Identification of Atomic Arbitrage on Solana

Jun (Cedric) Jiang

The Overall Approach

The solution uses an approach based on account token balance changes. The high level idea is to look at token balance changes in each account (except for the signer which is assumed to be the beneficiary if the transaction is an atomic arbitrage). The idea is based on observation that the arbitrage will involve a series of swaps. For example, the signer can swap n token T for m token S at exchange A, then swap m token S for k token R at exchange B, then swap k token R for h token T at exchange C. If you observe the three exchanges account token balance change, it'll be like this:

Exchange A: +n token T, –m token S
Exchange B: +m token S, –k token R
Exchange C: +k token R, –h token T

That is:

1. Every exchange will have one token balance increase and another token balance decrease
2. The token decrease on one exchange will have the same amount of token increase on another exchange, so we have a "chain" of exchanges (A => B => C in this case)
3. In this chain, the last exchange surrenders some token and the first exchanges receives some token – they have the same type and different amounts

Here we expect h > n for the signer to have (positive) gains and the amount (h – n) should match what is shown by the token balance change from the signer's side (with caveats discussed later along with advantages and limitations of this method).

Implementation Details

For simplicity, the code is written in Python using `requests`, a well-known HTTP library.

Rate Limiting

The RPC endpoint has implemented rate limiting. In our experience, the limit usually comes from the size of the response as the `getBlock` call returns a lot of data, and other limits (per time number of requests) do not really kick in. We surveyed the rate limiting solution on top of the `requests` package used, but it does not offer a "response size" based rate limiting (which probably makes sense because you cannot actually predict the response size before you make the request). We decided to just keep making RPC calls until a 429 response code is hit, then wait a period time as specified in the "retry after" header. (The actual implementation adds one second to this time for safety, which also handles the case where the response code is neither 200 or 429, or there's not "retry after" header, which never happened in our experience)

Because of this, there seems no need to try to optimize the runtime by parallelizing each block since the bottleneck is data acquisition, not processing.

Construction of Token Balance Changes

By making the `getBlock` RPC call using `accounts` level of details (instead of `full` which include instructions that are not used in this method). The response JSON will include `preTokenBalances` and `postTokenBalances` information in the metadata. Each item has the owner, mint and amount information that allows the code compute "per owner per mint" token balance change dictionary. Note to avoid annoyance of float computation and comparison, the intermediate result of the "amount" is represented as an integer amount and its decimals. Zero change is removed from the dictionary.

Construction of Account Linkage

This is the implementation of the overall method mentioned above. From the "per owner per mint" token balance change dictionary from the above step, we first remove the signer itself from the dictionary, then pick owners that satisfy the conditions that there are only two token balance changes and they must be one positive one negative. This will trim the dictionary to look like this:

```
{
    "ownerA": { "token1": 0.2, "token2": -5},
    "ownerB": { "token2": 5,   "token3": -100},
    "ownerC": { "token3": 100, "token1": -0.3},
    "ownerD": { "token3": 0.1, "token4": -500},  # this one is not relevant
}
```

To create the "linkage", we first transform the dictionary above to another dictionary where the key is the tuple of the token and the amount of change (in absolute) of values, and the value is a pair of owners that have outflow and inflow, so the above dictionary will be turned into:

```
{
    ("token1", 0.2): [None,     "ownerA"],
    ("token2", 5)  : ["ownerA", "ownerB"],
    ("token3", 100): ["ownerB", "ownerC"],
    ("token1", 0.3): ["ownerC", None    ],
    ("token3", 0.1): [None,     "ownerD"],
    ("token4", 500): ["ownerD", None    ],
}
```

For the dictionary above, we only care about the "value" part which is the owner pair, if neither is None, then it represents a link. We see two links here, that is A => B and B => C. Our next goal is to find the longest link, like A => B => C here. The algorithm works like this: pick an existing link, say B => C, then see if there are other links that starts at C or ends at B, and if there is one, we extend the link (here A => B => C). If we cannot find such "additional" link (including the candidate pool becomes empty), we know the algorithm has finished. The algorithm can be repeated if candidate pool has other links.

Note this implementation has some limitations which will be discussed later.

