Peer to Peer Systems - Final Project
TinyCoin: Simulating fraudolent mining strategies in a simplied Bitcoin Network

Matteo Bogo

July 12, 2017

Introduction. This report presents an implementation of a simplified version of the Bitcoin technology called TinyCoin. In this work the Bitcoin standard mining process is evaluated in comparison to a fraudulent version named Selfish Mining. The simulation has been made through the Peersim Simulator. During this report we will analyze all the choices made to model both TinyCoin standard and dishonest versions. Every issue that have been addressed during their implementation has been described. In the last part we will discuss some experiments made on the simulation realized.

Peersim. The P2P simulator Peersim has been exploited for setting up the simulation. In Figure 1 is presented a Schema that describes the most important classes realized or inherited from the libraries.

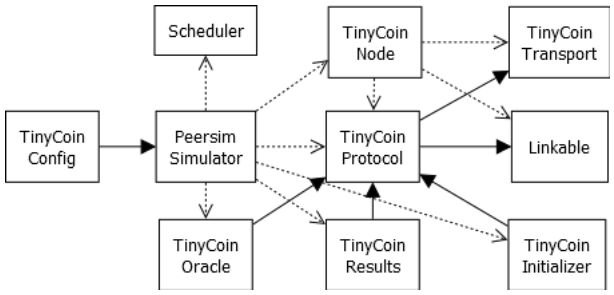

Figure 1: A simplified Schema of TinyCoin in Peersim

For this work both the Event-driven and Cycle-driven simulation engines have been adopted. The first one allowed us to use a scheduler for trigger events to the TinyCoin protocol of each node inside the network. Events like incoming transactions or blocks are handled through processEvent() method inherited from the EDProtocol class. The second one invokes the nextCycle() method obtained from the CDProtocol class. This method has been exploited for simulating the generation of transactions. Every k cycles each node has a probability to make a transaction against another node. This probability is managed through a threshold and will be discussed later.

The parameters needed to configure the entire simulation are stored inside the TinyCoin Config file. More details can be found inside the source code of the project.

The TinyCoinNode class implements the Node Interface needed by the simulator to build the Network. The default class GeneralNode has been extended with a new behavior for obtain the currently active TinyCoin Protocol of a Node during the simulation. This mechanism was necessary due to the separation of concerns applied to TinyCoin Protocol.

The protocol has been separated into distinct classes, such each class addresses a separate concern. In particular it was necessary to divide the logic of an honest miner from a fraudulent one. In this work three protocols have been developed. The first protocol represents a normal node that can make or receive transactions, obtaining blocks and broadcast messages to the network. The other two extend the previous one by adding an honest and a dishonest mining process behavior respectively. Figure 2 shows a Schema of the above-mentioned structure. The Base Miner protocol is simply an abstract class that contains common behaviors between the two strategies.

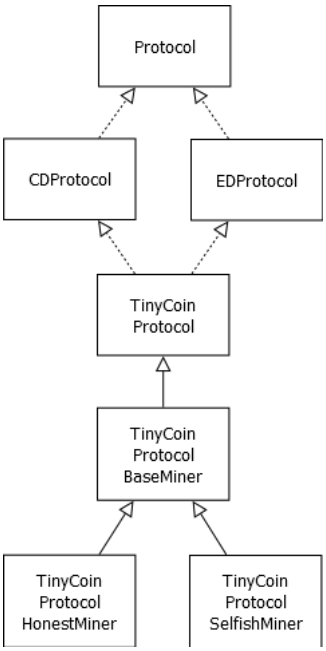

Figure 2: Schema of TinyCoin Protocol Inheritance

Each node has two extra protocols in common besides the one that models the TinyCoin infrastructure.

The first one implements the Linkable Interface and serves to form the overlay in the simulator framework. For this purpose the IdleProtocol has been chosen and it was associated to the WireKOut class, both obtained from Peersim Libraries. The WireKOut class generates a Graph with randomly k directed edges out of each node. Every neighbor is chosen without adding loop edges. The k factor is an important parameter that will be analyzed inside our experiments later. This protocol is used to discover neighbors during message broadcasting.

The second one implements the Transport interface. This interface represents a generic transport protocol for sending messages between linked nodes through the underlying network. Messages are scheduled using the EDSimulator with a delay (in cycles) for simulating network latency. Because one of the goals of the project is to simulate the latency increase due to increasing of the block size[1], it was necessary to define a new implementation to model the dynamic latency. The TinyCoinTransport class takes from the external configuration two constants, respectively to define the base latency for the block and the transaction propagation, then exposes a method to adjust the latency value before sending a message.

The TinyCoinInitializer class collects the simulation parameters from the external configuration file. It has the responsibility to generate the random distribution of nodes types and mining powers, handle the initialization of protocols inside nodes and spawn if requested the selfish pool used in some experiments. The distribution management will be discussed later. For convenience all parameters loaded from outside are saved in a singleton class named Parameters that acts as a data container. All other classes refer to this when a particular parameter is required.

