Bitcoin Mechanics II
Ethereum & Smart Contracts I

Alexander Schönhuth

UNIVERSITÄT BIELEFELD
Faculty of Technology

Bielefeld University
June 8, 2022

Recap Lecture 6

- *Medical Blockchain: Motivation*
    - Situation, risks, goals
    - Attribute Based Encryption
    - Key Aggregate Cryptography
    - Cloud based solutions
- *Medical Blockchain: Overview*
    - Nodes and data
    - Access rights
    - Transactions
    - Block structure
- *Medical Blockchain: Elements*
    - Transaction types: details
    - Tokens & rewards
    - Election
- *Bitcoin Mechanics I*
    - Transactions in detail
    - Metadata, Input, Output

Bitcoin Scripts
–
Syntax

Bitcoin Scripts
–
Applications

Ethereum Introduction

Smart Contracts

Overview

Introduction

- *Bitcoin Scripts Syntax*
  - Introduction
  - Pay-to-PubKeyHash
  - Opcodes
  - Pay-to-ScriptHash
  - Multisig
- *Bitcoin Scripts Applications*
  - Escrow Transactions
  - Micro Payments
  - Lock Time
- *Ethereum Introduction*
  - Transition Function
  - Turing-Complete Cryptocurrency
  - Blockchain Layers; Ethereum Virtual Machine
- *Smart Contracts*
  - Definition
  - Accounts
  - Account State Transitions

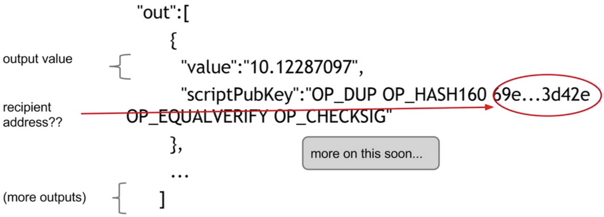
Bitcoin Scripts – Syntax

Bitcoin Scripts – Applications

Ethereum Introduction

Smart Contracts

Bitcoin Scripts: Introduction I

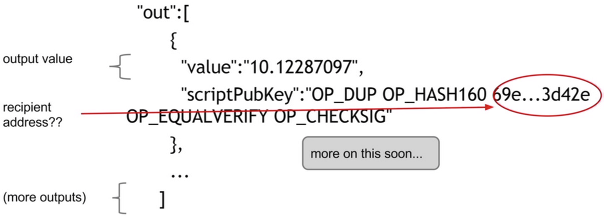


Transaction Output Syntax: Pay-to-PubkeyHash Script

From `bitcoinbook.cs.princeton.edu`

- Field specifying recipient(s) is a *script*
- Single elements (e.g. `OP_DUP`) are commands
- Run through interpreter, commands are executed

BITCOIN SCRIPTS: INTRODUCTION II

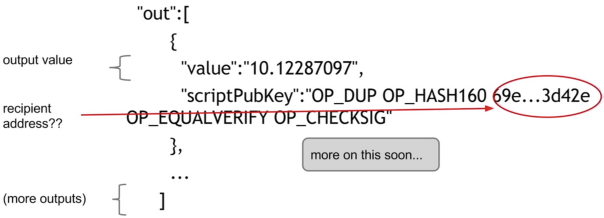


Transaction Output Syntax: Pay-to-PubkeyHash Script

From `bitcoinbook.cs.princeton.edu`

- Bitcoin specific; syntax adopted from scripting language *Forth*
- *Stack-based:* Commands executed in linear manner; *no looping!*
- Data is pushed onto stack

BITCOIN SCRIPTS: INTRODUCTION III

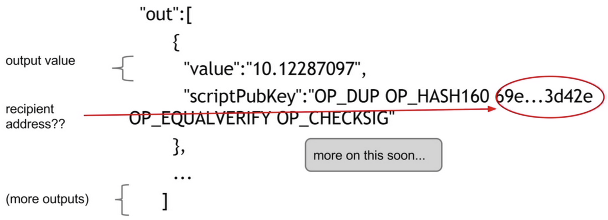


Transaction Output Syntax: Pay-to-PubkeyHash Script

From `bitcoinbook.cs.princeton.edu`

1. `OP_DUP`, then `OP_HASH160` are executed
2. Number `69e...3d42e` is pushed onto stack
3. `OP_EQUALVERIFY`, then `OP_CHECKSIG` are executed

