Augur: a Decentralized, Open-Source Platform for Prediction Markets

Dr. Jack Peterson & Joseph Krug
www.augur.net

Augur is a trustless, decentralized platform for prediction markets. It is an extension of Bitcoin Core's source code which preserves as much of Bitcoin's proven code and security as possible. Each feature required for prediction markets is constructed from Bitcoin's input/output-style transactions.

I. INTRODUCTION

A prediction market is a place where individuals can wager on the outcomes of future events. Those who forecast the outcome correctly win money, and if they forecast incorrectly, they lose money. People value money, so they are incentivized to forecast such outcomes as accurately as they can, so the price of a prediction market can serve as an excellent indicator of how likely an event is to occur [1, 2]. *Augur* is a decentralized platform for prediction markets.

A. Why decentralize?

Historically, the actions required of a prediction market – accepting wagers, deciding on the outcome of an event, then paying the wagers back out according to the results – have been centralized. The simplest approach to aggregating wagers is to have a trustworthy entity maintain a edger. The simplest way to determine the outcome of an event is to get the answer from a wise, impartial judge, whom all participants in the market trust.

Upon closer inspection, these straightforward answers fray at the edges: who is this 'someone' who is trustworthy enough to maintain a ledger for everyone else? Who is the judge that every participant trusts? And, even if such paragons were found and agreed on, how could participants be certain they will *remain* trustworthy once they are granted more power? "Opportunity makes the thief", after all. And, of course, "Power corrupts. Absolute power corrupts absolutely." [3]

In practice, a larger issue is that these trusted entities represent single points of failure. Prediction markets are often disliked by powerful interests. As the experience of centralized prediction markets – such as InTrade and TradeSports – over the past decade has shown, if governments or special interest groups want to shut down a website, they will find a way: InTrade, after all, was an Irish company, shut down as a result of the U.S. Commodity Futures Trading Commissions actions [4, 5]. Even the Defense Advanced Research Project Agency and Central Intelligence Agency were forced to end their foray into prediction markets as a result of Congressional interference [6].

Contrast this with the experience of Bitcoin [7]. Bitcoin is a cryptographic currency and payment platform that has found an enemy in powerful nation-states and financial institutions. Nevertheless, Bitcoin has thrived: as of November 2014, it has a market capitalization of over five billion U.S. dollars, and it is the anchor of a thriving ecosystem of startups, trade, and technological innovation. This is possible because Bitcoin is decentralized: once it was released, it could not be shut down. Even its pseudonymous creator, Satoshi Nakamoto, cannot stop the cryptocurrency. This is by design: he (or she) purposely did not make himself Bitcoin's single point of failure.

Augur is a decentralized prediction market platform. The Augur Project's goal is to revolutionize prediction markets, and, in doing so, change the way that people receive and verify 'truth'.

B. Strategy

Augur is built as an extension to the source code of Bitcoin Core.[1] It includes the features – the betting and consensus mechanisms – required by prediction markets. However, it is intentionally the same as Bitcoin in all other respects, to capitalize on Bitcoin's security and scalability. Our intention is to use the 'pegged sidechain' mechanism to make Augur fully interoperable with Bitcoin [8]; if this technology is not available, we will instead use our own internal currency as the store of value.

Our goal here is to provide a blueprint of a decentralized prediction market using Bitcoin's input/output-style transactions. Many theoretical details of this project are touched on lightly or not at all. The theoretical foundation of this project – in particular, the distributed consensus algorithm, and the game-theoretic underpinning – is explored in depth in [9].

C. Tokens

Augur has three types of tokens or *units*. To keep track of these units, every transaction input and output `value` fields is accompanied by a `units` field.

Although there are three types of tokens, users have a single cryptographic private key.[2] Augur addresses are

[1] Alternatives, such as smart contract-based implementations, are discussed in Appendix B.

