Intro to Crypto and Cryptocurrencies

Slides by Arvind Narayanan et al.

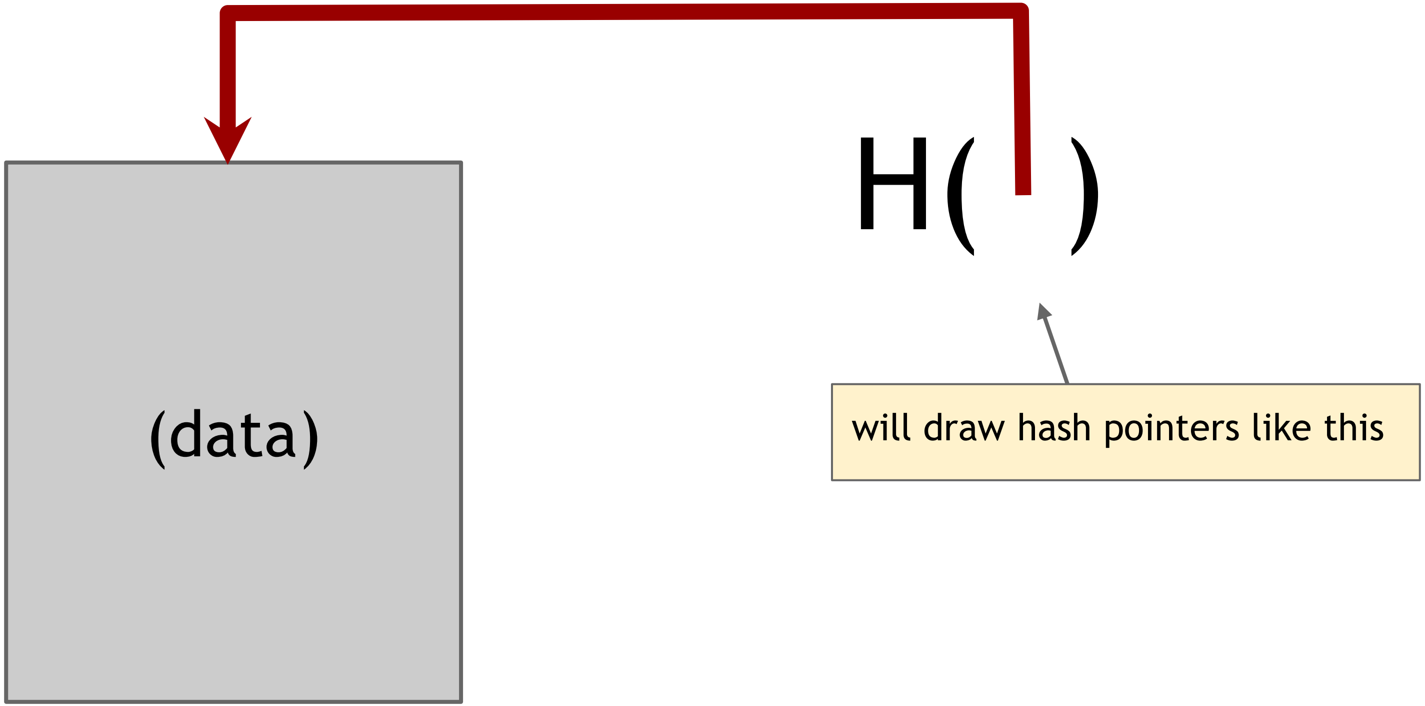
Hash Pointers and Data Structures

hash pointer is:

* pointer to where some info is stored, and
* (cryptographic) hash of the info

if we have a hash pointer, we can

* ask to get the info back, and
* verify that it hasn’t changed

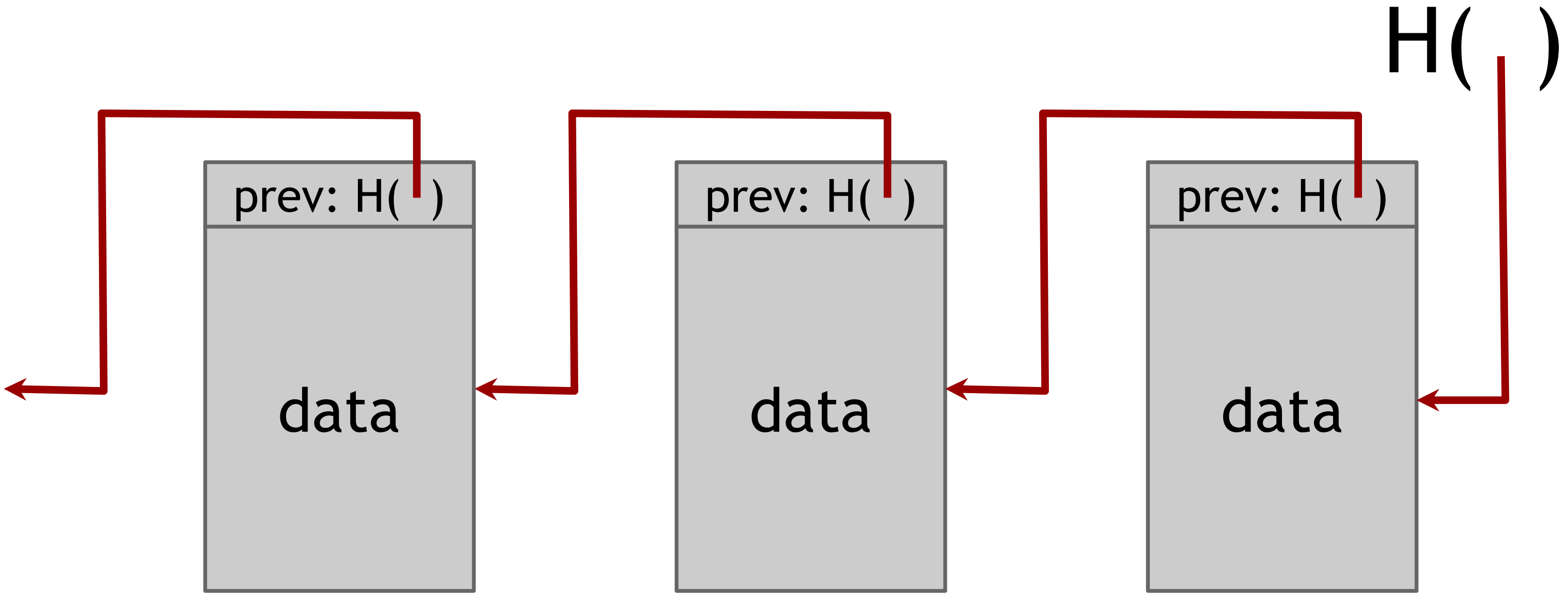


will draw hash pointers like this

key idea:

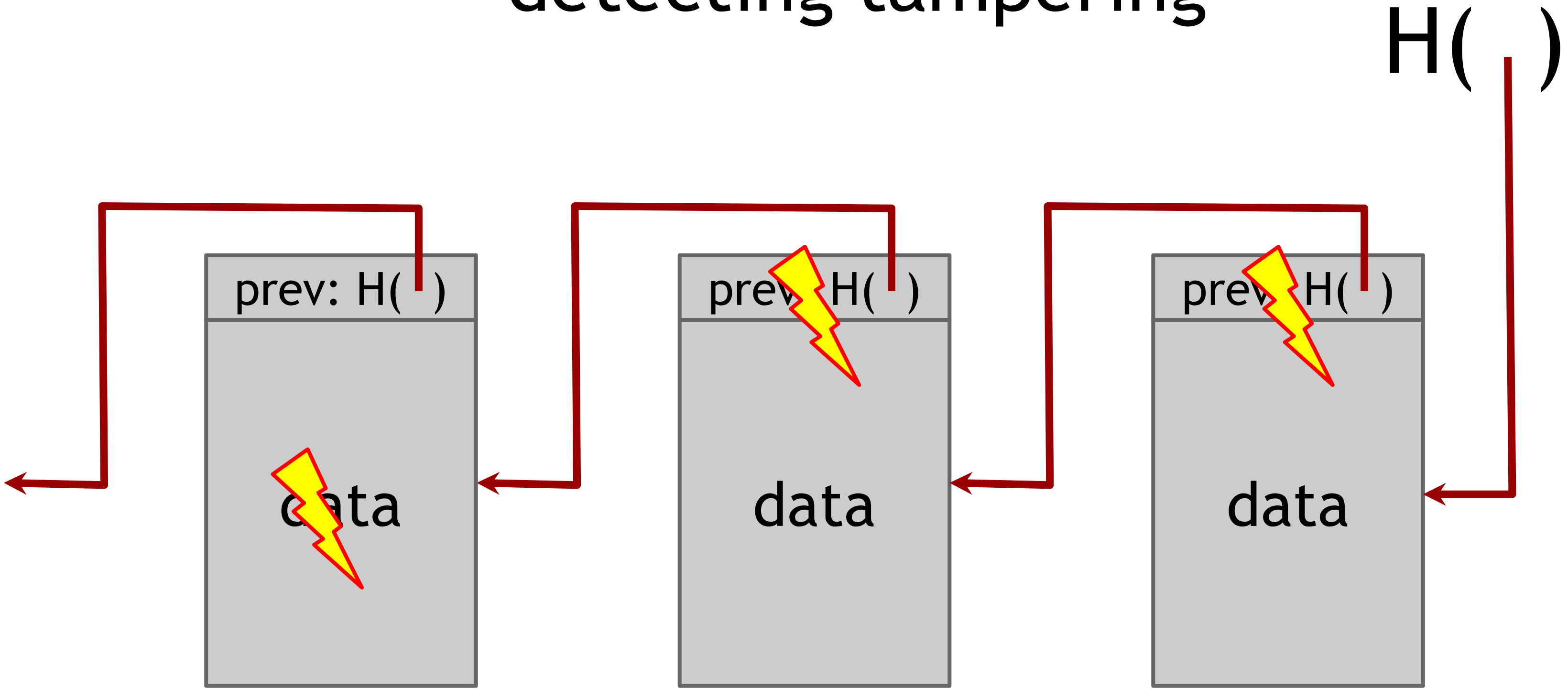
build data structures with hash pointers

linked list with hash pointers = “block chain”



