CS 5433 Homework 1

Due 18 March 2018

In this homework, we will use the concepts we've explored in class to build our own functioning mini-cryptocurrency.

Submit your work to CMS as a zip file similar to the one we provide, containing an additional **solutions.pdf** file for written answers. **You will work in ASSIGNED groups of 3-4 for this assignment; please see CMS for random group assignments, and do not discuss solution details outside of your assigned groups.** For all assignments, you make use of published materials, but must acknowledge all sources, in accordance with the Cornell Code of Academic Integrity. Additionally, you must ensure that you understand the material you are submitting; you must be able to explain your solutions to the course instructor or TA if requested.

Disclaimer

In general, many of the technologies we are using this semester will be poorly (if at all) documented, will be constantly changing, and will often suffer from broken or dead code or packages. This is par for the course for cryptocurrency. **Please start the assignment early**. We do not guarantee responses from the TAs or instructors on errors in the assignment or such broken packages without at least 48 hours lead time.

Problem 0 - Setting Up

First, download the provided zip in CMS.

The grading and testing code will be written in Python 3.6. **Full setup instructions and documentation for the provided code is available by opening `docs/index.html` in any web browser.**

Your solutions can be written in any programming language; we strongly recommend Python for its ability to natively interface with the provided infrastructure. If you would prefer to use a different programming language, you can use subprocess.communicate or other Python methods to call out to external binary code, including interpreters for other languages (an example is provided here: `https://stackoverflow.com/a/16770371`).

If you do choose a different programming language, please note that support from course instructors for language-related bugs may be limited, and it is your responsibility to ensure interoperability of input and output functionality throughout your code. If using a language other than Python, you are also required to provide a SETUP file with a list of Ubuntu or Debian packages required for your solution. We will not deduct points for problems setting

up your solution, but may contact you if such issues arise. You are responsible for remedying any problems in a timely way.

We encourage you to read and analyze the provided code that is not part of the assignment: please ask questions if anything is unclear to you!

Problem 1 - Signatures, Hashes, Sealing [40]

We will first explore the basic cryptography from **Lectures 2 and 3** and its applications to Bitcoin from **Lectures 4 and 5**.

We've learned that *hash functions* are used to secure blockchains through a mechanism called *Proof of Work*; our course textbook has more in-depth background on "PoW". We now implement proof of work and a popular alternative.

(1a) - Proof of Work

Recall from class that Proof of Work is required by the network for block validity; each miner in a cryptocurrency system is constantly attempting to produce valid blocks by finding a block such that:

```
H(block) < target.
```

More precisely, SHA256(SHA256(`block`)) < target (in BTC, "`H`" is SHA256^2).

One natural question is how blocks are represented in the system; our provided files `blockchain/block.py` and `blockchain/chain.py` provide an (unoptimized) infrastructure for representing and managing blocks.

In Proof of Work, what is hashed is in fact not the block but the **header** of the block; some binary metadata designed to be compact enough to efficiently communicate quickly through the network.

Our block headers take the following format:

```
block = height‘timestamp‘target‘parent_hash‘is_genesis‘merkle_root‘seal_data
```

represented as a Python string, with hexadecimal encodings for all binary values ("Merkle Root," i.e., root of a Merkle Tree, and signature) and no spaces.

The critical value here for header validity is the *seal data*, additional data which a protocol can specify as input to a function that checks whether a block is valid. In proof of work, a nonce counter is used for this seal data; a valid seal occurs when the header (including the seal data) hashes to below the target. Proof of Work mining then tries different seal data via brute force, checking if the block is valid until a valid seal (header hash) is found. A block is called *validly sealed* if its seal data causes the block to verify properly. In Proof of Work, the seal data is also called a nonce ("number that is used once").

An example value of this is:

```
H = 100‘1519668704‘
45231284858326638837332416019018714005183587760015845327913118753091066265 6‘
0074aa074e3fee902d5e2a251e90f37f528425ebcef4ae1212f988a388877acc‘False‘
8f212c0356b46c1df1d909d0ddeee09a923129dfa0c36b949df4c5fa357db158‘184
```

Notice that unlike what we covered in class, there are no transactions in these blocks. The Merkle Root is a short hash that summarizes all transactions in a block, which are propagated by the network separately. You can read more about Merkle roots here: `https://www.mycryptopedia.com/merkle-tree-merkle-root-explained/` or explore them in the bonus problem.

The mining lifecycle in our codebase for Proof of Work consists of the following process:

- A block object is created with full transaction data. This object is an *unsealed* block, as it starts with the default nonce of 0, and so may not be a valid PoW block.
- The mining loop checks whether the block's seal is valid (a.k.a. the hash of the block's header is below the target). If not, it increases the nonce/seal data of the block by 1 and tries again, until the block is valid.

An example of this process is given in `examples/add_single_pow_block.py`. You will first code some helper functions for this process.