Because the TinyCoin technology doesn't consider the Proof of Work like the Bitcoin system, it was necessary to implement an "Oracle" that periodically chooses a miner node as the winner of the computational race for mining a block that will be added in the blockchain. The choice is influenced by the mining power of every miner inside the network and it will also be discussed later. The only constant loaded by this class is the number of cycles to wait between two winning miners. The Oracle class is named TinyCoinOracle and it implements the Control interface.

The TinyCoinResults class is another one that implements the Control interface. Its main purpose is to collect all the statistics about the simulation that will be used later for evaluations.

Transactions. The TinyCoin transaction is much simpler compared to a real Bitcoin transaction. Fees, Scripts, Digital signatures and UTXO were not considered in our model. A TinyCoin transaction contains the sender address to identify the node who sent the money and a receiver address to recognize who will receive the credit. A variable to represent the amount is also present. A Schema is shown in Figure 3.

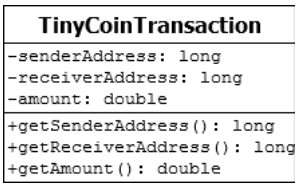

Figure 3: UML Schema of TinyCoin Transaction

Every time a node generates a transaction, the amount spent is immediately subtracted from his wallet. The wallet is simply represented by a variable maintained in the protocol instance of the node. When a node receives a transaction, it immediately proceeds to its validation. The validation steps implemented consist of:

- Addresses verification: each node has an address expressed as an integer, the value of the index associated by the simulator in the network array that contains all the nodes. This value must be greater or equal to zero.
- The amount of coins inside the transaction must be in range from the lower to the grater value of Satoshi. The thresholds considered are the same of the Bitcoin system: 0.00000001 and 21000000.
- Check if the transaction is already contained in the Mempool. In other words if the transaction has already been received, it's rejected to avoiding loops.
- Check if the transaction is already contained inside the local copy of the public Blockchain. (i.e. the transaction was already mined inside a Block)

Transactions received or generated are stored inside a data structure named Mempool. Transactions are removed only when a new block contains those transactions is added to the public Blockchain. The data structure chosen to represent Mempool is the HashSet. The HashSet allows a complexity of $\mathcal{O}(1)$ for add, remove, contains and size. Also HashSet permits only unique values, so it contains no duplicates. The latter behavior grants us to see if the transaction is already present by trying to add it, without iterate the entire data structure, because if it's already contained a boolean is returned.

[1]The block propagation time is increased by a constant amount for each transaction included in a block.

When a node acknowledges a transaction destined to itself either it's received by a neighbor or if it's contained in a block added to the public Blockchain, the node adds the transaction in a data structure named UnconfirmedTransactionMap. In the real Bitcoin system peers can adopt different strategies for consider a transaction "confirmed". The number of confirmations is equal of the number of blocks stored in the public Blockchain after the one that contains our transaction. Like the standard Bitcoin client we consider six blocks to confirm a transaction and then update the wallet. The structure chosen to represent the pool of unconfirmed transactions is the HashMap. Transactions are added with the pair $< TinyCoinTransaction, confirmations >$. Every time a block is received and added to the public Blockchain, all values in the Map are incremented by 1. The complexity is $\mathcal{O}(1)$ when a transaction is added or removed, $\mathcal{O}(n)$ when all values are incremented and $\mathcal{O}(n)$ for checking if a transaction has reached the minimum threshold of six confirmations.

In the last step of the algorithm, the transaction need to be transmitted to neighbors except the one from which it has arrived.

Blocks. The TinyCoin block is also different from the real Bitcoin block. The block doesn't contain the Proof of Work and his reward not decreasing over time. Also the miners don't insert the nonce, making the Blockchain a simple chain of blocks. The TinyCoin block includes an unique identifier obtained from Java UUID[2] (Universally Unique IDentifiers), the identifier of the previous block, the list of all transactions included in the block (Coinbase transaction also included) and the height inside the public Blockchain. Figure 4 shows a Schema of the TinyCoin block.

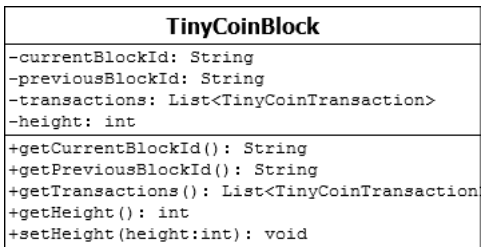

Figure 4: UML Schema of TinyCoin Block

The base reward of each block is fixed to 12.50 coins, according to the actual (From July 2016[2]) Bitcoin reward. To simulate transaction fees incentives, the base reward is added with a fixed amount multiplied for each transactions included in the block. This amount is 0.0008136 according to the average transaction fee in the real Bitcoin system[2].