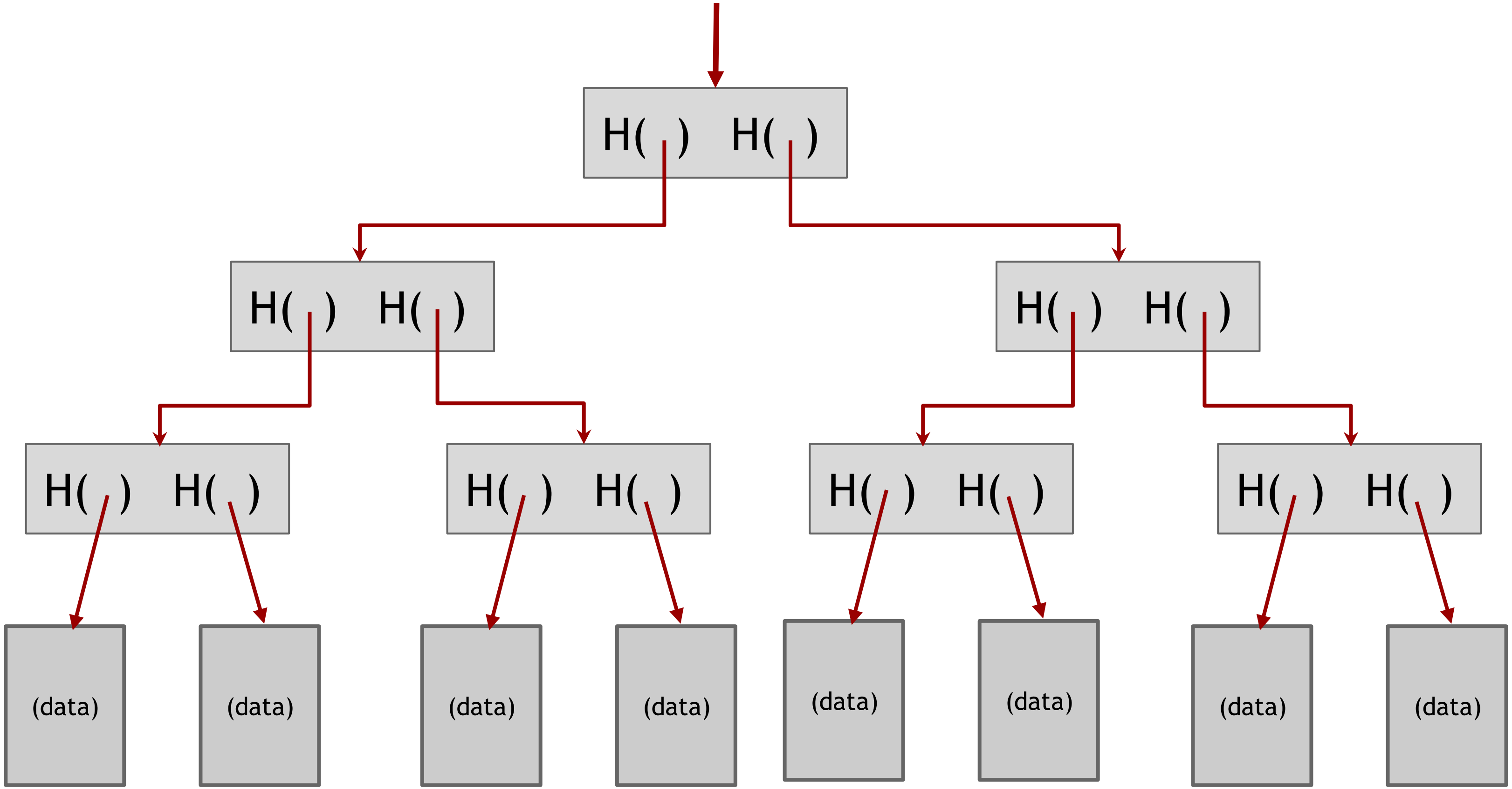
use case: tamper-evident log

detecting tampering

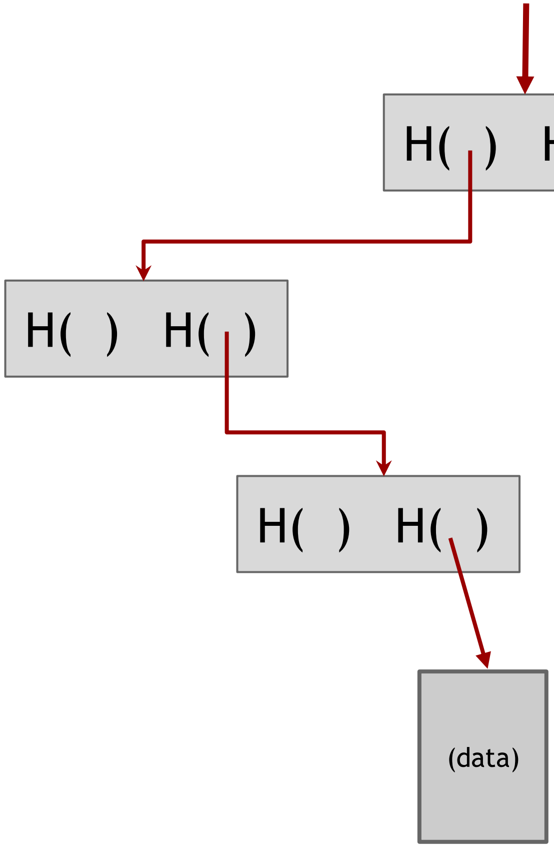


use case: tamper-evident log

binary tree with hash pointers = “Merkle tree”



proving membership in a Merkle tree



show O(log n) items

Advantages of Merkle trees

Tree holds many items

but just need to remember the root hash

Can verify membership in O(log n) time/space

Variant: sorted Merkle tree

can verify non-membership in O(log n)

(show items before, after the missing one)

More generally ...

can use hash pointers in any pointer-based data structure that has no cycles

GoofyCoin

Simple Cryptocurrencies

Obvious approach

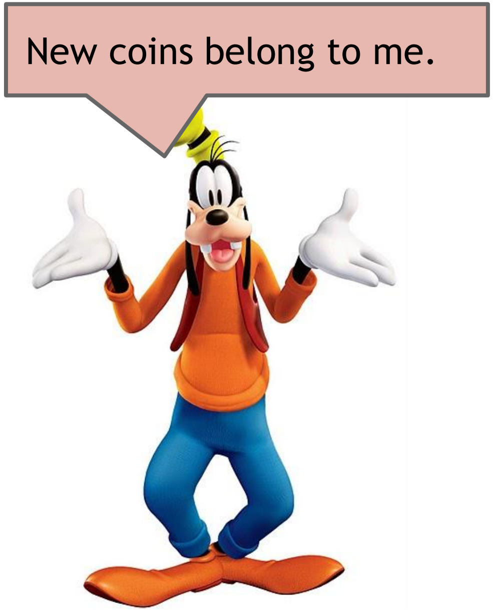
1. Use public keys as addresses
2. Sign to authorize transfer to new address

New coins created [somehow]

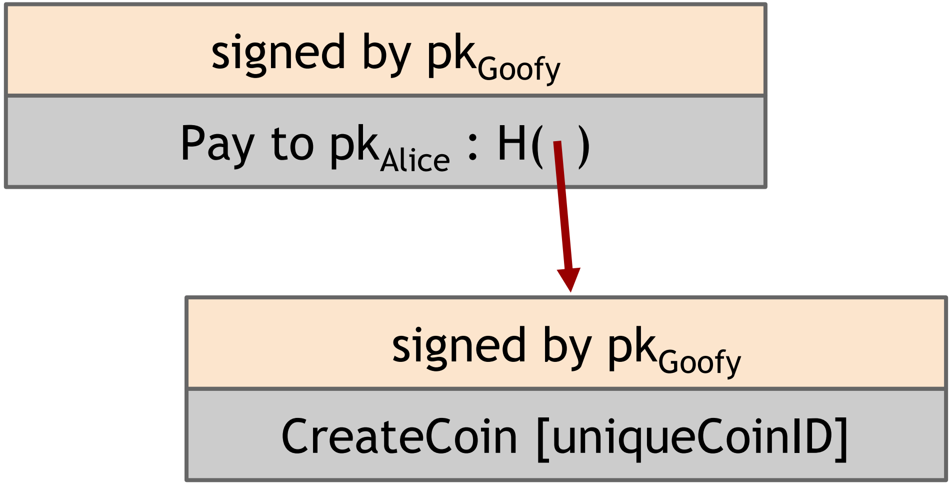
Goofy can create new coins

signed by pk$_{Goofy}$

CreateCoin [uniqueCoinID]



A coin’s owner can spend it.



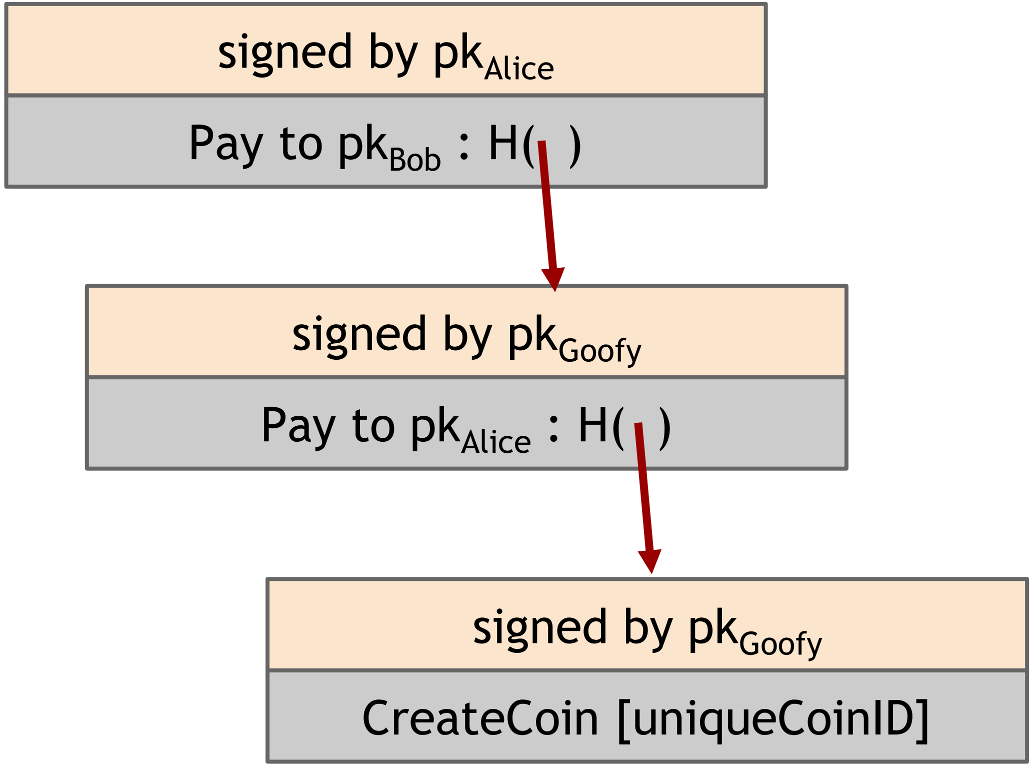
signed by pk_{Goofy}

Pay to pk_{Alice} : H( )

signed by pk_{Goofy}

CreateCoin [uniqueCoinID]

