Attack Of The Clones : Identifying and Negating Sybil Attacks

Ilyas Ridhuan 20770816   Thomas Smoker 20935166

1 Introduction

Imagine scrolling through eBay looking for a specific listing, but once you find it there are twenty different sellers all offering the same product at similar prices. In an attempt to distinguish the best seller to buy from, you look at feedback scores to determine who is the best, or the most trustworthy. However, not one seller has any negative feedback, with no scores less than 99.5% and most being 100% reputable. Further to this all the comments are overly positive: good buyer, prompt payment, highly recommended etc. How could this be happening, if they are all selling the same thing at the same price?

You have come across a common problem facing online market place: Sybil attacks. In such a scenario, all accounts are owned by a single user with the purpose being to engage in fake transactions to bolster the reputation. Given the relative online anonymity of these marketplaces Sybils can be very difficult to detect, but fortunately they leave behind transaction footprints which can be traced.

Our study attempts to explore the idea that instead of detecting these Sybil attacks and discounting them accordingly, a graph-theoretic algorithm can be used to ignore their effects. Given that Sybils are characterised by a cyclic graph, only analysing flows in and out of subgraphs can omit these cycles. We can show this by trialling our algorithm on the current real world Bitcoin network.

2 Background

Before we explain the algorithm and show application and results, its best to explain some of the concepts underpinning the reasoning behind it.

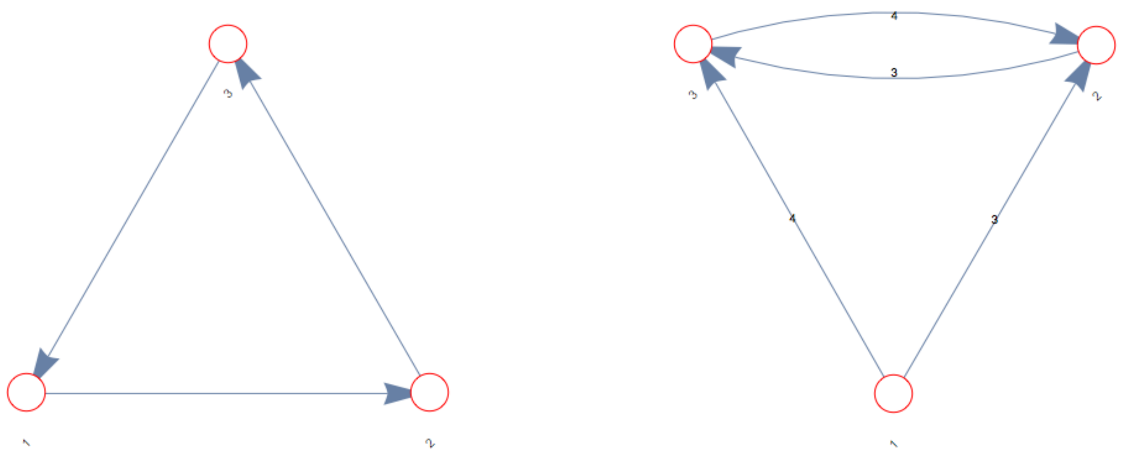
2.1 Sybils

In the information age it is increasingly easy to create online identities, as many and as specific as you choose. This fact is a big driver of internet globalisation and the meteoritic rise of e-commerce. You or I could create multiple accounts on sites such as eBay or Amazon and be buying and selling in a matter of minutes. Most of these accounts only require en email address, of which many can be easily created. While this system is encouraging for wider engagement, it falls fault to dishonest sellers who can easily create several fake identities online. Sybil attacks are a specific use of such fake accounts: all of these accounts effectively collude (even through they are all centrally controlled) to transact with each other, which boosts the reputation of either all or a few of these accounts. To the external community these users seem to be particularly effective and reputable buyers and/or sellers, which are more likely to be engaged with.

This level of collusion leaves breadcrumbs, however, which can be used to exclude these Sybils and restore the validity of a reputation-based marketplace. As Sybils mostly only interact with other Sybils before engaging in a legitimate transaction, these early contrived interactions typically show themselves as a cycle in a network graph. While cycles are east to detect in undirected graphs, a problem arises in real world applications of transactional networks; these are multi directed graphs, meaning searching for cycles will not give a complete solution.

2.2 Weighted MultiDirected Graphs

Within these graphs there can be multiple directed edges between nodes, and these edges will have a weighting. This makes the search for cycles harder, as directed cycles traditionally begin and end at the same node (Fig 1(a)). This analysis will not capture all Sybils (Fig 1(b)) and therefore would render the solution incomplete and possibly inapplicable. However, a featur of real world transactional networks can bridge this gap: they are closed systems.