Transactions inserted in the blocks are taken from the Mempool. Due to simulate the block size, in this implementation each block have two threshold: a minimum and a maximum amount of transactions can be placed. Those thresholds will be seen later in our experiments because they affect the propagation of blocks in the network. Transactions are also prioritized in descending order by the amount of coins contained.

When a candidate block is generated by a miner, it also includes a Coinbase transaction. This kind of transaction contains the amount of reward for mining the block as discussed previously. Coinbase is inserted on the top of the transactions list.

Like transactions, when a node receives a block a validation process must be performed. It consist of several stages:

- Check if the list of transactions is empty or absent, if so the block is rejected.
- Check if the list of transactions size is less or equals to the maximum threshold
- Check if the first transaction is the Coinbase transaction. The node computes the reward of the block and then compares it to the amount of coins included in the first transactions contained in the list.

If the block is considered valid, the node will look for duplicates inside the Blockchain or in the Orphan pool. If the block is a fresh one, the node manages it within the Blockchain.

The last two steps consist to transmit the managed block to neighbors except the one which it came from and the management of the orphan blocks. The latter stage will be discussed later.

Blockchain. As mentioned previously, the TinyCoin Blockchain is a simple chain of blocks. Every node has its own local copy and when the simulation begins the Blockchain is initialized with the Genesis block.

In an early version of the TinyCoin Blockchain, the idea of using an array list was taken into consideration to model the chain of blocks. That's because it seemed to be the most natural representation. Also the complexity was good enough: $\mathcal{O}(1)$ to check if the previous block pointed by the previous ID is at the head of the public Blockchain and $\mathcal{O}(1)$ to add a new block in the head. The biggest difficulties come with the management of forks and swaps. When the previous identifier of the received block doesn't point to the head of the public Blockchain, it's necessary to iterate the array list backwards to find the forked block. At this point a new fork is generated and inserted into a list of lists. More troubles if the previous identifier is contained into a fork, the list of lists need to be iterated to find the previous block, with a complexity of $\mathcal{O}(mk)$, where m is the number of forks and k their average length. If the fork contains another fork, the problem becomes more complex. Other difficulties come from the branches swapping process.

[2]For UUID only after generating 1 billion UUIDs every second for the next 100 years, the probability of creating just one duplicate would be about 50%.

Every time a fork need to be placed in the public Blockchain, we have to handle the swap with sublists from the main branch to side branch and vice versa. The complexity is $\mathcal{O}(2k)$ where k is the number of blocks swapped.

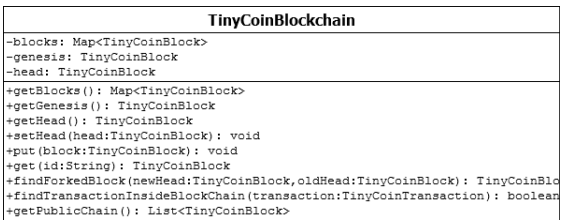

Figure 5: UML Schema of TinyCoin Blockchain

The current version of TinyCoin Blockchain uses a LinkedHashMap to store unique blocks in pairs $< BlockID, TinyCoinBlock >$ and a pointer to the public head (i.e. the last inserted block in the main branch of the Blockchain). Figure 5 shows a Schema of the TinyCoin Blockchain. This approach compared to the previous one is less complex and more efficient. Checking duplicates blocks requires only $\mathcal{O}(1)$ both in main branch and side branches (all blocks are stored in one structure). Check if the previous block is contained in the head of the public Blockchain is also $\mathcal{O}(1)$. Same constant time for putting block in the map and for changing the head pointer.

When a received block connects to somewhere other then the head of the public blockchain, a fork need to be handled. Because there is only a data structure with all blocks, the findForkedBlock() method reconstructs the paths from the public head and the last inserted block to the forked block. This is guaranteed because the blocks have only one parent. Figure 6 shows a visual representation of this process. This method has a complexity in the worst case of $\mathcal{O}(2k+1)$, where k is the number of blocks swapped off from the main branch. Get blocks through the previous ID is $\mathcal{O}(1)$ in a Map.

```
findForkedBlock(oldHead,newHead):

  oldPointer = oldHead;
  newPointer = newHead;

  while(oldPointer != newPointer):
    if(oldPointer.height >
        newPointer.height) then
      oldPointer =
          map.get(oldPointer.previousID);
    else
      newPointer =
          map.get(newPointer.previousID);
  return oldPointer;
```

Listing 1: Pseudo-code of findForkedBlock()

Once the forked block has been found, the algorithm compares the heights. If the fork has an height greater than one, it will be swapped in the main branch of the Blockchain. The complexity is $\mathcal{O}(1)$ because it requires only to change the head pointer of the Blockchain. No sublists or supporting data structures are required.

However the process of swapping two branches involves several necessary steps that increase the complexity of the algorithm.

The first step is to iterate the old main branch from the head to the forked block. During this iteration all transactions contained inside blocks are stored again in the Mempool. Every block iterated decreases the confirmations counter of all transactions in UnconfirmedTransactionMap structure. When the iteration is over, all transactions with counter less or equal zero are removed.

