Steps to Reproducing the Demo:

1. Mining and coin creation:

In **MinerApp.py,** miners can initiate the mining action with the **start_mining** function.

MinerApp.py

```
159 @app.route('/start_mining')
160 def start_mining():
161     myMiner.isMining = True
162     time.sleep(2)
163     while True:
164         time.sleep(0.5)
165         # try:
166         res = myMiner.start_mining()
167         if res == "receive new block":
168             continue
169         elif res == "error":
170             continue
171         # except Exception as e:
172         #     return {"Exception": str(e)}, 500
173         announce(res)
174 
```

Line 166: the start_mining function from **Miner.py** is called.

Line 163: If a new block has been found, miner announces it via the announce function.

In the **start_mining** function in **Miner.py**, each miner first checks if other miners have managed to find a block. If other miners have indeed found a block, the miner will stop what he is doing and validates said block. Else, the miner will collect all valid transactions, verifies it, and tries to verify the proof of work to successfully create a new block.

Miner.py

```
19      def start_mining(self):
20          ## RECEIVE NEW ANNOUNCEMENT ##
21          for block in Miner.new_block_queue:
22              print('new block queue')
23              to_hash = block[0].serialize()
24              digest = hashlib.sha256(to_hash.encode('utf-8')).hexdigest()
25              print('hash of announced block', digest)
26              if self.blockchain.verify_pow(digest, block[1]):
27                  if self.blockchain.validate_block(block[0]):
28                      self.blockchain.add_block(block[0])
29              print("NEW BLOCK QUEUE", self.miner_id)
30              Miner.new_block_queue.remove(block)
31          list_of_trans = []
32          bal_map = self.blockchain.blockchain_graph[self.blockchain.longest_header]["balance_map"]
33          ## COLLECT VALID TRANSACTIONS ##
34          if len(Miner.trans_pool) > 0:
35              for trans in Miner.trans_pool:
36                  if bal_map[trans.sender] - trans.amount > 0:
37                      list_of_trans.append(trans)
38                  else:
39                      print("Insufficient Balance")
40                  if len(list_of_trans) > 3:
41                      break
42          ## GENERATE NONCE ##
43          new_block = Block(
44              list_of_trans, self.blockchain.longest_header, self.miner_id)
45          counter = 0
46          while True:
47              if counter % 100000 == 0:
48                  print("attempt:", counter)
49              counter += 1
50              if Miner.new_block_queue:
51                  return "receive new block"
52              generate_nonce = str(random.randint(0, 300000))
53              new_block.header['nonce'] = generate_nonce
54              to_hash = new_block.serialize()
55              digest = hashlib.sha256(to_hash.encode('utf-8')).hexdigest()
56              ## TRY ADDING BLOCK TO MINER'S BLOCKCHAIN ##
57              # try:
58              if self.blockchain.verify_pow(digest):
59                  if self.blockchain.validate_block(new_block):
60                      self.blockchain.add_block(new_block)
61                      for trans in list_of_trans:
62                          if trans in Miner.trans_pool:
63                              Miner.trans_pool.remove(trans)
64                      return new_block
```

Lines 21-30: If another miner has found a block, the current miner will stop what it is doing, and tries to validate the block.

Lines 34-41: Collects valid transactions and checks if the transactions are valid

Lines 43-55: Miner attempts to create a new block with the transactions collected

Lines 59-64: Miner attempts to add the block to miner’s blockchain

Finally, once a miner has successfully added a block to the blockchain, he will announce the update to other miners.

