code "import pandas as pd" in Figure 2.3 means we can call the module using two letters "pd" instead of the full six letters "pandas" for brevity. "pd" is widely used as abbreviation for "pandas".

2.3 Data Import

For the demonstration, we can use a simplified version of transaction data and transaction ID, but the idea is the same: miners need to import the transaction data first and select the transaction IDs before they can proceed to the mining.

Assume that the transactions available, or the transaction pool in bitcoin, are stored in a local file named "transactions.txt". We can use the "with open" command to import the text file into Python (see Figure 2.4). We can include the full directory of the file (e.g., "C:/$\cdots$/transaction.txt") or we can include the file in the same Python project folder and use only the file name "transaction.txt". The "with open" command is also applicable if the data are stored in other formats such as csv. We can also use other methods such as the csv or pandas module to import a csv file.

Blockchain and Smart Contracts Downloaded from www.worldscientific.com
by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

```
# From local file
with open('transactions.txt', 'r', encoding='utf-8') as f:
    tx = f.readlines()
# To create an empty list
tx_list = []
# To change the tx data from txt file to a list
for t in tx:
    t = t.replace('\n', '')
    tx_list.append(t.split(', '))

# To change it to the data frame type using the "pandas" library for better
display
tx_data = pd.DataFrame(tx_list, columns=['id', 'content', 'tx_fee'])
```

Figure 2.4: Importing transaction data.

The data in the text file are then stored as a list in *tx*, with each line as an element of the list (see Figure 2.4). To clean the transaction data, we would like to store each row in different columns (similar to Excel) and remove the line breaker "\n" at the end of each line. So, we create a new list named *tx_list* and leave it empty ("[]" refers to a list). For each element t in *tx*, we delete the line breaker "\n" by replacing it with an empty string. *t.split* will convert each line (t) in transaction data (*tx*) into different columns using the separator — comma. Then, we store the separated line data in the list *tx_list* with each line as a list.

In the last line of Figure 2.4, we convert the list to dataframe type by calling the pandas module, with the column names as "id", "content" and "tx_fee" (here we use simplified transaction data; the column names can be specified according to the data using the following format). The transaction data are now well organised with transaction id, transaction content and transaction fee data in separate columns.

Next, with reference to Figure 2.5, we choose the transaction IDs from the transaction data. The input() function allows users to use the keyboard to key in data, which will be stored as strings. The sentence in the parenthesis is what shows on the screen preceding the cursor. The "\n" at the end of the line allows the user to key in data in a new line. We can split the string input by space and get a

Blockchain and Smart Contracts Downloaded from www.worldscientific.com by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

```
tx_in = input("Please key in the transaction IDs (separated by spacing):\n")

tx_in_list = tx_in.split(' ')
tx_in_list = [t.replace('\\s+', '') for t in tx_in_list]  # To remove one or
more space
tx_in_list = [int(t) for t in tx_in_list]

instr_list = []
for t in tx_in_list:
    print(t)
    if t not in list(tx_data['id']):
        print('The ID of %s is not a valid input and has been excluded. ' % t)
        continue
    instr_list.append(t)

print('='*30)
print('The list of valid transaction IDs selected is: ', instr_list)

in_data = pd.DataFrame()
for id in instr_list:
    in_data = in_data.append(tx_data[tx_data['id']==id])

print(in_data)
```

Figure 2.5: Selecting transaction IDs.

list of transaction IDs selected. There may be extra spacing in *tx_in* and thus *instr_list*, so we delete the spacing from *instr_list*. This will generate a list of transaction IDs without extra spaces, but they are still string type resulting from the input() function. So, we convert the IDs to integer type using the int() function.

Transaction IDs that are not included in the transaction data *tx_data* should be removed from the list (see Figure 2.5). So, in the *for* loop, for every element *t* or every transaction ID in the *tx_in_list*, we use the *if* statement to skip the step of appending the transaction ID to the ID output list *instr_list* using "continue" (which means to skip the following lines in the loop and continue to the next loop) if that transaction ID is not in the ID column of the transaction data (*tx_data*['id']). If all the transaction IDs selected are valid, all the IDs will be included in the final ID output *instr_list*.

Then, we create an empty data frame using *pandas* to store the full transaction records for the transaction IDs selected

Blockchain and Smart Contracts Downloaded from www.worldscientific.com
by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

using the "*for* loop" (see Figure 2.5). The *for* loop and *tx_data[tx_data['id']==id]* return only the rows in *tx_data* where the ID in *tx_data['id']* appears in the *instr_list* (the list that stores the transaction IDs selected). The print output will be the list of valid transaction IDs selected and the corresponding transaction data. Printing 30 equation symbols is just to make the layout nicer.

Then, we can have the transaction data imported and nicely presented in Python. This is a demonstration of data files import and data clean in Python. In the following sections, we will use a simple four-transaction example to further elaborate the mining process.

2.4 Transaction

Before we proceed to deal with the transaction data in Python, let us run through the information contained in a Bitcoin transaction and how it works. As mentioned earlier, transaction data will be recorded in the bitcoin blockchain. Like any other conventional transaction, a bitcoin transaction needs information on the receiver, sender and the amount of money. First, a sender needs to have a certain amount of bitcoins to spend, and he can initiate a transaction by specifying a bitcoin amount and a sender. This transaction will then go to a transaction pool for miners to validate and be included in a block with other transactions. The Bitcoin blockchain ensures that transaction data included in a block cannot be altered. How can it achieve that? The answer is the bitcoin blockchain structure and information included in the transaction. Whether the transaction data included in a specific block have been tampered with can be easily verified using the Merkle Hash Tree method in the cryptography. Understanding transaction, transaction data and Merkle Hash Tree is important to understand bitcoin mining and blockchain.