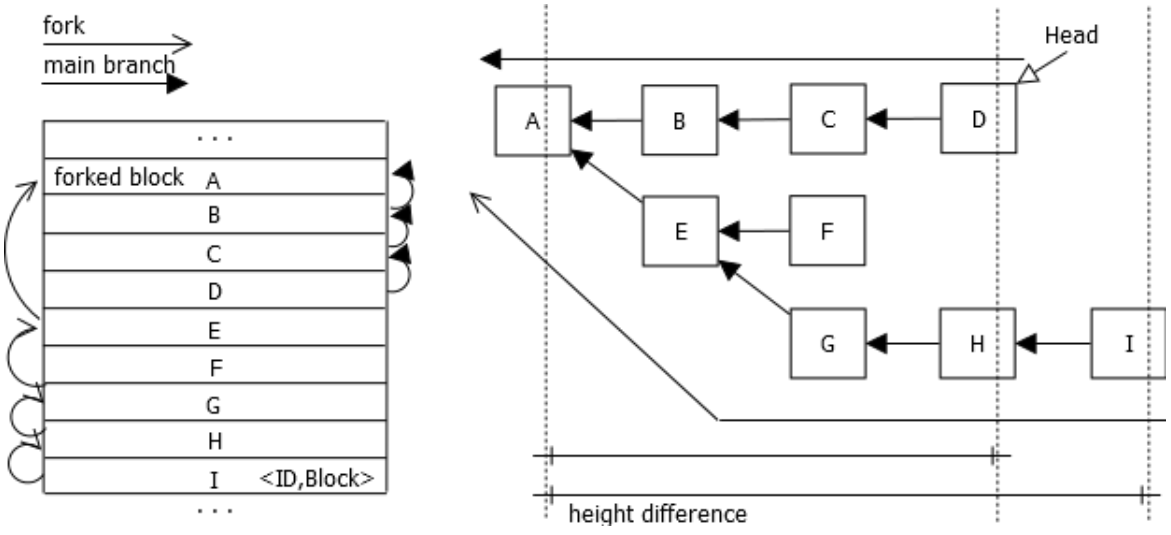

Figure 6: Forks management with TinyCoin Blockchain

The second step is to iterate the new main branch (old side branch) from the new head to the same forked block. If a block iterated contains a transaction with the local node as receiver, then the transaction is added into UnconfirmedTransanctionMap with the confirmations counter equals to the difference between the new head height and the current block iterated height. Transactions inside blocks are also removed from the Mempool. When the iteration is over the UnconfirmedTransactionMap is verified and if contains "confirmed" transactions, the wallet is updated.

Orphans. Blocks received without an existing parent inside the Blockchain are considered "orphans". If these blocks are valid they aren't rejected, instead they are added into an another data structure named Orphans Pool. Like the Mempool, for the Orphan Pool an HashSet was adopted. The Set guarantees a complexity of $\mathcal{O}(1)$ when checking duplicates, adding or retrieving orphans.

When a node finishes to handle the received block and even if the last block isn't an orphan, the Orphan pool is iterated to find blocks that have the last block added as parent. Complexity is of course $\mathcal{O}(m)$ where m is the number of orphans in the Pool. For each ex-orphan retrieved the entire algorithm is invoked again recursively with those blocks as parameters.

Mining Process. Periodically a miner is chosen by the Oracle as the winner of the mining race. In the second TinyCoin protocol developed (i.e. the honest miner protocol) when a miner is chosen, the mined block is added in the main branch of the public Blockchain. With our approach the node simply puts the block into a Map and changes the head pointer. All with a complexity of $\mathcal{O}(1)$.

Before the node proceeds to mine the next block, some several verification steps are required. The first step is to check if the mined block contained some transactions with the local node as receiver. These transactions are then added into UnconfirmedTransactionMap with confirmations counter set to one. Already contained transactions receive an increase in the confirmations counter. At this point the structure is iterated to find "confirmed" transactions, if so the wallet is updated. The second step involves the removal of the transactions contained in the mined block from the Mempool. The last step is the transmission of the mined block to all neighbors and the generation of a new candidate block.

Selfish Mining. The third TinyCoin protocol developed resumes the selfish mining strategy described in [1]. As the authors says: the key idea behind the selfish mining is to keep discovered blocks private, thereby intentionally forking the chain. This strategy leads honest miners to waste resources and permits selfish miners to get more revenue compared to their mining power.

In the Selfish Mining implementation the Blockchain is slightly different compared to the one used in the first and second protocols. Three new parameters have been added: a private head to keep track of the last element added in the private chain, a counter to handle the length of the private chain and a pointer to the last published block.