(a) A cycle in a directed graph (b) Sybil behaviour that does not form a cycle

Figure 1: The shortcomings of looking for only cycles

2.3 Net Network Flow and Closed Systems

To help this solution we can consider net network flow, or divergence, of transactional networks. This is the difference between the values found from the inflow and outflow edges considered, and can be calculated with eqn(1) from based on the work done by Beuchler et al [1].

$$N(G_s, E_s, E_p) = \sum_{e_s \in E_s}^{s} w(e_s) - \sum_{e_p \in E_p}^{p} w(e_p) \tag{1}$$

Where:

N is the net network flow

G_s: Is the subgraph of a graph of G

E_s: Are the edges from the subgraph to its successor nodes

E_p: Are the edges fto the subgraph from its predecessor nodes

w : Is the weight of a given edge

For example, eBay and Bitcoin are closed systems as their feedback and currency, respectively, is not able to leave the network. Because of this, the divergence of the whole system is zero, which implies that the probability of a subgraph also having zero divergence is improbably and this would hint at collusion (or something of similar effect) within a subgraph. Further to this, divergence can also be used to exclude transactions from nodes within the subgraph when viewed from nodes outside the subgraph. Therefore if divergence is considered within reputation calculations these exploits can be omitted.

2.4 Networkx

The algorithm created is written in Python and utilises a graphical package called Networkx[2]. It was chosen for several reasons: its algorithms (numerical and statistical) are written in C and Fortran, making it very efficient; it uses native Python data structures such as a dictionary of dictionaries makes it relatively seamless and scalable for large and connected graphs; and it plugs in very effectively to Numpy, SciPy and Matplotlib for easy of analysis and advanced plotting[3].

3 Implementation

Our initial planning was to run this algorithm on a list of eBay transactions, to as closely mimic the effect that we believe this type of analysis could have. However, while the eBay developer API does give transactional information it does not provide the value of the transaction or the transaction score, which means the graph edges would be unweighted. Given this, we decided on the Bitcoin network. As Bitcoin utilises the blockchain so all necessary information is readily available and searchable; with 130 million transactions since 2008 between 360,000 unique addresses there is more than enough information to validate the algorithm and concept.

The code works by starting from a node the user selects to transact with and calculating the net flow of that node. From here, connected edges are iteratively added using Breadth First Search (BFS) to form a new subgraph. Then the net flow of this created subgraph is calculated, and the process is repeated with the new nodes in the subgraph and so on. The difference between the current subgraphs net flow and the previous net flow should not be zero unless the subgraph is effectively the entire network. The implication of a difference of zero means a lack of connectivity: either their is collusion (Sybils) within the subgraph, or the user in question is adding no value; regardless, this is not a node that you would want to interact with.

While using Python is helpful for this example in terms of ease and power there are some limitations, such as the lack of recursive tail optimisation. This would cause a recursive BFS of the graph (for large graphs) to result in a stack overflow, due to all of these stack frames accumulating. However, we understood this the begin with, and to combat this the graph traversal portion of the code was written iteratively.

```
//netflow.py
def limitedNetFlow(G,subgraph,maxDepth):
    flowArr = []
    depth = 0
    sub = set(subgraph.nodes()) #Set containing all nodes in current subgraph
    fringe = set(subgraph.nodes()) #Set containing only nodes on the current boundary
    while depth < maxDepth:
        neighbour = set() #Deduplicated list of all neighbours of nodes in the fringe
        currFlow = 0

        for node in fringe:
            currFlow +=
                G.in_degree(node,weight="weight")-G.out_degree(node,weight="weight")
                #Running total of inflow - outflow for each node
            neighbour |= set(G.predecessors(node))
            neighbour |= set(G.successors(node))

        fringe = neighbour - sub #fringe equal to nodes which are neighbours but not
             already in the subgraph
        sub |= neighbour #subgraph is now the deduplicated list of subgraph and the new
             neighbour set
        # try is here only for the first interation where depth-1 = -1
        try:
            ## add previous flow to the current flow to get flow across current fringe.
                Addition is used because outflow is negative
             # and inflow is positive
            flowArr.append(currFlow+flowArr[depth-1])
        except:
            # Occurs when there is no previous flow
            flowArr.append(currFlow)

        depth += 1
    return flowArr
```

Another consideration of the algorithms limitations is for it to be properly effective it needs to run on a variety of everyday machines, with varying computations strengths. With this in mind we kept efficiency of the code a priority, and kept the ability of users to trade computational intensity for accuracy. A solution for this is memoization; maintaining a store of information already calculated and then adding to it only what is necessary, which is essentially remembering prior calculation. In our case, at each iteration the size of the subgraph increases by once edge, which means only flow entering and leaving form nodes outside the subgraph are necessary to calculate the new convergence. By simply adding this net flow to the previously calculated and stored net flow of the previous subgraph we have an answer, which saves many repeated expensive calculations.