The recipient can pass on the coin again.



signed by pk_{Alice}

Pay to pk_{Bob} : H( )

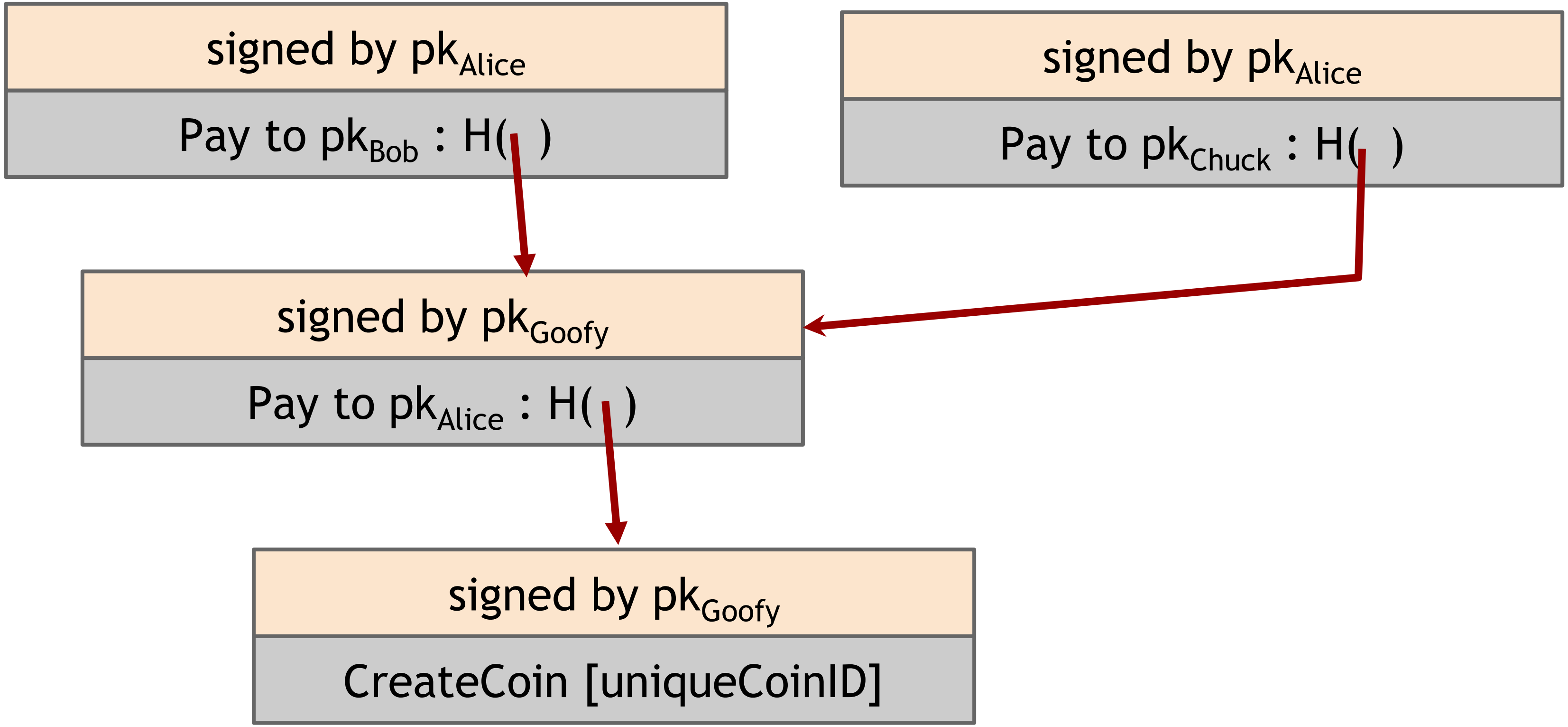
signed by pk_{Goofy}

Pay to pk_{Alice} : H( )

signed by pk_{Goofy}

CreateCoin [uniqueCoinID]

double-spending attack



signed by pk_{Alice}

Pay to pk_{Bob} : H( )

signed by pk_{Alice}

Pay to pk_{Chuck} : H( )

signed by pk_{Goofy}

Pay to pk_{Alice} : H( )

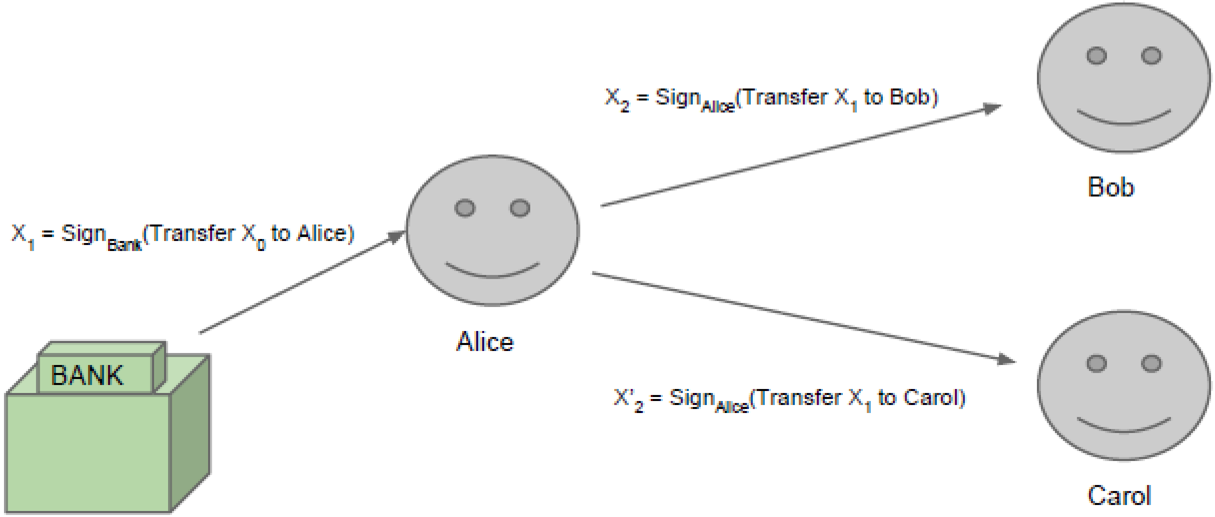
signed by pk_{Goofy}

CreateCoin [uniqueCoinID]

