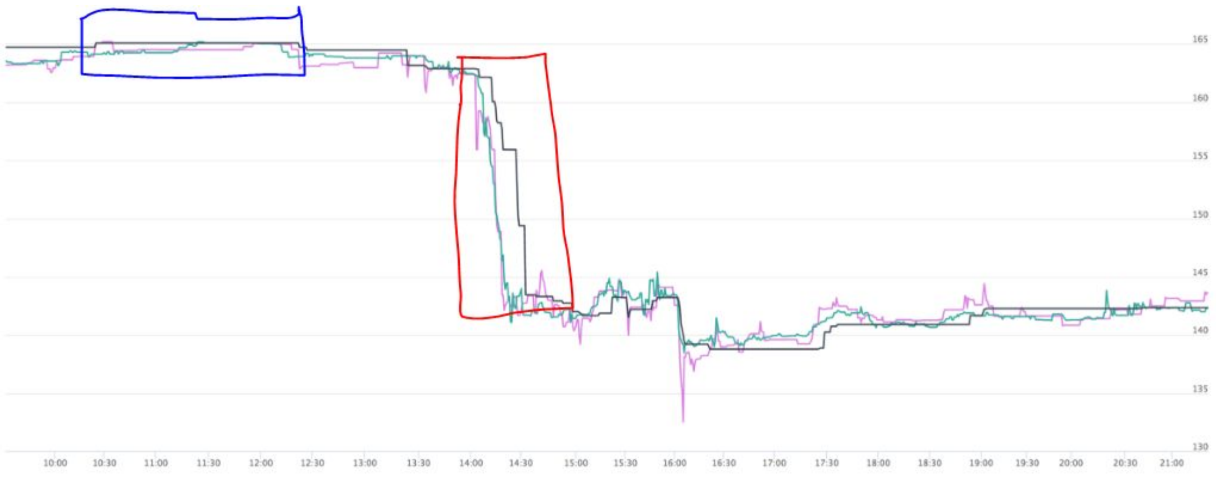
A Decentralized Cryptocurrency Pricing Algorithm

Hackers:
Marco Aguilar, Dean Wasil, Jonah Yamato

Our project was to develop a decentralized pricing algorithm for cryptocurrency that is protected against market manipulation attacks while simultaneously remaining up to date with the current exchange rate. A similar system named Polaris has already been implemented, but there was room for improvement. Due to limited time and limited background with the Polaris documentation, our code is untested and the system is not fully implemented. It will be assumed that the reader is familiar with Polaris. All concepts should be perceived as improvements to Polaris, not as an entire new system, as most of Polaris will remain in place. Concepts of the algorithm will be described piece by piece in order to show each individual improvement to the decentralized cryptocurrency pricing algorithm.

Problems with Polaris

1. It is slow to respond to drastic market changes, such as during a crash. This can be seen in the red box in the picture below, taken from the Medium blog on Polaris when the market crashed. Notice that Polaris lagged behind this crash by ~10 minute.
2. It does not adjust the median when it is less than 1% off, even for extended periods of time. This can be seen in the blue box in the picture below. Notice that Polaris does not change its median for ~2 hours, and during this time the median is off the real price by about 1.5$, just less than 1%.



Poke Conditions

Recall the Polaris poke conditions:

1. The current price on Uniswap is >1% different from the median price.
2. The current price on Uniswap is >1% different from the median of all checkpoints in the last 3.5 minutes
3. The current price on Uniswap is >1% different from the last checkpoint
4. It has been >3 hours since the last price checkpoint was registered

Our poke conditions are as follows:

1. The current price on Uniswap is >1% different from the median price and it has been > 15 seconds since the last price checkpoint was registered
2. It has been >30 minutes since the last price checkpoint was registered

Below is the code implementation of our poke conditions:

```
uint public constant PENDING_PERIOD = 30 minutes;
uint public constant QUICK_PEND_PERIOD = 0.25 minutes;
```

```
function _willRewardCheckpoint(address token, Checkpoint memory
checkpoint) internal view returns (bool) {
    Medianizer memory medianizer = medianizers[token];
    return (
        medianizer.prices.length < MAX_CHECKPOINTS ||
        (block.timestamp.sub(medianizer.pendingStartTimestamp) >=
QUICK_PEND_PERIOD && _percentChange(medianizer.median, checkpoint) >=
MIN_PRICE_CHANGE) ||
(block.timestamp.sub(medianizer.pendingStartTimestamp) >=
PENDING_PERIOD
    );
}
```

<u>Checkpoints</u>

Like Polaris, we remember the last 15 checkpoints. However, the timeframe between checkpoints may be anywhere between 15 seconds to 30 minutes due to our poke conditions.