A use case of our solution is as follows: a user enters the desired partners address and the depth at which they are willing to search. The code will then return the net flow at each depth until the required depth has been traversed. Allowing the user the specify depth gives them the option of time and accuracy, as further depth will give a more correct result, but take longer in computation and therefore time. There are also checks in place to make sure the user passes in legitimate information, otherwise exceptions are thrown.

4 Bitcoin Network

For the final test, the code is run (and validated) against the current Bitcoin transaction network. We ran a full Bitcoin node on an Amazon Web Services EC2 instance, with the blockchain data fully synced and indexed. After this, the python rpc interface is used to query for address and its transactional information. Figure 2 shows a graph of a subsection (100 transactions) of the main Bitcoin transaction network[4].

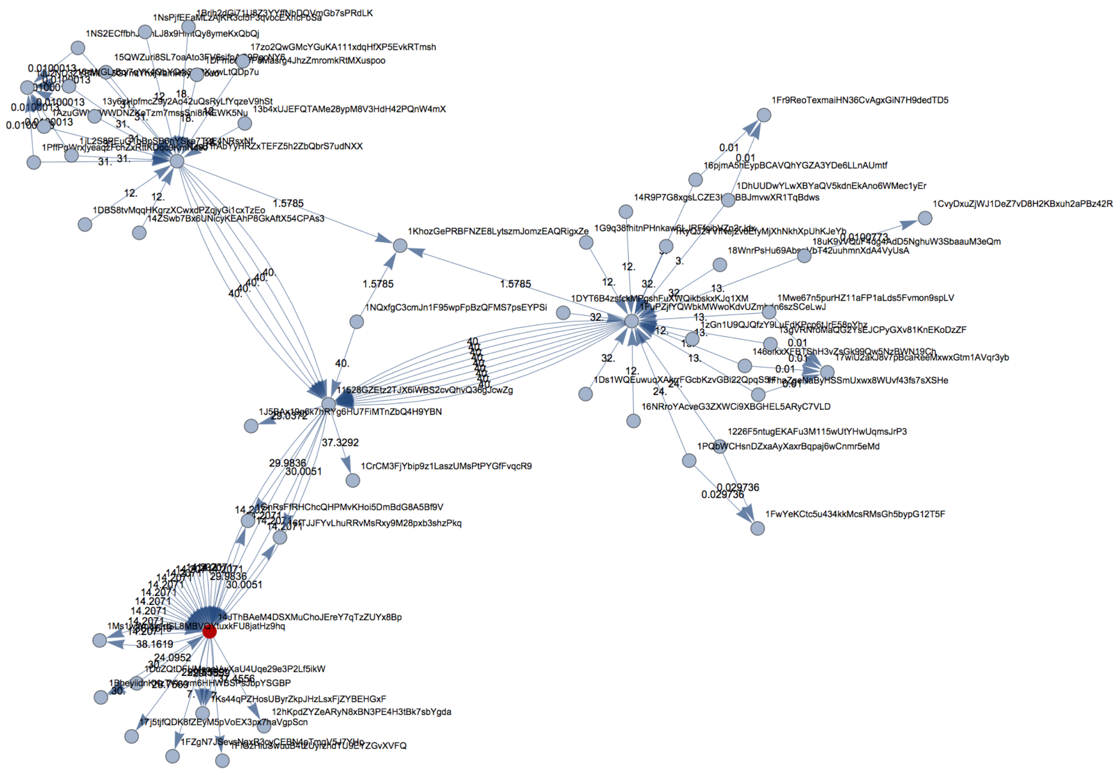


Figure 2: First 100 transactions originating from red node

The net network flow as function of search depth is show in figure 3, as can be seen the relatively low network connectivity of the starting node (highlighted in red) is displayed in this low net network flow levels between depths one and five. While the flow does pick up at higher levels this is more as a result of its neighbours connecting to highly connected nodes. We did not expect to find much sybil behaviour in the Bitcoin network as there is no incentive, financial or otherwise, to doing so. Much of the repeat transactions between two nodes are through users having change addresses that are separate from their spending address.