When a dishonest miner adds a received block in the main branch of the public Blockchain , the selfish mining algorithm computes the delta parameter before the insertion:

$$\Delta = PrivateHead.Height - PublicHead.Height$$

At this point there are four conditions need to be handled:

- $\Delta = 0$: The honest miners win. The public chain has one more block then the selfish private chain. The selfish miner withdraws and continue mining on the last block of public chain. The algorithm sets the private head pointer equals to the public head pointer, the private chain length counter is reset to zero and the last published block points also to the public head. The dishonest node transmits the public block to neighbors. Figure 7 shows an example.

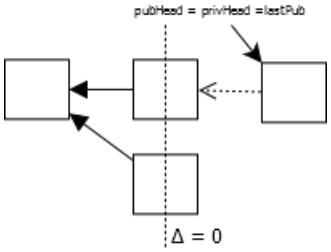

Figure 7: Selfish Miner receives a block, case $\Delta = 0$

- $\Delta = 1$: A tie has been reached. The honest miners have recovered the disadvantage and now the public chain has the same length of the selfish private chain. The selfish miner tries the fate and hopes that a portion of the network will choose its block. The node immediately transmits the last block of its private chain (i.e. the private head). Figure 8 shows an example.

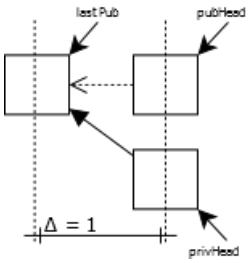

Figure 8: Selfish Miner receives a block, case $\Delta = 1$

- $\Delta = 2$: The selfish miner win due to the lead of 1. The node transmits all private blocks to their neighbors. The algorithm sets the head of the public chain and the last published block pointer equals to the head of its private chain. The private chain length counter is reset. Figure 9 shows an example.

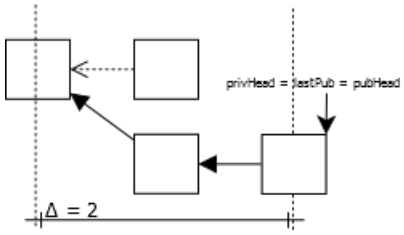

Figure 9: Selfish Miner receives a block, case $\Delta = 2$

- $\Delta > 2$: The selfish miner win due to a lead greater then one. This is the condition where the last published block pointer is useful. The selfish Blockchain iterates from the private head to the last published block. The block before the last published is the first unpublished. The found block is transmitted to neighbors and becomes the new last published block. Figure 10 shows an example.

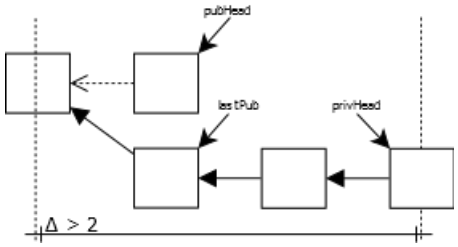

Figure 10: Selfish Miner receives a block, case $\Delta > 2$

The Complexity of all conditions is always $\mathcal{O}(1)$, excepts for the latter that requires to iterate backwards the private chain to found the unpublished block, with a complexity of $\mathcal{O}(k)$ where k is the number of unpublished blocks.

When the selfish miner is chosen by the Oracle, again the delta parameter is computed. At this point the algorithm puts the mined block into the Blockchain Map, sets the private head pointer to the mined block and increments the private chain length counter by 1. The Node continues to mine the next block and repeats the previous steps until a specific condition is reached. After the mined block is added to the private chain, if the delta is equal to zero and the private chain length is equal to two, means that the dishonest node moves from a tie to a victory. To avoid further risks the algorithm decides to transmit all private blocks. The private chain length counter is reset, the head of the public chain and the last published block pointer are moved to the private head. Figure 11 shows an example of this particular condition.

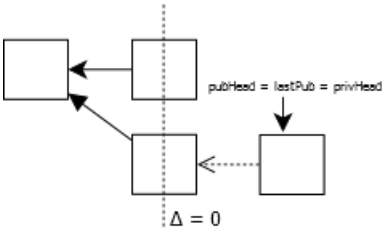

Figure 11: Block is mined, case $\Delta = 0 \wedge len = 2$

A last note about the receipt of transactions, the selfish miner inherits the method from the standard protocol but it adds a new behavior: every time a transaction is received the node must check if its also contained inside the private chain before processing it.

Probabilistic Distribution. In this work several probabilistic choiches are made for setting up the simulation. For this purpose a random weighted collection has been introduced. The collection is developed as Generic, so any entity can be weighted during probabilistic choiche. This structure includes a NavigableMap instantiated as a TreeMap, an istance of Random to generate a stream of pseudorandom numbers (to preserve the same randomization, CommonState.r from Peersim has been used) and a variable that mantains the total weight. Figure 12 shows how random weighted collection works.