2.5 Bitcoin Transaction

Similar to the bank account in a conventional online banking transaction, each bitcoin transaction has a sender and receiver (Nakamoto, 2019). But, a bitcoin transaction contains more information and it can have several transactions. So, the term input and output are used

Blockchain and Smart Contracts Downloaded from www.worldscientific.com by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

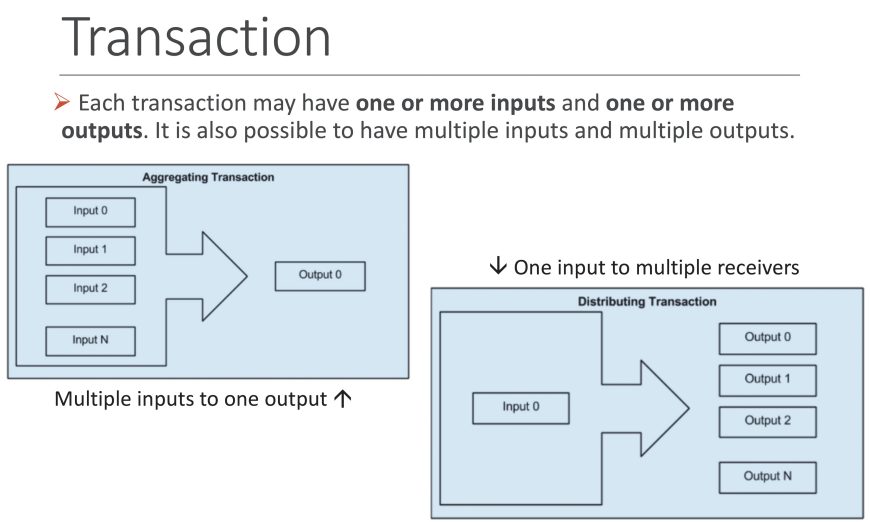

Figure 2.6: Inputs and outputs in a transaction.

in Bitcoin and a bitcoin transaction may have one or more inputs and one or more outputs (see Figure 2.6). Consider it as a one-way flow of bitcoins. Input shows where the bitcoins came from (could be multiple sources) and output points to where they are going (could be multiple receivers). The number of inputs in a transaction is called "Input Counter" and the number of outputs "Output Counter", both of which will be included in the bitcoin transaction data.

As we all know, a key aspect of blockchain is that once the information is recorded, it is there forever and cannot be changed. This spirit of blockchain can be viewed in a single transaction as well. Although the bitcoins are non-fungible (meaning we do not distinguish between different bitcoins), the sender needs to specify the source of the transacted amount of bitcoins as part of the input information in a transaction, and the concept of remaining account balance is not applicable in bitcoin. Unlike a bank account, we can see the balance left in a lump sum and we can spend it all together without linking to previous transactions; the balance of bitcoins goes by transactions. In each transaction, the input information will point to previous transactions, to which a specific amount of bitcoins is

Blockchain and Smart Contracts Downloaded from www.worldscientific.com by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

➢ The concept of a change in bitcoin.

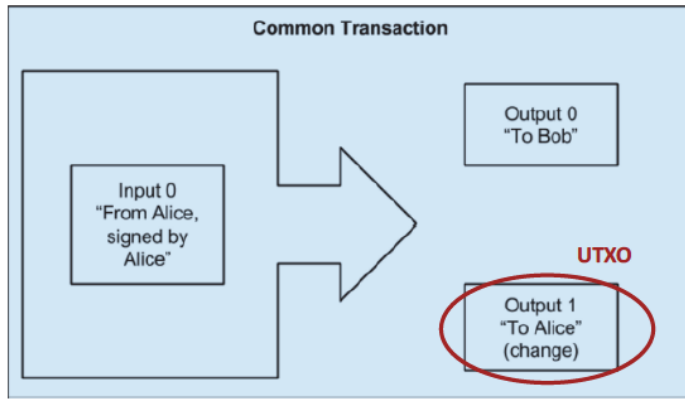

Figure 2.7: UTXO and the concept of change.

linked. In this way, it would be very effective and easy for miners to verify whether the sender owns the bitcoins.

Furthermore, the receiver information, or the transaction output in bitcoin, is different from conventional online transactions. In a bank transfer, we can simply transfer the money to another account with the rest remaining unchanged as long as the amount we are trying to send is no more than the total amount available, while in bitcoin, we need to include the bitcoins left as part of the transaction output. For each transaction, if there are unspent bitcoins left, this amount will be sent to the sender's address again as an unspent transaction output (UTXO), which refers to bitcoins from a previous transaction that has not yet been spent (or unspent) and can be the input of another transaction in the future (Figure 2.7). All transaction inputs in bitcoin are UTXOs and can be linked to previous transactions that indicate the source of unspent bitcoins.

In summary, a typical bitcoin transaction contains the following information as shown in Figure 2.8:

- Version that indicates the consensus rules to follow.
- Input counter that shows the total number of inputs.

Blockchain and Smart Contracts Downloaded from www.worldscientific.com by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

Blockchain and Smart Contracts Downloaded from www.worldscientific.com
by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

Version		Rules (consensus) to follow	
Input Counter		Number of inputs	
Inputs	Prev output hash	Previous UTXO identifier	
	Prev output index	Previous UTXO index	
	scriptSig size	Size of sender's signature	Proof of Ownership
	scriptSig	Sender's signature	
	Sequence number	Currently disabled; set "FF FF FF FF"	
Output Counter		Number of outputs	
Outputs	Amount	Bitcoin amount to send	
	scriptPubKey size	Size of receiver's scriptPubKey	
	scriptPubKey	Condition to spend the output (Receiver's public key)	
Locktime		Time before the tx can be added to blockchain	

Figure 2.8: Information contained in a Bitcoin transaction.