Bitcoin Scripts: Introduction IV

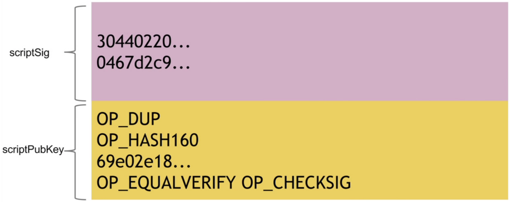


Transaction Output Syntax: Pay-to-PubkeyHash Script

From `bitcoinbook.cs.princeton.edu`

- Simple & compact; but limits on time / memory
- Support for cryptography
- *Here:* Checking whether earlier output agrees with later input

Bitcoin Scripts: Pay-to-PubkeyHash I

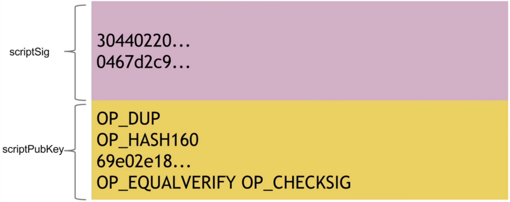


Connecting Input with Output

From `bitcoinbook.cs.princeton.edu`

- *scriptSig:* Input from current transaction
  - push `30440220...` → push `0467d2c9`
- *scriptPubKey:* Output from earlier transaction
  - `OP_DUP` → `OP_HASH160` → push `69e02e18...` → ...
    ... → `OP_EQUALVERIFY` → `OP_CHECKSIG`

Bitcoin Scripts: Pay-to-PubkeyHash II



Connecting Input with Output

From `bitcoinbook.cs.princeton.edu`

- *Validating transaction:*
  - Scripts executes without errors: include in your block
  - Executing script yields error: reject transaction
- Renders validating robust and convenient

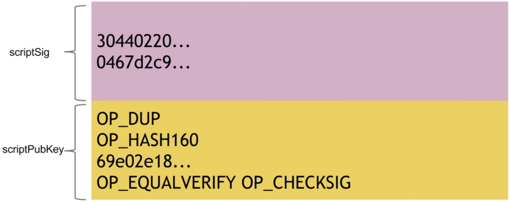
Bitcoin Scripts: Opcodes I

- Bitcoin scripting language is *small*
- Instructions referred to as *opcodes*
  - Room for 256 opcodes
    ☞ each one represented by one byte
  - Currently, 15 disabled, 75 reserved
    ☞ 166 in use
- Has basic arithmetic and basic logic
  - E.g. if-then logic
- Supports throwing errors and returning early

BITCOIN SCRIPTS: OPCODES II

- OP_DUP – Duplicates topmost item on stack
- OP_HASH160 – Replaces topmost item on stack by its hash
  - Hashes twice: first SHA-256, then RIPEMD-160
- OP_EQUALVERIFY – Returns true if two topmost elements agree
  - Marks transaction as invalid otherwise; stops executing script
- OP_CHECKSIG – Verifies signature:
  - Takes first (topmost) element of stack as public key
  - Takes second element as signature
  - Verification based on public key, signature and entire transaction
- OP_CHECKMULTISIG: True if k of specified signatures valid

Bitcoin Scripts: Pay-to-PubkeyHash (P2PKH)

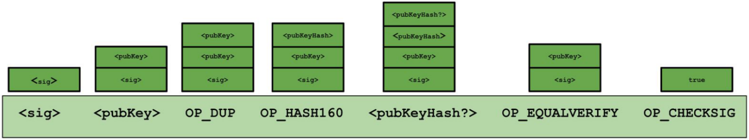


Connecting Input with Output

From `bitcoinbook.cs.princeton.edu`

- *In the following:*
  - Signature 30440220... denoted as `<sig>`
  - Public key 0467*d*2*c*9... denoted as `<pubKey>`
  - Hash of public key 69*e*02*e*18... denoted as `<pubKeyHash?>`