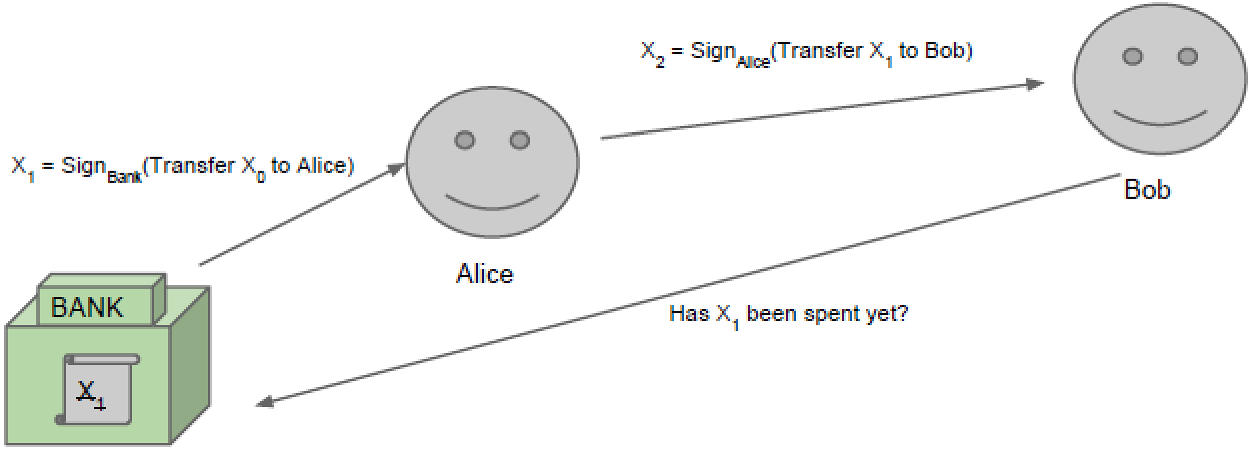
double-spending attack

the main design challenge in digital currency

Double-spends must be prevented

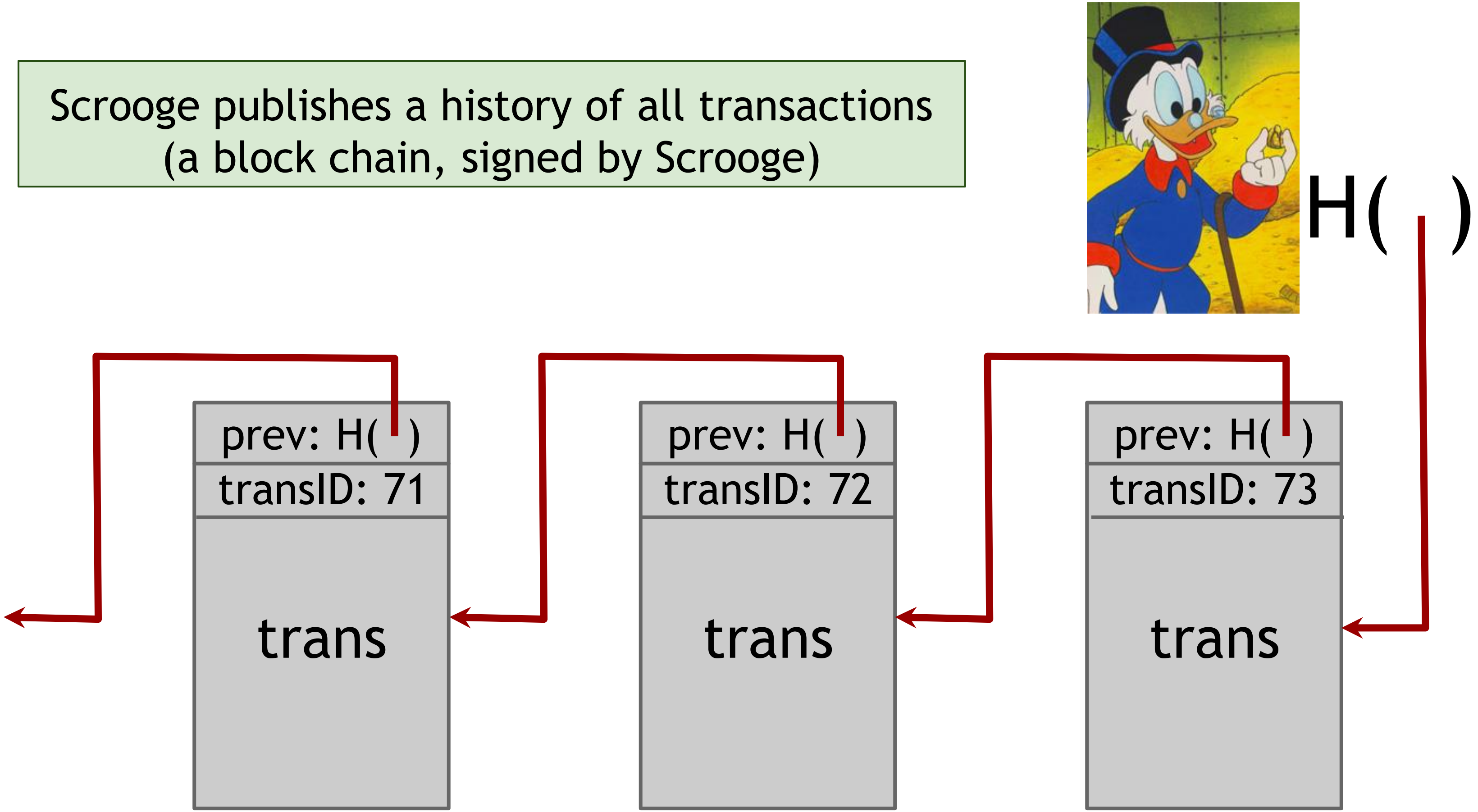

X_2 = $\text{Sign}_{\text{Alice}}$(Transfer X_1 to Bob)
Bob
X_1 = $\text{Sign}_{\text{Bank}}$(Transfer X_0 to Alice)
Alice
BANK
X'_2 = $\text{Sign}_{\text{Alice}}$(Transfer X_1 to Carol)
Carol

Traditional approach: talk to the issuer


X_2 = Sign$_{Alice}$(Transfer X_1 to Bob)
X_1 = Sign$_{Bank}$(Transfer X_0 to Alice)
Alice
Bob
BANK
X_1
Has X_1 been spent yet?

ScroogeCoin

Scrooge publishes a history of all transactions
(a block chain, signed by Scrooge)



H( )

prev: H( )

transID: 71

trans

prev: H( )

transID: 72

trans

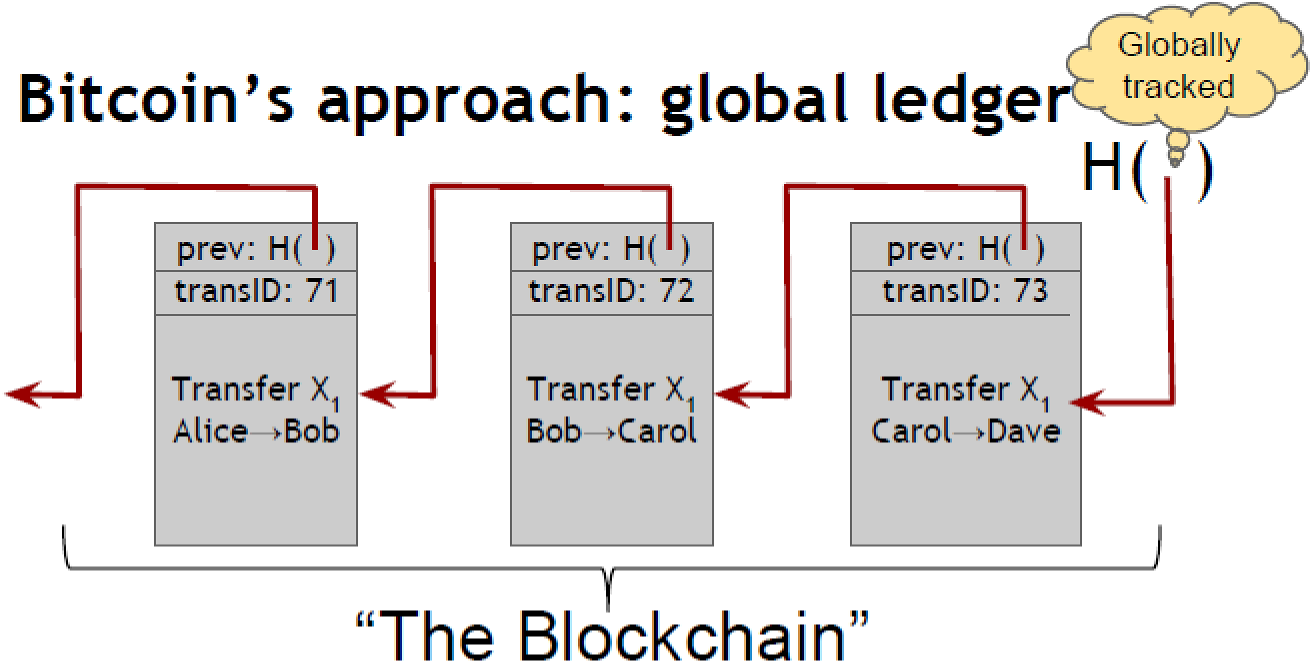
prev: H( )

transID: 73

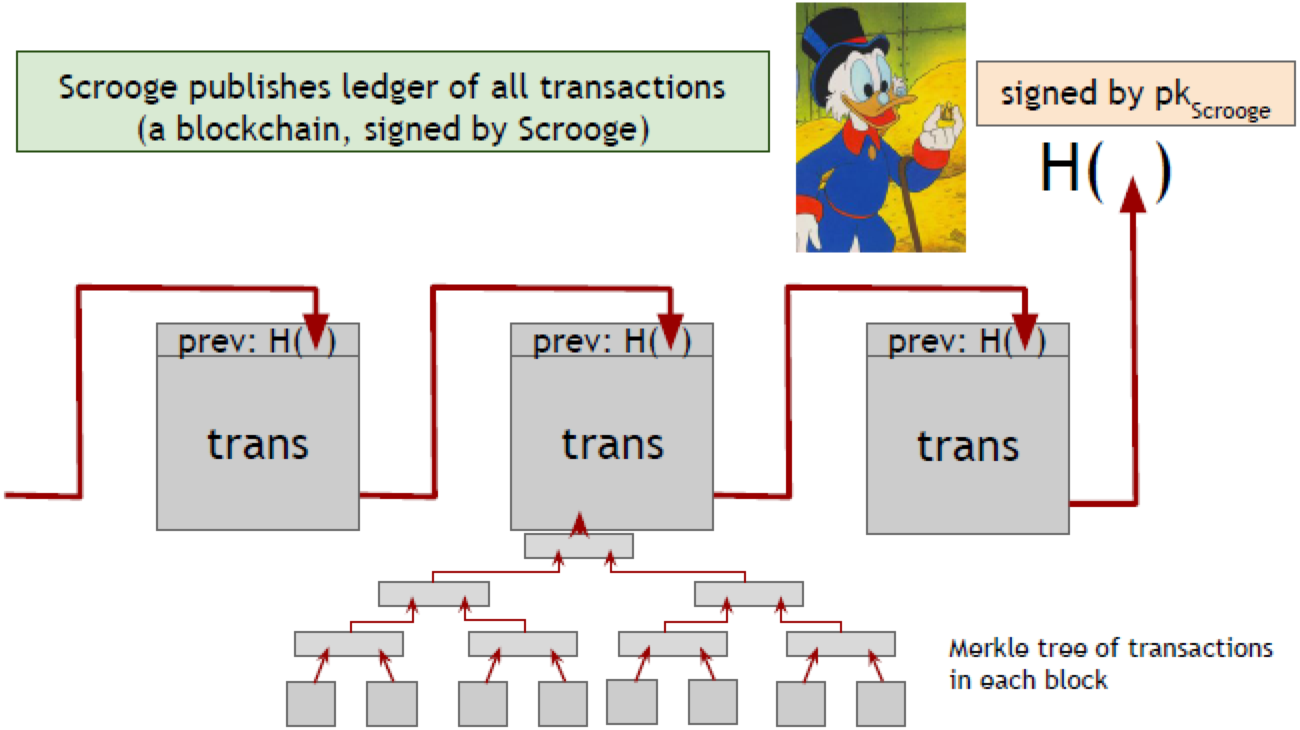
trans

optimization: put multiple transactions in the same block

Bitcoin’s approach: global ledger



Scrooge publishes ledger of all transactions
(a blockchain, signed by Scrooge)



signed by $pk_{Scrooge}$

Merkle tree of transactions
in each block

CreateCoins transaction creates new coins

transID: 73 type:CreateCoins

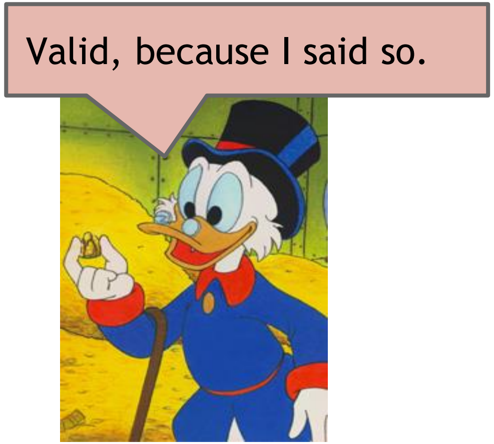
coins created

num	*value*	*recipient*
0	3.2	0x...
1	1.4	0x...
2	7.1	0x...

