EIP 792

The standard for Arbitra(tion/ble) smart contracts.

A Primer on Blockchains

They:

A Primer on Blockchains

They:

- Are distributed data stores that can be used to store any sort of data (e.g. Bitcoin stores balances for a bunch of addresses.).

A Primer on Blockchains

They:

- Are distributed data stores that can be used to store any sort of data (e.g. Bitcoin stores balances for a bunch of addresses.).
- Can be edited as long as certain rules are followed (e.g. No double spending, no spending of others' funds, etc.).

A Primer on Blockchains

They:

- Are distributed data stores that can be used to store any sort of data (e.g. Bitcoin stores balances for a bunch of addresses.).
- Can be edited as long as certain rules are followed (e.g. No double spending, no spending of others' funds, etc.).
- Guarantee provable data integrity, commonly through different consensus protocols based on `cryptography + economics = cryptoeconomics`.

Smart? Contracts

They:

Smart? Contracts

They:

- Are turing complete, stateful programs that get sent in a special deployment `TX` and become immutable.

Smart? Contracts

They:

- Are turing complete, stateful programs that get sent in a special deployment `TX` and become immutable.
- Have their own address to which you can send `TX`s with some input data and value and have some logic executed on-chain.

Smart? Contracts

They:

- Are turing complete, stateful programs that get sent in a special deployment `TX` and become immutable.
- Have their own address to which you can send `TX`s with some input data and value and have some logic executed on-chain.
- Guarantee provable data and state transition integrity through the chain they live on, in our case, Ethereum.

A Note on Smart Contract Design

A Note on Smart Contract Design

Traditional Back Ends:

- Do most of the heavy lifting for the front end, because computation is cheap and faster than on the client.

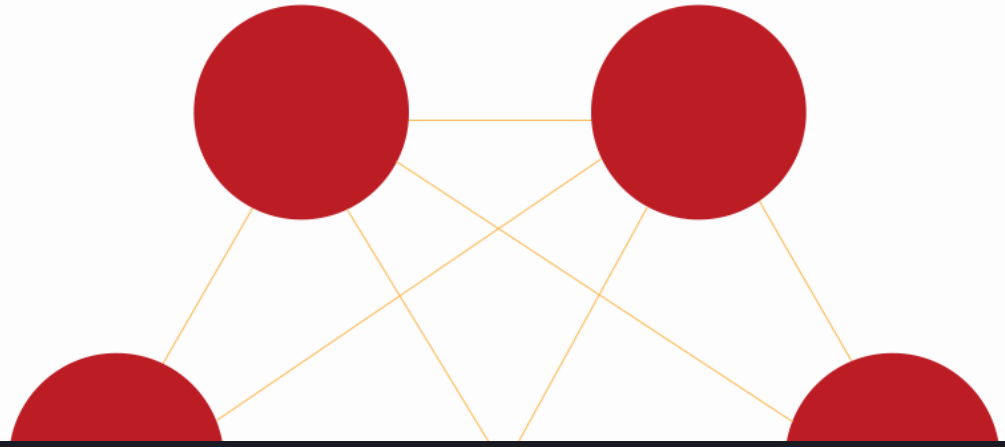
A Note on Smart Contract Design

Traditional Back Ends:

- Do most of the heavy lifting for the front end, because computation is cheap and faster than on the client.

Smart Contracts Back Ends:

- Leave most of the heavy lifting for the front end, because computation is expensive and slower than on the client.