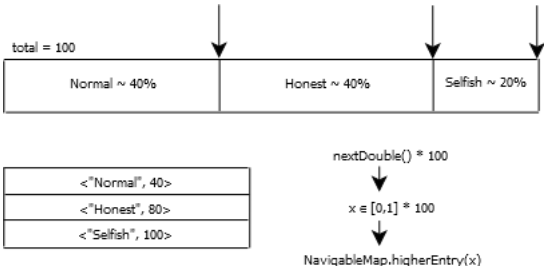

Figure 12: Random Weighted Collection for Node Type

Once the Map is initialized with weights, a random double in range [0,1] has been chosen, then multiplied with the total weight. NavigableMap provides higherEntry(x) method that returns the mapping associated with the least key strictly greater than the given key x.

Three random weighted collections have been exploited during our simulation initialization. The first collection is used to choose the node type. There are three type of nodes: a Normal node, an Honest Miner and a Selfish Miner. Each node is initialized with a different TinyCoin protocol as seen previously. The second collection is used to choose the type of miner. There are four different types based on the computing hardware powers: CPU, GPU, FPGA and ASIC. The last collection is used by The Oracle to choose the winning miner: each kind of mining hardware have a weight according to its power. Multiple nodes with the same hardware type reduce the single probability of a node to be chosen. Figure 13 shows an example of the probability to choose an ASIC node as winner with 0,9 power probability. All this weights will be discussed later during our experiments.

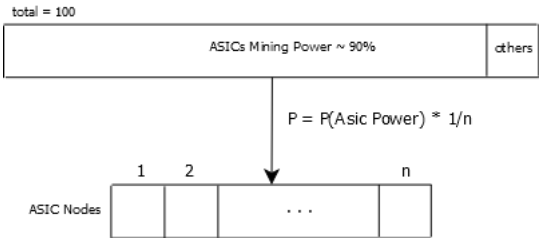

Figure 13: Choosing an ASIC node as winner

To simulate the revenue of a single pool that uses the selfish-mine strategy inside the network, the distribution of miners and hardware types follows a specific approach. One node is chosen as the selfish pool (i.e. the first element of the Peersim network array), then its probabilistic weight is computed according to mining power assigned from configuration file, with this following formula:

$$y = \frac{\%Power * TotalWeight}{1 - \%Power}$$

The selfish pool is added to the third random weighted collection with a weight of y. Let's consider a selfish pool with 0,5 power (i.e. the 50% computing power of the network). The node 0 is chosen as the candidate node to simulate the pool with the selfish mining protocol. The remaining 50% of the computational power is divided between other miners. Assume we have a distribution of 90% ASIC, 8% FPGA, 2% GPU and 1% CPU computational power for every group of miners of the same hardware type. The third random weighted collection has four weights: $(1, PWR\_CPU)$, $(3, PWR\_GPU)$, $(11, PWR\_FPGA)$, $(101, PWR\_ASIC)$. Before adding the selfish pool is necessary to compute its weight. From previously Formula we obtain $y = 101$. At this point the pair $(202, POOL)$ is inserted in the Navigable Map. Range from 101 to 201 corresponds to the 50% probability to choose the selfish pool as winner during the simulation.

The generation of transactions also follows a probabilistic approach. Every time the nextCycle() method is invoked by the Cycle-driven scheduler, there is a probability to generate a transaction. All types of node can do transactions (both miners and normal nodes). If a node has enough coins a random double between 0 and 1 is chosen. The node can do the transaction if this number is below a previously configured threshold. An example: with a threshold of 0.01 every 25 cycles and a network size of 500 nodes, the system generates 19870 total transactions. The transaction amount is also chosen randomly in range from the smallest Satoshi measurement unit and the current value of the wallet.

The initial wallet generation is also random. When the simulation is initialized each node receives a number of coins between two previously configured thresholds.

Experiments and Results. During the experiments several parameters influenced the results of the simulation. The most important are:

- The k factor of the graph (i.e. the randomly k directed edges out of each node)
- The block delay (in cycles) and its maximum number of transactions. (i.e. the block propagation time)
- The interval between each mined block chosen by the Oracle (in cycles)
- The Network size (i.e. the total number of nodes)
- Distribution of Node Types, Miner Types and Hardware Powers

In the first experiment we've considered a single selfish pool. The pool revenue has been calculated with different mining powers. The simulation has been configured in the following way:

- Network Size: 1000 nodes
- Mining interval: 500 cycles
- Simulation End: 100000 cycles
- K-factor: 4
- Block base propagation: 0,10
- Max Transactions per Block: 100
- Min Transactions per Block: 10
- Node Types Distribution: 0% selfish miners (only 1 selfish pool), 40% honest miners, 60% normal nodes
- Miner Types Distribution: 20% CPU, 30% GPU, 30% FPGA, 20% ASIC
- Hardware Powers: 1% CPU, 2% GPU, 8% FPGA, 90% ASIC

The choice of the cycle interval between two consecutive mined blocks was made in agreement with [Blockchain.info] stats. If we consider a Peersim cycle as a second in real Bitcoin system, then 500 cycles correspond to 8.3 minutes for computing a block.

