- Funding: up-front funding provided by the market's creator, which covers the creator's maximum possible loss in this Market. This is calculated using the *loss limit*, a parameter specified by the user.
- Loss limit: this parameter is set by the market's creator. The loss limit (ℓ) sets the maximum amount of money the market's creator can lose if the number of shares goes to zero:

$$\text{maximum possible loss} = \ell \times \log N_{\text{out}} \qquad (1)$$

  N_{out} refers to the number of outcomes, across all events, in this market. Greater values of ℓ translate to greater market liquidity, but greater potential loss for the market's creator.
- Creator: the address of the market's creator.

CreateMarket Transaction

As shown in Fig. 2, the CreateMarket transaction has an input and an output for each event included in the market, plus one additional input and output for the user's Bitcoin payment, which provides the market's initial liquidity. If N_{E} is the number of events included in the market, then the CreateMarket transaction has $N_{\text{E}} + 1$ outputs:

1. The market's input funding is sent to a special *market pool* address, which is a hash of the market's data fields.
2. There is a coinbase output for each event that is included in the market. These coinbase outputs create an essentially unlimited number of shares for each event.[8] Including an event in a market spends the event transaction's fee output, so that it is no longer available to be included in other markets.

This is stored in a transaction as follows:

```
{
    "type": "CreateMarket",
    "loss_limit": 1.2,
    "vin": [
        {
            "n": 0,
            "value": 27.72588722,
            "units": "bitcoin",
            "tradingFee": 0.005,
            "scriptSig": "<Joe's signature>
                          <Joe's public key>"
        }
    ],
    "vout": [
        {
            "n": 0,
            "value": 27.72588722,
            "units": "bitcoin",
            "script": "OP_DUP
                       OP_HASH160
                       OP_EVENTLOOKUP
                       OP_ISSHARES
                       OP_MARKETCHECK"
        },
        {
            "n": 1,
            "value": 10^9,
            "units": "shares",
            "event": "<event-1 hash>",
            "branch": "politics",
            "script": "OP_DUP
                       OP_HASH160
                       OP_EVENTLOOKUP
                       OP_ISBITCOIN
                       OP_MARKETCHECK"
        },
        {
            "n": 2,
            "value": 10^9,
            "units": "shares",
            "event": "<event-2 hash>",
            "branch": "politics",
            "script": "OP_DUP
                       OP_HASH160
                       OP_EVENTLOOKUP
                       OP_ISBITCOIN
                       OP_MARKETCHECK"
        },
        ...
    ],
    "id": "<market hash>",
    "creator": "<Joe's address>"
}
```

The `tradingFee` field is the transaction fee to buy or sell shares in the market. This is specified by the market's creator, and is expressed as a multiple of transaction volume. Here, Joe has set `tradingFee` at 0.01, so traders in this market will pay a 1% fee. These fee payments are divided in half, and split into two Buy and Sell transaction outputs each: one sent to the market's creator, and one sent to the market pool.

The locking Scripts for the CreateMarket outputs are somewhat different than those used by Bitcoin:

```
OP_DUP
OP_HASH160
OP_EVENTLOOKUP
OP_ISBITCOIN (or OP_ISSHARES)
OP_MARKETCHECK
```

The purpose of Bitcoin locking Scripts is to verify that the recipient controls the private key associated with the public key hash recorded in the Script – that is, that they are the legitimate owner of the unspent output. By contrast, to spend shares (or Bitcoins) from the CreateMarket outputs, the requirement is that the sender supply (1) the event ID (hash) of which they are Buying or Selling shares, and (2) the number of shares or Bitcoins which they wish to trade.

[8] Our initial implementation will include 10^9 shares per event; this can be increased later if needed.

`OP_EVENTLOOKUP` ensures that the target event ID matches one of the events in the market. `OP_ISBITCOIN` verifies that the units field in the unlocking input Script is `bitcoin`; this instruction is in the Script for the shares outputs. Similarly, `OP_ISSHARES` verifies that the incoming units field is `shares`, and is in the equivalent Script for the market pool (Bitcoin) output.

