Cryptocurrency Cabal: Problem Set 2

Alec Grieser
ahg3de@virginia.edu

October 9, 2015

1. To support Satoshi's claim, it must be the case that the action of the theoretical greedy attacker in order to secure their wealth would be detectable. Given the public nature of the Bitcoin network, this *should* be the case, but we could imagine a large attacker with many addresses (and possibly many IPs from which to work) masking the fact that it is reversing its own transactions by mixing in random other transactions each time and using different public addresses with each transaction. If an attacker could successfully hide their actions, then the apparent integrity of the Bitcoin network would not be compromised as no one would be aware that transactions were being surreptitiously reversed.

   Another assumption that Satoshi makes is that the incentive (the mining reward with transaction fees) remains at a level higher than could be successfully gotten by reversing transactions. Now, it is true that actions that undermine the integrity of the system in general will devalue the currency held personally by the attacker, but this attacker could attempt to avoid such a problem by transferring their wealth into traditional currency. If they did this, the only thing keeping them honest in the system is the idea that they might increase their wealth by remaining in the system and playing by the rules. If the fee is smaller than could be gained by reversing their own transactions, some greedy miners may stay in only to dishonestly do the latter.

2. This means that if we have a greedy attacker controlling 45% of all available hashing power who is attempting to reverse a payment they made, then if one want the probability that that transaction will be reversed by the production of an alternate version of the blockchain to be less than 0.1%, then one should wait until 340 blocks have been added to the chain after the one that includes your transaction.

3. The same table but with the target of $P < 0.05$ would look like this:

q	0.10	0.15	0.20	0.25	0.30	0.35	40	0.45
z	3	3	5	6	10	17	36	137

   This was generated by running the code in Satoshi's paper but by cutting off the search for z when the probability of a successful attack exceeded 0.05 rather than the value of 0.001 used in the paper.

4. (a) The problem with this naïve approach (if we don't include the index in each chunk) is that the cloud storage provider could simply drop records (or rearrange them) everything would pass our tests. This is because each record only contains information pertaining to its own contents and doesn't have any information regarding its relationship with other records.

   (b) We can't be certain that the data match what we wrote. If the cloud storage engine returns any record that we wrote, it will return a piece of data that matches the signature we gave it, so it could return a record different from the one we wrote and we would be unaware. Additionally, even if we included information specifying which index each record refers to, we could be certain that the database includes information that we wrote to the given position, but we can't be certain it is the latest write. The cloud service could simply return a previous version of the record which would match both the signature associated with it and would have the needed position data.

5. (a) If we wished to read from the database and verify its contents, we would have to read the entire database each time. We would then recompute the hash of all the data concatenated together, storing along the way any records we actually want to read the contents of. Then we can just check the value of the computed hash with the value we have stored locally to verify that the data read are the same as the data previously written. To write to a position i, we would have to read from every position and recompute the hash substituting our new data for the data at position i. We then update the value of the hash of the data that we store locally and write our new value to position i in the database.

   (b) Note that we compute the hash for both reading and writing by keeping track of a running version of the hash as we iterate through the data and the updating it with the new data from each new record, we never have to store more locally than the running hash, two records (the one we are actually interested in reading or writing and the next one we are adding to the hashed contents), and the global hash, so this can be done with $\Theta(1)$ local memory. But as we increase the number of records n, because each verification stage requires re-reading the entire database, the time to verify will increase as per the number of records, i.e., reading and writing require $\Theta(n)$ time (assuming the physical actions of reading and writing to the cloud storage engine are $\Theta(1)$ operations).

6. (a) So, to clarify, the procedure requires that we store a copy of the Merkle tree on the cloud server and a copy of the Merkle root locally. The procedure for reading the data at position i is then to retrieve the data from the cloud server and compute its hash. Now, for any node in the Merkle tree, we can compute the hash stored in its parent by taking the hash of the hash in the original node concatenated in the correct order with its sibling node in the tree (which can be retrieved by requesting its value from the cloud server). If we repeat this process starting with the node with only the hash of the data at position i until we get to the root of the Merkle tree, we can compute the Merkle tree of the data stored on the cloud server (or at least, the Merkle root of the tree containing $data[i]$). We can then verify that there

hasn't been data corruption by verifying that the computed value of the Merkle root is the same as the Merkle root that we keep locally. For writes, we need to make a few different writes to the cloud server. First, if we are writing to position i, we need to first update $data[i]$ in the cloud storage server and then we need to update the node in the Merkle tree containing its hash with the new hash. From there, using the method described above, we need to update the hash stored in that node's parent in the Merkle tree, and then that node's parent, and so on, until we recompute the hash stored in the Merkle tree's root node. From there, we need to update the value of the Merkle root stored locally to the new value.