P2PKH: EXECUTION I

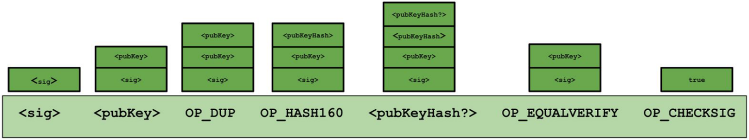


Pay-to-PubkeyHash Script Execution and Stack

From `bitcoinbook.cs.princeton.edu`

1. `<sig>` is the first number from *scriptSig* → pushed onto stack
2. `<pubKey>` is the second number from *scriptSig* → pushed onto `<sig>`
3. `OP_DUP` duplicates topmost `<pubKey>`
4. `OP_HASH160` replaces `<pubKey>` by its hash `<pubKeyHash>`

P2PKH: EXECUTION II



Pay-to-PubkeyHash Script Execution and Stack

From `bitcoinbook.cs.princeton.edu`

1. `<pubKeyHash?>` pushes data from *scriptPubKey* onto stack
2. `OP_EQUALVERIFY` compares `<pubKeyHash?>` with `<pubKeyHash>`
   - Script continues only if they agree
3. `OP_CHECKSIG` verifies signature
   - ”Consumes” `<pubKey>` and `<sig>` from stack
   - Pushes `<true>` only if signature valid
   - Throws error otherwise

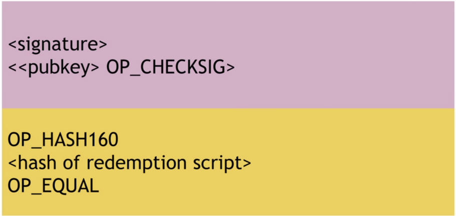
Bitcoin Scripts: Theory & Practice

- *Theory:*
  - Scripts can specify various conditions to spend coins
  - Whatever is possible through stack based arrangement
  - *However:* Scripting language is not Turing-complete
    ☞ We'll get to that later – in a lot more detail!
- *Practice I:*
  - 99.9% of scripts are of type "Pay-to-PubkeyHash (P2PKH)"
  - `MULTISIG` gets used a little bit
  - Pay-to-Script-Hash (P2SH) gets used a litle bit
- *Practice II:*
  - Many nodes maintain "white lists" of standard scripts
  - They refuse non-white-listed scripts
  - Usage of non-white-list scripts still possible, but harder

Bitcoin Scripts: Pay-to-Script-Hash (P2SH)

- *Situation:* Recipient wants to use ”fancy” script to redeem coins
- *Solution:* Recipient tells sender to send coins ...
  - ... not to hash of public key (see above)
  - ... but to hash of ”fancy” script
- P2SH has two parts:
  1. Hashes script provided in *scriptSig* provided by recipient and compares with hash of script provided by sender in *scriptPubKey*
  2. Re-interprets (”deserializes”) script in *scriptSig* and executes it
- *Advantage:* Tracking output scripts by miners
  - Keep track of unspent coins
  - Hashing scripts pushes complexity to input scripts

Bitcoin Scripts: Pay-to-Script-Hash II

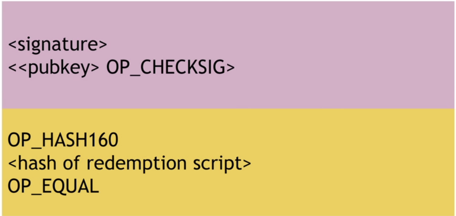


P2PKH as P2SH

From `bitcoinbook.cs.princeton.edu`

- Recipient provides in *scriptSig* (purple)
  - his signature
  - redemption script `<<pubKey> OP_CHECKSIG>`
- Sender provides output script in *scriptPubKey* (yellow)