[2] Augur uses the same public/private key cryptography system as

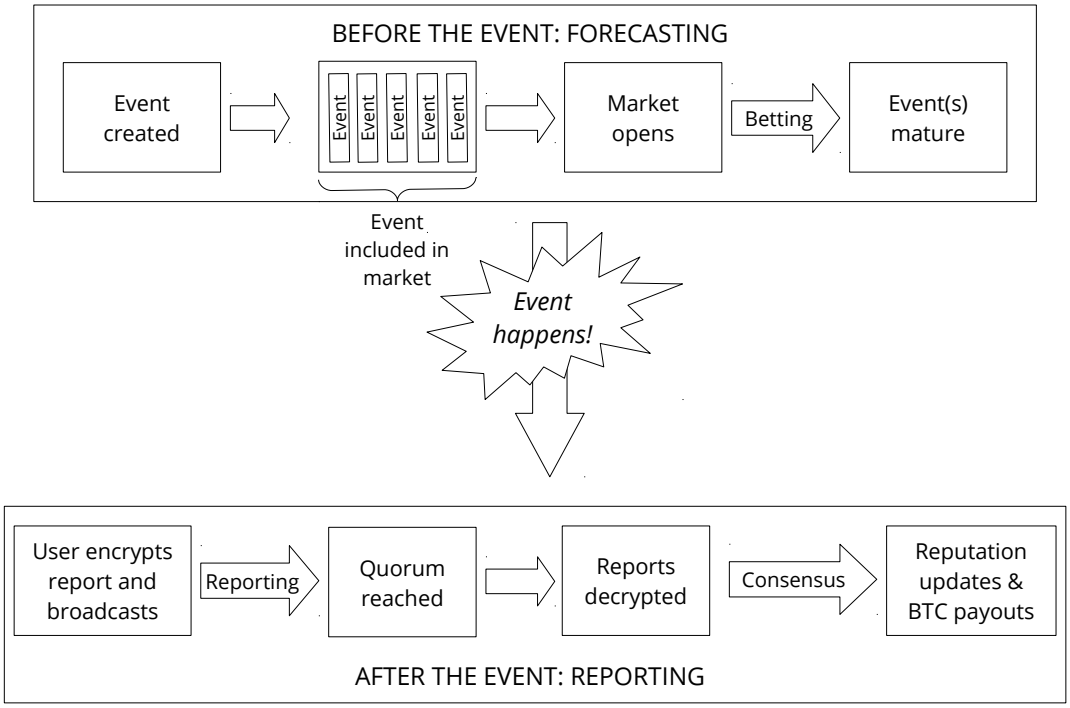

FIG. 1. Simplified outline of the actions in a prediction market, separated into before and after the event.

simply base-58 hashes of their cryptographic public keys, to which users can send tokens. The different types of token are distinguished from one another solely through the 'units' field. Therefore, users sign the transaction data of a Reputation payment in exactly the same way as they would sign a Bitcoin payment.

The first token is Bitcoin. *Sidechains* allow us to transfer Bitcoin by creating a transaction on the Bitcoin blockchain that locks up the Bitcoin in an address [8]. Then a transaction is made on our blockchain with an input that has a cryptographic proof that the lock was made on the Bitcoin blockchain along with the genesis blockhash of its originating blockchain. A simplified payment verification (SPV) scheme is used to verify these locks. A SPV has a list of block headers for proof-of-work and cryptographic proof that the locking transaction was created on one of the blocks in the header list.

A key feature of Augur is tradeable *Reputation*. The total amount of Reputation is a fixed quantity, determined upon the launch of Augur. Holding Reputation entitles its owner to report on the outcomes of events, after the events occur. Reputation tokens are similar in other respects to Bitcoins: they are divisible to eight decimal places, they are accounted for by summing over unspent transaction outputs, and they can be sent between users.

A significant way that Reputation differs from Bitcoin is that Reputation is not intended to be a stable source of value: Reputation tokens are gained and lost depending on how reliably their owner votes with the consensus. Reputation holders are obligated to cast a vote every time the network 'checks in' with reality (by default, this is every 8 weeks). Another significant difference is that Reputation is not mined. Instead, Reputation will be distributed by means of an auction, as proposed in [9]. This solves two problems. First, it will provide funding to complete the development of the Augur network. Second, it bootstraps the network with an initial group of Reputation-holders who are invested in the network's success.

Using a Bitcoin sidechain is a straightforward way to purchase shares. Users will simply transfer Bitcoins to the Augur network and use them to buy shares of a prediction. However, this poses a problem. If two people want to make a wager on whether event X will occur within a year from now, they would be exposed to the price volatility of Bitcoin. A potential solution is a seigniorage modeled coin. This would create a new cryptocurrency, say 'USDCoin', and put them on a few exchanges to be traded. If the price were above one dollar, the coin algorithm would print more coins; if below

Bitcoin: private keys are random 256-bit numbers, with a valid range governed by the secp256k1 ECDSA standard.

a dollar, the algorithm would discontinue printing. This allows the price of the coin to stay relatively close to a dollar.[3]

Another pertinent question is how to distribute the coins used to buy shares of predictions. A few options are available: 1) having a Bitcoin sidechain be the mechanism to buy shares, 2) using a seigniorage model to create a 'Cashcoin' or 'USDcoin' that maintains its value with respect to the US Dollar, and 3) using a hybrid approach of 1 and 2.

In our view, the best option is to sidechain Augur to Bitcoin, and have a new seigniorage based coin to allow users to both wager with Bitcoin, *and* another currency which would be used more for longer term wagers in which volatility would become an issue. This provides the benefits of allowing interoperability with Bitcoin holders as well as a currency suited for holding shares of predictions.

II. PREDICTION MARKET CREATION

The life of a prediction market can be separated into two principal 'phases' (Fig. 1): before and after the event. To walk through the life cycle of a typical prediction market, consider a user, Joe, who wants to wager on the outcome of the 2016 U.S. Presidential Election.