MinerApp.py

```
96  def announce(block):
97      for port in PORT_LIST:
98          if port == args.port:
99              continue
100         print("I am announcing to: {}".format(port))
101         try:
102             data = {}
103             data['header'] = block.get_header()
104             data['difficulty'] = sutdcoin.old_target
105             transactions_to_send = []
106             for trans in block.merkle_tree.past_transactions:
107                 transactions_to_send.append(trans.serialize())
108             data['transactions'] = transactions_to_send
109             data['miner_id'] = myMiner.miner_id # to get the longest one, used by /headers endpoint
110 
111             form = json.dumps(data)
112             r = requests.post(
113                 "http://localhost:{}/listen".format(port), json=form)
114             print('done sending')
115         except Exception as e:
116             return {"Exception": str(e)}, 500
117     latest_block = sutdcoin.blockchain_graph[sutdcoin.longest_header]["block"]
118     latest_chain = sutdcoin.create_chain_to_parent_block(latest_block)
119     chain_of_headers = []
120     for i in latest_chain:
121         head = i.get_header()
122         chain_of_headers.append(head)
123     d1 = {}
124     d1['list_headers'] = chain_of_headers
125     d1['miner_port'] = PORT
126     d1 = json.dumps(d1)
127     r = requests.post("http://localhost:{}/headers".format(8080),json=d1)
128 
129     if r.ok:
130         return 'success', 200
131     else:
132         return 'not ok', 400
```

Lines 101-113:

2. Fork Resolution

Forks are resolved in the **resolve** function in **Blockchain.py**

Blockchain.py

```
127     def resolve2(self):
128         """
129         get longest header and compute the longest chain through recurrence
130         update balance map
131         return longest chain through recurrence
132         """
133         highest_level_n = 0
134         highest_level_n_digest = []
135         for digest, node in self.blockchain_graph.items():  # finding the highest level_n
136             if len(node["children"]) > 1:
137                 pass
138                 # print('Fork is found...')
139             # check transaction
140             if node["height"] > highest_level_n:
141                 highest_level_n_digest = []  # getting nodes with highest level_n
142                 highest_level_n_digest.append(digest)
143                 highest_level_n += 1
144             elif node["height"] == highest_level_n:
145                 highest_level_n_digest.append(digest)
146             else:
147                 continue
148         self.longest_header = highest_level_n_digest[random.randint(
149             0, len(highest_level_n_digest) - 1)]  # return a random header
150         print("The longest header is now ", self.longest_header)
```

Lines 140 - 145: Checks in the blockchain for the nodes with the highest level in the graph. A node with the highest level indicates that it is the node that is on the longest chain

Lines 148 - 149: If there are multiple nodes with the highest level, one node is randomly selected to be the node in the longest chain.

3. Transaction resending protection (to verify that the transaction has not been reused)

In **MerkleTree.py,** each tree stores its past transaction hashes.

MerkleTree.py

```python
class MerkleTree():

    def __init__(self):
        self.past_transactions = []
        self.past_hashes = []
        self.tiered_node_list = []
        self.root = None
```

A **validate_block** function is created in the Blockchain class to check for duplicates

Blockchain.py

```python
42      def validate_block(self, block):
43          """
44          ensure proof of work is valid, transactions are new, timestamp
45          return block isValidated
46          """
47          # ## CHECK INCOMING BLOCK IS MINED AFTER THE PARENT BLOCK ##
48          if block.get_header()["timestamp"] <= self.blockchain_graph[block.get_header()["prev_header"]]["block"].get_header()["timestamp"]:
49              return False
50          ## CHECK FOR DUPLICATE TRANSACTIONS ##
51          for trans in block.merkle_tree.past_transactions:
52              for block_ in self.create_chain_to_parent_block(block):
53                  for trans_ in block_.merkle_tree.past_transactions:
54                      if trans.serialize() == trans_.serialize():
55                          return False
```

Lines 51-55: Blockchain checks for the block if it contains any duplicate transactions but iterating through the blocks in the longest chain.

Finally, the **validate_block** method is called in **Miner.py** for miners to validate their own block that they found.

Miner.py

```python
58                  if self.blockchain.verify_pow(digest):
59                      if self.blockchain.validate_block(new_block):
60                          self.blockchain.add_block(new_block)
61                          for trans in list_of_trans:
62                              if trans in Miner.trans_pool:
63                                  Miner.trans_pool.remove(trans)
64                          return new_block
```

Line 59: validate_block function is called in **Miner.py.**

4. Payments between miners and SPV Clients

a. Transaction validation

The SPV client stores the all the block headers in the longest chain. How this is done is that when Miners announce a block to others, they also announce the chain of headers to the SPVapp. The SPVapp receives it and checks if its longer than the list of headers it currently stores (initialized to be []). If the incoming chain is longer, it will replace its current list of headers with the incoming one.