- Input information such as previous output hash (previous UTXO identifier), previous output index (previous UTXO index), scriptSig size (size of the sender's signature), scriptSig (sender's signature) and sequence number (currently disabled).
- Output counter that shows the total number of outputs.
- Output information such as amount (bitcoin amount to send), scriptPubKey size (size of receiver's public key) and scriptPubKey (condition to spend the output which is the receiver's public key).
- Locktime that suggests the time before the transaction can be included in a block and added to the blockchain.

Among them, the inputs and outputs in a transaction provide the information about the sender and receiver, including the sender's source of wealth, proof of ownership (see Figure 2.8) or proof that the sender owns the bitcoin by a digital signature signed using the sender's private key, receiver's address or public key, and the amount to receive. The input counter shows the number of sources or previous transactions linked (where the bitcoins come from) in this transaction

and the output counter shows the number of outflows (where the bitcoins go) with unspent amount transferred to the sender's own address.

Blockchain and Smart Contracts Downloaded from www.worldscientific.com by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

2.6 Hash

Since there is a lot of information included in a transaction, bitcoin adopts a more efficient way to incorporate the transaction data in the mining while ensuring the integrity of the data at the same time. Namely, the transaction information will then be taken into a hash function and the generated hash value of the information will be used as transaction ID that is used to identify the transaction. The hash value of any input has the same length and any slight changes in the input will cause the resulted hash value to be completely different with no pattern. Such features of hash ensure that as long as we have the same transaction ID, the data and information in a transaction have not been tampered with.

What is hash and why is used in bitcoin transactions? Hash, or hashing, refers to a function that takes data input of any length and returns a value of fixed length (Sobti & Geetha, 2012). The value returned by a hash function is also known as hashes, hash values, hash codes or digests. We can create hashes of all kinds of digital content of any length and form (e.g., numbers, letters, and symbols; documents, images and music) and the output will be a value of fixed length, which makes the calculation process neater and easier. The use of a hash function can also easily suggest whether the transaction information remains unchanged or not. This is because hashing the same input will always generate the same hash values and any changes, even the slightest, in the input will result in a totally different hash value.

The hash function used in bitcoin is SHA-256, which returns a fixed length of 256 bits (32 bytes). So, the transaction ID in bitcoin would look like this:

B94d27b9934d3e08a52e52d7da7dabfac484efe37a5380
ee9088f7ace2efcde9

Blockchain and Smart Contracts Downloaded from www.worldscientific.com
by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

Now, that we use the hash values to represent a transaction (at transaction level), is there an efficient way to aggregate multiple transactions or hash values (at block level) as many transactions may be included in a single block? In fact, for each block, the transactions selected by the miner are aggregated by Merkle Hash Tree and reflected as a single number, the Merkle Root.

The Merkle Hash Tree is a binary tree that would calculate the hash value of every two transactions until the Merkle Root is reached. The Merkle Root summarises all the transactions gathered within this block and is used as an efficient way to determine if a particular transaction is included within a block and to verify if all transactions have not been tampered with (Merkle, 1980). A simplified version of Merkle Hash Tree is shown in Figure 2.9. If the information within a transaction is changed, the hash value of the transaction will change (also known as the leaf) and the second layer (or the branch) will change, leading to a different Merkle Root because any changes in the input of a hash function will generate a completely different output or hash values. We will demonstrate this process in Python using an example of four transactions.

What does this imply? This means that if an attacker tries to modify a transaction (say Alice to Bob: 1 btc), he is effectively trying to find another transaction with the same hash value as Alice to Bob: 1 btc in order to be not detected. This is not possible because it is computationally not feasible as to find two different inputs

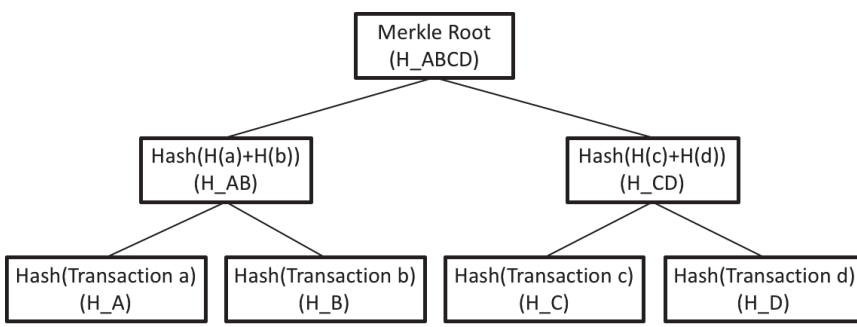

Figure 2.9: Structure of a Merkle Root.

(or transaction data) with the same hash output (or hash values) is extremely difficult with current computing power.

Same as earlier, we need to import the packages needed. Since Bitcoin uses SHA-256 cryptographic hash function, we can import it from the Python library available named "hashlib" (see Figure 2.10). Note that if we use "import hashlib" instead of "from hashlib import sha256" at the beginning, the code would be "hashlib.sha256()" instead of "sha256()", and if we would like to call the function as Python, it would only recognise "hashlib" not "sha256" as a valid function.