Below is our code implementation of updating our checkpoints:

```
bool public quick_pend = false;


if ((quick_pend == true &&
block.timestamp.sub(medianizer.pendingStartTimestamp) >
QUICK_PEND_PERIOD) || (quick_pend == false &&
block.timestamp.sub(medianizer.pendingStartTimestamp) >
PENDING_PERIOD) || medianizer.pending.length == MAX_CHECKPOINTS) {
    medianizer.pending.length = 0;
    medianizer.tail = (medianizer.tail + 1) % MAX_CHECKPOINTS;
    medianizer.pendingStartTimestamp = block.timestamp;
}
```

```
... Update Median ...

if (_percentChange(medianizer.median, checkpoint) >=
MIN_PRICE_CHANGE) {
      quick_pend = true;
}
else {
      quick_pend = false;
}
```

Poke Protocol

Our protocol uses two different algorithms depending on if the price difference is > or < 1%. Our protocol operates as follows:

1. When the current price on Uniswap is >1% different from the median price, the median is updated with our S curve algorithm
2. Else the current price is updated with our meet-in-the-middle algorithm

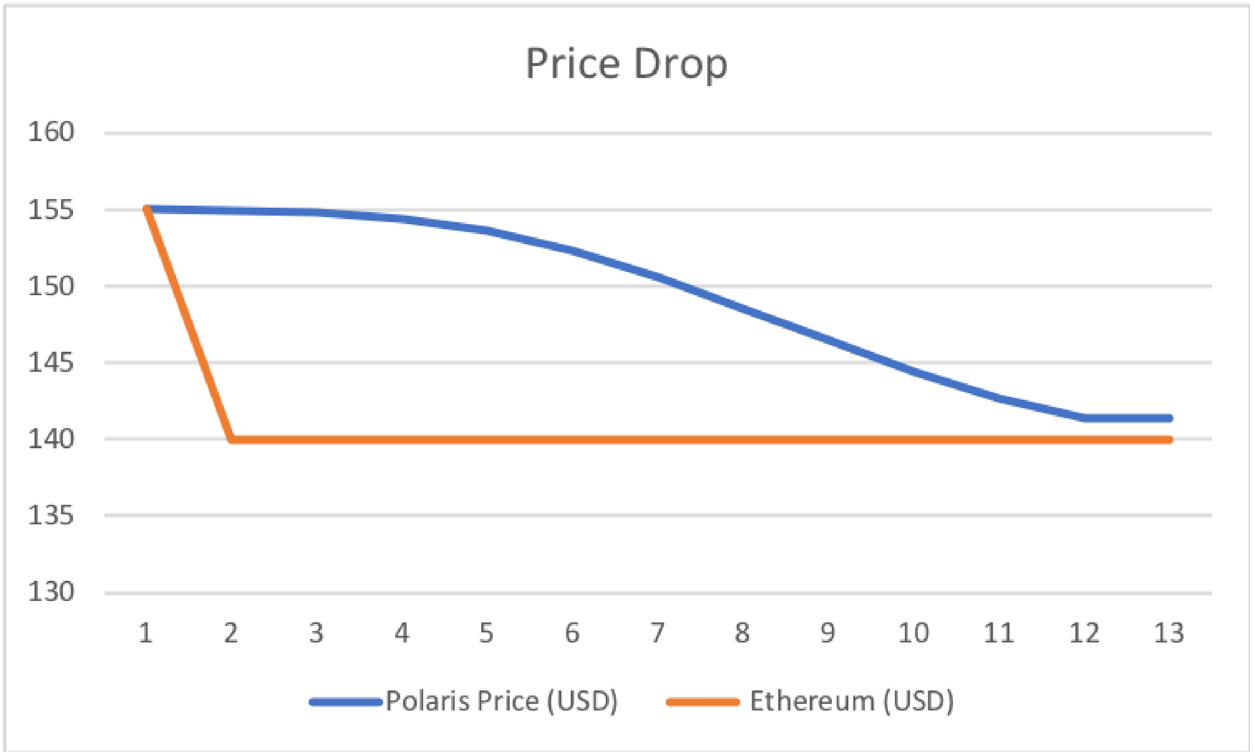
S Curve Algorithm

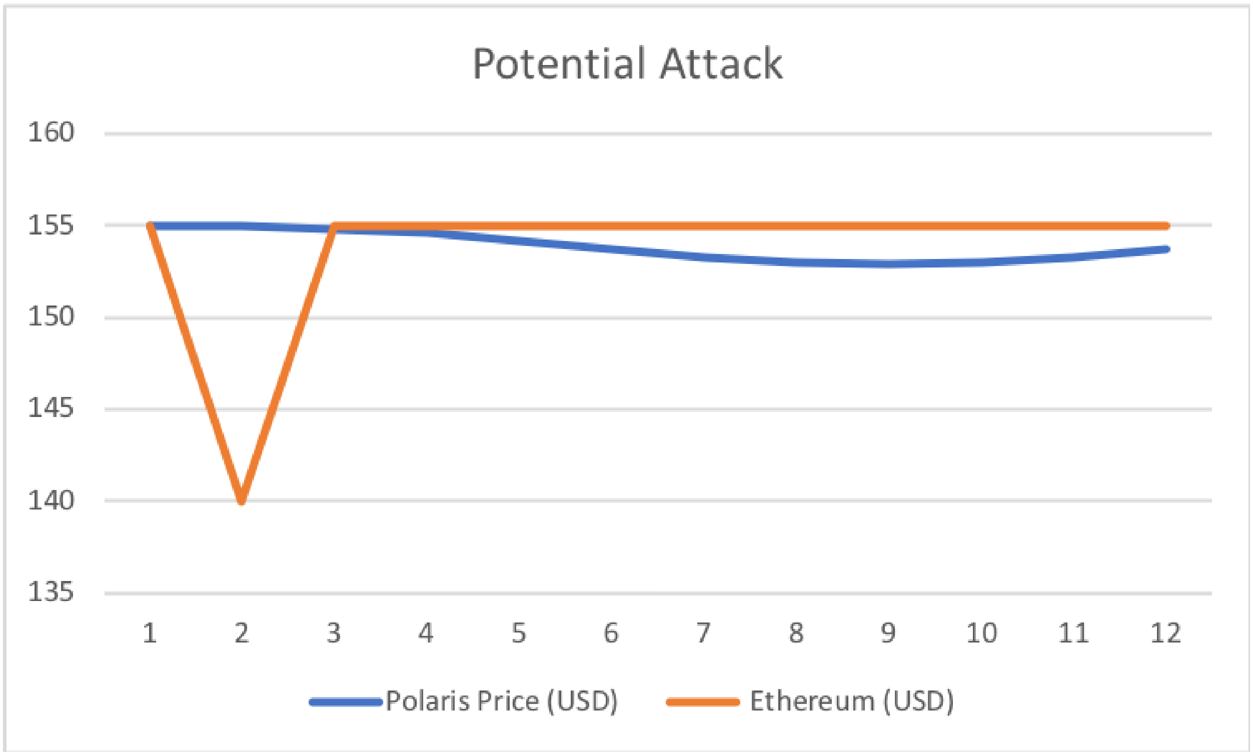
The determining factors for this algorithm are the drop in price and the amount of time that such a drop remains consistent. When the drop is > 1%, our S curve algorithm begins to process the data and adjust the median until it reaches a difference <1% between the median and Uniswap (actual) price. The weights are larger towards the middle of the processed checkpoints and smaller at the beginning and oldest checkpoints (hence the S curve). This is because the S curve algorithm must compensate for the price drop by changing our median price the longer it remains deviant from the actual price. Simultaneously, this algorithm causes large, erratic changes in the market price to barely affect the median, while allowing the median to quickly follow large, long-term market changes. In other words, market manipulation which causes these large, erratic changes will not affect the median, while real, quick, large changes, such as during a market crash, will cause the median to quickly converge on the real, new price. This algorithm solves problem 1 with Polaris, by reducing the time it takes to respond to large market changes from ~10 minutes to ~ 2.5 minutes.