To decide which block to use as the next block for mining and transacting on, the network uses what is called the "best chain" model. Because blocks can have varied difficulties, just because a chain is longer does not mean more work was expended to produce it (it is possible to have a high-difficulty short chain if all targets are low, or a low-difficulty long chain if all targets are high). To solve this, each block is given a "weight", or an estimate of how hard the block was to create. The best block in the chain is then called the "heaviest" block, and is the block that is on the chain with the most weight. A block B's "chain" is the string of blocks between genesis and B, obtained by repeatedly following the parent hash pointers in the block header.

You should now complete the following functions (we recommend completing them in this order):

- `blockchain/util.py/sha256_2_string(string_to_hash)`: returns the SHA256(SHA256)) hash of a given string. .
  Provided tests: `tests/hash.py`
- `blockchain/pow_block.py/get_weight(self)`: Gets the "consensus weight" of a block's chain, as described above. In Proof of Work, the "weight" is the same as the total work on the chain. Calculate the weight using the ratio of a block's target (`Block.target`) to the maximum possible target, where the max possible target is 2^{256}. Notice that in the current codebase, all PoW mining is constant-difficulty, but your method should be able to handle later extensions to include blocks with different targets.
  Provided tests: `tests/weight.py`
- `blockchain/blockchain.py/get_chain_ending_with(self)`: Gets a list of block-hashes in the blockchain ending with the block represented by `block_hash`. The first item in the list should be the provided block hash, and the last item in the list should be the hash of the genesis block (as above, check with the `Block.is_genesis` attribute).
  Provided tests: `tests/get_chain.py`

Please refer to the provided Python docs for full documentation of all the data structures and methods required for this problem; for more complex methods, we provide some hints, but you may need to explore the documentation additionally on your own.

It is not enough for a block's seal to be valid; for a block to be accepted into the chain, the block must also obey all the rules of the currency system it is running (such as enforcing that no money is created out of thin air or no input is spent twice). Coding this validation will be the most substantial component of this homework. Please complete the following method:

- `blockchain/block.py/is_valid(self)`: returns True if a block is valid. A list of conditions for block validity is provided as comments in the included code, as well as a list of methods and datastructures you may find helpful. Note that the provided test file includes multiple unit tests; we will be assigning partial credit for implementing partial validation and passing some but not all tests.
  Provided tests: `tests/validity.py`

Add your solutions to the appropriate files, at (1a) placeholders therein.

(1b) - Proof of Authority

Recall from class that Proof of Work is an inherently wasteful system. In many systems that use blockchains, such as supply-chain blockchains (`https://www.ibm.com/blockchain/supply-chain/`), sometimes participants share a single root of trust or set of trusted parties. In such use cases, it is possible to construct a blockchain without the use of Proof of Work. Instead of the above seal validity conditions, a Proof of Authority seal is valid when

```
Verif(PK, [block, sig]) = accept
```

(where Verif is a function in the digital signature primitive described in Lectures 1 and 2).

For a block to be valid, it must be *signed* by the public key of a valid authority, who is trusted by all blockchain participants. Some version of this scheme is used in many enterprise deployments, where the ability to produce blocks can be cleanly delegated to a trusted party, but where all system participants want the ability to rigorously and cryptographically audit the actions of this party.

You will implement the `mine` method corresponding to the provided method in the PoW block class; we provide utility methods `get_private_key` and `get_public_key` in the authority file; in a real consensus system, node implementations would only need the public key, with the corresponding private key available only to the trusted authority. To simulate this, notice that we **do not** use the `get_private_key` function in the provided `is_valid_seal`.

Tests are provided in `tests/poa.py`.

Add your solution to the `blockchain/poa_block.py` file in the root of the project directory, replacing the (1b) placeholders therein.

(1c) - Hybrid Systems

There are a variety of system designs possible beyond the proof of work and authority sealing mechanisms we've explored here. Name one security problem with the proof of authority sealing mechanism you created above. Can you think of a technically sound approach that could help mitigate this problem (imperfect mitiagions are fine)? (Please do not describe

existing protocols, including but not limited to Proof of Stake protocols, Tendermint, or PBFT-based systems; originality is highly valued here!)

Add your solution to a `solutions.pdf` file in the root of the project directory.

Problem 2 - UTXO Management in Wallets [15]

In Lecture 6, we learned about the representation of Bitcoin as inputs and outputs. Additional reading on this "UTXO" model of representing account state is available in the textbook.

With Cornell Chain, we have provided some code that generates a random proof-of-work blockchain; in this code, we simulate various users transacting with wallet software, choosing which UTXO to use randomly from the UTXOs available to a given user.

Unfortunately, this strategy is clearly not efficient: wallets want to maintain as few UTXOs as possible in Bitcoin, since transactions are charged according to size, and using / creating more output costs money.

