Ethereum Oracle Standard for time-limited events

High level description:

This is a model how oracles can provide smart contracts with the outcome of an event. First the event is described by a set of attributes which are stored in a JSON object. A deterministic fingerprint (hash) of this object serves as an identifier. This identifier is used to reference this event on the blockchain. The oracle publishes the outcome/result of the event with a signature. Anyone can (as needed) put the signed outcome on the blockchain. The smart contact will evaluate the signature and then accept the signed result as input data for its functions.

Event description

The event description is served as a stringified JSON. The JSON object can have arbitrary properties, however some are required. Required properties are:

- **title**:
  A human readable short description of the event. Preferably in one interrogative clause.
- **description**:
  A human readable detailed description of the event containing all information to make the event unambiguous.
- **resolutionDate**:
  The date when the event takes place. After this date the oracle should publish the signed event result. If an event is postponed, the oracle may delay the publishing of the event result. The date is represented as a datetime string following the ISO 8601 standard combining date and time in UTC. One example is 2015-12-15T07:36:25Z.

There are two different types of events:

- **Discrete events**:
  Events with outcomes that are chosen from a group of outcomes. For example the winner of the World Cup is chosen from the group of participating teams. These include binary outcomes which resolve to either 0 (“no”) or 1 (“yes”).
- **Ranged events**:
  Events with a numeric value as a result. Foyr< example the Bitcoin price at date X is a ranged event resulting in a numeric price.

Depending on the event type either a list of outcomes for discrete events or a unit/decimals information for ranged events is required:

- **outcomes**:
  A list of possible distinct outcomes of the event. The probability of all outcomes together should be 100%. There always have to be at least 2 outcomes.
- **unit**:
  The unit in which the result of the event will be published. E.g. “MilliBit” is one possible unit of the event “What will be the Bitcoin price at date X?”
- **decimals**:
  Because solidity does not support decimals, results with decimals have to be multiplied to get an integer value. 10**decimals represents this multiplier.

A possible example JSON with all required properties for a ranged event is:

```
{
	"title": "What will be the Bitcoin price end 2015",
	"description": "What will be the Bitcoin price end 2015 according to Bitstamp.net.",
	"resolutionDate": "2015-12-31T24:00:00Z",
	"unit": "MilliBit",
	"decimals": 10
}
```

A possible example JSON with all required properties for a discrete event is:

```
{
	"title": "Who will win UEFA Champions League 2015/16?",
	"description": "Who will win UEFA Champions League 2015/16 according to http://www.uefa.com/uefachampionsleague",
	"resolutionDate": "2015-05-28T23:59:59Z",
	“outcomes”: [“Bayern Munich”, “Real Madrid”, “Barcelona”, “Others”]
}
```

JSON objects are not sorted. This causes a problem because the JSON needs to be hashed and signed by the Oracle and therefore the signed information has to be unambiguous. For this reason the oracle serves a stringified version of the JSON object. This representation is used to hash and sign the event description.

A stringified version of the previous ranged event would be:

```
‘{"title":"What will be the Bitcoin price end 2015","description":"What will be the Bitcoin price end 2015 according to Bitstamp.net.","resolutionDate":"2015-12-31T24:00:00Z","unit":"MilliBit","decimals":10}’
```

Event signing

An oracle has to publicly sign the event description it offers, to be able to verify on the blockchain that an oracle is committed to publish a signed result for the event. To sign an event, the event description JSON is hashed with the sha3 hash function:

event_hash = sha3(event_description_json)

Important: The sha3 implementation in ethereum is using a 256-bit length and is based on the Keccak implementation. Find more information here: http://legrandin.github.io/pycryptodome/Doc/3.3.1/Crypto.Hash.keccak-module.html.

The event_description_hash is concatenated with a fee charged by the oracle. This fee is represented by an integer and a fee token address. The concatenation is then hashed with the sha3 function again:
event_fee_hash = sha3(event_hash + fee_value + fee_token_address)

The event_description_fee_hash is then signed by the oracle using the Elliptic Curve Digital Signature Algorithm:

v, r, s = ecdsa_raw_sign(event_fee_hash, oracle_private_key)

This information is provided by the oracle as a JSON in the following format:

```
{
        “signature”: {
                “address”: “<oracle address>”,
                “v”: v,
                “r”: r,
                “s”: s
        },
        “fee”: Integer,
        “eventHash”: event_hash
}
```

Event result signing

The oracle has to sign the result in order to publish it. The signature process is very similar to the event signing. Instead of the fee, the result is concatenated with the event hash and hashed and signed by the oracle:

hex_result = format(result, 'x').zfill(64).decode('hex')
event_result_hash = sha3(event_hash + hex_result)
v, r, s = ecdsa_raw_sign(event_result_hash , oracle_private_key)

This information is provided by the oracle as a JSON in the following format:

```
{
        “signature”: {
                “address”: “<ethereum address>”,
```

```
            "v": v,
            "r": r,
            "s": s
        },
        "result": Integer,
        "eventHash": event_hash
}
```

Discrete events: The result is the index of the winning outcome in the list starting with 0. In case the first outcome in the list is winning, the result will be 0.
Ranged events: The result is the resulting numeric value of the event, multiplied with 10**decimals.