A. Event Creation

Joe turns on his Augur program, which plugs him directly into the peer-to-peer network, and clicks on the 'politics' group. This brings up a list of all active political prediction markets, along with their current prices. No one has created a market for the 2016 election yet, so Joe decides to take the initiative. He clicks on 'Create Event'. Here, he enters the necessary information about this event (Fig. 2):

- Description: a short description of the event.
- Type: binary (yes or no), scalar (numeric), or categorical
- Range of valid answers: yes/no/maybe for boolean, the minimum and maximum allowed values for scalar.
- Topic: the category the event falls into. Topics are created by users, and can be as fine-grained as users want them to be – examples might be 'politics', 'science', 'funny cat videos', 'serious cat videos', and so on.
- Fee: a small fee (on the order of 0.01 Bitcoin) required to create an event. Half of this fee goes to users who Report on the results of the event, after it happens (details in Section IV).
- Maturation: duration of the forecasting phase. This is the period where users buy and sell shares of the market. In Augur, time is marked by the block interval, which is expected to be constant, on average. Users have the option of entering a 'block number' (which is exact, but not intuitive) or an 'approximate end time' (which is not quite exact, since the block interval is not always *exactly* the same length, but is much more intuitive).
- Creator's address: the address of the event's creator.

CreateEvent Transaction

Joe pays a fee of 0.01 Bitcoin to create this event. As shown in Fig. 2, this is the CreateEvent transaction's only input. The CreateEvent transaction also has a single output, which contains the event's data.

The output's data includes the event's automatically assigned unique ID, which is a 160-bit hash[4] of the other event output fields. To prevent accidental duplicate event creation (due to network latency), the same user is not permitted to create the same event again, with the same expiration date and description. The value of this output is the user's event creation fee. The output can then be spent by including the event in a CreateMarket transaction (see below).

These inputs and outputs are combined into a Bitcoin-style transaction:[5]

```
{
    "type": "CreateEvent",
    "vin": [
        {
            "n": 0,
            "value": 0.01000000,
            "units": "bitcoin",
            "scriptSig": "<Joe's signature>
                          <Joe's public key>"
        }
    ],
    "vout": [
        {
```

[3] See Nubits for an example.

[4] SHA256 followed by RIPEMD160, usually referred to simply as 'hash-160'.

[5] Like Bitcoin, Augur's transactions are broadcast and stored in serialized form. A deserialized, rawtransaction-like JSON format is shown here for readability. For brevity, fields such as `locktime` and `sequence` are omitted from our examples when they are not directly relevant. Also, for Script fields (such as `scriptSig` and `scriptPubKey`) only the `asm` subfield – i.e., the input to `CScript()` – is shown.