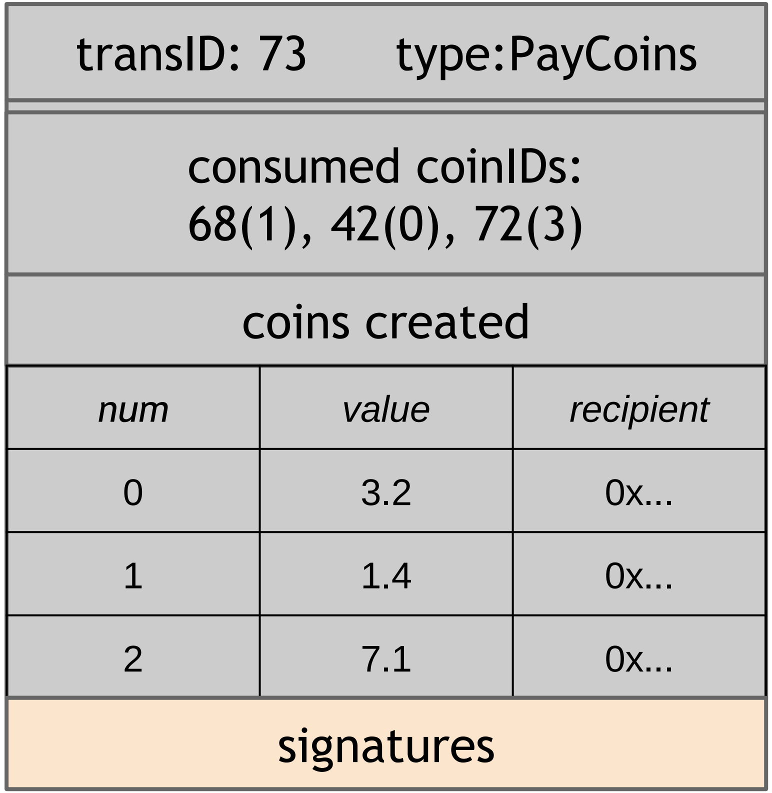
coinID 73(0)

coinID 73(1)

coinID 73(2)



PayCoins transaction consumes (and destroys) some coins,
and creates new coins of the same total value



transID: 73 type:PayCoins

consumed coinIDs:
68(1), 42(0), 72(3)

coins created

num	*value*	*recipient*
0	3.2	0x...
1	1.4	0x...
2	7.1	0x...

signatures

Valid if:
- -- consumed coins valid,
- -- not already consumed,
- -- total value out = total value in, and
- -- signed by owners of all consumed coins

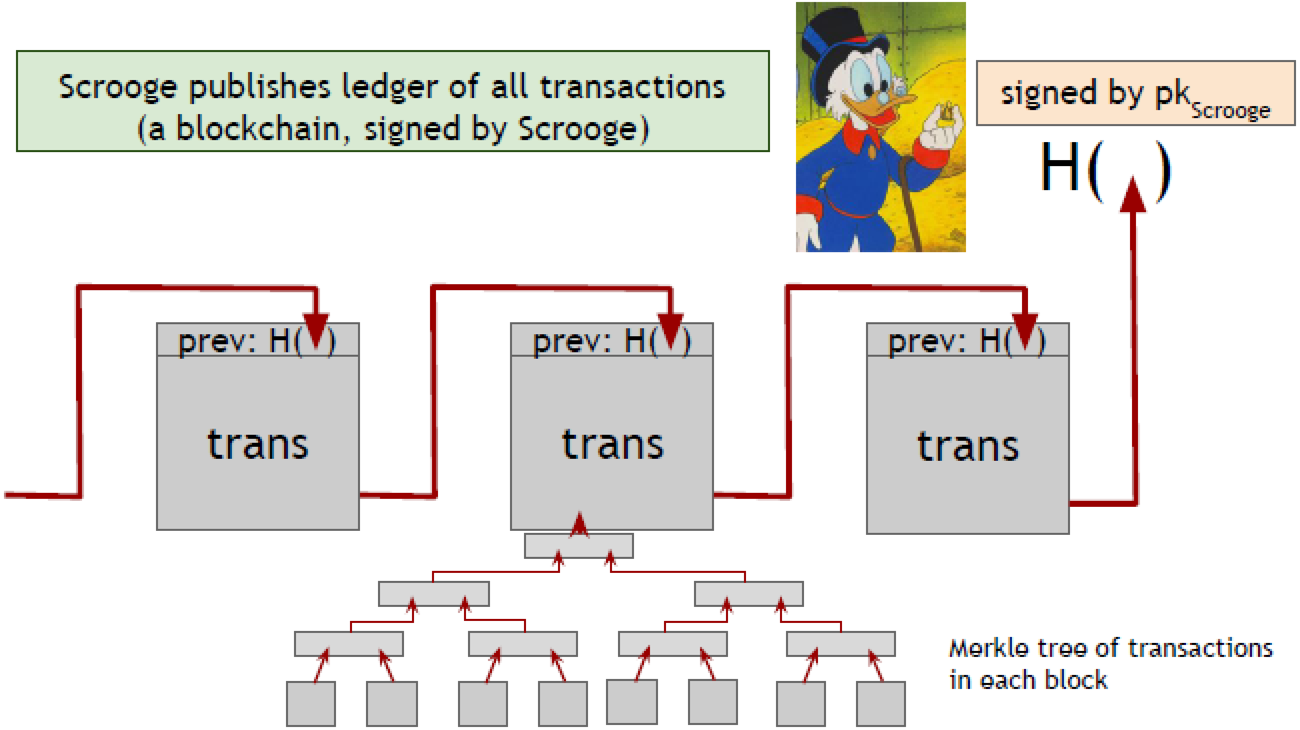
Immutable coins

Coins can’t be transferred, subdivided, or combined.

But: you can get the same effect by using transactions

- to subdivide: create new trans
  - consume your coin
  - pay out two new coins to yourself

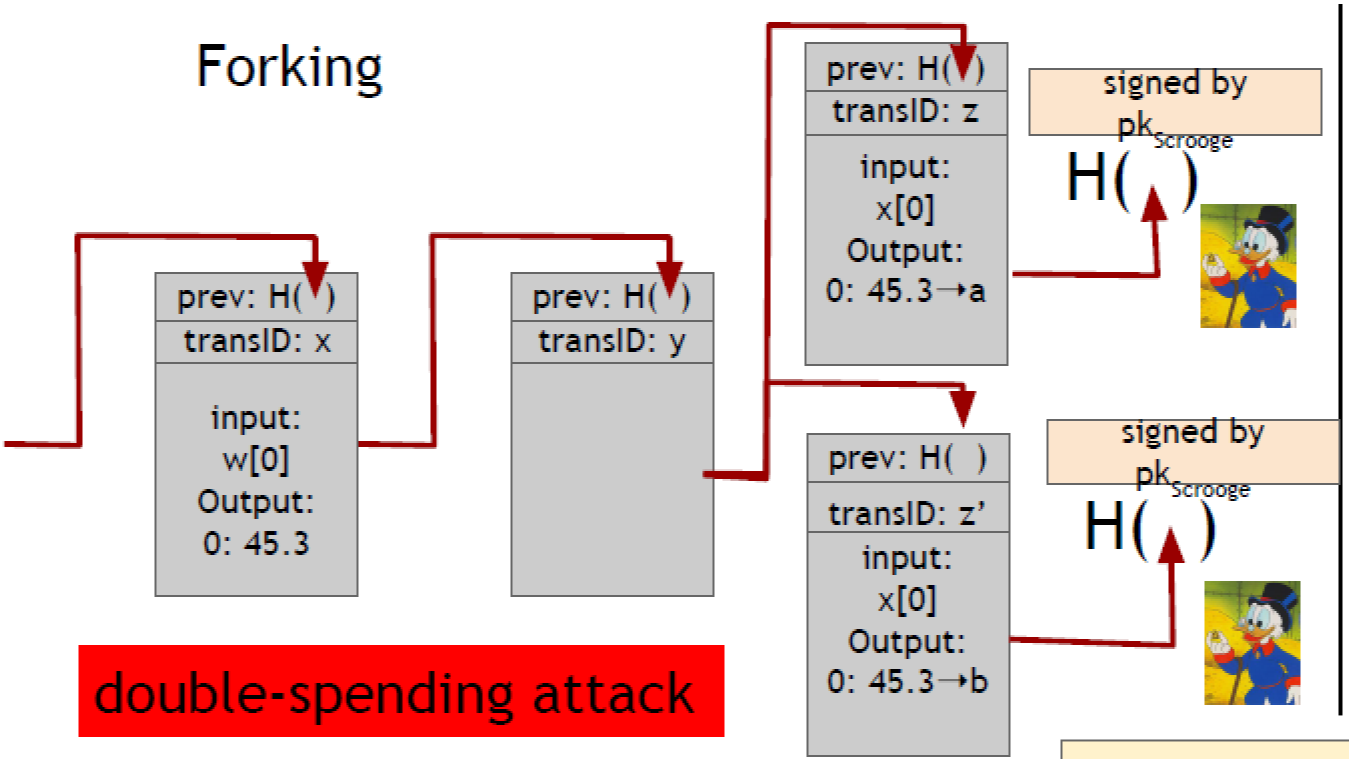
Scrooge publishes ledger of all transactions
(a blockchain, signed by Scrooge)



Merkle tree of transactions
in each block

Don’t worry, I’m honest.

Forking



double-spending attack

What if Scrooge is malicious?

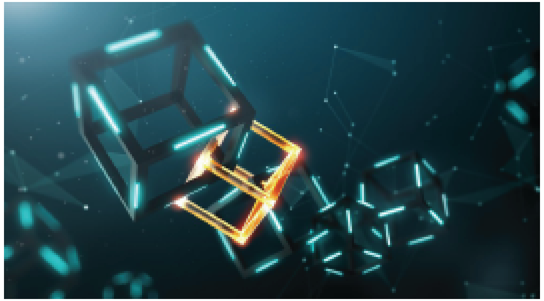
Don’t worry, I’m honest.