Bitcoin Scripts: Pay-to-Script-Hash III

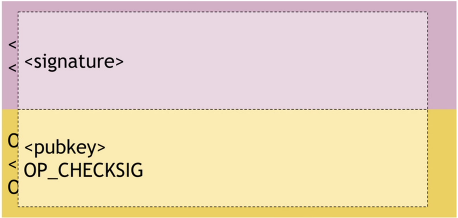


P2SH: Comparison Stage

From `bitcoinbook.cs.princeton.edu`

- *Comparison* – `<<pubKey> OP_CHECKSIG>` *taken as data:*
  1. Push `<signature>` onto stack
  2. Push `<<pubKey> OP_CHECKSIG>` onto stack (data!)
  3. `OP_HASH160` hashes data `<<pubKey> OP_CHECKSIG>`
  4. Push `<hash of redemption script>` onto stack
  5. `OP_EQUAL` compares the two topmost values

Bitcoin Scripts: Pay-to-Script-Hash IV



P2SH: Redemption Script Execution Stage

From bitcoinbook.cs.princeton.edu

- *Execution* – <<pubKey> OP_CHECKSIG> *taken as script:*
  1. Push <signature> onto stack
  2. Push <pubKey> onto stack
  3. Execute OP_CHECKSIG
- *Summary:* Both stages together simulate common P2PKH script

Bitcoin Scripts: Multisig Transactions I

- *Idea:* Create output that can be redeemed by specifying n public keys out of which m provide signatures
- Implementation requires two transactions ((i) & (ii))
  - (i) *MULTISIG transaction:*
    - Owner provides coins to be spent in *scriptSig* (as usual)
    - In *scriptPubKey*, owner specifies n public keys, and minimum number m of signatures
  - (ii) *Redemption transaction:*
    - m out of n public keys reach agreement (offline)
    - In *scriptSig*, they put their m public keys and their m signatures
    - In *scriptPubKey*, they specify the recipient

BITCOIN SCRIPTS: MULTISIG TRANSACTIONS II

- *Note:* Combining *scriptPubKey* of MULTISIG with *scriptSig* of redemption yields *Pay-to-Multisig-Script (P2MS)*
- *Another note:* P2MS can be performed using P2SH
- For *illustrations* (opcodes etc.) see e.g. `https://learnmeabitcoin.com/technical/p2ms`

Bitcoin Scripts
–
Syntax

Bitcoin Scripts
–
Applications

Ethereum Introduction

Smart Contracts

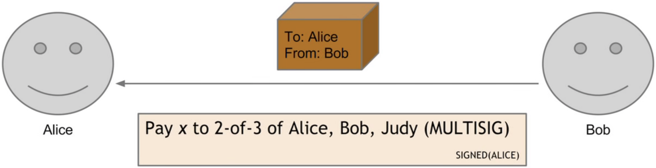
Script Applications

- *Escrow transactions:*
  - Alice wants to pay Bob for goods
  - Alice does not want to pay before having received goods
  - Bob does not want to send goods before having been paid
  - *Solution:* Introduce third party and perform escrow transaction
- *Micro payments:*
  - Alice wants to continually pay Bob small amounts
  - *Example:* Bob is Alice's phone provider; Alice needs to pay for every minute
  - Sending one transaction per minute costs Alice too many fees
  - *Idea:* Combine all small payments into one big payment at the end

Script Applications

- *Lock time:*
  - Alice releases MULTISIG transaction that never gets redeemed
  - *Example:* Escrow transaction never released by sufficiently many signatures
  - *Consequence:* Coins remain locked
  - *Solution:* Coins returned to Alice after some maximum lock time
- *Smart contracts:*
  - General term for contract-type transactions
  - Bitcoin scripts have limits
    ☞ They do not support Turing-complete language
  - *Idea:* Support running programs on blockchain
    ☞ Again: we will get to that in more detail!