Identification of Arbitrage

Given the "longest" link of owners, the way to identify arbitrage is to see if the first owner has "received" certain token that is the same type as the last owner has "given out". If so, the amount difference will be the arbitrage amount. We would like to match that amount to the signer's token balance change. However, we noticed that this check may not work as expected since there may be other instructions that make transfers of a part of the gain like "tipping" (e.g., in this transaction). For `WSOL`, there are also possible wrapping and unwrapping regarding native `SOL`. This will unfortunately bring in some false positives.

The code will output the information in the CSV format, where the schema is `signature, beneficiary, mint, amount`. Signature is the (first) signature of the transaction, which is typically considered as the ID of the transaction. Beneficiary is the (first) signer of the transaction which we assume is the beneficiary of the arbitrage. Mint is the token type and the amount is, well, the amount of gain through this arbitrage.

Given the information in the CSV data, it's fairly straightforward to compute statistics. By running this code for 100 slots, we observed arbitrages predominantly on `WSOL` and some on stable coins (`USDT` / `USDC`), since for some statistics we need to assign a dollar value for the token, we used a hardcoded amount for them and in case we encounter other tokens (likely false positives mentioned above due to avoiding the "sanity check" for reasons), they are ignored, for simplicity.

Discussion and Future Work

There are a few benefits with using the approach. The first one is simplicity as it only uses account data. This makes the code simpler (no need to worry about parsing instructions which will be a huge task), and also allow us to set the level of details parameter in `getBlock` RPC call to be `accounts` instead of `full` to save response size and avoid getting rate limited too soon, and there is no need to query account info of any kind.

This approach is also somewhat "principled". It does not require "hardcoding" well-known exchanges or whatever, just tries to figure out what might be an exchange that is involved in the chain, and there is no limit on the number of exchanges involved.

The drawback of this approach is also obvious. The biggest one is it cannot find failed arbitrages. The reason is failed transactions will not incur any balance changes and thus cannot be found. There is no "as if succeed" balance change information that can be used. This unfortunately leads to the solution unable to answer the "percentage of failure" question.

Another limitation is the accuracy. Because we cannot inspect instructions to see what actually happens, we only have access to the "end result". One notable miss is for example we failed to identify this transaction as arbitrage, because the swap using `WSOL` for `Nailong` happens on two different accounts ("Meteora (SOL) Vault Authority" and "Meteora (Nailong) Vault Authority", respectively) – they belong to the same owner "Meteora Vault Program". This may be solved by querying account information to figure out the "ownership" tree and "consolidate" accounts if necessary.

The implementation of account linkage is a simplified version and thus has limitations. The actual algorithm should not remove owners with more than 2 token balance changes, as there may be some changes not relevant to the arbitrage. Also, the link extension should be modeled as a graph search problem, like you select a node in the graph and try to find paths that involve this node. The algorithm here does not handle the case where there can be multiple choices when you try to extend. Think about the case:

```
{
    "ownerA": { "token1": 0.2, "token2": -5},
    "ownerB": { "token2": 5,   "token3": -100},
    "ownerC": { "token3": 100, "token1": -0.3},
    "ownerD": { "token3": 100, "token4": -500},  # this one is a confounder
}
```

Which should be the in/out flow for `("token3", 100)`? In our current implementation, the code will produce either `["ownerB", "ownerC"]` or `["ownerB", "ownerD"]`, depending on which comes later (will overwrite existing). This isn't right because you actually want to keep both. In this case, we should have three links A => B, B => C, and B => D, then the "extension" algorithm should obtain two chains A => B => C and A => B => D and consider both of them and realize the latter isn't right because A's inflow token type differs from B's outflow token type, while the former is correct. This distracting owner D can happen in cases where there are mint and burn instructions.

In short words, our current code works well on simple arbitrage transactions, where there are no other things happening at the same time (or there are but they happen not to affect our algorithm – say a tip payment through native `SOL` balance transfer). The improvement discussed above helps to make the method more robust, but still cannot guarantee complete accuracy due to the inherent drawback of this approach: with access to only "total" balance change per owner/token, you do not know what exactly happens in the transaction – you only get a "final" view, not per step view.