Next, looking at the functions in Figure 2.11, we define two functions: dsha() and rev(). Bitcoin uses double hashing, meaning for the input, we need to use the sha256() function twice. In Python, "sha256(tx).digest()" returns the hash value, or digest, of the input "tx". So, for bitcoin, the code would be "sha256(sha256(tx).digest()).digest()". Instead of writing so many strings every time, we can define a function named "dsha()" to return the double hash results. Similarly, we can define a function named "rev()" to reverse the elements in the input "buf", which will be used later in the calculation. "buf[::−1]" means to cover all elements (by default as there are no numbers specified before each colon), from the last to the first and one at each time (determined by the "−1" after the second colon).

```
from hashlib import sha256  # The hash algorithm used in Bitcoin
```

Figure 2.10: Importing *sha256*.

```
def dsha(tx):  # Double hashing the transaction data 'tx' in Bitcoin
    return sha256(sha256(tx).digest()).digest()

def rev(buf):  # Reverse the elements in 'buf'
    return buf[::-1]
```

Figure 2.11: Defining functions *dsha* and *rev*.

Blockchain and Smart Contracts Downloaded from www.worldscientific.com by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

Let us start with the following four transactions, A, B, C and D (see Figures 2.12 and 2.13). The strings in the parenthesis are the corresponding transaction IDs. The "bytes.fromhex()" function converts hexadecimal, or hex, strings to binary strings.

The hash values of the four transactions can be calculated using the following commands (see Figure 2.14). Note that output from the "dsha()" function is binary, but it is easier for us to see the hash

```
tx_A =
bytes.fromhex('0000000003ce0000000000000000000000000000000000000000000000000000
00000000000ffffffff08044c86041b020602ffffffff0100f2052a010000004341041b0e8c256
7c12536aa13357b79a073dc4444acb83c4ec7a0e2f99dd7457516c5817242da796924ca4e99947
d087fedf9ce467cb9f7c6287078f801df276fdf84ac00000000')

tx_B =
bytes.fromhex('0000000001032e38e9c0a84c6046d687d10556dcacc41d275ec55fc00779ac8
8fdf357a187000000008c493046022100c352d3dd993a981beba4a63ad15c209275ca9470abfcd
57da93b58e4eb5dce82022100840792bc1f456062819f15d33ee7055cf7b5ee1af1ebcc6028d9c
db1c3af7748014104f46db5e9d61a9dc27b8d64ad23e7383a4e6ca164593c2527c038c0857eb67
ee8e825dca65046b82c9331586c82e0fd1f633f25f87c161bc6f8a630121df2b3d3ffffffff020
0e32321000000001976a914c398efa9c392ba6013c5e04ee729755ef7f58b3288ac000fe208010
000001976a914948c765a6914d43f2a7ac177da2c2f6b52de3d7c88ac00000000')
```

Figure 2.12: Transaction A and B.

```
tx_C =
bytes.fromhex('0f00000001c33ebff2a709f13d9f9a7569ab16a32786af7d7e2de09265e41c6
1d078294ecf010000008a4730440220032d30df5ee6f57fa46cddb5eb8d0d9fe8de6b342d27942
ae90a3231e0ba333e02203deee8060fdc70230a7f5b4ad7d7bc3e628cbe219a886b84269eaeb81
e26b4fe014104ae31c31bf91278d99b8377a35bbce5b27d9fff15456839e919453fc7b3f721f0b
a403ff96c9deeb680e5fd341c0fc3a7b90da4631ee39560639db462e9cb850fffffffff0240420
f00000000001976a914b0dcbf97eabf4404e31d952477ce822dadbe7e1088acc060d2110000000
01976a9146b1281eec25ab4e1e0793ff4e08ab1abb3409cd988ac00000000')

tx_D =
bytes.fromhex('0c400000670b6072b386d4a773235237f64c1126ac3b240c84b917a3909ba1c
43ded5f51f4000000008c493046022100bb1ad26df930a51cce110cf44f7a48c3c561fd977500b
1ae5d6b6fd13d0b3f4a022100c5b42951acedff14abba2736fd574bdb465f3e6f8da12e2c53039
54aca7f78f3014104a7135bfe824c97ecc01ec7d7e336185c81e2aa2c41ab175407c09484ce969
4b44953fcb751206564a9c24dd094d42fdbfdd5aad3e063ce6af4cfaaea4ea14fbbffffffff014
0420f00000000001976a91439aa3d569e06a1d7926dc4be1193c99bf2eb9ee088ac00000000')
```

Figure 2.13: Transaction C and D.

Blockchain and Smart Contracts Downloaded from www.worldscientific.com
by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

```
ha = dsha(tx_A)
hb = dsha(tx_B)
hc = dsha(tx_C)
hd = dsha(tx_D)

# To print the hash values in hex string of the four transactions
print("H_A:", ha.hex())
print("H_B:", hb.hex())
print("H_C:", hc.hex())
print("H_D:", hd.hex())
```

Figure 2.14: Calculating hash values of transactions.

```
H_A: 607fd0327a3b7773690219764d9df3188b3df63dc927a62e088295838c5c52d2
H_B: f5a205e1aa269d59c0e4f17b6d3f934d5c3e32826949b777710cf61276512ef9
H_C: 59e0e7fbd76c65b24371899472ffa5f39514a8eca2de48e800c65dd53ea85940
H_D: bc2a504bcbf5e3e4c938394cd7b53f39f34acc73870dc23dac34b1540de1ddee
```

Figure 2.15: Output of the four transactions.

```
hab = dsha(ha+hb)
hcd = dsha(hc+hd)

print("H_AB:", hab.hex())
print("H_CD:", hcd.hex())

habcd = dsha(hab+hcd)

print("H_ABCD:", habcd.hex())
print("Merkle root:", rev(habcd).hex())
```

Figure 2.16: Calculating the Merkle Root.

values in hex strings, so we present the result in hex strings using ".hex()" function.