Escrow Transactions I

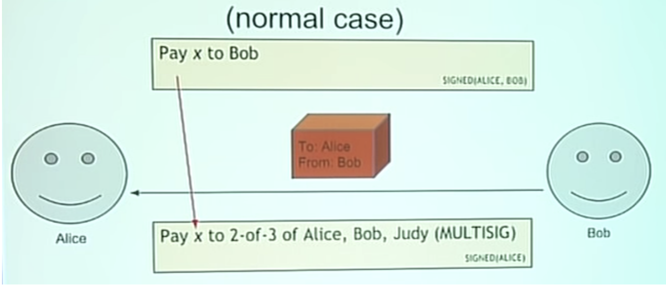


Escrow Transaction: 2-of-3 MULTISIG transaction

From `bitcoinbook.cs.princeton.edu`

- *Goal:* Alice pays Bob for services without anyone's damage
- *Idea:* Involve Judy, as a third-party arbitrator
- Alice launches MULTISIG transaction:
  - Spends x coins, price of Bob's services
  - 2 out of 3 signatures from Alice, Bob, Judy required

Escrow Transactions II

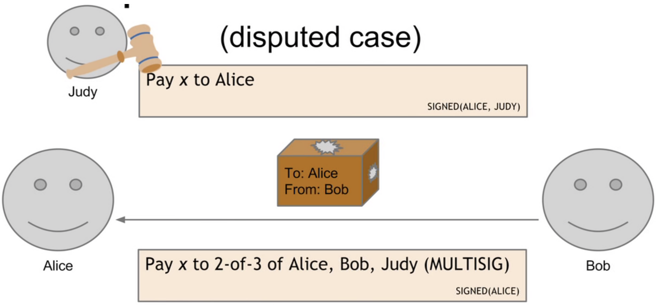


Merchandize Received OK: Alice & Bob Sign Redemption

From `bitcoinbook.cs.princeton.edu`

- *Scenario 1:* Bob's services all right, Alice happy to pay
- *Implementation:* Alice & Bob both sign redemption script
- *Result:* Bob gets paid x coins

Escrow Transactions III

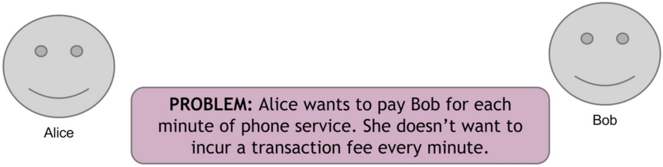


Merchandize Damaged: Alice & Judy Sign Redemption

From `bitcoinbook.cs.princeton.edu`

- *Scenario 2:* Bob's services insufficient, Alice does not intend to pay
- *Implementation:* Alice & Judy both sign redemption script
- *Result:* Alice gets her x coins returned

Micro Payments I

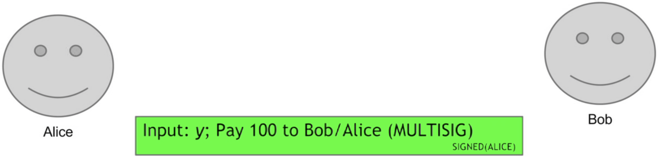


Micro Payments: Initial Scenario

From `bitcoinbook.cs.princeton.edu`

- *Situation:* Alice wants to pay Bob per unit of time of service
- *Example:* Bob runs phone service
  - Service needs to be paid per minute
  - Alice cannot anticipate length of call
- *Issue:* One transaction per minute incurs excessive fees

MICRO PAYMENTS II



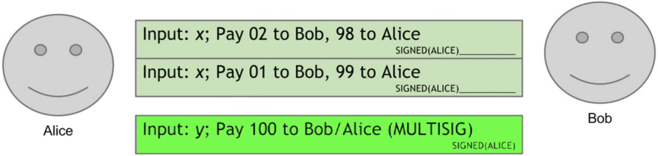
Alice Launches MULTISIG Transaction

From `bitcoinbook.cs.princeton.edu`

Solution Part I

- Alice launches MULTISIG transaction
- Specifies maximum amount to be paid for service (here: 100)
- 2-out-of-2 MULTISIG
- So both Alice and Bob need to sign redemption script