SPVApp.py (get_headers())

```python
@app.route('/headers', methods=["POST"])
def get_headers():
    global list_of_headers
    global longest_miner
    # wait for miners to send the longest chain
    res = json.loads(request.get_json())
    chain_length = len(res['list_headers'])
    if chain_length > len(list_of_headers):
        list_of_headers = res['list_headers']
        longest_miner = res['miner_port']
    # print(chain_length)
    # print(list_of_headers)
    return "200"
```

We can also pass a public key to the SPVclient, for it to return all transactions associated with it (sender and receiver) by calling the function below. (get_related__transactions)

This function posts a request to the miner and then miner will call the function listening_to_my_transactions, which will traverse the current longest chain and return all transactions that contain the public key to the SPV app, along with all the other hashes required to reconstruct the merkel root.

```python
@app.route('/list_transactions', methods=['POST'])
def get_related_transactions():
    # send request to miners
    # this is the longest guy
    data = request.get_json()
    # print(longest_miner)
    r = requests.post(
        "http://localhost:{}/get_transactions".format(5005), json=data)
    res = r.json()
    print("THE TRANSACTION LIST FROM MINER:", res)
    transaction_list = []
    pub_key = data['pub']
    # After u get the txn params here, have to initialize a new txn
    for trans_ in res['transaction_list']:
        trans = json.loads(trans_[0])
        sender = trans["sender"]
        receiver = trans["receiver"]
        amount = trans["amount"]
        comment = trans["comment"]
        timestamp = trans["timestamp"]
        signature = trans["signature"]
        t = Transaction(sender, receiver, amount,
                        comment, timestamp, signature)
        transaction_list.append(t)

        txn_and_proofs[pub_key] = txn_and_proofs.get(pub_key, [])

        txn_and_proofs[pub_key].append( {"transaction" : t, "nodes" : trans_[1], "neighbo

    # this should get test
    print(transaction_list)
    return 'success', 200
```

```python
@app.route('/get_transactions', methods=['POST'])
def listening_to_my_transactions():
    user_pub_key = request.get_json()['pub']
    print(user_pub_key)
    # need to serialize transaction and then make on other side
    transactions_list = []
    data = {}
    # print(sutdcoin.blockchain_graph)
    last_block = sutdcoin.blockchain_graph[sutdcoin.longest_header]["block"]
    longest_chain = sutdcoin.create_chain_to_parent_block(last_block)
    longest_chain.insert(0,last_block)
    #blockchain_graph_items = sutdcoin.blockchain_graph.items()
    for block in longest_chain:
        for trans in block.merkle_tree.past_transactions:
            if trans.sender == user_pub_key or trans.receiver == user_pub_key:
                nodes, neighbour, index = block.merkle_tree.get_min_nodes(
                    trans)
                transactions_list.append(
                    [trans.serialize(), nodes, neighbour, index])

    data['transaction_list'] = transactions_list
    print("THE TRANSACTION LIST IS:", data)
    data = json.dumps(data)
    return data, 200
```