Oracle contracts

Prediction markets or other contracts using oracle services have to evaluate oracle results. Those results can be either off-chain in form of a signed result, which can be verified in a contract or in from of on-chain oracles, for example an oracle for block difficulty. Those contracts should implement the following four functions to be compatible:

- getFee(bytes32[] data) returns (uint fee, address token);
    - data: Array of values identifying an event. Based on this information oracle contracts can calculate a fee charged for resolving an event.
    - Returns fee: Oracle contracts return the fee amount charged for resolution.
    - Returns token: Oracle contracts return the token, which has to be used to pay fees.

- registerEvent(bytes32[] data) returns (bytes32 eventIdentifier);
    - data: Array of values identifying an event. Based on this information oracle contracts can decide if it can resolve this event.
    - Returns eventIdentifier: In case an oracle accepts an event, it returns an event identifier, which references the event in the oracle contract.

- isOutcomeSet(bytes32 eventIdentifier) constant returns (bool isSet);
    - eventIdentifier: The event identifier returned by the oracle can be used to retrieve the information if the outcome has been decided.
    - Returns isSet: Oracle returns if outcome has been decided.

- getOutcome(bytes32 eventIdentifier) constant returns (int outcome);
    - eventIdentifier: The event identifier returned by the oracle can be used to retrieve the information of the resolved outcome. It is either the index of the outcome in case of discrete events or the outcome as numeric value in case of a ranged event.

    - Returns outcome: Oracle returns the decided outcome.

- getEventData(bytes32 eventIdentifier) constant returns (bytes32[] data);
    - eventIdentifier: The event identifier returned by the oracle can be used to retrieve the information about the event.
    - Returns data: Data, which is used to resolve the event. E.g. off-chain-oracle addresses or a block-number.

Recommended additions to improve security of off-chain-oracles

The private key of an off-chain oracle could be compromised and an attacker could use stolen private key to sign event results of arbitrary events. This would be especially harmful in case of oracles used to resolve prediction markets with high trading volume. For this reason it is strongly recommended to offer an additional on-chain-oracle, which can be used in case of a key compromise. When registering a new event or resolving the outcome of an event using off-chain-oracles, the fallback on-chain-oracle should be consulted first. If the fallback on-chain-oracle indicates that a key was compromised (isValidSigner returns false) the event should not be registered and the off-chain result should be ignored and replaced with the result from the fallback on-chain-oracle. The fallback on-chain-oracle used for validation should have the following functions:

- isValidSigner(bytes32 eventIdentifier, address signer) returns (bool);
    - eventIdentifier: The event identifier returned by the oracle can be used to retrieve the information if the off-chain-oracle provider decided to switch the oracle to the on-chain-oracle.
    - signer: Signer address of the signed event description or result.
- isOutcomeSet(bytes32 eventIdentifier) returns (bool);
    - eventIdentifier: The event identifier returned by the oracle can be used to retrieve the information if the outcome has been decided.
    - Returns isSet: Oracle returns if outcome has been decided.
- getOutcome(bytes32 eventIdentifier) returns (int outcome);
    - eventIdentifier: The event identifier returned by the oracle can be used to retrieve the information of the resolved outcome. It is either the index of the outcome in case of discrete events or the outcome as numeric value in case of a ranged event.
    - Returns outcome: Oracle returns the decided outcome.

An event registration in a service using off-chain-oracles should fail if an oracle was compromised:

```
if (!fallbackOracle.isValidSigner(oracle)) throw;
```

An event resolution should use the fallbackOracle if the singer is not valid anymore:

```
if (!fallbackOracle.isValidSigner(oracle) && fallbackOracle.isOutcomeSet(eventIdentifier)) {
        result = fallbackOracle.getOutcome(eventIdentifier);
        // use on-chain result
}
else if (fallbackOracle.isValidSigner(oracle)) {
        // process off-chain result
}
```

The service using off-chain-oracle results should offer off-chain-oracles to register their fallback-oracle with them:

- registerFallbackOracle(address oracle) returns (bool);
    - oracle: Address of the fallback-oracle. The address is assigned to the message sender, which should be the off-chain-oracle.