MICRO PAYMENTS III



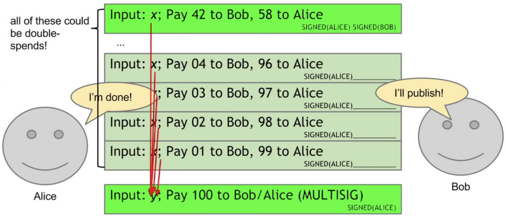
Alice Broadcasts Redemption Transactions Every Minute

From `bitcoinbook.cs.princeton.edu`

Solution Part III

- Alice broadcasts redemption transaction every minute
- Each one appropriately breaks down amounts to be paid
- As long as Alice keeps calling
  - Bob does not sign redemption scripts
  - Therefore, transactions not published on blockchain

Micro Payments IV



Alice Done Calling After 42 Minutes

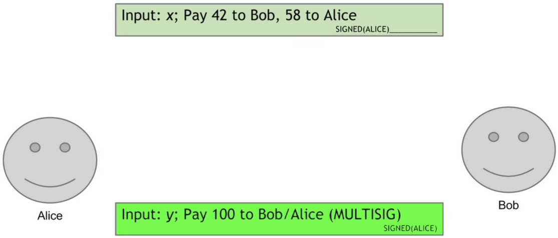
From `bitcoinbook.cs.princeton.edu`

Solution Part IV

- When Alice is done calling, Bob signs most recent transaction
- Alice done after 42 minutes: Bob receives 42, Alice 58 coins
- All redemption transactions possible double-spends
- *But:* Earlier transactions invalid; only last transaction gets published

UNIVERSITÄT BIELEFELD

Lock Time I

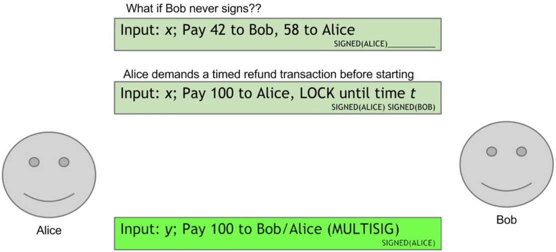


Issue: Bob Never Signs Redemption Script

From `bitcoinbook.cs.princeton.edu`

- *Possible Issue:* Bob never signs any redemption script
- *Consequence:* Coins (here: 100) remain in escrow; Alice unable to spend them otherwise

Lock Time II

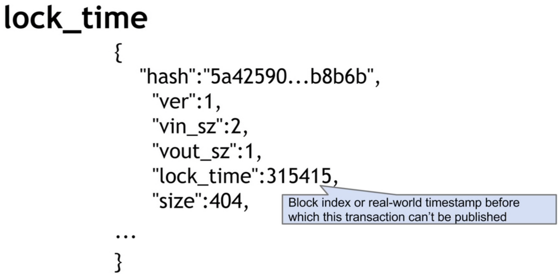


Alice and Bob sign Timed Refund Transaction

From `bitcoinbook.cs.princeton.edu`

- *Solution:* Alice and Bob sign timed refund transaction
- After time t, Alice gets full amount in return
- So Bob needs to hurry to sign redemption; otherwise no pay

Lock Time III



Lock Time Specified in Metadata

From `bitcoinbook.cs.princeton.edu`

- Lock time is specified in the metadata part of transaction
- Transaction cannot be published before

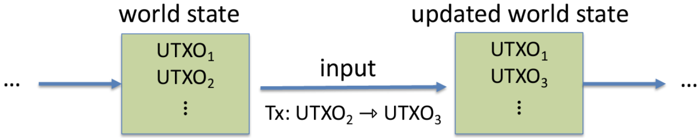
Bitcoin Scripts
–
Syntax

Bitcoin Scripts
–
Applications

Ethereum Introduction

Smart Contracts

BITCOIN: TRANSITION FUNCTION

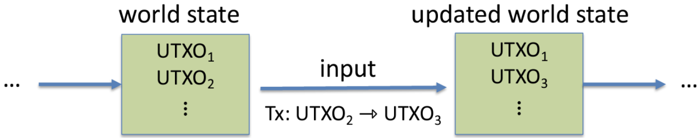


Bitcoin Blockchain as Sequence of States

From `cs251.stanford.edu`

- A *UTXO* is short for U nspent T ransa X ion O utput
- Keeping track of all UTXO's: tracking of Bitcoin ownerships
- *State:* All UTXO's at some point in time
- *Transition:* Executing transactions in one block