EIP 792 Arbitrator: Interface

```
contract Arbitrator {
    function createDispute(
        uint _choices, bytes _extraData
    ) public requireArbitrationFee(_extraData) payable returns(uint disputeID) {};

    function arbitrationCost(bytes _extraData) public view returns(uint fee);

    function appeal(
        uint _disputeID, bytes _extraData
    ) public requireAppealFee(_disputeID,_extraData) payable {
        emit AppealDecision(_disputeID, Arbitrable(msg.sender));
    }

    function appealCost(uint _disputeID, bytes _extraData) public view returns(uint fee);

    function appealPeriod(uint _disputeID) public view returns(uint start, uint end) {}

    function disputeStatus(uint _disputeID) public view returns(DisputeStatus status);

    function currentRuling(uint _disputeID) public view returns(uint ruling);
}
```

EIP 792 Arbitrator: Types and Events

```
contract Arbitrator {
    enum DisputeStatus { Waiting, Appealable, Solved }

    event DisputeCreation(uint indexed _disputeID, Arbitrable indexed _arbitrable);

    event AppealPossible(uint indexed _disputeID, Arbitrable indexed _arbitrable);

    event AppealDecision(uint indexed _disputeID, Arbitrable indexed _arbitrable);
}
```

EIP 792 Arbitrable: Interface and Events

```
contract Arbitrable {
    event Ruling(Arbitrator indexed _arbitrator, uint indexed _disputeID, uint _ruling);

    function rule(uint _disputeID, uint _ruling) public onlyArbitrator;
}
```

Evidence Standard:

Link

Composed Arbitrable Contracts:

Link

EIP 792 Arbitrator: Simple Example

Centralized Arbitrator: Set Up

```
contract CentralizedArbitrator is Arbitrator {
    struct Dispute {
        Arbitrable arbitrated;
        uint choices;
        uint fee;
        uint ruling;
        DisputeStatus status;
    }

    address public owner = msg.sender;
    uint arbitrationPrice;
    uint constant NON_PAYABLE_VALUE = (2 ** 256 - 2) / 2;
    Dispute[] public disputes;

    modifier onlyOwner {require(msg.sender == owner, "Can only be called by the owner."); _;}

    constructor(uint _arbitrationPrice) public {arbitrationPrice = _arbitrationPrice;}
}
```

EIP 792 Arbitrator: Simple Example

Centralized Arbitrator: Setters and Getters

```
contract CentralizedArbitrator is Arbitrator {
    function setArbitrationPrice(uint _arbitrationPrice) public onlyOwner {
        arbitrationPrice = _arbitrationPrice;
    }
    function arbitrationCost(bytes _extraData) public view returns(uint fee) {
        return arbitrationPrice;
    }
    function appealCost(uint _disputeID, bytes _extraData) public view returns(uint fee) {
        return NON_PAYABLE_VALUE;
    }
    function disputeStatus(uint _disputeID) public view returns(DisputeStatus status) {
        return disputes[_disputeID].status;
    }
    function currentRuling(uint _disputeID) public view returns(uint ruling) {
        return disputes[_disputeID].ruling;
    }
}
```

EIP 792 Arbitrator: Simple Example

Centralized Arbitrator: Creating and Ruling Disputes

```
contract CentralizedArbitrator is Arbitrator {
    function createDispute(uint _choices, bytes _extraData) public payable returns(uint disputeID)  {
        super.createDispute(_choices, _extraData);
        disputeID = disputes.push(DisputeStruct({
            arbitrated: Arbitrable(msg.sender),
            choices: _choices,
            fee: msg.value,
            ruling: 0,
            status: DisputeStatus.Waiting
        })) - 1;
        emit DisputeCreation(disputeID, Arbitrable(msg.sender));
    }

    function giveRuling(uint _disputeID, uint _ruling) public onlyOwner {
        DisputeStruct dispute = disputes[_disputeID];
        require(_ruling <= dispute.choices, "Invalid ruling.");
        require(dispute.status != DisputeStatus.Solved, "The dispute has already been ruled.")

        dispute.ruling = _ruling;
        dispute.status = DisputeStatus.Solved;
        msg.sender.send(dispute.fee);
        dispute.arbitrated.rule(_disputeID, _ruling);
    }
}
```

EIP 792 Arbitrator: Complex Example

Kleros Liquid:

EIP 792 Arbitrator: Complex Example

Kleros Liquid:

- Has a tree shaped hierarchy of subcourts, where jurors can choose paths to participate in.