(b) Note that the locally memory stored will only every contain the data from one record (the data we are interested in reading or writing) and up to three hashes (the locally stored Merkle root, the hash of one node in the Merkle tree, and the hash of that node's sibling). This won't change as the number of records n increases, so this requires $\Theta(1)$ memory. Now, because Merkle trees are complete binary trees, the depth of the tree will be $\lceil \log_2(n) \rceil$. This means that for every read from the database, to verify we will have to do at most $\lceil \log_2(n) \rceil + 1$ additional reads and compute that same number of hashes (as well as do one hash comparison). This means that reading with verification is a $\Theta(\log n)$ operation. Likewise, whenever we write and update the data we use to verify, we will have to do $\lceil \log_2(n) \rceil + 1$ additional writes to the cloud server coupled with $\lceil \log_2(n) \rceil + 1$ additional reads (plus one local write). This means that writing with verification is a $\Theta(\log n)$ operation. (This is assuming, as before, that each read or write from the database is a $\Theta(1)$ operation.)

7. (a) We expect the blocks to be distributed fairly evenly amongst all nodes because of the randomness of the process of finding blocks. Therefore, if a mining pool controls 15% of all hashing power, it should be expected to find about 15% of all blocks mined. If we find blocks every 10 minutes, then the total number of blocks found by the mining pool is:

$$\alpha \cdot \frac{1\,\text{day}}{10\,\text{minutes/block}} = (0.15) \cdot \frac{24\,\text{hours}}{1\,\text{day}} \frac{60\,\text{minutes}}{1\,\text{hour}} \frac{1}{10\,\text{minutes/block}} = 21.6\,\text{blocks}.$$

So, we should expect to see about 21.6 blocks found per day by the mining pool.

(b) So, within t minutes, we expect that approximately $\frac{t}{10}$ blocks will be found by the network in general. Then, if blocks are evenly distributed (weighted by hashing power), then a mining pool that controls a fraction α of all available hashing power should be able to find $\alpha \frac{t}{10}$ blocks in t minutes.

8. If we assume everyone is honest, then we will get an orphaned block from the pool with hashing power α if and only if the pool produces a block in the time between when the rest of the network finds a block and the pool learns about it or the rest of the network finds a block in the time between when the pool finds a block and the rest of the network hears about it. Now, as both of these intervals (i.e., the time between the mining pool finding the block and the network hearing about it and the time between the network finding a block and the mining pool hearing about it) are equal to the

latency of L seconds in the network, we can calculate the number of blocks by noting that:

(i) Suppose that someone in the network not in the pool finds a block. Then, there are $\frac{L}{60}$ minutes for the pool to find an alternate block. Using the formula from the last problem, we would then expect the pool to find $\alpha\frac{L}{600}$ blocks in this time. Now, because there are $24 \cdot 60 = 1440$ minutes in a day and thus we expect to find $\frac{1440}{10} = 144$ blocks in one day in total, we expect the network not in the pool to find about $144(1-\alpha)$ blocks in one day, so we expect the pool to have an opportunity to produce an orphan block $144(1-\alpha)$ times a day. This leads to an expected number of

$$144(1-\alpha)\alpha\frac{L}{600} = 0.24\alpha(1-\alpha)L$$

alternative blocks.

(ii) Suppose that someone in the pool finds a block. Then there are $\frac{L}{60}$ minutes for the rest of the network to find an alternative block. In this time, we expect the network to find $(1-\alpha)\frac{L}{600}$ blocks. We expect that the pool will find 144α blocks in the day (using logic from above), so that means the rest of the network will find

$$144\alpha(1-\alpha)\frac{L}{600} = 0.24\alpha(1-\alpha)L$$

alternative blocks.

So, we expect that there will be about $0.48\alpha(1-\alpha)L$ blocks a day that become orphaned blocks. (Note that it is not necessarily the case that the second block generated is orphaned each time we have two blocks announced at the same length, but it is the case that one of these two blocks will end up orphaned.) For more concrete numbers, if we assume $\alpha = 0.15$ as in the last problem and $L = 30$ (not an unreasonable latency in the Bitcoin network), then we get about $0.48 \cdot 0.15 \cdot (1-0.15) \cdot 30 = 1.836$ orphaned blocks per day. If we assume our pool is larger, lets say $\alpha = 0.35$, then we get about 3.276 orphaned blocks per day.