The output is displayed as follows in Figure 2.15.

Following the Merkle Hash Tree, the Merkle Root can then be calculated as the hash of the sum of *hab* and *hcd* (see Figure 2.16).

The output is shown in Figure 2.17.

Blockchain and Smart Contracts Downloaded from www.worldscientific.com by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

```
H_AB: 33878862fd223395836ecb79b090370f571a579cea8330cc34eb0515c6038e28
H_CD: 24770cb6ff08789585d492d7e4fd4b2f19f78bce57fb17b8259fdefb37d470ac
H_ABCD: e947ad0314c5d58e0165282159584c04f0eb58200743a4f2c4fa3adf6a10f4c1
Merkle root: c1f4106adf3afac4f2a443072058ebf0044c5859212865018ed5c51403ad47e9
```

Figure 2.17: Output of the Merkle Root.

Therefore, the Merkle Root of our four transactions is

"c1f4106adf3afac4f2a443072058ebf0044c5859212865018 ed5c51403ad47e9"

This will then be used to mine a block and stored in the block header.

Why do we need the reverse function for the Merkle Root? Basically, this has something to do with how data is stored in the computer and how we prefer it to appear. When considered as input of the hash, or double hash, in the case of Bitcoin function, transaction IDs need to be in binary strings and little endian machines, which refer to an order that stores the low-order byte of the number ("little-end", or the least significant value in the sequence) first (at the lowest address) and high-order byte ("big-end", or the most significant value in the sequence) last (at the highest address). But, the blockchain explorers display the data in hexadecimal and big endian machines, which store the "big-end" first and "little-end" last. Therefore, we need to reverse the characters in the hexadecimal Merkle root strings to convert it back to big endian machines, thus the usage of reverse function defined earlier on.

2.7 Block Structure

Each block consists of a block header and transaction data (see Figure 2.18). Full transaction data are included in a block, but only the Merkle Root is included in the block header as an aggregation and presentation of the transactions in that block. Some nodes can keep and sync merely the block header information, instead of the full transaction data, to save space. The block header is crucial in blockchain and mining as each block is cryptographically chained

Blockchain and Smart Contracts Downloaded from www.worldscientific.com by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

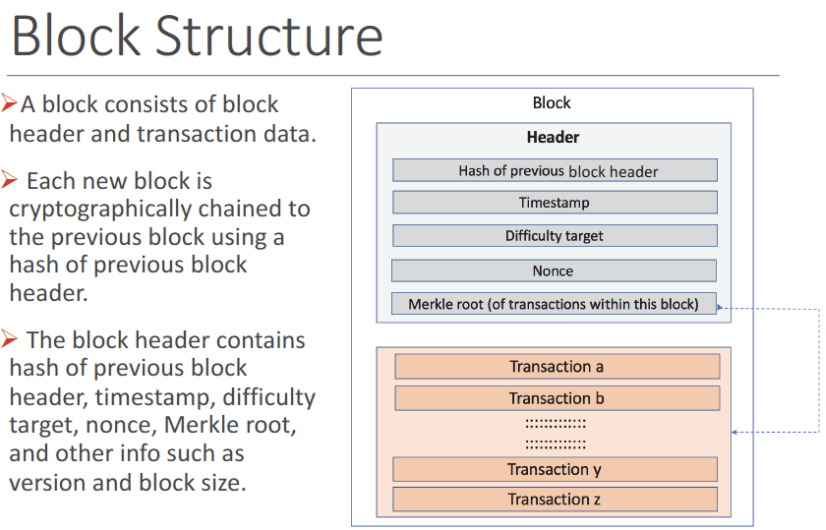

Figure 2.18: Overview of block structure.
Source: https://luxsci.com/blog/understanding-blockchains-and-bitcoin-technology.html.

to the previous block using a hash of the previous block header. It contains the hash of the previous block header, timestamp, difficulty target, nonce, Merkle Root and other information such as version and block size.

The blockchain is designed in such a way that is constantly appending new blocks of validated transactions to the end of its chain. The chain can only grow and blocks cannot be removed or amended once confirmed and appended. Since every block contains a unique hash value of the previous block, if an attacker tries to change content of the previous block, its hash value will change. Thus, the attack will be detected because all subsequent hash values will not match. Just as the hash of a transaction reflects the information within the transaction and the Merkle Root reflects the transactions included in a block, the hash value of the block header reflects all the information in the block.

Here, looking at the codes in Figure 2.19, we include the version, the hash value of previous block and time information, to be included

Blockchain and Smart Contracts Downloaded from www.worldscientific.com by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

```
ver = 2
prev_block =
"000000000000000117c80378b8da0e33559b5997f2ad55e2f7d18ec1975b9717"
time_ = 0x5D21A868  # 2019-07-07 08:08:08
```

Figure 2.19: Given information to be included.

in the block header later. The above number for *time_* is the Unix epoch (or Unix time or POSIX time or Unix timestamp), which is the number of seconds that have elapsed since January 1, 1970 (midnight UTC/GMT), not counting leap seconds (in ISO 8601: 1970-01-01T00:00:00Z). It is equivalent to 2019-07-07 08:08:08.