EIP 792 Arbitrator: Complex Example

Kleros Liquid:

- Has a tree shaped hierarchy of subcourts, where jurors can choose paths to participate in.
- Supports subcourt appeals and appeals to higher subcourts.

EIP 792 Arbitrator: Complex Example

Kleros Liquid:

- Has a tree shaped hierarchy of subcourts, where jurors can choose paths to participate in.
- Supports subcourt appeals and appeals to higher subcourts.
- Supports governance for global and subcourt specific settings like fees and time outs.

EIP 792 Arbitrator: Complex Example

Kleros Liquid:

- Has a tree shaped hierarchy of subcourts, where jurors can choose paths to participate in.
- Supports subcourt appeals and appeals to higher subcourts.
- Supports governance for global and subcourt specific settings like fees and time outs.
- Implements the evidence standard and another set of events designed to power a rich client notification system.

EIP 792 Arbitrator: Complex Example

Kleros Liquid:

- Has a tree shaped hierarchy of subcourts, where jurors can choose paths to participate in.
- Supports subcourt appeals and appeals to higher subcourts.
- Supports governance for global and subcourt specific settings like fees and time outs.
- Implements the evidence standard and another set of events designed to power a rich client notification system.

Link

EIP 792 Arbitrator: Other Examples

EIP 792 Arbitrator: Other Examples

- An arbitrator with a back up arbitrator in case it doesn't rule on time.

EIP 792 Arbitrator: Other Examples

- An arbitrator with a back up arbitrator in case it doesn't rule on time.
- An arbitrator that appeals to another arbitrator.

EIP 792 Arbitrator: Other Examples

- An arbitrator with a back up arbitrator in case it doesn't rule on time.
- An arbitrator that appeals to another arbitrator.
- An arbitrator that assigns weights to different arbitrators and computes an average ruling.

EIP 792 Arbitrator: Other Examples

- An arbitrator with a back up arbitrator in case it doesn't rule on time.
- An arbitrator that appeals to another arbitrator.
- An arbitrator that assigns weights to different arbitrators and computes an average ruling.
- Any sort of complex system or DAO that gives rulings.

EIP 792 Arbitrable: Simple Example

Two Party Arbitrable Escrow Payment: Storage and Modifiers

```
contract TwoPartyArbitrableEscrowPayment is Arbitrable {
    address public sender;
    address public receiver;
    uint public value;
    bytes public extraData;
    Arbitrator public arbitrator;
    uint public disputeID;
    bool public disputed;
    bool public appealed;
    bool public executed;
    uint public createdAt;
    uint public timeOut;

    modifier onlySenderOrReceiver{
        require(msg.sender == sender || msg.sender == receiver, "Can only be called by the sender or the receiver.");
        _;
    }
}
```

EIP 792 Arbitrable: Simple Example

Two Party Arbitrable Escrow Payment: Payment Lifecycle

```
contract TwoPartyArbitrableEscrowPayment is Arbitrable {
    constructor(address _receiver, bytes _extraData, Arbitrator _arbitrator, uint _timeOut, string _metaEvidence) public payable {
        sender = msg.sender;
        receiver = _receiver;
        value = msg.value;
        extraData = _extraData;
        arbitrator = _arbitrator;
        createdAt = now;
        timeOut = _timeOut;
        emit MetaEvidence(0, _metaEvidence);
    }
    function raiseDispute() public payable onlySenderOrReceiver {
        emit Dispute(arbitrator, arbitrator.createDispute.value(msg.value)(2, extraData), 0);
    }
    function submitEvidence(string _evidence) public onlySenderOrReceiver {
        require(disputed, "The payment has to be disputed.");
        require(!appealed, "The payment can not be appealed.");
        emit Evidence(arbitrator, disputeID, msg.sender, _evidence);
    }
    function appeal() public payable onlySenderOrReceiver {
        arbitrator.appeal.value(msg.value)(disputeID, extraData);
        if (!appealed) appealed = true;
    }
}
```