Crucial question:

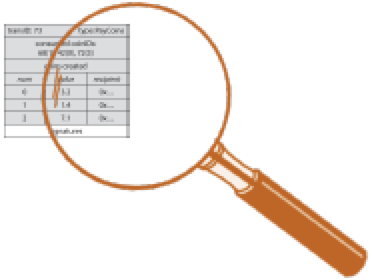
Can we descroogify the currency, and operate without any central, trusted party?

The path to decentralization

- technology & incentive design



Who maintains the ledger of transactions? (and how?)



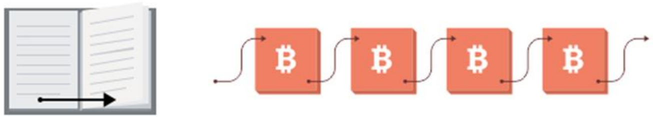
Who determines the validity of transactions to be included in the ledger?

Who creates new Bitcoins?

Mechanics of Bitcoin

What is blockchain, and why does it matter?

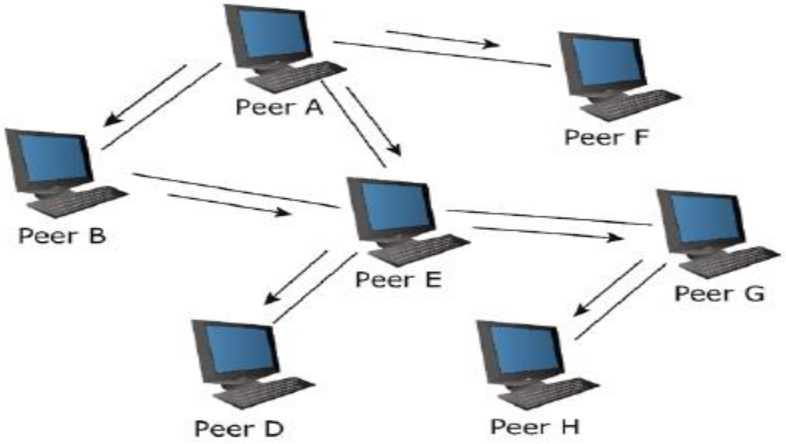
- A blockchain is a historical record of transactions, much like a database
- Blocks in a chain = pages in a book.
- Each page in a book contains:
  - The text: the story
  - Each page has information about itself: title of the book, chapter title, page number, etc. (e.g. the “metadata”)

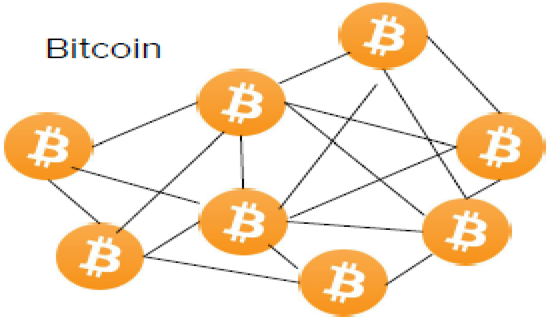


“Bits on Blocks”, Blog by Antony Lewis, https://bitsonblocks.net/

- Similarly, in a blockchain, each block has:
  - A *header* which contains the data about the block: e.g. technical information, a reference to the previous block, and a digital fingerprint (aka “hash”) of the data contained in this block, among other things. This hash is important for ordering and block validation.
  - The *contents* of the block, e.g. information about the transaction(s)

Peer to Peer Network



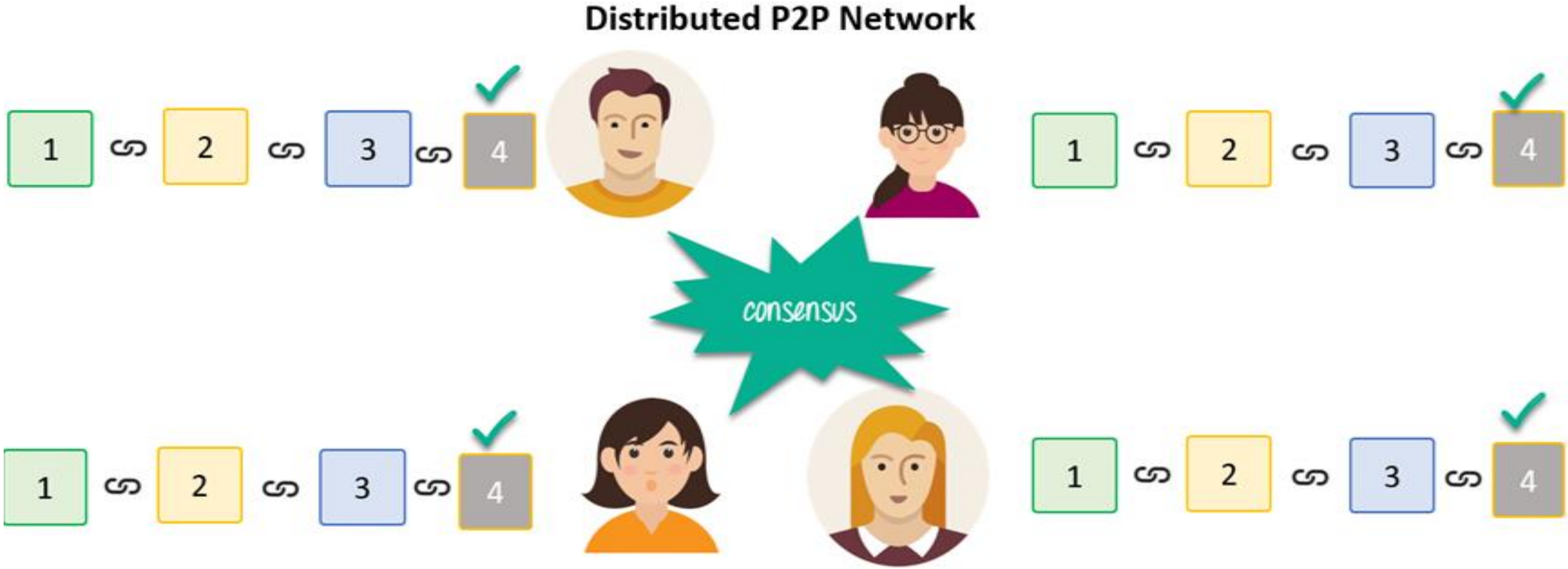


Peer to Peer Network

A distributed network architecture may be called a Peer-to-Peer (P-to-P, P2P,.) network, if the participants share a part of their own hardware resources (processing power, storage capacity, network link capacity, printers,.). These shared resources are necessary to provide the Service and content offered by the network (e.g. file sharing or shared workspaces for collaboration). They are accessible by other peers directly, without passing intermediary entities. The participants of such a network are thus resource (Service and content) providers as well as resource (Service and content) requestors (Servent-concept).

- Rüdiger Schollmeier, 2002

How to achieve consistency?



Public Keys as Identities

Why don’t Bitcoin nodes have identities?

No Central Authority

Useful trick: public key == an identity

if you see *sig* such that *verify(pk*, *msg*, *sig)==true*,

think of it as

pk says, “*[msg]*”.

to “speak for” *pk*, you must know matching secret key *sk*

How to make a new identity

create a new, random key-pair *(sk, pk)*

pk is the public “name” you can use

[usually better to use Hash(pk)]

sk lets you “speak for” the identity

you control the identity, because only you know *sk*

if *pk* “looks random”, nobody needs to know who you are

Decentralized identity management

Public Key as Idenitity!

anybody can make a new identity at any time - make as many as you want!

These identities are called “addresses” in Bitcoin.

Pseudonymity is a goal of Bitcoin

Identity is hard in a P2P system — Sybil attack

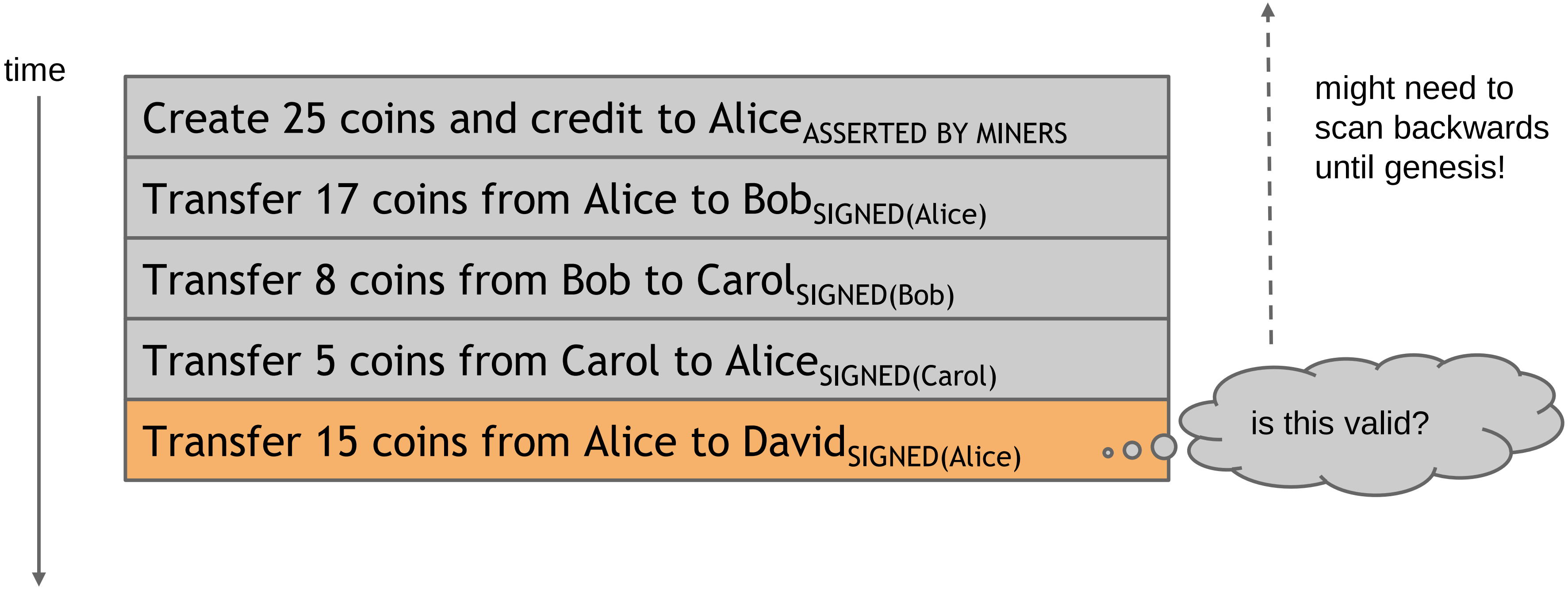
Privacy

Addresses not directly connected to real-world identity.