Consider an enterprise business called Moonbase, which must make and receive hundreds of thousands of transactions to millions of users every day, paying millions of dollars in network fees. Moonbase's strategy is as follows: for every user that wants to deposit money, Moonbase receives a new UTXO from (and created by) that user. For every user that wants to withdraw money, they create a withdrawal transaction funded by a random UTXO that is large enough to fund this transaction.

1. Identify a denial-of-service vector in this process, i.e., an adversarial attack strategy by dishonest and wealthy users that could result in failed withdrawals for legitimate users of Moonbase. Provide a fix for this DoS vector and argue informally that user withdrawals will never fail.

2. For each user withdrawal in the provided scheme, recall from class that *two UTXOs* actually need to be generated: one paying the target user, and one that is kept by Moonbase representing any leftover "change". Provide a modification to the above strategy that will reduce the number of UTXOs Moonbase must maintain in its database.

Add your solution to a `solutions.pdf` file in the root of the project directory.

Problem 3 - Consensus [30]

Consider the following simple n-party BA protocol.

- Round 1: The sender, given a message m as input, multicasts m to all receivers.
- Round 2: Each receiver i, upon receiving a message m_i from the sender sends multicast a message (called a vote) of the form (VOTE, m_i) to everyone; the sender also multicasts a bit for m.
- Round 3: Each player i, checks whether they received votes for some message m from at least $\frac{3}{4}n$, and if so they output m. Otherwise, they output $\perp$.

Answer the following questions:

1. Under what corruption threshold does the protocol satify validity? For instance, does it satisfy validity when $< n/6$, $< n/4$ or even $< n/3$ of the players are faulty? Prove it, and provide an attack showing that your bound is optimal. (Recall that providing validity means that if the sender is honest, then all honest players need to output the sender's input.)

2. Show that the protocol does not satisfy consistency even if just the sender is faulty. Provide an explicit attack. (Recall that breaking consistency means coming up with an attack that leads 2 honest players to output different values; for instance, one of the could output some message m, and a different one outputs $\perp$.)

3. **Bonus**: Show that the protocol satisfies a weaker form of consistency where all honest players either output the same message m, or they output $\perp$, even if $< n/2$ of the players are faulty. (That is, you need to show that if 2 honest players output messages m_1, m_2, neither of which is $\perp$, then $m_1 = m_2$.) Hint: prove a variant of the quorum intersection lemma shown in class.

Add your solution to a `solutions.pdf` file in the root of the project directory.

Problem 4 - Visualizing Bitcoin Data [15]

One rich space of exploration in the Bitcoin scientific literature is that of measurement and measurement studies. We will now explore several visualizations of the Bitcoin network.

The data we are using is sourced from the BlockSci package (`https://github.com/citp/BlockSci`), a tool for high performance scientific analysis of Bitcoin blockchain data. Because running BlockSci requires a synchronized Bitcoin full node and the syncing process usually takes several days, we provide you some sample data for analysis.

You will recall in class that we calculated the profitability of mining. In the `data/difficulty.csv` file, we provide a list of difficulties for each block on the Bitcoin network. Each line is a block in the format `block_number,difficulty`.

Complete the provided `data/profitability.py/calculate_profitability(...)` function. When you are done, running `python3.6 data/profitability.py` should output a graph of mining profitability over time. Test your function with `tests/data.py`.

Note that there are two major impacts on profitability that show up on the graph: new hardware release, and network-wide reward halvening (described in the course textbook).
Add your solution to the `data/profitability.py` file in the root of the project directory, replacing the (4) placeholder therein.

Grading Scripts and Testing

You can run all provided grading scripts and tests in this assignment by running `python3.6 run_all_tests.py` from the root of your project directory. Note that we may have left edge cases untested. The provided tests will be worth 80% of your score, so passing these will guarantee at least a 80% across the coding component of this assignment (unless you tailor solutions specifically to tests, which is worth no credit).

(BONUS) - Merkle Trees

In problem (1), we explored block header formats, which included a "Merkle Root" summarizing all transactions in a merkle tree.

Currently, our implementation of `blockchain/block.py/calculate_merkle_root` is incomplete; when asked to generate a Merkle Tree for a list of transactions, this method just returns a hash of the transactions as strings, in sequential order!

For 10 points of extra credit,

- Implement and test `get_merkle_tree`.
- Describe, at a high level, the algorithm you would use for generating a Merkle proof of transaction inclusion in a block. You can read background about Merkle proofs in the course textbook.

Add your solutions to the `blockchain/block.py` and `solution.pdf` files in the root of the project directory, replacing the (BONUS) placeholder therein.

Evaluation

To help us tune future homeworks in the class, please answer the following in your **solutions.pdf**:

- Did you find the homework easy, appropriately difficult, or too difficult?
- How many hours total (excluding breaks :)) were spent on the completion of this assignment?
- Did you feel there was too much coding, the appropriate amount of coding, or not enough coding?

Any other feedback on the homework or class logistics are appreciated!