EIP 792 Arbitrable: Simple Example

Two Party Arbitrable Escrow Payment: Execution

```
contract TwoPartyArbitrableEscrowPayment is Arbitrable {
    function executePayment() public onlySenderOrReceiver {
        require(now - createdAt > timeOut, "The timeout time has not passed yet.");
        require(!disputed, "The payment is disputed.");
        require(!executed, "The payment was already executed.");
        executed = true;
        receiver.send(value);
    }
    function executeRuling(uint _disputeID, uint _ruling) internal {
        require(disputed, "The payment is not disputed.");
        require(_disputeID == disputeID, "Wrong dispute ID.");
        require(!executed, "The payment was already executed.");
        executed = true;
        if (_ruling == 2) receiver.send(value);
        else sender.send(value);
        emit Ruling(arbitrator, disputeID, _ruling);
    }
}
```

EIP 792 Arbitrable: Complex Example

Two Party Arbitrable Escrow Payment: New Features

EIP 792 Arbitrable: Complex Example

Two Party Arbitrable Escrow Payment: New Features

- Multiple sending and/or receiving parties.

EIP 792 Arbitrable: Complex Example

Two Party Arbitrable Escrow Payment: New Features

- Multiple sending and/or receiving parties.
- Shared and collaterized fees.

EIP 792 Arbitrable: Complex Example

Two Party Arbitrable Escrow Payment: New Features

- Multiple sending and/or receiving parties.
- Shared and collaterized fees.
- Support for insurance providers to cover fees and collect rewards.

EIP 792 Arbitrable: Complex Example

Two Party Arbitrable Escrow Payment: New Features

- Multiple sending and/or receiving parties.
- Shared and collaterized fees.
- Support for insurance providers to cover fees and collect rewards.
- Different evidence submission rules.

EIP 792 Arbitrable: Complex Example

Two Party Arbitrable Escrow Payment: New Features

- Multiple sending and/or receiving parties.
- Shared and collaterized fees.
- Support for insurance providers to cover fees and collect rewards.
- Different evidence submission rules.
- Multiple payments. I.e. have the contract store payments instead of it being the payment.

EIP 792 Arbitrable: Complex Example

Two Party Arbitrable Escrow Payment: Set Up

```
contract TwoPartyArbitrableEscrowPayment is MultiPartyInsurableArbitrableAgreementsBase {
    struct Payment {
        uint value;
        uint createdAt;
        uint timeOut;
    }
    event PaymentExecuted(bytes32 indexed _paymentID, address indexed _sender, address indexed _receiver, uint _value);
    mapping(bytes32 => Payment) public payments;
    constructor(
        Arbitrator _arbitrator,
        bytes _arbitratorExtraData,
        address _feeGovernor,
        uint _stake
    ) public MultiPartyInsurableArbitrableAgreementsBase(_arbitrator, _arbitratorExtraData, _feeGovernor, _stake) {}
}
```

EIP 792 Arbitrable: Complex Example

Two Party Arbitrable Escrow Payment: Creating Payments

```
contract TwoPartyArbitrableEscrowPayment is MultiPartyInsurableArbitrableAgreementsBase {
    function createPayment(
        bytes32 _paymentID, string _metaEvidence, address _to, uint _arbitrationFeesWaitingTime, Arbitrator _arbitrator, uint _timeOut
    ) external payable {
        require(msg.value > 0, "Payment must be more than zero.");
        address[] memory _parties = new address[](file:///home/travis/build/kleros/kleros-presentations/src/2);
        _parties[0] = msg.sender;
        _parties[1] = _to;
        _createAgreement(
            _paymentID,
            _metaEvidence,
            _parties,
            2,
            new bytes(0),
            _arbitrationFeesWaitingTime,
            _arbitrator
        );
        payments[_paymentID] = Payment({
            value: msg.value,
            createdAt: now,
            timeOut: _timeOut
        });
    }
}
```