9. Observe the following:

(i) Whenever the selfish miner finds a new block, it will never release it immediately to the network as this would node would extend the chain (which is against the point of selfish mining). This means the event of a selfish miner finding a node can never create an orphan block.

(ii) Suppose the selfish mining pool is zero blocks ahead of the other supernode. If the rest of the network finds a new block, then because of the spy notifying the pool instantly of the new block, unless the mining pool finds a new block at exactly the same moment as the rest of the network (which is highly unlikely), the mining pool will consider this to be the new block and it will be added to the consensus chain. Thus, no orphan blocks are created.

(iii) Suppose the mining pool is exactly one block ahead of the other supernode. Then if the rest of the network finds a new block, then spy will notify the pool that the new block was found, and the mining pool will move on to working on a new block with the hidden block discarded having never been announced to the network. Thus, no orphan blocks are created.

(iv) Suppose that the selfish mining pool is two or more blocks ahead of the other supernode. Then if the other supernode releases a new block, the selfish pool can release one of its blocks, and this block will register as a conflict in that it will compete with the block released by the rest of the network as the head of the list. If the selfish miner still has a lead of two or more, then this is the end of what happens. If the selfish miner now only has a lead of one block, it will reveal that block as well to assert that the chain it found is the consensus chain. Now, it is possible that the other supernode will find a block in the L seconds that it takes for this information to be transmitted. In general, we would expect it to find about $(1-\alpha)\frac{L}{600}$ blocks during that time. Should a block be found, the mining pool will have an advantage in that it has already started working on the next block in the chain, but the other supernode has more hashing power which may counteract the head start the mining pool has had. In any case, I will assume that the selfish miner will succeed in getting both of its blocks registered by the whole network when it reveals them as the number of times the other supernode preempts it should be rather small and it complicates analysis significantly. Now, this means that when the reveal of two blocks happens, the consensus chain will be the one that the selfish miner has been building. If this chain has length F (that is, there have been F blocks mined by the selfish miner since the last time it was only zero blocks ahead), the other miner will have mined $F-1$ nodes, all of which become orphaned blocks. As discussed in class, the expected value of F is $2+\frac{\alpha}{1-2\alpha}$, so we expect $1+\frac{\alpha}{1-2\alpha}$ blocks to be orphaned every time the selfish miner gets to be two blocks ahead of the rest of the network.

(v) So, if we are in a state where the selfish pool is zero blocks ahead of the rest of the network, there is a $1-\alpha$ probability that the selfish pool does not get the next block, so no orphaned blocks are found. There is a probability of $\alpha(1-\alpha)$ that the next block is won by the selfish pool, but the following block is found by the other supernode, so no orphaned blocks are generated. Then, with probability α^2, the selfish pool will generate two blocks and then $1+\frac{\alpha}{1-2\alpha}$ orphaned blocks are generated. So, the proportion of blocks that are orphaned will be

$$\frac{\text{expected number of orphaned blocks}}{\text{total expected number of blocks}} = \frac{\alpha^2\left(1+\frac{\alpha}{1-2\alpha}\right)}{(1-\alpha^2)+\alpha^2\left(2+\frac{\alpha}{1-2\alpha}\right)}.$$

If we multiply this proportion by 144, which is the number of blocks we expect to mine per day (as there are $24\cdot 60 = 1440$ minutes per day and 10 minutes required to mine per block), this means we expect to find:

$$144\frac{\alpha^2\left(1+\frac{\alpha}{1-2\alpha}\right)}{(1-\alpha^2)+\alpha^2\left(2+\frac{\alpha}{1-2\alpha}\right)}$$

orphaned blocks per day.

So, to put concrete numbers on this, if $\alpha = 0.15$, as before, then we expect that there will be about 3.83 orphaned blocks per day and about 30.2 orphaned blocks per day when $\alpha = 0.35$.

10. (This answers part A.) One other scheme for paying miners would be to give a small payout for every near solution produced by a miner equal calculated by looking at the expected number of these near solutions that we are expected to find for each real solution. This would mean that every computation is worth the same no matter how long it has been since the last solution was found (and thus discourages pool hopping). However, one problem with this incentive scheme is that it requires the pool manager to pay out rewards to the pool workers before any coins have actually been won through mining.