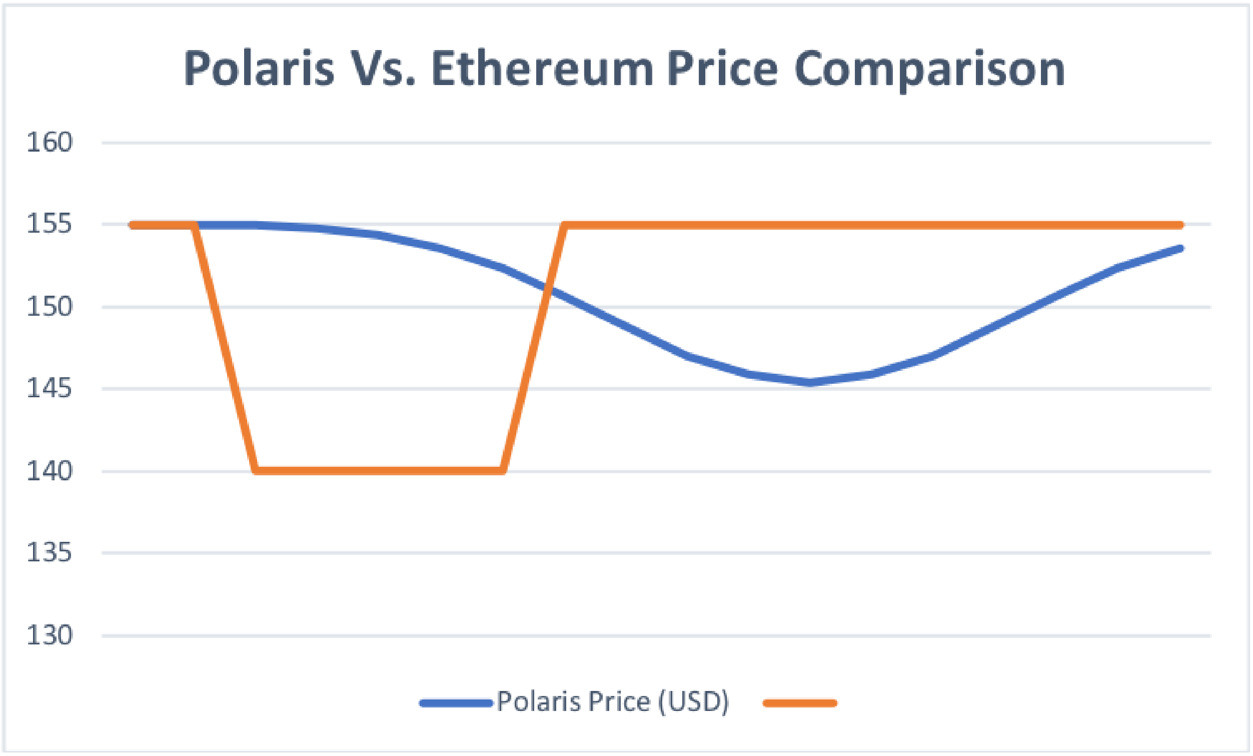
Below is the code implementation of our S curve algorithm:

```
uint8 result = 0;
if (_percentChange(medianizer.median, checkpoint) >=
MIN_PRICE_CHANGE) {
      uint8[16] weights = [0,1,4,9,19,30,41,47,50,47,41,30,19,9,4,1];
      uint8 count = 15;
      uint8 weights_sum = 0;
      while (count >= 0) {
            result += medianizer.prices[count]*weights[count];
            weights_sum += weights[count];
            count -= 1;
      }
      result /= weights_sum;
}
...
medianizer.median = result;
```

Below are graphs of how the S curve algorithm responds to drastic market changes:







<u>Meet-in-the-Middle Algorithm</u>

This algorithm takes the current median and the average of the real price since the last checkpoint and moves the new median to be directly in the middle of them. This allows the algorithm to slowly keep up with the real price, while keeping subscription prices low due to infrequent pokes. This algorithm solve problem 2 of Polaris.