Bitcoin: Transition Function



State Transition: Performing Transactions

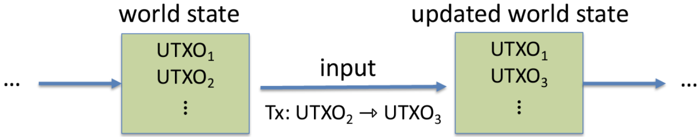
From `cs251.stanford.edu`

- Let S be all possible Bitcoin states; let s_0 be the genesis state
- Let I be all possible inputs ☞ An input is a set of transactions
- The *Bitcoin state transition function*

$$F_{BTC} : S \times I \longrightarrow S \tag{1}$$

maps state s to new state $F_{BTC}(s, i)$ when given input i

Ethereum: Motivation



State Transition: Performing Transactions

From `cs251.stanford.edu`

- $F_{BTC} : S \times I \rightarrow S$ has limits imposed by scripting language
- So, F_{BTC} not *Turing-complete*; e.g. no looping
- *Idea:* Implement *Turing-complete* transition function

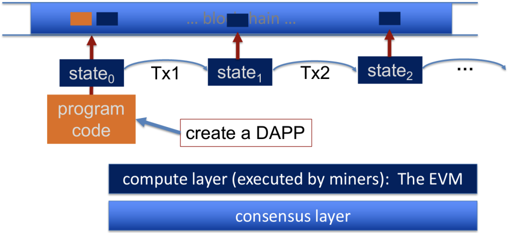
☞ ***Ethereum*** is a *"Turing-complete cryptocurrency"*

"Turing-Complete Cryptocurrency"

Things to Consider

- How to get computer programs onto/into blockchain?
- How to execute these programs?
  - Programs may have several different functionalities
  - Should be reusable, immutable etc.
- *Turing machines:* Infinite loops, halting problem?
- How to arrange states? Transaction based ledger?

Ethereum: Transition Function



Ethereum: Transition of States

From `cs251.stanford.edu`

- *DAPP:* "Decentralized Application"
- *EVM:* Ethereum Virtual Machine
- Blockchain records states; EVM performs transitions

Bitcoin Scripts – Syntax

Bitcoin Scripts – Applications

Ethereum Introduction

Smart Contracts

SMART CONTRACTS I

- *Motivation:* Nodes execute programs via transactions
- *Solution:* Make programs nodes in their own right
- *Explicit, Signed Types of Transactions:*
  - *User to User – Money Transfer:* Simple transfer of ether (ETH)
  - *User to Program – Deployment:* User releases (”deploys”) program
    ☞ Program becomes node
  - *User to Program – Execution:* User executes program functionality
    ☞ User interacts with program node
- *Implicit, Unsigned Types of Transactions:*
  - *Program to User:* Execution leads to money transfer
  - *Program to Program:* Execution leads to execution of other program

SMART CONTRACTS: DEFINITION

DEFINITION [SMART CONTRACT]: A *smart contract* is the program that gives rise to a program node.

Remarks:

- Turing-completeness implies that smart contracts can implement arbitrary functionality
- Smart contracts are supported by programming languages that take Turing-completeness into account
- *Example languages:* Solidity, Web3 (Python), Brownie

Smart Contracts: Example

```
contract NameRegistry {
      mapping(bytes32 => address) public registryTable;
      function claimName(bytes32 name) {
            if (msg.value < 10) {
                  throw;
            }
            if (registryTable[name] == 0) {
                  registryTable[name] = msg.sender;
            }
      }
}
```

From `bitcoinbook.cs.princeton.edu`

Transactions

- *Deploying:* Code writer node broadcasts code to network
- *Execution:* Node calls function `claimName[name]`
  - `name` is name of choice
  - Provided value `msg.value` must be sufficient, otherwise error
  - `RegistryName[name]` stores `msg.sender` (sender's public identity)