Finally, verify transactions can be called, by using the function, verify_transaction() . E will pass in the transaction tas a json, and the code attempt s to reconstruct it from line 96-105. Then it iterates through the list of transactions and proofs that we obtained in the earlier two functions above. If any the transactions are matched, it will attempt to rebuilt the root and then match it to any of the roots in the block header. Once it finds a match, it will print “Transaction Verified” and we can know that the transaction has been mined and added to the longest chain.

```python
@app.route('/verify', methods=['POST'])
def verify_transactions():
    trans = request.get_json()
    sender = trans["sender"]
    receiver = trans["receiver"]
    amount = trans["amount"]
    comment = trans["comment"]
    timestamp = trans["timestamp"]
    signature = trans["signature"]
    t = Transaction(sender, receiver, amount,
                        comment, timestamp, signature)

    for pub_key_trans in txn_and_proofs.values():
        for trans in pub_key_trans:
            if t == trans["transaction"]:
                print('found matching transaction')
                #VERIFY IT
                to_hash = trans["transaction"].serialize()
                digest = hashlib.sha256(to_hash.encode()).hexdigest()
                if trans['nodes'] == [] and trans['neighbour'] == None:
                    #check if digest in longest chain
                    for header in list_of_headers:
                        if header["root"] is None:
                            continue
                        print(header["root"])
                        if digest == header['root'][0]:
                            print("Transaction Verified")
                            return "success",200
                else:
                    if trans["index"] % 2 == 0:
                        # the node is on the left
                        to_hash = str(digest) + str(trans["neighbour"])
                        digest = hashlib.sha256(to_hash.encode()).hexdigest()
                    else:
                        # node is on the right
                        to_hash = str(trans['neighbour']) + str(digest)
                        digest = hashlib.sha256(to_hash.encode()).hexdigest()
                    if trans["nodes"][1] == 'left':
                        to_hash = str(trans["nodes"][0]) + str(digest)
                        digest = hashlib.sha256(to_hash.encode()).hexdigest()
                    else:
                        to_hash = + str(digest)+ str(trans["nodes"][0])
                        digest = hashlib.sha256(to_hash.encode()).hexdigest()
                    for header in list_of_headers:
                        if header["timestamp"] < timestamp:
                            continue
                        if digest == header['root'][0]:
                            print("Transaction verified")
                            return "success",200
    return "error",500
```

5. Attacks

a. 51% Attack

The 51% Attack (Majority Attack) occurs when more than half the total miners who are mining a blockchain network create a branch from a specific block and continue mining from there instead isolated from the rest of the blockchain network until the branch becomes long enough to win the democracy resolve before broadcasting it back to the main blockchain network.

```
@app.route('/start_mining_51')
def start51Mining():
    myMiner.isMining = True
    good_length = 0
    
    while good_length < 3:
        #counter += 1
        # try:
        res = myMiner.startMining()
        if res == "receive new block":
            continue
        elif res == "error":
            continue
        good_length = len(sutdcoin.createChainToParentBlock(sutdcoin.blockchain_graph[sutdcoin.longest_header]["block"])) + 1
        # except Exception as e:
        #     return {"Exception": str(e)}, 500
        announce([res])

    print("COMMENCING EVIL DEEDS")
```

In this simulation, we will initialize 2 malicious miners and 1 normal miner. The function first initialized the malicious miners to act normally, however, they are continuously counting the number of normal blocks that have been mined. This is to simulate that a transaction has been made on the second block which the malicious miners want to revoke to create double-spending.

```python
    print("COMMENCING EVIL DEEDS")
    evil_length = 0
    evil_head = sutdcoin.blockchain_graph["4a18a58e969d9281c65ff9fdc9443f23ce2484c532329a99c14f58b5eaef5120"]["children"][0]
    while evil_length <= good_length + 2 and not stopEvil:
        print('good chain length:', good_length)
        print('evil chain length:', evil_length)
        res = myMiner.start51Mining(evil_head)
        if res == "receive new block":
            print('received a good block')
            good_length += 1
            continue
        elif res == "receive evil block":
            evil_length += 1
            continue
        elif res == "error":
            continue
        if stopEvil:
            res = ""
            print("leaving evil deeds...")
            break
        print("EVIL FOUND NEW BLOCK ", res.hash_header()[-6:])
        # except Exception as e:
        #     return {"Exception": str(e)}, 500
        evil_head = res.hash_header()
        evil_length = len(sutdcoin.createChainToParentBlock(res)) + 1
        announce([res], isEvil=True)
        if evil_length > good_length+2:
            stop51Attack()

    if type(res) is not str:
        evil_block_list = sutdcoin.createChainToParentBlock(res)
        evil_block_list.insert(0, res)
        if not stopEvil:
            evil_block_list.reverse()
            announce(evil_block_list)
    print("EVIL DEED IS DONE")
    r = showGraph()
    while True:
```