Happy with the market, Joe then publishes it. This deducts the $\ell \log N_{\mathrm{E}}$ market funding payment from his account, and broadcasts his CreateMarket transaction to the Augur network. Miners can then pick up and include this transaction in a block like any other transaction. Once it is incorporated into a block, and attached to the blockchain, Joe's Market becomes visible to all users of the Augur network.

III. BEFORE THE EVENT: FORECASTING

Joe's prediction market has now been created. When the market is first created, the events contained in it have not yet occurred. This is the *forecasting* phase. It lasts from the time the market is created until the expiration specified by each event. Typically, event expiration should coincide with the occurrence of the event: Joe's event is set to expire at midnight on November 5, 2016 (recorded as a Unix timestamp, 1478329200).

In this phase, the market's participants are forecasting or predicting the outcome of the event by making wagers. The way that they make these wagers is by buying and selling the outcome's *shares*. In this section, we first discuss market making, then delve into the details of the Buy and Sell transactions.

A. Market Maker

Real-world prediction markets have often had problems with liquidity [10]. Liquidity is an issue for prediction markets, because, generally, a market's forecasts are more accurate the more liquid the market is [11]. To avoid the liquidity issues that arise from simple order books, we instead use the *logarithmic market scoring rule* (LMSR) [12, 13].

The LMSR is the *market maker* for the entire market: all buy and sell orders are routed through the LMSR, rather than matching buy and sell orders between traders. The key feature of the LMSR is a *cost function* (C, in units of Bitcoin, or BTC), which varies according to the number of shares purchased for each possible outcome:

$$C\left(q_1, q_2, \ldots, q_N\right) = \ell \log \left(\sum_{j=1}^{N} \mathrm{e}^{q_j/\ell} \right), \tag{2}$$

where q_j denotes the number of shares for outcome j, N is the total number of possible outcomes, and ℓ is the loss limit, which is determined by the market's creator. When the q_j's are all zero,

$$\sum_{j=1}^{N} \mathrm{e}^0 = N \implies C(0, 0, \ldots, 0) = \ell \log N. \tag{3}$$

This is the maximum possible loss.

The amounts paid by traders who buy and sell shares in the market are the *changes* in the cost function caused by increasing or decreasing the total number of shares. The cost to the user to buy x shares of outcome k is the difference between the cost of the new set of shares outstanding and the old:

$$C\left(q_1, q_2, \ldots, q_k + x, \ldots, q_N\right) - C\left(q_1, q_2, \ldots, q_k, \ldots, q_N\right)$$

If a user wants to sell x shares of outcome k, the cost to the user is the difference:

$$C\left(q_1, q_2, \ldots, q_k - x, \ldots, q_N\right) - C\left(q_1, q_2, \ldots, q_k, \ldots, q_N\right)$$

Since this difference is negative, the user receives Bitcoin from this transaction, in return for shares. The current price for outcome i ($p\left(q_i\right)$) is the derivative of Eq. 2:

$$p\left(q_i\right) = \frac{\mathrm{e}^{q_i/\ell}}{\sum_{j=1}^{N} \mathrm{e}^{q_j/\ell}}. \tag{4}$$

A more in-depth discussion of the LMSR, and the equivalence of prices and probabilities, is in Appendix C.

Special Cases

Eq. 2 covers events that can have arbitrary, categorical outcomes. Our other two event types are special cases of Eq. 2. *Binary* events are simply categorical events with only two possible outcomes. For binary events, the cost function simplifies to:

$$C_{\text{binary}}\left(q_1, q_2\right) = \ell \log \left(\mathrm{e}^{q_1/\ell} + \mathrm{e}^{q_2/\ell} \right), \tag{5}$$

If we restrict our attention to events with *scalar* outcomes, the exponential sum in Eq. 2 becomes an integral:

$$C_{\text{scalar}}\left(q(x)\right) = \ell \log \left(\int_a^b \mathrm{e}^{q(x)/\ell}\, dx \right), \tag{6}$$

where a and b are the lower and upper bounds on the scalar's value. Although the integral in Eq. 6 cannot be evaluated analytically for unknown $q(x)$, good model extraction and numerical approximation methods exist [14–17].