Ethereum Accounts

- *Issue:* Public key identities do not work for program nodes
- *Solution:* Nodes become *accounts*
- Accounts generalize concept of node
- *Types of Accounts:*
  - *Owned:* Ordinary user accounts; controlled by (S_k, P_k) key pair
  - *Contracts:* Program accounts; controlled by code
- *Account Data:*
  - *Owned:* Balance of account only
  - *Contracts:* Full spectrum of values assigned to variables

Smart Contract Account Data I

```
contract NameRegistry {
      mapping(bytes32 => address) public registryTable;
      function claimName(bytes32 name) {
            if (msg.value < 10) {
                  throw;
            }
            if (registryTable[name] == 0) {
                  registryTable[name] = msg.sender;
            }
      }
}
```

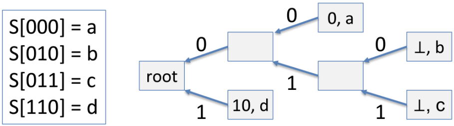
From `bitcoinbook.cs.princeton.edu`

Example: Contract Account Data

- *Creator Identity:* Stored as hash of public key of deploying node
- *Code:* Stored as code hash
- *Variables:* Store `nameRegistry`

Smart Contract Account Data II

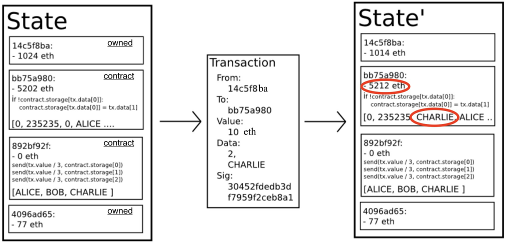
- *Issue:*
  - Transaction based ledger only works for owned accounts
  - Contract accounts require account based ledgers
  - "Fancy" data structures necessary for efficient transitions
- *Solution:*
  - Each contract maintains storage array S; entries hold 32 bytes
  - S can hold 2^{256} entries $S[i], i = 0, ..., 2^{256} - 1$ (in theory)
  - S arranged as *Merkle Patricia tree*



Account Merkle Patricia Tree

From `cs251.stanford.edu`

Account State Transitions



From `cs251.stanford.edu`

- Owned account `14c5f8ba` calls function of contract account `bb75a980`
- Provides money `msg.value` and input parameters
- Leads to adjustment of
  - Contract account balance
  - Values stored in Patricia Merkle tree

ETHEREUM: BLOCK OF TRANSACTIONS

0xa4ec1125ce9428ae5...	0x2cebe81fe0dcd220e...	0 Ether	0.00404405
0xba272f30459a119b2...	Uniswap V2: Router 2	0.14 Ether	0.00644563
0x4299d864bbda0fe32...	Uniswap V2: Router 2	89.839104111882671 Ether	0.00716578
0x4d1317a2a98cfea41...	0xc59f33af5f4a7c8647...	14.501 Ether	0.001239
0x29ecaa773f052d14e...	CryptoKitties: Core	0 Ether	0.00775543
0x63bb46461696416fa...	Uniswap V2: Router 2	0.203036474328481 Ether	0.00766728
0xde70238aef7a35abd...	Balancer: ETH/DOUGH...	0 Ether	0.00261582
0x69aca10fe1394d535f...	0x837d03aa7fc09b8be...	0 Ether	0.00259936
0xe2f5d180626d29e75...	Uniswap V2: Router 2	0 Ether	0.00665809

From `cs251.stanford.edu`

- *Columns 1-2:* Sender (`msg.sender` in contract) and recipient
- *Columns 3-4:* Money transferred (`msg.value` in contract) and transaction fees

Materials / Outlook

- See *Bitcoin and Cryptocurrency Technologies*, 3.2 & 3.3, 10.7
- See `cs251.stanford.edu`, Lecture 7
- See also
  - `https://bitcoinbook.cs.princeton.edu/`
  - `https://ethdocs.org/en/latest/index.html`
  - `https://ethereum.org/en/developers/docs/`

  for further resources
- Next lecture: "Ethereum Mechanics & Solidity"