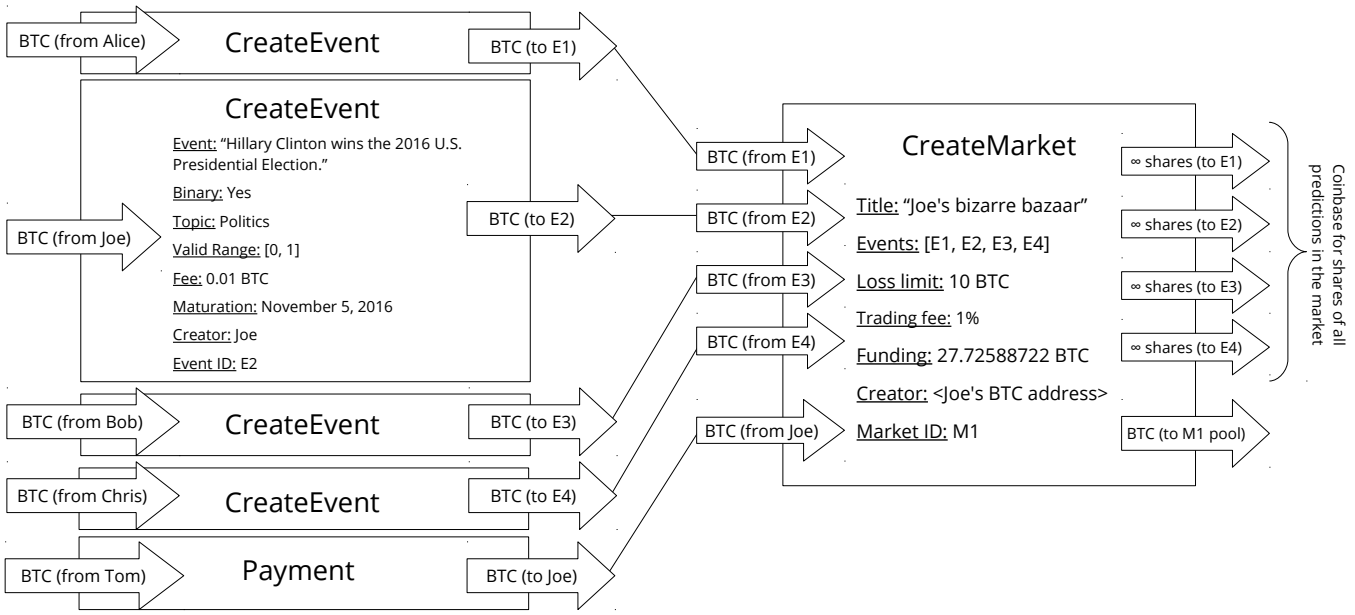

FIG. 2. CreateEvent transactions are payments from the user's address to a newly-generated event address. A CreateMarket transaction is a coinbase for shares of all the events contained in the market. Any user who has created one or more Events can incorporate their event(s) into a new market. In this simplified diagram, arrows represent inputs and outputs, and lines represent spending an unspent output. Note that events and markets have been given simplified IDs (E_1, E_2, etc.) here to make the figure easier to read; actual event and market IDs are 160-bit hashes of the event or market data fields, respectively.

```
            "n": 0,
            "value" : 0.01000000,
            "units": "bitcoin",
            "event": {
                "id": "<event hash>",
                "description": "Hillary Clinton
                    wins the 2016 U.S. Presidential
                     Election.",
                "branch": "politics",
                "is_binary": True,
                "valid_range": [0, 1],
                "expiration": 1478329200,
                "creator": "<Joe's address>"
            },
            "address": "<base-58 event ID>",
            "script": "OP_DUP
                       OP_HASH160
                       <event hash>
                       OP_EQUALVERIFY
                       OP_MARKETCHECK"
        }
    ]
}
```

The CreateEvent output's Script is keyed to the Event's ID:[6]

```
OP_DUP
OP_HASH160
<event hash>
OP_EQUALVERIFY
OP_MARKETCHECK
```

Since the event ID is the hash-160 of the Event's data, the CreateMarket transaction's matching input Script is:

```
<market hash>
<market data>
<event data>
```

The raw event data is supplied by the user to be included in a market, then is pushed onto the stack during the subsequent CreateMarket transaction. `OP_MARKETCHECK` first calculates the hash-160 of the market's fields (excluding the market ID), then verifies that this hash matches the actual market ID. The CreateEvent transaction's output is automatically sent to the event's *address*, which is a base-58 string derived from the event ID in the same way that Bitcoin addresses are derived from the public key's hash-160.

B. Market Creation

Once the event is created, he then clicks on 'Create Prediction Market', where he can include his new Event in a prediction market. Here he enters other information about the market:

- Title: label/brief description of the market.
- Events: list event IDs to be included in the market.

[6] Transaction Scripts are conventionally written left-to-right. However, we use a top-to-bottom format here for the sake of readability.

- 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:

$$\text{maximum possible loss} = \ell \times \log N_{\text{out}} \tag{1}$$

  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.[7] 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.

[7] 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.

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:

$$C_{\mathrm{binary}}\left(q_1, q_2\right) = \ell \log \left(\mathrm{e}^{q_1/\ell} + \mathrm{e}^{q_2/\ell} \right), \tag{4}$$

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

$$C_{\mathrm{scalar}}\left(q(x)\right) = \ell \log \left(\int_{a}^{b} \mathrm{e}^{q(x)/\ell}\, dx \right), \tag{5}$$

where a and b are the lower and upper bounds on the scalar's value. Although the integral in Eq. 5 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 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:

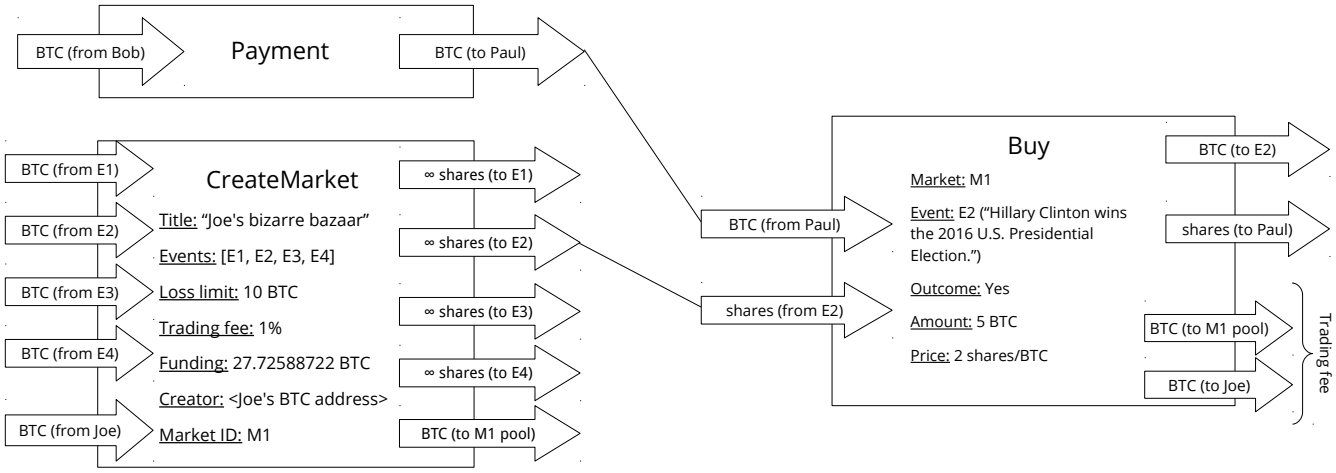

FIG. 3. Example of a Buy transaction. Here, Paul buys shares in the 'Yes' outcome of the Event 'Hillary Clinton wins the 2016 U.S. Presidential Election'. The shares used as an input to the transaction come from an unspent output belonging to event E_2.

- 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.[8] 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.

[8] 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!

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.