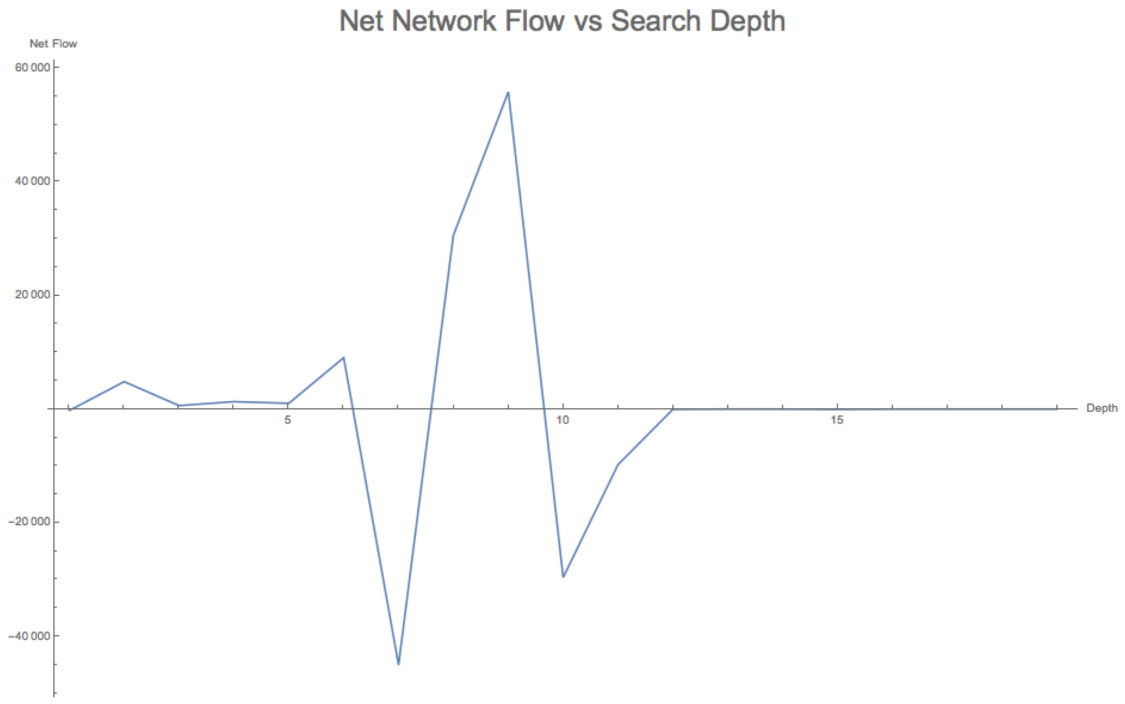


Figure 3: Net Network Flow of Bitcoin as a function of search depth

5 Conclusion And Discussion

From the results gained from the Bitcoin experiment and correlating it to the Bitcoin network, it can be seen that the algorithm produces accurate results, and works as we intended it to. We can omit the results where repeated transactions occur from the net network flow that is observed by an external node. The ability to exclude these results is very powerful, as it is easy to disincentive Sybil attacks as they would no longer have any effect on a score such as reputation.

In regard to improvements on our implementation there are some we have identified. Firstly, the current algorithm takes in an entire graph of the network and traverses it. However, with real-world examples such as eBay and Bitcoin these networks are unrealistically large, making it very unlikely a user will want or need to cover it in its entirety. We could then truncate the graph at n+1 edges, with n being the edge depth specified by the user, as this is all the user is concerned with.

Another improvement related to the Bitcoin graph. The mechanism of payment for every transaction in Bitcoin has the change from that transaction sent to a specific change address, which is most commonly the same address that sent it. In other words, all of the Bitcoin value is consumed in the transaction, and so often Bitcoin are just circled back to the original sender. This causes a problem as these edges are traversed but instantly cancelled out. The algorithm should be adjusted to not follow edges that have the same source and target nodes, which would remove a significant portion of the calculations.

Finally, to further mimic real world interactions the net network flow could be normalised at each depth. This is to allow for the fact that transactions further from the desired node are further from the users web of trust, and should therefore be weighted less than edges closer to the original node. This would give most likely more accurate results, and could easily be optional for the user.

Potential uses we see of this algorithm is in the area of reputation management for online transactional networks. If instead of the current global reputation system (which, as we have shown, is vulnerable to Sybil attacks) a transactional network used a relative reputations in which the user assess and validates the counter-partys reputation based on connectivity, fake transactions could be ignored.

References

[1] M. Buechler, M. Eerabathini, C. Hockenbrocht, and D. Wan, "Decentralized Reputation System for Transaction Networks," tech. rep., University of Pennsylvania, Philadelphia, 2014.

[2] "Networkx." `https://networkx.github.io/`. Accessed: 2016-05-25.

[3] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," pp. 11 – 15, 2008.

[4] J. Garzik, "python-bitcoinrpc." `https://github.com/jgarzik/python-bitcoinrpc`, 2011.