Malicious miners target the previous block and start mining from there independently from the other miners. Malicious miners record the targeted header as the longest header and continue mining, broadcasting their findings to each other until their branch becomes long enough to beat the single miner on the other branch.

```python
    def start51Mining(self, evil_header):
        ## RECEIVE NEW ANNOUNCEMENT ##
        print('===EVIL new block queue===')
        for block in Miner.new_block_queue:
            print(block)
            to_hash = block[0].serialize()
            digest = hashlib.sha256(to_hash.encode('utf-8')).hexdigest()
            print(self.miner_id[-6:], ', hash of announced block', digest)
            if self.blockchain.verifyPow(digest, block[1]):
                if self.blockchain.validateBlock(block[0]):
                    self.blockchain.addBlock(block[0])
            if block[2] == "evil":
                print("received block is evil")
                evil_chain_length = len(self.blockchain.createChainToParentBlock(block[0])) + 1
                print(evil_chain_length)
        Miner.new_block_queue = []
            
        print("----------------")

        try:
            if evil_chain_length > len(self.blockchain.createChainToParentBlock(self.blockchain.blockchain_graph[evil_header]
                    ["block"]) + 1):
                self.blockchain.longest_header = digest
            else:
                self.blockchain.longest_header = evil_header
        except:
            self.blockchain.longest_header = evil_header

        list_of_trans = []
        bal_map = self.blockchain.blockchain_graph[self.blockchain.longest_header]["balance_map"]
```

Malicious miners have their own mining function that is similar to the original mining function. Malicious miners, however, do not update their longest header based on the blockchain network. Instead, they continue mining from the separate branch.

```python
@app.route('/stop_being_bad')
def stopBeingBad():
    print("=19*&$*7981&^)$=")
    global stopEvil
    stopEvil = True
    print("=cleansing successful=")
    return "success", 200
```

```python
def stop51Attack():
    print("===AnnouncingEVIL===")
    for port in PORT_LIST_51:
        if port == args.port:
            continue
        print("I am announcing to: {}".format(port))
        r = requests.get("http://localhost:{}/stop_being_bad".format(port))
    print("----------------")
    if r.ok:
        return 'success', 200
    else:
        return 'not ok', 400
```

Once the malicious branch is long enough, the malicious miners broadcast their new branch to the normal miner in one go. The normal miner will then update these new blocks into their blockchain. Due to their mining rule, the normal miner will then continue to mine from the longest chain, which is the new malicious blocks on a separate branch, revoking the transactions that were made in the other branch.

b. Selfish Mining

A selfish miner will not announce his blocks until he has gained a lead of 2 blocks. Else, he performs like a normal miner. The selfish miner adds blocks that he found into a private list, and once it reaches a length of two, it announces to the other miners and earn the rewards.

```python
@app.route('/start_selfish_mining')
def start_selfish_mining():
    myMiner.isMining = True
    list_of_priv_blocks = [] ## the lead, and not announced chain branch
    while True:
        list_of_priv_blocks = []
        # show that selfish miner has worse computational power
        time.sleep(2)
        res = myMiner.startMining()
        if res == "receive new block" or myMiner.new_block_queue:
            # continue mining on public blockchain
            print('selfish recieve new block')
            continue
        else:
            while(not myMiner.new_block_queue):
                list_of_priv_blocks.append(res)
                res = myMiner.startMining()
                if type(res) is not str:
                    list_of_priv_blocks.append(res)
                    if len(list_of_priv_blocks)>=2:
                        announce(list_of_priv_blocks)
                        list_of_priv_blocks = []
                        print("SELFISH MINING SUCCESS")
                        r = showGraph()
```

Differences between Bitcoin and SUTD Coin:

- Maximum transaction per block in SUTD Coin is 4, for Bitcoin, it is 1MB worth of transactions (over 2700 transactions)
- Difficulty adjustment of for SUTD Coin is based on 1 block, whereas Bitcoin is based on 2016 blocks
- Block reward for SUTD coin is fixed at 100 coins per block and while for bitcoin, it is originally 50 coins. SUTDcoin does not go through halving like bitcoin and will never run out.
- SUTDcoin does not use a bloom filter (like bitcoin) to check if a transaction is in the longest chain, but manually performs string matching.
- SUTD coin uses an address : balance model unlike bitcoin’s UTXO model.
- There are no transaction fees in the SUTDCoin unlike bitcoin.