There are a few unusual things about the Report transaction. The first is the structured `quorum` field:

```
"quorum": {
    "matured": true,
    "reported": 0,
    "required": 2,
    "met": false
}
```

There are two requirements for *quorum* to be met:

Reputation: 42

Event	Your Report
Hillary Clinton will win the 2016 U.S. Presidential election.	NO
The unemployment rate will be lower at the end of 2017 than at the end of 2016.	YES
If Hillary Clinton is elected President in 2016, the unemployment rate will be lower at the end of 2017 than at the end of 2016.	YES
Hillary Clinton sucks!	INVALID

Submit Report

FIG. 4. Sample Report. Reputation owners report on the *actual outcome* of an event, after the event has occurred. Each 'Your Report' entry is initially set to `NO REPORT ENTERED`; if submitted that way, the user will not receive credit for reporting on that event. `INVALID` reports are permitted – in fact, encouraged! – for poorly-worded and/or indeterminate questions, since truth-by-consensus works well only for outcomes which are easily and objectively determinable. Since this user's Reputation is 42, his/her ballot has a *weight* of 42. Another user who has a Reputation of 105 would also only cast a single ballot, but his/her opinions would be given 2.5 times as much weight as the first user's.

1. The events being reported on must have passed their maturity time. Typically, if people are reporting on an event, this will be the case – events' maturity times should be set at or after the (expected) time of occurrence. In this example, the Report transaction has been made after the maturity time, as expected.

2. The minimum number of required reports must be met. In this simple example, there are only two reports required to meet quorum, zero of which have been provided so far.[9]

Once quorum is met, the market closes, and no more Bitcoin transactions are accepted. Report broadcasting continues for a pre-specified period of time. Upon completion of this phase, each Report is decrypted, and the Reports are gathered into a *report matrix*. This matrix has users as rows and events as columns; the values of the matrix are the users' observations.

The other unusual feature of the Report transaction is its output Script, which includes several new commands:

```
OP_DUP
OP_HASH160
<Jane's hash-160>
OP_EQUALVERIFY
OP_DATACHECK
OP_CONSENSUS
OP_PCACHECK
OP_EQUALVERIFY
```

The first, `OP_DATACHECK`, calculates the hash-160 of the Report's fields (excluding the report ID), then verifies that this hash matches the actual Report ID. This is to ensure that the transaction's contents were not tampered with during the delay from the time the Report was broadcast until the market closes.

The second new command, `OP_CONSENSUS`, initiates the consensus algorithm, which is described in detail in the following section. `OP_CONSENSUS` requires the report matrix as an input. The report matrix does not yet exist when the reports are first broadcast; instead, it is pieced together when the Reports are decrypted. Once the report matrix is assembled, it is written to the associated input's `scriptSig`.

`OP_PCACHECK` verifies that the outcome of the consensus algorithm is correct.[10] PCA is simpler to verify than to compute, as all that is needed to verify that the PCA results are correct is to multiply three matrices together. The final opcode, `OP_EQUALVERIFY`, compares the product with the original (centered) report matrix and ensures that they match.

[9] Not counting the current report, which has just been broadcast to the network, and is not yet included in a block.

[10] The consensus algorithm is a modified form of a common statistical technique called PCA (Principal Component Analysis) [18], hence the name `OP_PCACHECK`.