2.8 Mining

Mining refers to the process of finding a valid block to add to the current blockchain and it follows certain rules or consensus rule, which is Proof of Work (PoW) in bitcoin. Miners are the nodes that do mining. Since there are no centralised parties like banks or governments to control the issuance or validate the account balance and transactions, the consensus rule, or PoW, is what the bitcoin network relies on. The nodes in the bitcoin network reach the consensus that a block is genuine through the block header's hash value — whether it satisfies a given condition or meets a given target (Nguyen and Kim, 2018). Just like solving a math problem, this condition is explicit and stated in the bitcoin protocol so that everyone in the community must comply, which ensures trust among the bitcoin community.

In Bitcoin's Proof-of-Work mechanism, the blockchain that was created with the greatest cumulative effort expended is the one considered to be the latest and to which new blocks are attached. That is to say, the longest blockchain in the system is the accepted state of the bitcoin ledger as it requires the greatest effort to create. Since the rest of the information included in the block header hash value computation is more or less given, the most variable factor is the nonce, so miners will keep trying different nonce values to generate different hash values of the block header. That said, without changing any other data in the block header (definitely not the

Blockchain and Smart Contracts Downloaded from www.worldscientific.com
by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

transactions), the Proof-of-Work process is a competition among miners to find "the Golden Nonce" that will result in a block fingerprint (i.e., the hash value of the block header) that satisfies a given condition or meets a given target.

2.9 Target and Mining Difficulty

In the bitcoin blockchain, the target is a 256-bit number that all bitcoin nodes share. It is sometimes referred to as the difficulty target as it measures how difficult it is to mine or generate a new block. The SHA-256 hash of a block's header must be lower than or equal to the current target for the block to be accepted by the network. It is usually expressed in hexadecimal. An example of target is as follows:

0x0000000000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF.

The more the leading zeros, the lower the target (meaning in decimal the number is lower), and thus the more difficult it is to find a block fingerprint that satisfies the condition or meets the target. What is hexadecimal? The hexadecimal numeral system, often shortened to "hex", is a numeral system made up of 16 symbols (base 16). The standard numeral system is called decimal (base 10) and uses ten symbols: 0,1,2,3,4,5,6,7,8,9. Hexadecimal uses the decimal numbers and six extra symbols (see Figure 2.20). We often see the block hash represented as hexadecimal numbers (64 digits) which result from SHA-256 hashing. Each hexadecimal number is four bits, so the output of SHA-256 is a 256-bit number (64 hexadecimal digits $\times$ 4 = 256 bits), and thus the name.

Sometimes, a target is given in "bits" or a "compact" format of the target such as "1903a30c" or 0x1903a30c. The exponent is 0x19 and the coefficient is 0x03a30c. The target can be calculated using the following formula:

$$Target = Coefficient * 2^{8*(Exponent-3)}$$

So, for a difficulty bits value of 0x1903a30c, we have

$$Target = 0x03a30c * 2^{0x08*(0x19-0x03)}$$

Blockchain and Smart Contracts Downloaded from www.worldscientific.com by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

Blockchain and Smart Contracts Downloaded from www.worldscientific.com
by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

Hexadecimal	Binary	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Figure 2.20: Hexadecimals, binary and decimal.

The target is then 0x0000000000000003A30C0000000000000000000000000000000000000000000000000000.

What does "0x" *mean? It simply means that what follows is in hexadecimal, to avoid confusion with other data types such as decimal or strings.*

Since it is difficult to compare two targets using the hex strings or decimal values as they are very large integers, difficulty can be viewed as another representation of the target that makes the comparison easier.

Difficulty is expressed as follows:

$$difficulty = \frac{difficulty\_1\_target}{current\_target}$$

Difficulty_1_targets is defined as the easiest allowed target value, 0x1d00ffff, which gives us a hex target of

$$0x00ffff * 2 ** (8 * (0x1d - 3))$$
$$= 0x00000000FFFF0000000000000000000000000000000000000000000000000000$$

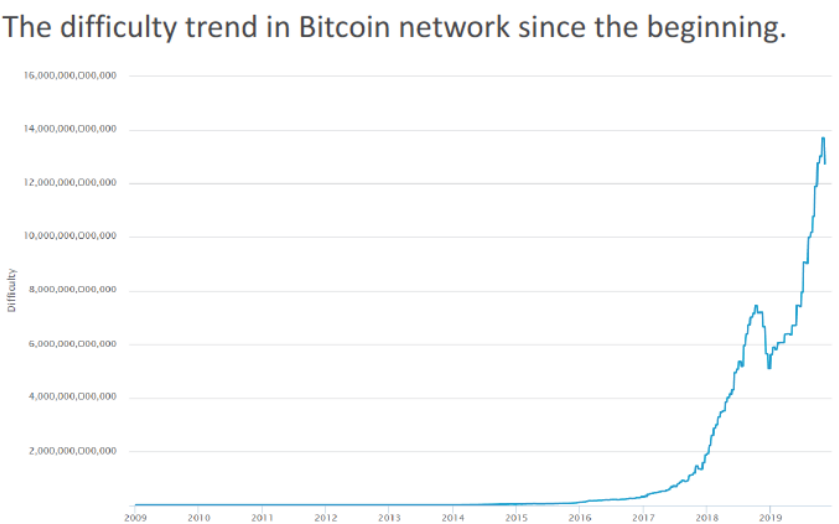

Figure 2.21: Increase in difficulty of mining Bitcoins.
Source: https://www.blockchain.com/charts/difficuly?timespan=all.

Difficulty is a network-wide setting that controls how much computation is required to produce a proof of work. It is a relative measure of how difficult it is to find a hash below a given target or to find a new block. The bitcoin network has a global block difficulty, which is adjusted periodically as a function of how much hashing power has been deployed by the network of miners to ensure that it takes 10 minutes on average to add a new block to the bitcoin blockchain (O'Dwyer and Malone, 2014).