But observer can link together an address’s activity over time, make inferences.

Bitcoin transactions

An account-based ledger (*not* Bitcoin)



Create 25 coins and credit to Alice$_{\text{ASSERTED BY MINERS}}$

Transfer 17 coins from Alice to Bob$_{\text{SIGNED(Alice)}}$

Transfer 8 coins from Bob to Carol$_{\text{SIGNED(Bob)}}$

Transfer 5 coins from Carol to Alice$_{\text{SIGNED(Carol)}}$

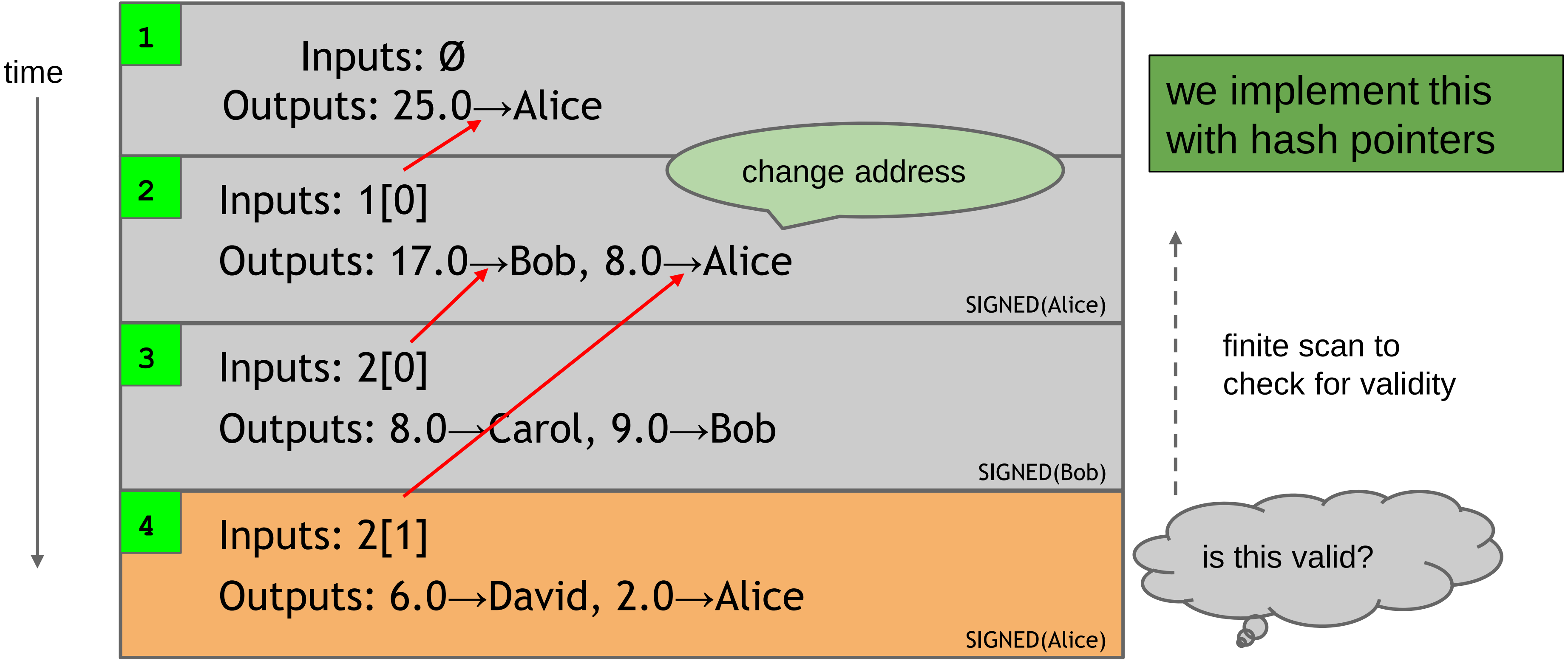
Transfer 15 coins from Alice to David$_{\text{SIGNED(Alice)}}$

is this valid?

might need to scan backwards until genesis!

SIMPLIFICATION: only one transaction per block

A transaction-based ledger (Bitcoin)



time

1

Inputs: Ø
Outputs: 25.0→Alice

2

Inputs: 1[0]
Outputs: 17.0→Bob, 8.0→Alice
SIGNED(Alice)

change address

3

Inputs: 2[0]
Outputs: 8.0→Carol, 9.0→Bob
SIGNED(Bob)

4

Inputs: 2[1]
Outputs: 6.0→David, 2.0→Alice
SIGNED(Alice)

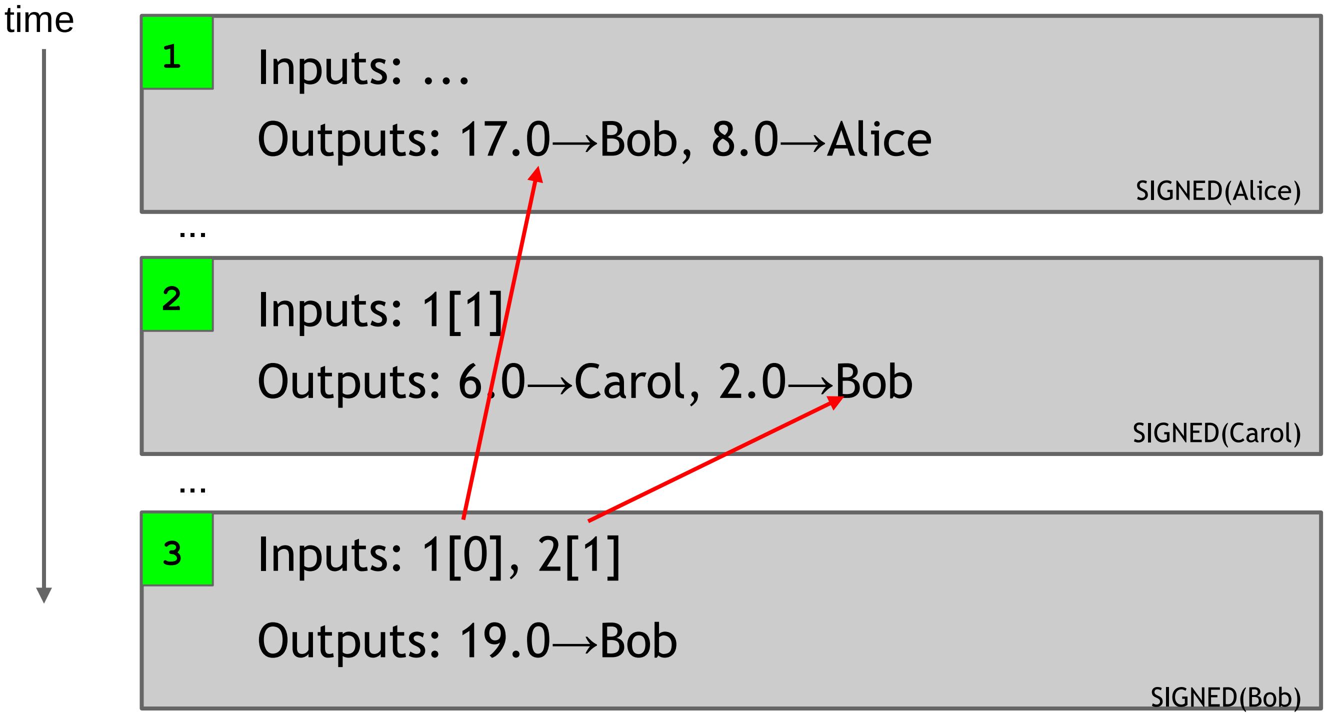
we implement this with hash pointers

finite scan to check for validity

is this valid?

SIMPLIFICATION: only one transaction per block

Merging value



time

1

Inputs: ...

Outputs: 17.0→Bob, 8.0→Alice

SIGNED(Alice)

...

2

Inputs: 1[1]

Outputs: 6.0→Carol, 2.0→Bob

SIGNED(Carol)

...

3

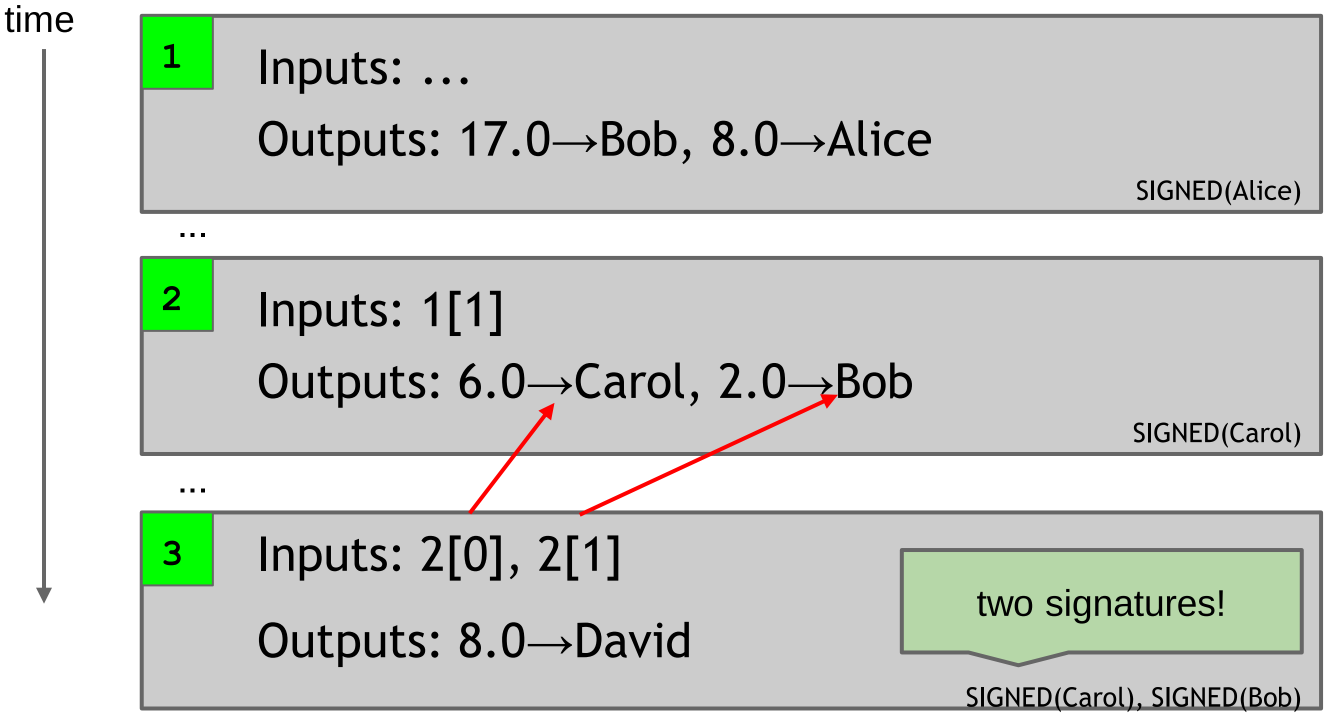
Inputs: 1[0], 2[1]

Outputs: 19.0→Bob

SIGNED(Bob)

SIMPLIFICATION: only one transaction per block

Joint payments



time

1

Inputs: ...