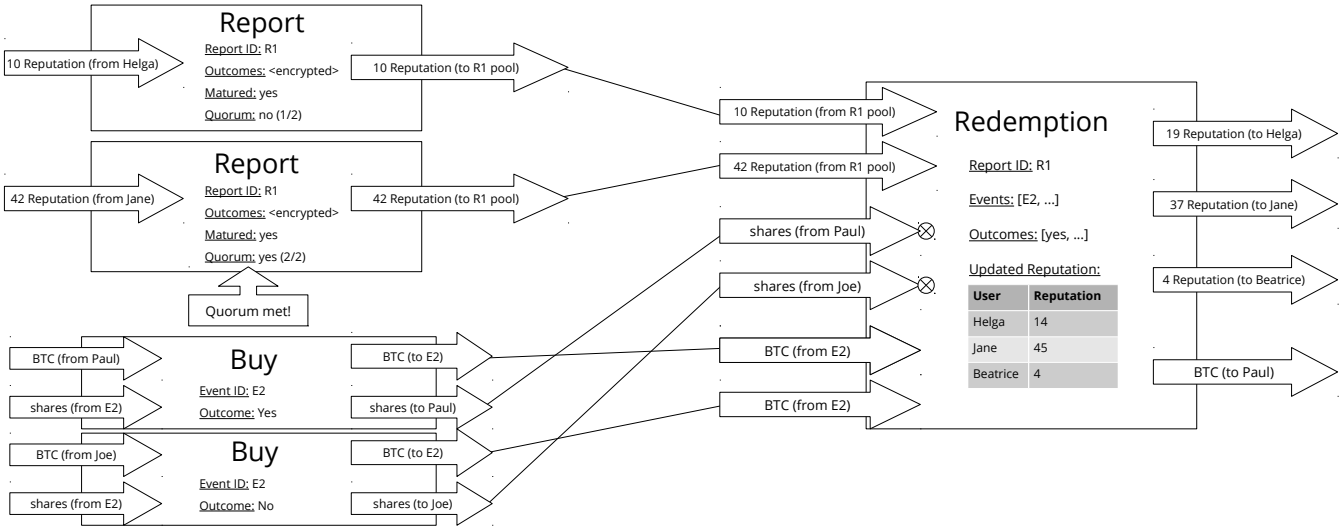

FIG. 5. Report and Redemption transactions. In this simple example, there are only 2 users reporting, Helga and Jane. (In reality, their names would not be known.) Helga broadcasts her report first, after the event has reached its maturity date. Since she is the first user to broadcast her report, only 1/2 users have reported, and quorum has not been reached, so the markets associated with the events listed on the ballot remain open. Later, Jane fills out her ballot and broadcasts it. At this point 2/2 users have reported, and quorum is reached. This automatically triggers the Redemption transaction, which makes use of special commands in the Report transactions' output Scripts to reach and verify consensus. Notice there are actually 3 users total; the third user, Beatrice, neglected to submit a Report, and therefore lost Reputation.

2. *Redemption Transaction*

The Report output's matching input Script is as follows:

```
<Jane's public key>
<report matrix>
<centered report matrix>
```

The *centered report matrix* is the report matrix with the per-column weighted mean subtracted from each column.

This Script is found in the *Redemption* transaction (Fig. 5), a special transaction which is initiated when quorum is reached:

```
{
    "type": "Redemption",
    "vin": [
        {
            "n": 0,
            "value": 10,
            "units": "reputation",
            "scriptSig": "<Helga's public key>
                         <report matrix>
                         <centered report matrix>"
        },
        {
            "n": 1,
            "value": 42,
            "units": "reputation",
            "scriptSig": "<Jane's public key>
                         <report matrix>
                         <centered report matrix>"
        },
        <all outstanding shares>,
        <all outstanding wagers>
    ],
    "vout": [
        <all reputation>,
        <all wagers>
    ]
}
```

The Redemption transaction is quite large: it has two inputs for every outstanding wager – one for Bitcoin, one for shares.[11] Reputation balances are updated using a modified version of the *Sztorc consensus algorithm* [9].[12] This algorithm's purpose is to reward users whose reports are consistent with the consensus, and punish those whose reports are not.

Per-share values are fixed according to the consensus-determined outcomes. The fixed share-price values are then used to determine the payout sent to all users holding shares in these events. Payout values are assembled in a matrix containing the events in columns, and the share owners' addresses in rows. When this payout matrix is complete, the market broadcasts it to the Augur network. Once it is incorporated into a block and added to the blockchain, the payouts appear in the recipients' accounts.

[11] Various hard-coded size limits on transaction and block size are lifted specifically for the Redemption transaction.

[12] Details of the consensus algorithm are shown in Appendix A.

B. Third Party APIs

As the network's popularity grows, people will be required to report on more events. Branches limit the number of things users will be required to report on; however, as individual branches become more popular, this problem returns. One solution is for wagers to be resolved by third party feeds.

This seems contrary to the spirit of our project at first, due to centralization. However, we propose using a variety of third party feeds and comparing their results. If a feed disagrees with another feed, a traditional vote is called. Additionally, any market participant can pay a small fee to call a traditional vote at any time. This provides the best of both worlds: people are not required to report all the time on simple, repetitive outcomes (such as sports games), but people are still required to report on murkier – and, we suspect, weightier – issues. If a malicious feed exists, or is suspected to exist, users can simply call a vote.

We propose that each peer that holds Reputation would periodically make calls to the third party APIs that deliver event outcomes on the branches to which their Reputation belongs. Then these results would be weighted according to the amount of reputation their respective wallets hold. We can weigh them by broadcasting a message with the weight and the result they claim to have received from the API, all wrapped up in a signed message. Then other peers can check that this signed message did, in fact, come from an address with a certain amount of Reputation that resulted in the respective weight. Provided a sufficient threshold of users agree – since these are API results this can be high, perhaps 95% – the outcome is decided. If the consensus is below this threshold, manual reporting is automatically invoked.

ACKNOWLEDGMENTS

The authors thank Vitalik Buterin, Zack Hess, Alan Lu, Joe Costello, Jeremy Gardner, Kinnard Hockenhull and Paul Sztorc for helpful discussions and feedback. Financial support for this project was provided by Joe Costello.

[1] C. Manski. Interpreting the predictions of prediction markets. *NBER Working Paper No. 10359*, 2004.

[2] J. Wolfers and E. Zitzewitz. Interpreting prediction market prices as probabilities. *NBER Working Paper No. 10359*, 2005.

[3] M. Felson and R.V. Clarke. Opportunity makes the thief: practical theory for crime prevention. *Police Research Series, Paper 98*, 1998.

[4] U.S. Commodity Futures Trading Commission. CFTC charges Ireland-based "prediction market" proprietors Intrade and TEN with violating the CFTC's off-exchange options trading ban and filing false forms with the CFTC. Nov. 26, 2012.