How often does the network difficulty change? It changes roughly every two weeks, depending on the network block generation speed or the overall computing power (see Figure 2.21).

The formula for a new difficulty level is as follows:

$$\text{New difficulty} = \text{Old difficulty} * \left(\frac{\text{actual time taken for last 2016 blocks}}{\text{20160 minutes}} \right)$$

Let us assume a simpler target to demonstrate how the target is calculated using bits value in Python (see Figure 2.22). Exponent

Blockchain and Smart Contracts Downloaded from www.worldscientific.com by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

```
bits = '0x1e03a30c'  # 5 zeros difficulty target
exponent = bits[2:4]
coefficient = bits[4:]
exponent2 = int("8", 16) * (int(exponent, 16) - int("3", 16))

target = int(coefficient, 16) * (int("2", 16)) ** exponent2
print("The target is: ", target)

target1 = format(target, 'x')  # To convert the string back to hex
print("The target (in hex) is: ", str(target1).zfill(64))
```

Figure 2.22: Calculating the difficulty target.

```
The target is
25100974094617268335075366062347480202344008198530079569242512873750528
The target (in hex) is:
000003a30c0000000000000000000000000000000000000000000000000000000000000
```

Figure 2.23: Output of the target.

is the third and fourth digit in bits, so it is the second and third element in the *bits* string since the number starts with 0 in a list and *bits[2:4]* means the second and third in *bits*, which is 0x1e. *bits[4:]* gets the 5th digit to the last digit in the bits value, which is 0x03a30c. Then, we can calculate the target using the abovementioned formula. While *int(var)* gives the integer of *var* in decimal, *int(var, 16)* tells the program that this integer *var* is in hexadecimal.

The output will be as follows (see Figure 2.23). We can see that the target in decimal is an extremely large number. It is much more neatly expressed in hex strings, which is why hex strings are widely used in the bitcoin blockchain. Without the zfill(64), the target would be 000003a30c and the function fills the rest of the places so that the final number is 64 digits.

2.10 Finding the Golden Nonce

If we recall the whole mining process again, once the miner selected the transactions, the Merkle Root will be calculated and remain the same as long as the transactions selected do not change. In previous sections, we calculated the difficulty target. The next step is to put

Blockchain and Smart Contracts Downloaded from www.worldscientific.com
by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

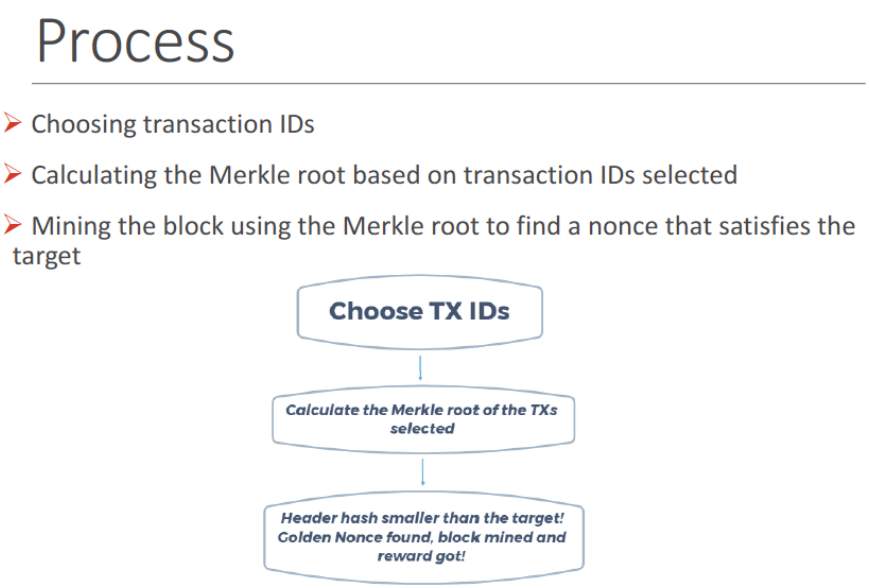

Figure 2.24: Process of Bitcoin consensus.

the rest of the information into the header and start finding the Golden Nonce in the mining process (see Figure 2.24).

The target value in bytes is stored in *target_byte* for block mining later (see Figure 2.25). We can also prepare the *partial_header* using the information we have now, including version, hash of previous block header, Merkle Root that aggregates the transactions included in this block and timestamp. *struct.pack("< L", x)* returns the input x as little-endian ordered unsigned long value. As mentioned in the transaction section, the endianness is a convention that determines in which direction a sequence of bits is interpreted as a number: from (Big-endian) left to right, or from (Little-endian) right to left. This is to follow the rules in bitcoin network.

From the *partial_header* (see Figure 2.25), we can see that among the information included in a block header, all the rest of the data are given and can be easily obtained except for the nonce. So, to successfully mine a new block is equivalent to finding a correct nonce that makes the hash value of the block header equal to or smaller than the target. Any change to the block header input (such as the

Blockchain and Smart Contracts Downloaded from www.worldscientific.com by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

```
target_byte = bytes.fromhex(str(target1).zfill(64))
partial_header = struct.pack("<L", ver) \
                 + bytes.fromhex(prev_block)[::-1] \
                 + bytes.fromhex(mrkl_root)[::-1] \
                 + struct.pack("<LL", time_, int(bits, 16))
```

Figure 2.25: Partial header and block header information.

nonce) will make the block hash completely different. Since it is believed infeasible to predict which combination of bits will result in the right hash, many different nonce values will be tried and the hash recomputed for each nonce value until a hash less than or equal to the current target of the network is found. As this iterative calculation requires time and resources, the presentation of the block with the correct nonce value (known as the golden nonce) constitutes PoW in Bitcoin (MacKenzie, 2019). The mining process can also be referred to as the process of finding the golden nonce.