EIP 792 Arbitrable: Complex Example

Two Party Arbitrable Escrow Payment: Payment Time Outs

```
contract TwoPartyArbitrableEscrowPayment is MultiPartyInsurableArbitrableAgreementsBase {
    function executePayment(bytes32 _paymentID) external {
        Agreement storage agreement = agreements[_paymentID];
        Payment storage payment = payments[_paymentID];
        require(agreement.creator != address(0), "The specified payment does not exist.");
        require(!agreement.executed, "The specified payment has already been executed.");
        require(!agreement.disputed, "The specified payment is disputed.");
        require(now - payment.createdAt > payment.timeOut, "The specified payment has not timed out yet.");
        agreement.parties[1].send(payment.value); // Avoid blocking.
        agreement.executed = true;
        emit PaymentExecuted(_paymentID, agreement.parties[0], agreement.parties[1], payment.value);
    }
}
```

EIP 792 Arbitrable: Complex Example

Two Party Arbitrable Escrow Payment: Ruling Execution

```
contract TwoPartyArbitrableEscrowPayment is MultiPartyInsurableArbitrableAgreementsBase {
    function executeAgreementRuling(bytes32 _agreementID, uint _ruling) internal {
        super.executeAgreementRuling(_agreementID, _ruling);
        Agreement storage agreement = agreements[_agreementID];
        PaidFees storage _paidFees = paidFees[_agreementID];
        Payment storage payment = payments[_agreementID];
        address _receiver;
        if (_paidFees.stake.length == 1) { // Failed to fund first round.
            // Send the value to whoever paid more.
            if (_paidFees.totalContributedPerSide[0][0] >= _paidFees.totalContributedPerSide[0][1]) _receiver = agreement.parties[0];
            else _receiver = agreement.parties[1];
        } else { // Failed to fund a later round.
            // Respect the ruling unless the losing side funded the appeal and the winning side paid less than expected.
            if (
                _paidFees.loserFullyFunded[_paidFees.loserFullyFunded.length - 1] &&
                _paidFees.totalContributedPerSide[_paidFees.totalContributedPerSide.length - 1][0] - _paidFees.stake[_paidFees.stake.length - 1]
            ) _receiver = agreement.parties[_ruling == 2 ? 0 : 1];
            else _receiver = agreement.parties[_ruling == 2 ? 1 : 0];
        }
        _receiver.send(payment.value); // Avoid blocking.
        agreement.executed = true;
        emit PaymentExecuted(_agreementID, agreement.parties[0], _receiver, payment.value);
    }
}
```

EIP 792 Arbitrable: Other Examples

EIP 792 Arbitrable: Other Examples

- Arbitrable contracts that binarize multiple choice disputes for binary arbitrators.

EIP 792 Arbitrable: Other Examples

- Arbitrable contracts that binarize multiple choice disputes for binary arbitrators.
- Linear oracles that recursively ask an arbitrator if the correct answer lies under or over the median of suggested values, until it reaches the desired precision.

EIP 792 Arbitrable: Other Examples

- Arbitrable contracts that binarize multiple choice disputes for binary arbitrators.
- Linear oracles that recursively ask an arbitrator if the correct answer lies under or over the median of suggested values, until it reaches the desired precision.
- Curated lists with a dispute challenge mechanism for registrations and clearings.

EIP 792 Arbitrable: Other Examples

- Arbitrable contracts that binarize multiple choice disputes for binary arbitrators.
- Linear oracles that recursively ask an arbitrator if the correct answer lies under or over the median of suggested values, until it reaches the desired precision.
- Curated lists with a dispute challenge mechanism for registrations and clearings.

See the `kleros-interaction` repo for more inspiration.

EIP 792

Now it's your turn.