[5] M. Philips. What's behind the mysterious intrade shutdown? *Bloomberg Businessweek*, Mar. 11, 2013.

[6] P.F. Yeh. Using prediction markets to enhance us intelligence capabilities: a "standard & poors 500 index" for intelligence. 50, 2006.

[7] S. Nakamoto. Bitcoin: a peer-to-peer electronic cash system. *https://bitcoin.org/bitcoin.pdf*, 2008.

[8] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timon, and P. Wuille. Enabling blockchain innovations with pegged sidechains. *http://www.blockstream.com/sidechains.pdf*, 2014.

[9] P. Sztorc. Truthcoin: trustless, decentralized, censorship-proof, incentive-compatible, scalable cryptocurrency prediction marketplace. *https://github.com/psztorc/Truthcoin*, 2014.

[10] C. Slamka, B. Skiera, and M. Spann. Prediction market performance and market liquidity: a comparison of automated market makers. *IEEE Transactions on Engineering Management*, 60:169–185, 2013.

[11] D.M. Pennock, S. Lawrence, C.L. Giles, and F.A. Nielsen. The real power of artificial markets. *Science*, 291:987–988, 2001.

[12] R. Hanson. Logarithmic market scoring rules for modular combinatorial information aggregation. *Tech. Rep., George Mason University, Economics*, pages 1–12, 2002.

[13] R. Hanson. Combinatorial information market design. *Information Systems Frontiers*, 5:107–119, 2003.

[14] J. Shore and R. Johnson. Axiomatic derivation of the principle of maximum entropy and the principle of minimum cross-entropy. *IEEE Transactions on Information Theory*, 26(1):26–37, 1980.

[15] J. Skilling. Data analysis: the maximum entropy method. *Nature*, 309:748–749, 1984.

[16] S. Presse, J. Lee, and K.A. Dill. Extracting conformational memory from single-molecule kinetic data. *J. Phys. Chem. B*, 117:495–502, 2013.

[17] S. Presse, J. Peterson, J. Lee, P. Elms, J.L. MacCallum, S. Marqusee, C. Bustamante, and K. Dill. Single molecule conformational memory extraction: P5ab RNA hairpin. *J. Phys. Chem. B*, 118:6597–6603, 2014.

[18] J. Shlens. A tutorial on principal component analysis. *http://arxiv.org/abs/1404.1100*, 2009.

[19] G.R. Price. Extension of covariance selection mathematics. *Annals of Human Genetics*, 35:485–490, 1972.

[20] V. Buterin. Ethereum white paper: a next generation smart contract & decentralized application platform. *https://www.ethereum.org/pdfs/EthereumWhitePaper.pdf*, 2013.

[21] G. Wood. Ethereum: a secure decentralised generalised transaction ledger, proof of concept VI. *http://gavwood.com/paper.pdf*, 2014.

[22] J. Shore and R. Johnson. Properties of cross-entropy minimization. *IEEE Transactions on Information Theory*, 27:472, 1981.

[23] Principles of maximum entropy and maximum caliber in statistical physics. *Reviews of Modern Physics*, 85:1115–1141.

Appendix A: Consensus Algorithm

The algorithm outlined here is a modified version of the *Sztorc consensus algorithm*, originally presented in [9]. The heart of the algorithm is unchanged. In the original algorithm, the first eigenvector of the covariance matrix (the principal component) was solely used, regardless of the amount of variance it explained. Our modification is that a fixed *fraction of explained variance* is specified. This should make the rewards/punishments for reporting with/against the consensus more consistent across different Branches.

Suppose there are N total events being reported on. Let R be the total amount of Reputation, among all users, in the Branch which is being reported on:

$$R = \sum_{i=1}^{N} r_i. \tag{A1}$$

If the vector of all centered reports for event i is $\mathbf{c}_i$, then the centered report matrix is given by

$$\mathbf{C}^* = \mathbf{c}_i - \langle \mathbf{c}_i \rangle. \tag{A2}$$

The unbiased, weighted covariance matrix ($\mathbf{\Sigma}$) is [19]:

$$\mathbf{\Sigma} = \frac{R}{R^2 - \sum_{i=1}^{N} {r_i}^2} \sum_{i=1}^{N} r_i \left(\mathbf{c}_i - \langle \mathbf{c} \rangle\right)^T \left(\mathbf{c}_i - \langle \mathbf{c} \rangle\right), \tag{A3}$$

where T denotes the transpose operation, and $\langle \mathbf{c} \rangle$ is the weighted mean vector:

$$\langle \mathbf{c} \rangle = \frac{1}{R} \sum_{i=1}^{N} r_i \mathbf{c}_i. \tag{A4}$$

Each row (and column) in $\mathbf{\Sigma}$ corresponds to the variability *across* users, within a single event. Since there are N events, $\mathbf{\Sigma}$ is an $N \times N$ matrix.