Below is the code implementation of our meet-in-the-middle algorithm:

```
else {
      result = (medianizer.median +
medianizer.prices[medianizer.tail]) / 2;
}
medianizer.median = result;
```

<u>A Crypto Weight</u>

One solution we came up with but did not have the time to implement is the idea of creating a weight for the median price that is tied to the crypto market as a whole. The idea is that large price swings in the currencies like Bitcoin will reverberate throughout the market and send all cryptocurrencies in one direction. By tracking this we can match the swings closer. It also hedges the median from attacks against a single currency because there will be that counter weight tied to the whole crypto market. It is more than likely that any major changes in the crypto market as a whole will be organic. We were not able to implement this crypto weight.

<u>The Incentive Problems</u>

One of the major problems of this model of rewarding pokers is that these costs are incurred by the subscribers, and to grow the service we would like a low subscription cost. Previously a flat rate cost of 5 Ethereum per month was the transaction fee to have access to the prices provided by Polaris and that 5 Ethereum per month has been paying the pokers to continuously update our prices. We have implemented a solution to lower the cost of subscription by dividing the 5 Ethereum by the number of subscribers so that as more subscribers subscribe to the service, the monthly fee becomes lower. This model makes sense because no matter how many subscribers our service attracts, the costs of pokes should remain the same and thus we can lower the subscription costs. There are other options such as to reward the pokers with a larger share of the profits should that be appropriate, but this would also disturb the natural equilibrium of the number of pokers.

Below is our code implementation of a variable subscription fee:

```
uint public constant MONTHLY_SUBSCRIPTION_FEE = (5
ether)/accounts.size;
```

Another problem we identified is that users can redeem their Oracle Coin as soon as the transaction has been recorded on the ledger. By allowing users to arbitrarily redeem coins at any time essentially, it has the potential to fluctuate the value of an Oracle Coin because there may be more coins available at certain times than others. The solution we propose is to limit when users can redeem their coins to a certain time of day or week so that there is more stability in the cost of an Oracle Coin, as everyone must redeem at the same time. Perhaps we could alter the smart contract so that we automatically burn all the coins in existence at a point in time that we deem appropriate if that doesn't violate the nature of decentralization. We were not able to implement this in code.

Incentive Costs

Our costs for incentives will be high because we allow poking only when necessary. When there are large price movements like a stock plummet or rally, we allow more pokes. However, these dramatic crypto movements are generally limited and smoother situations are the norm. Looking at an average day for Ethereum, it is unlikely that within 30 minutes the price difference would be >1%. Therefore, it is unlikely that there will be more than one successful poke every 30 minutes, which keeps payments at a minimum. This ensures that the price is kept up to date with subtle differences in prices while also minimizing costs.

Weighing Bots Low and Humans High

When the median is being updated, it would be helpful to check if the the market is moving primarily due to bots, and react to those movements less than because they are erratic and manipulative. A method of implementing this would be to check the buyer in each transaction to see how long it has been since their last sell. If the timeframe between this buy and their last sell is low (< ~15 seconds), it is likely that they are a bot that is quickly buying and selling to influence the market. If the time frame is larger, it is likely that they are a human making real transactions. This can be done vice versa for the seller too. Knowing if the timeframe was short or long would allow us to change the influence of the a transaction on our median. This would be done by adding weights to each transaction as the transactions are being averaged to form the checkpoint. If the timeframe is low, the weight will be very low, and if the time frame is larger than a ~15 seconds, the weight would be higher. The longer the time frame the higher the weight to a certain point. An S curve similar to the one used in the S curve algorithm would suffice for the weights. We were not able to implement these weights in code.

Neural Network Concept

A Neural Network would allow Polaris to create the optimal solution to remedying the problem of continuous real time update and resistance to price manipulation through hacks. One would incentivize the network to follow trends in real time while also providing a penalty for following it too closely and falling for price manipulation attacks. One of the hardest tasks would be to create a data set that demonstrates the natural movements of the market and feeding it into the neural network to learn from. This data set would need to have examples of attacks, which themselves may be hard to identify regular organic trends, including big jumps or losses. The element of time also makes it more complicated than simple image recognition tasks and would be similar in difficulty to speech recognition. A specific type of Recurrent Neural Network called a Long Short Term Memory neural network would be probably one of the more promising networks to look into for solving this problem. Once trained, the neural network would have weights that optimize the tradeoff between real time accuracy and stability. We were not able to implement this neural network.