Algorand Smart Contracts

Algorand Smart Contract Design and Features

September 27, 2022 Joe Polny

Algorand™
FOUNDATION

Tech Stack

- Algorand Virtual Machine (AVM)
  - Running on every node
  - Not compatible with Ethereum Virtual Machine
- Transaction Execution Approval Language
  - Assembly-like language for writing smart contracts
- PyTeal and beaker
  - Python library and framework for writing Algorand smart contracts
  - Ultimately compiles down to teal

Algorand Virtual Machine

- Available data
  - Transaction information (sender, fee, amount, etc.)
  - Global variables (current round, latest timestamp, etc.)
  - Application state
- AVM and TEAL are turing complete
- Constraints
  - Static fees mean we need to constrain execution in another way
  - Constraints are hardcoded into AVM to limit computational complexity

TEAL

```
#pragma version 6
byte "hello " // ["Hello "]
byte "world" // ["World", "Hello "]
concat // ["Hello World"]
log // [] "Hello World" will be logged on chain
int 1 // [1]
return
```

PyTeal

```python
return Seq(
  Log(Bytes("Hello World")),
  Approve()
)
```

Modes of Use

- Stateless - Smart Signature
    - Signs transactions conditionally based on smart contract logic
    - Delegated approval: sign transactions from any account that signs the logic
    - Contract account: sign transactions from contract-specific account
- Stateful - Applications
    - Saved state
    - Logging
    - Inner transactions

Application State

- Global state
    - 64 key/value pairs
    - Limited to 128 bytes per key/value pair
- Local state
    - 16 key/value pairs *per account*
    - Limited to 128 bytes per key/value pair
    - Accounts must opt-in
    - Can be cleared by end-user

Inner Transactions

- An application can send any transaction type
    - This includes application calls
- An application can send up to 16 transactions
    - Inner transactions are atomic with the outer transactions
    - One failure will cause all to fail
- Every application has its own contract address it can send transactions from

Logging

- Applications can log data during execution
- Logs are only saved upon completion
- Other applications can read logged data

Randomness

- Random numbers can be generated off-chain
- `vrf_verify` opcode can be used verify number on-chain
- Oracles can provide random numbers through smart contracts

Constraints

- Opcode budget
    - Every opcode has a cost proportional to computational complexity
    - Budget is pooled in grouped application calls
- State access
    - Caller must predefine what the smart contract will be accessing
        - accounts
        - applications
        - assets

App Call Anatomy

- App arrays
    - Defines what state can be accessed
- Arguments array
    - Arguments that can be read by the application
- OnComplete
    - Action to take upon execution of the logic

On Completions

OnComplete	Program	Action
NoOp	Approval	Nothing
OptIn	Approval	Allocates local state for sender
CloseOut	Approval	Clear local state of sender
ClearState	Clear	Clear local state of sender regardless of logic result
UpdateApplication	Approval	Updates the approval and clear programs
DeleteApplication	Approval	Deletes the application

App Creation Anatomy

- TEAL Programs
  - Approval program defines primary logic for application creation/calls
  - Clear program defines logic for clearing local application state
- Schema
  - Defines the number of key/value pairs that store integers or bytes
  - Defined for both global state and local state
  - Schema can not be updated

ARC-0004: ABI

- Standardizes encoding/decoding methods for types beyond Uint64 and Bytes
    - UintN, tuples, decimals, booleans, etc.
- Provides standard way of method calling
- JSON schema for defining available methods
- Logging for return values