Next, we diagonalize $\mathbf{\Sigma}$,

$$\mathbf{\Sigma} = \mathbf{S} \mathbf{\Lambda} \mathbf{S}^T, \tag{A5}$$

revealing $\mathbf{\Lambda}$, an $N \times N$ matrix with $\mathbf{\Sigma}$'s eigenvalues on its diagonal, and zeros elsewhere:

$$\mathbf{\Lambda} = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_N \end{bmatrix}. \tag{A6}$$

The eigenvalues are in descending order, $\lambda_j > \lambda_{j+1}$.

The eigenvectors of $\mathbf{\Sigma}$ are the columns of the similarity matrix, $\mathbf{S}$. The column of $\mathbf{S}$ associated with the largest eigenvalue (λ_1) is called the *principal component*, $\mathbf{s_1}$. This component $\mathbf{s_1}$ is a unit vector oriented in the direction of as much of the report matrix's variability as can be captured in a single dimension. This direction of maximum variability might be thought of as a hypothetical user who maximally contributed as much as possible to the variability in this particular set of reports.

The projection $\mathbf{p_1}$ of the centered report matrix ($\mathbf{C}$) onto the principal component $\mathbf{s_1}$ tells us how much each user's reports contributed to this direction of maximum variability. The same is true for $\mathbf{p_2}$: now the projection is onto the direction of next-largest-variability.

Our strategy to achieve consensus is as follows. The cumulative fraction of variance (α_k) explained by the k^{th} component is given by a standard formula from Principal Component Analysis [18],

$$\alpha_k = \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{j=1}^{N} \lambda_j}. \tag{A7}$$

Setting a fixed *variance threshold* (α) allows us to extract the number of components (n) needed to explain at least $\alpha \times 100\%$ of the report matrix's total variance. The *coordination vector* $\mathbf{v}$ is calculated by summing together the first n components:

$$\mathbf{v} = \sum_{i=1}^{n} \mathbf{s_i} \, H\left[\alpha - \alpha_i\right], \tag{A8}$$

where H is the discrete (Heaviside) step function.

The remainder of our consensus calculation, and Reputation redistribution calculation, is identical to the specification of [9].

Appendix B: Smart Contracts

A decentralized prediction market can be constructed in a relatively straightforward way using smart contracts, such as Ethereum [20, 21]. Due to the recent release of Serpent contracts on Counterparty, we are also testing an alternative implementation of Augur using Serpent. Counterparty's announcement frees us to use Bitcoin as the transactional currency on our platform, instead of having to use Ethereum's ether – this will also be remedied on Ethereum itself when sidechains are released.

We think smart contracts may provide an alternative method for implementing Augur. However, there are a few barriers related to security and cost. We are currently testing our contract implementation and will update the whitepaper with our results.

Appendix C: LMSR Derivation

Here we present a simple derivation of the logarithmic market scoring rule (LMSR), using the principle of Maximum Entropy [23].

Let q_i denote the number of shares outstanding of outcome i, and $P(q_i)$ denote the *probability* that there are q_i outstanding shares of outcome i.

The *entropy* of the market is given by the standard form [14, 22]:

$$S(\{q_i\}) = -\sum_i P(q_i) \log P(q_i), \tag{C1}$$

where the sum is over all possible outcomes in this market. Suppose there are two constraints on the entropy. The first is the average number of shares outstanding

$$\langle q \rangle = \sum_i q_i P(q_i), \tag{C2}$$

and the second is normalization:

$$\sum_i P(q_i) = 1. \tag{C3}$$

Therefore, the market's Lagrangian is $\Lambda = S - \langle q \rangle - 1$.

The optimization condition is given by setting the functional derivative of the Lagrangian ($\nabla_i \Lambda$) equal to zero,

$$\sum_i \nabla_i \log P(q_i) + 1 + \alpha + \beta \sum_i \nabla_i q_i = 0, \tag{C4}$$

where the multipliers α and β enforce normalization and constraint C2, respectively. ∇_i is a convenient shorthand notation for the functional derivative with respect to $P(q_i)$,

$$\nabla_i = \frac{\delta}{\delta P(q_i)}.$$

Since Eq. C4 is required to hold for all possible i, this becomes:

$$\log P(q_i) + 1 + \alpha + \beta q_i = 0. \tag{C5}$$

Using Eq. C3 to eliminate the normalization parameter $e^{-1-\alpha}$, reveals the Boltzmann distribution:

$$P(q_i) = \frac{e^{-\beta q_i}}{\sum_j e^{-\beta q_j}} = Q^{-1} e^{-\beta q_i}, \tag{C6}$$

where Q is the canonical partition function:

$$Q = \sum_i e^{-\beta q_i}. \tag{C7}$$

Using the LMSR's loss limit parameter to set the Lagrange multiplier β,

$$\beta = -\frac{1}{\ell}, \tag{C8}$$

we see that Eq. C6 is identical to the LMSR price function $p(q_i)$:

$$p(q_i) = Q^{-1} e^{q_i/\ell}. \tag{C9}$$

In other words, the *price* offered for shares of outcome i is equal to the *probability* that outcome i will occur.

The LMSR's cost function is analogous to another common thermodynamic function, the Helmholtz free energy:

$$\langle q \rangle + \ell S = -\ell \log Q = \ell \log \left(\sum_i e^{-\beta q_i} \right). \tag{C10}$$