B. Buy and Sell Transactions

Now that Joe's new market is active, anyone can buy shares of it using Bitcoin. Another user, Paul, looks up

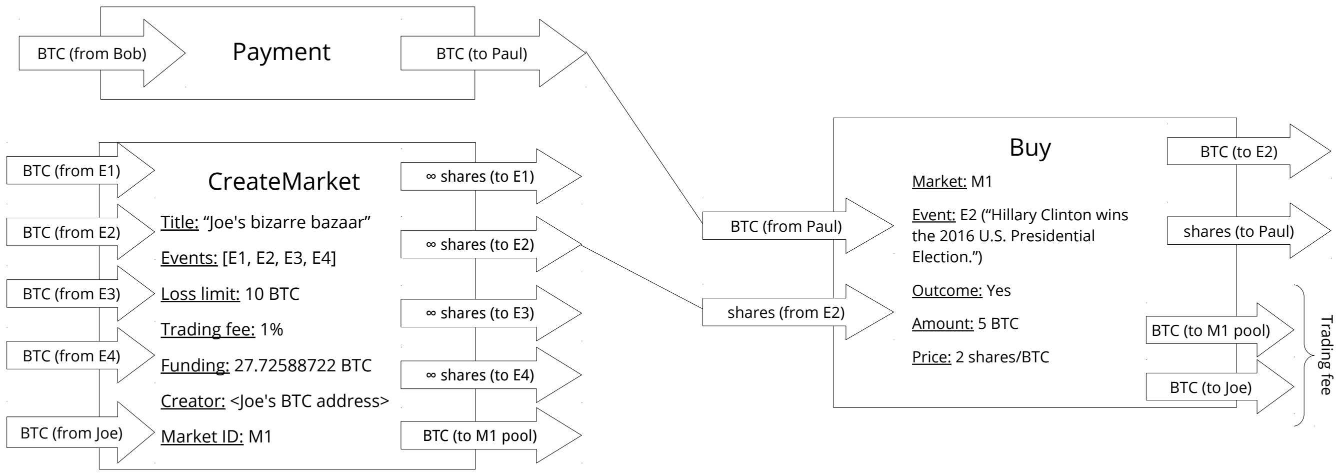

FIG. 3. Example of a Buy transaction. Here, Paul buys shares in the 'Yes' outcome of the Event 'Hillary Clinton wins the 2016 U.S. Presidential Election'. The shares used as an input to the transaction come from an unspent output belonging to event E_2.

the current price, which is 2 shares/Bitcoin, and creates a *Buy* transaction, trading 5 Bitcoin for 10 shares of Joe's event.

A Buy transaction is initiated by a Bitcoin owner. It has two inputs and two outputs, as shown in Fig. 3. A Buy transaction sends Bitcoin from the user's address to a specified event address, and shares of the event to the user's address. It contains the following information:

- Event: the event for which the user is buying shares.
- Outcome: the outcome of the event that the user is buying shares of.
- Amount: the number of shares to buy.
- Price: the price per share, calculated using the LMSR.