The distribution of hardware powers was also made in accordance with the gap between different generations of mining hardware. At the time of this work, ASIC is the only choice for a competitive miner to have some chances of mining a block. An ASIC AntMiner S9 has a power of 14 terahash per second. The computational difference is huge compared to an high-end CPU [4]. For this reason the selfish pool is assumed to be a single ASIC miner.

The K-factor is chosen according to Peersim Ball-Expansion controller after a validation of the overlay connectivity. With a $K = 4$ and 1000 nodes, the maximum distance between nodes is 5.

Nodes	Interval	Sim End	K	Block delay	Pool Power	Avg Forks	Revenue
1000	500	100000	4	0.10	50%	28	84%
1000	500	100000	4	0.10	45%	21	72%
1000	500	100000	4	0.10	40%	20	61%
1000	500	100000	4	0.10	30%	21	45%
1000	500	100000	4	0.10	20%	21	16%
1000	500	100000	4	0.10	10%	17	3%

Figure 14

Figure 15 shows a plot of the values contained in 14. The Selfish pool increases strongly the revenue when its computational power reaches the threshold of 30%.

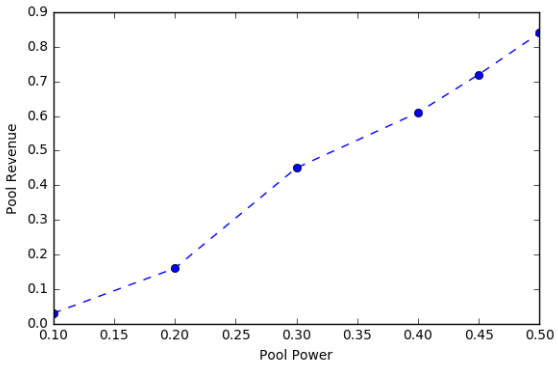

Figure 15: Selfish Pool power-revenue single test

Changing the random seed, other tests have been made for demonstrate the trend of selfish pool revenue. Results are shown in Figure 16.

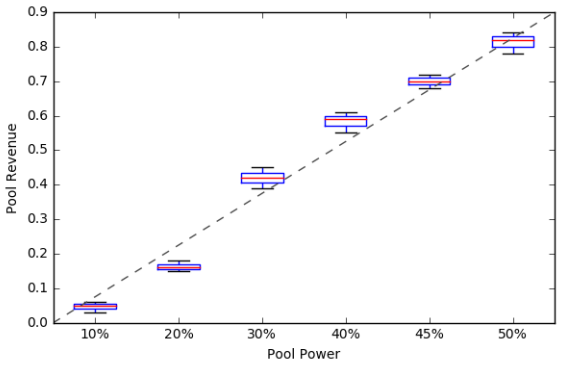

Figure 16: Selfish Pool power-revenue multiple tests

Figure 17 shows the average number of forks made by nodes. The trend is steady until the pool reaches a power of 40%, then the number of forks decreases drastically and remains constant. This behavior could be explained in fact that the selfish pool releases less frequently its private blocks. When the probability of being chosen by the oracle is 1/2, the pool tends to have a delta greater than two. The advantage over the rest of the network is more accentuated. Figure 18 shows the average number of cycles to resolve a Fork. Again when the selfish pool reaches a computational power over 40% affects the other network to generate longer forks and to resolve them in more cycles. Figure 20 highlights this phenomenon and confirms the goal of selfish mining: make honest miners to waste as many possible computational cycles.