What is a nonce? In cryptography, a nonce is an arbitrary number that can be used just once in a cryptographic communication (Gordon and Jeffrey, 2004). It is similar in spirit to a nonce word, hence the name. The "nonce" in a bitcoin block is a 32-bit (4-byte) field whose value is adjusted by miners so that the hash of the block will be less than or equal to the current target of the network. The rest of the fields may not be changed, as they have a defined meaning. It is important to realise that block generation is not a long, set problem (like doing a million hashes), but more like a lottery. Each hash basically generates a random number between 0 and the maximum value of a 256-bit number (which is huge). If the corresponding hash is below the target, the miner wins. If not, the miner needs to increment the nonce (completely changing the hash) and try again. Therefore, the process requires a huge amount of computing power.

The next step is to find the golden nonce, or to examine whether the block hash meets the target using different nonce values. In Python, we can start with a nonce that we choose by the *input()* function. The *while True* loop ensures that the user can keep keying in values if there are any wrong entries (see Figure 2.26).

Blockchain and Smart Contracts Downloaded from www.worldscientific.com
by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

```
print('='*30)

# To specify a nonce
while True:
    user_input = input('Please key in an non-negative integer: \n')
    try:
        nonce = int(user_input)
        if nonce < 0:
            print('It is a negative integer. Please key in again. ')
            continue
        else:
            break
    except ValueError:
        print('That is not an integer. Please key in again. ')
        continue
```

Figure 2.26: To input a nonce.

Blockchain and Smart Contracts Downloaded from www.worldscientific.com by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

The *nonce = int(user_input)* statement is necessary to convert the user input to integer. Otherwise, what the user keyed in will be a string type. It is possible that the user keyed in strings or other symbols that cannot be used for the *int()* function, which will raise a value error message. So, the "*except ValueError:*" statement is meant for such cases and allows the user to key in again without the program being ended by the error message.

Once we have keyed in a nonce value, we can start the trial with this nonce value and keep trying to use a *while* loop. 0x100000000 in hex is 4294967296, so this is the max number of loops run if the golden nonce is not found. With the nonce, we can have the hash of block header (used to determine if the target is met). Do note that in this example, we use a relatively easy target. In reality, to mine a block in bitcoin is very difficult and the number of loops or the time it takes to find the golden nonce is much larger and longer than our example.

Generating a *mining_start_time* variable that shows the start time is for us to see how long it takes to find a golden nonce or mine a block with the target set compared to the time when we find the golden nonce (see Figure 2.27). Current time minus start time is the

```
mining_start_time = time.time()
while nonce < 0x100000000:  # The number is 4294967296
    header = partial_header + struct.pack("<L", nonce)
    hash = dsha(header)

    if nonce in range(50000, 20000000, 50000):
        mining_time = time.time() - mining_start_time
        hash_rate = nonce / mining_time
        print(nonce, hash[::-1].hex(), hash_rate, " per second")

    if hash[::-1] < target_byte:
        print('Success!', nonce, binascii.hexlify(hash[::-1]))  # in hexstring
        break

    nonce += 1
```

Figure 2.27: Finding the golden nonce.

time taken. We will show the nonce, hash of the block header and the hash rate per second every 50,000 times the loop runs.

For every loop, we try a larger nonce value until the golden nonce is found or nonce value reaches 4294967296 (can set to larger value with smaller target). If the block header hash is smaller than the target, the loop will be broken with a "Success!" message. The nonce that satisfies the condition is the golden nonce.

This is an example of trying different nonce values to see if we get a hash value that is smaller than the target given. Here, we start with a small nonce and try a bigger nonce every time the loop runs. Of course, it can be other methods to try out different nonce values, such as to randomly select a nonce number in a range or to start with a large number and try a smaller value in every round.

Once the target is met, miners can broadcast the new block they find to the network. If recognised, more miners will use this block as the latest block and calculate the hash value of the next block to mine another new block. Successful miners can collect the reward generated with the new block and the transaction fees via the transactions included in and confirmed with the block. This is the economic incentive for miners to keep the bitcoin network running.

Blockchain and Smart Contracts Downloaded from www.worldscientific.com
by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.

References

Gordon, A. D. and Jeffrey, A. (2004). Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3–4), 435–483.

MacKenzie, D. (2019). Pick a nonce and try a hash. *London Review of Books*, 41(8), 35–38.

Merkle, R. C. (1980). Protocols for public key cryptosystems. *In Proc. 1980 Symposium on Security and Privacy, IEEE Computer Society*, (pp. 122–133).

Nakamoto, S. (2019). Bitcoin: A peer-to-peer electronic cash system. *Manubot*.

Nguyen, G. T. and Kim, K. (2018). A survey about consensus algorithms used in blockchain. *Journal of Information Processing Systems*, 14(1), 101–128.

O'Dwyer, K. J. and Malone, D. (2014). Bitcoin mining and its energy footprint. Conference working paper.

Sobti, R. and Geetha, G. (2012). Cryptographic hash functions: A review. *International Journal of Computer Science Issues* (IJCSI), 9(2), 461.

Blockchain and Smart Contracts Downloaded from www.worldscientific.com
by 186.12.44.8 on 01/17/22. Re-use and distribution is strictly not permitted, except for Open Access articles.