This is organized into a transaction as follows:

```
{
    "type": "Buy",
    "vin": [
        {
            "n": 0,
            "value": 5,
            "units": "bitcoin",
            "scriptSig": "<Paul's signature>
                          <Paul's public key>"
        },
        {
            "n": 1,
            "value": 10,
            "units": "shares",
            "outcome": true,
            "scriptSig": "<event ID>"
        }
    ],
    "vout": [
        {
            "n": 0,
            "value": 5,
            "units": "bitcoin",
            "script": "OP_DUP
                       OP_HASH160
                       <event ID>
                       OP_EQUALVERIFY
                       OP_MARKETCHECK"
        },
        {
            "n" : 1,
            "value" : 10,
            "units": "shares",
            "outcome": true,
            "script" : "OP_DUP
                        OP_HASH160
                        <Paul's hash-160>
                        OP_EQUALVERIFY
                        OP_CHECKSIG"
        }
    ]
}
```

Similarly, a Sell transaction sends unspent shares from the user back to the event, and Bitcoin from the event to the user:

```
{
    "type": "Sell",
    "vin": [
        {
            "n": 0,
            "value": 10,
            "units": "shares",
            "outcome": true,
            "scriptSig": "<Paul's signature>
                          <Paul's public key>"
        },
        {
            "n": 1,
            "value": 5,
                "units": "bitcoin",
                "scriptSig": "<market ID>
                             <market data>
                             <event data>"
            }
        ],
        "vout": [
            {
                "n": 0,
                "value": 5,
                "units": "shares",
                "outcome": true,
                "script": "OP_DUP
                          OP_HASH160
                          <event ID>
                          OP_EQUALVERIFY
                          OP_MARKETCHECK"
            },
            {
                "n" : 1,
                "value" : 10,
                "units": "bitcoin",
                "script" : "OP_DUP
                            OP_HASH160
                            <Paul's hash-160>
                            OP_EQUALVERIFY
                            OP_CHECKSIG"
            }
        ]
    }
```

Buy and Sell transactions are *atomic*, in the sense that either the entire transaction succeeds, or the entire transaction fails. That is, it is impossible for Paul to send Bitcoin to the event address, and not receive shares in return. In traditional database terms, either the entire transaction is committed – broadcast to the network, included in a block, added to the blockchain – or it is rolled back. There is no way for only *some* of the information in the transaction to be written to the blockchain.

IV. AFTER THE EVENT: REPORTING

The *reporting* phase occurs *after* the event takes place.[9] In this phase, the event's outcome is easily determinable – to continue with our example of the U.S. Presidential Election, by Googling for the results of the Presidential election, after the election has finished.

A. Reporting

A Report transaction consists of:

- Outcomes: encrypted report that contains the sender's observations.
- Reputation: the sender's Reputation.

To prevent collusion, the contents of Augur reports must be kept secret. To achieve this, after the user inputs his/her observations, the Report is encrypted by his/her local Augur software. After it is encrypted, the Report transaction is broadcast to the network.

1. *Report Transaction*

Report data is stored as follows:

```
{
    "type": "Report",
    "vin": [
        {
            "n": 0,
            "value": 40,
            "units": "reputation",
            "scriptSig": "<Jane's signature>
                          <Jane's public key>"
        },
        {
            "n": 1,
            "value": 2,
            "units": "reputation",
            "scriptSig": "<Jane's signature>
                          <Jane's public key>"
        }
    ],
    "vout": [
        {
            "n": 0,
            "value": 42,
            "units": "reputation",
            "report": {
                "id": "<report hash>",
                "outcomes": "<encrypted>",
                "quorum": {
                    "matured": true,
                    "reported": 1,
                    "required": 2,
                    "met": false
                }
            },
            "script": "OP_DUP
                      OP_HASH160
                      <Jane's hash-160>
                      OP_EQUALVERIFY
                      OP_CHECKDATA
                      OP_CONSENSUS
                      OP_PCACHECK
                      OP_EQUALVERIFY"
        }
    ]
}
```

The Report ID is the hash-160 of the Report's data fields. Since Reputation is tradeable, it can be sent between users; here, the user (Jane) has two Reputation inputs into her Report transaction. One of them comes from her last Report's Redemption transaction (see below). The other, smaller, Reputation input is Reputation that was sent to her by a friend.

[9] Technically, reporting is allowed any time after the Market is created. However, reporting on an outcome prior to the event's occurrence would be a spectacularly unnecessary gamble!