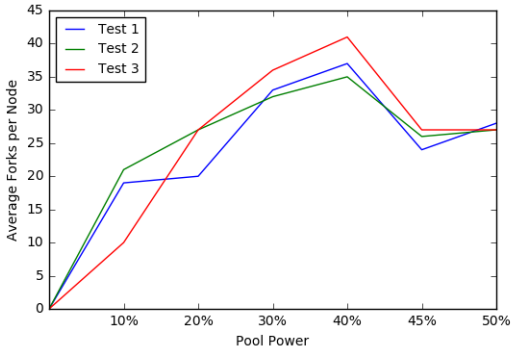

Figure 17: Average Forks per Node

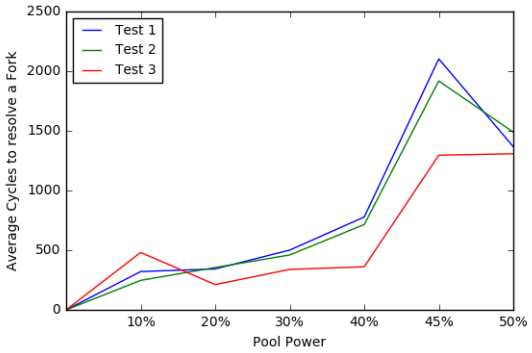

Figure 18: Average Cycles to resolve a Fork

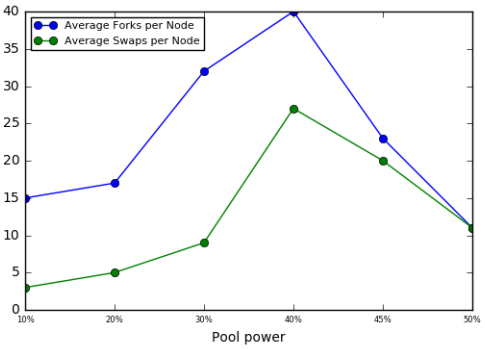

Figure 19: Average Forks/Swaps per Node

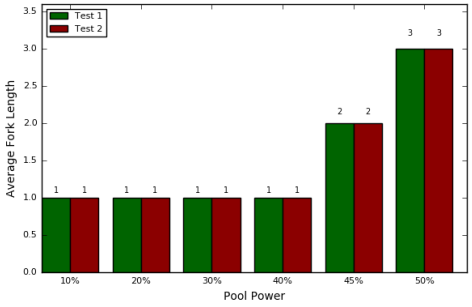

Figure 20: Average Fork Length

In the second part of our experiments, the single selfish pool is abandoned in favor of multiple selfish miners together with honest miners. Several experiments have been made. The distributions chosen are the following:

- Test 1: 0% selfish miners, 100% honest miners
- Test 2: 20% selfish miners, 80% honest miners
- Test 3: 40% selfish miners, 60% honest miners
- Test 4: 60% selfish miners, 40% honest miners
- Test 5: 80% selfish miners, 20% honest miners
- Test 6: 100% selfish miners, 0% honest miners

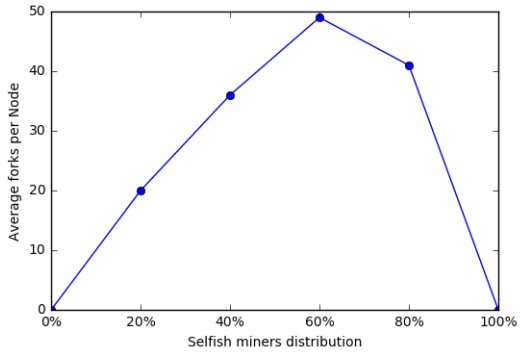

Figure 21: Average Fork per Node (Selfish Distributions)

According to Figure 21, Test 1 and Test 6 are not important. In the first case, due to good latency, if the miners follow the honest protocol forks are rare. In the second case, if the miners are all dishonest, no one put mined blocks into the public chain. In other cases is observable that more selfish miners are in the network, more forks are generated. This behavior demonstrates that selfish mining perturbs the network negatively.

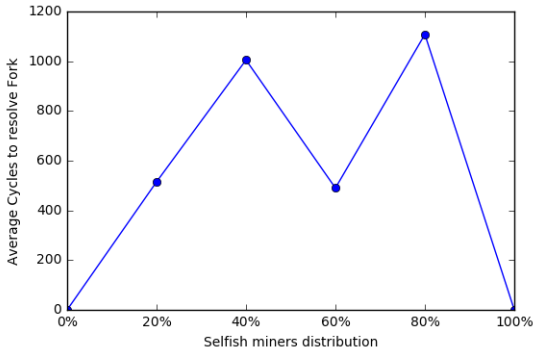

Figure 22: Avg cycles per Fork (Selfish Distributions)

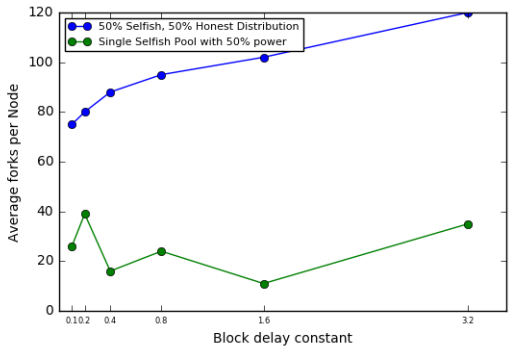

Figure 23: Avg forks per Node with different latency

Final Considerations. After this analysis we can confirm the hypotheses made by the authors of [1]: once a selfish pool's mining power exceeds a threshold, it can increase its revenue. Rational miners will join the selfish pool to increase their revenues. Once the miner pool reaches the majority, it controls the blockchain. As seen in previous charts, the presence of selfish miners also affects negatively the network. The number of forks drastically increases both with single selfish pool and multiple selfish miners distributed along the network.

References

[1] *Majority is not Enough: Bitcoin Mining is Vulnerable*, Ittay Eyal and Emin Gun Sirer, Department of Computer Science, Cornell University

[2] Blockchain.info Statistics
`https://blockchain.info`

[3] The Bitcoin Wiki
`https://en.bitcoin.it/wiki/`

[4] Bitcoin mining hardware comparison in 2016
`https://99bitcoins.com/2016-bitcoin-mining-hardware-c`

[5] *Mastering Bitcoin*, Andreas M. Antonopoulos. 2014. O'Reilly. ISBN: 978-1-449-37404-4

[6] Matplotlib: Python plotting
`https://matplotlib.org`