Outputs: 17.0→Bob, 8.0→Alice

SIGNED(Alice)

...

2

Inputs: 1[1]

Outputs: 6.0→Carol, 2.0→Bob

SIGNED(Carol)

...

3

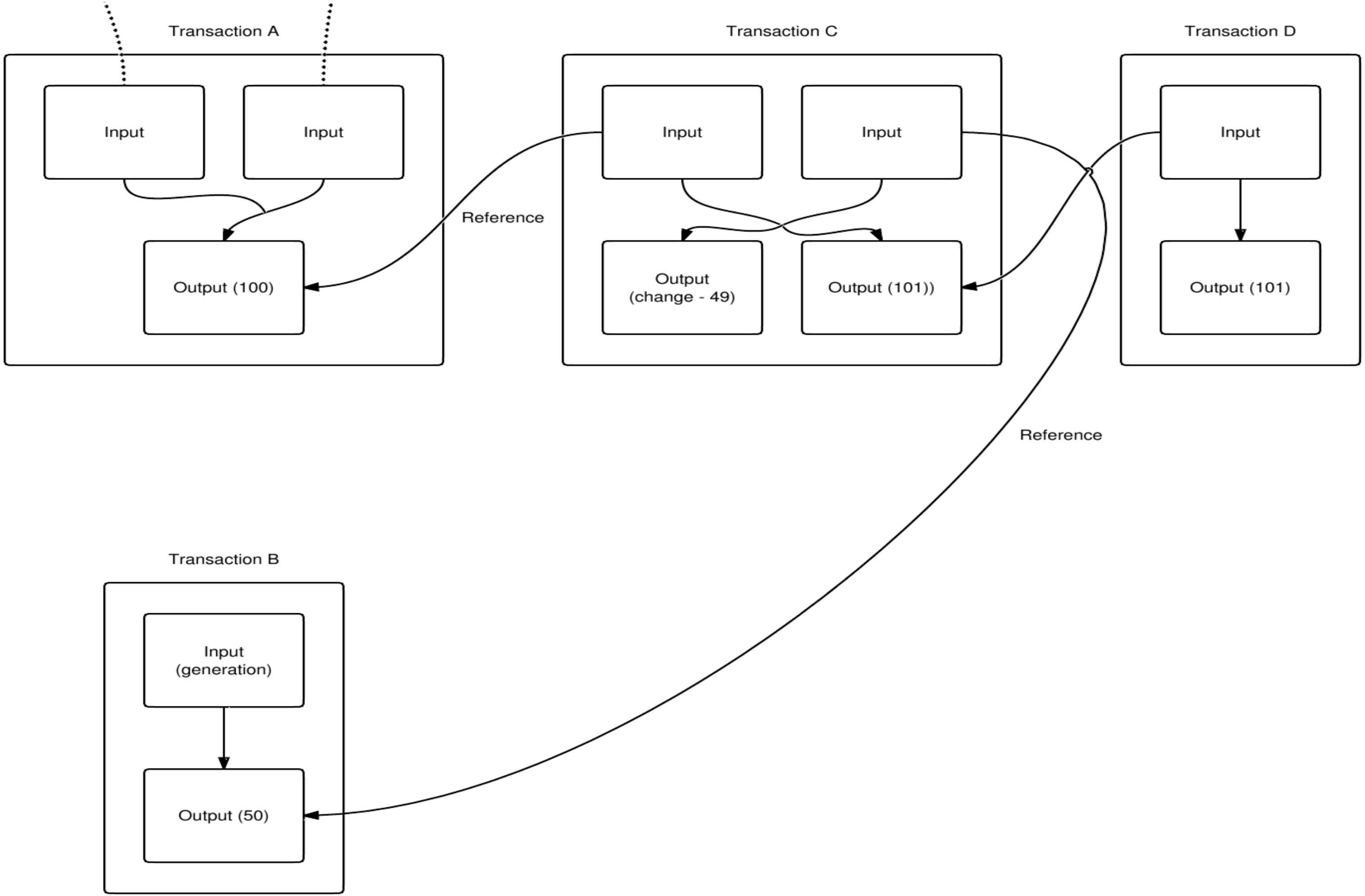
Inputs: 2[0], 2[1]

Outputs: 8.0→David

two signatures!

SIGNED(Carol), SIGNED(Bob)

SIMPLIFICATION: only one transaction per block


Transaction A
Input
Input
Output (100)
Reference
Transaction C
Input
Input
Output
(change - 49)
Output (101))
Transaction D
Input
Output (101)
Reference
Transaction B
Input
(generation)
Output (50)

The real deal: a Bitcoin transaction

```
            {
               "hash":"5a42590fbe0a90ee8e8747244d6c84f0db1a3a24e8f1b95b10c9e050990b8b6b",
               "ver":1,
               "vin_sz":2,
metadata       "vout_sz":1,
               "lock_time":0,
               "size":404,
               "in":[
                {
                 "prev_out":{
                  "hash":"3be4ac9728a0823cf5e2deb2e86fc0bd2aa503a91d307b42ba76117d79280260",
                  "n":0
                 },
                   "scriptSig":"30440..."
input(s)        },
                {
                 "prev_out":{
                  "hash":"7508e6ab259b4df0fd5147bab0c949d81473db4518f81afc5c3f52f91ff6b34e",
                  "n":0
                 },
                 "scriptSig":"3f3a4ce81...."
                }
               ],
               "out":[
                {
output(s)        "value":"10.12287097",
                 "scriptPubKey":"OP_DUP OP_HASH160 69e02e18b5705a05dd6b28ed517716c894b3d42e OP_EQUALVERIFY OP_CHECKSIG"
                }
               ]
            }
```

The real deal: transaction metadata

```
                        {
transaction hash            "hash":"5a42590...b8b6b",
                              "ver":1,
housekeeping                  "vin_sz":2,
                              "vout_sz":1,
“not valid before”            "lock_time":0,
housekeeping                  "size":404,

                        ...
                        }
```

The real deal: transaction inputs

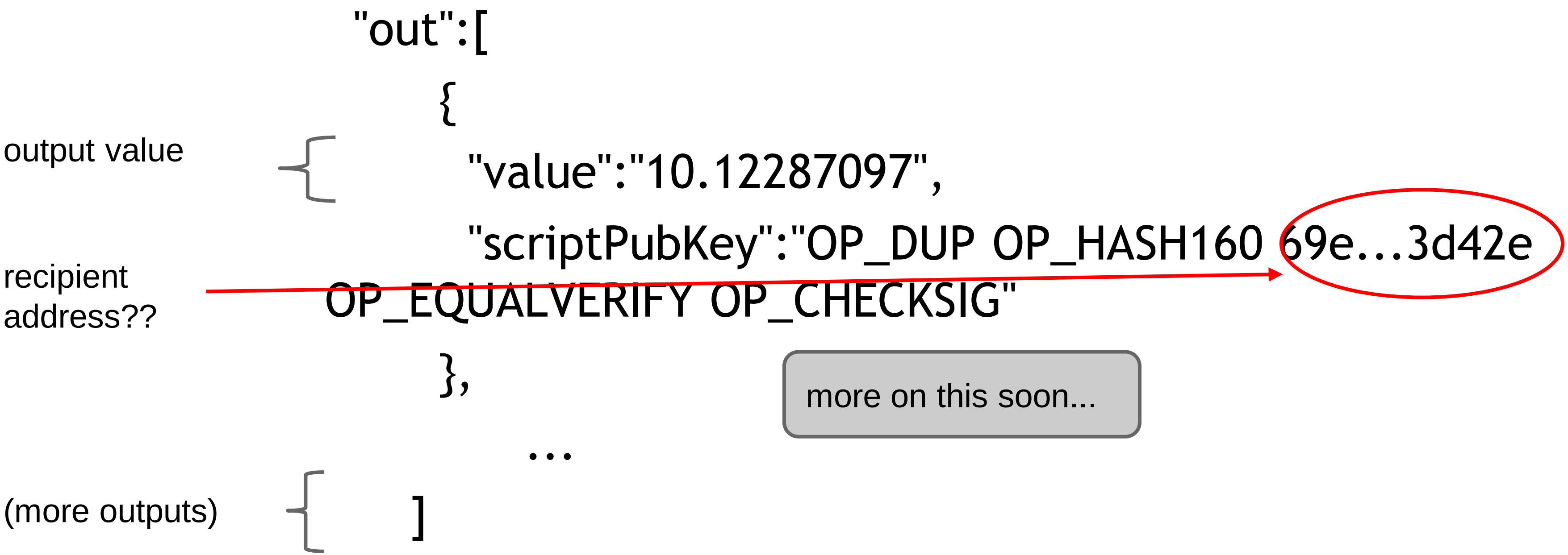
```
"in":[
  {
    "prev_out":{
      "hash":"3be4...80260",
      "n":0
    },
"scriptSig":"30440....3f3a4ce81"
  },
…
],
```

previous
transaction

signature

(more inputs)

The real deal: transaction outputs



```
"out":[
        {
          "value":"10.12287097",
          "scriptPubKey":"OP_DUP OP_HASH160 69e...3d42e
OP_EQUALVERIFY OP_CHECKSIG"
        },
              ...
     ]
```

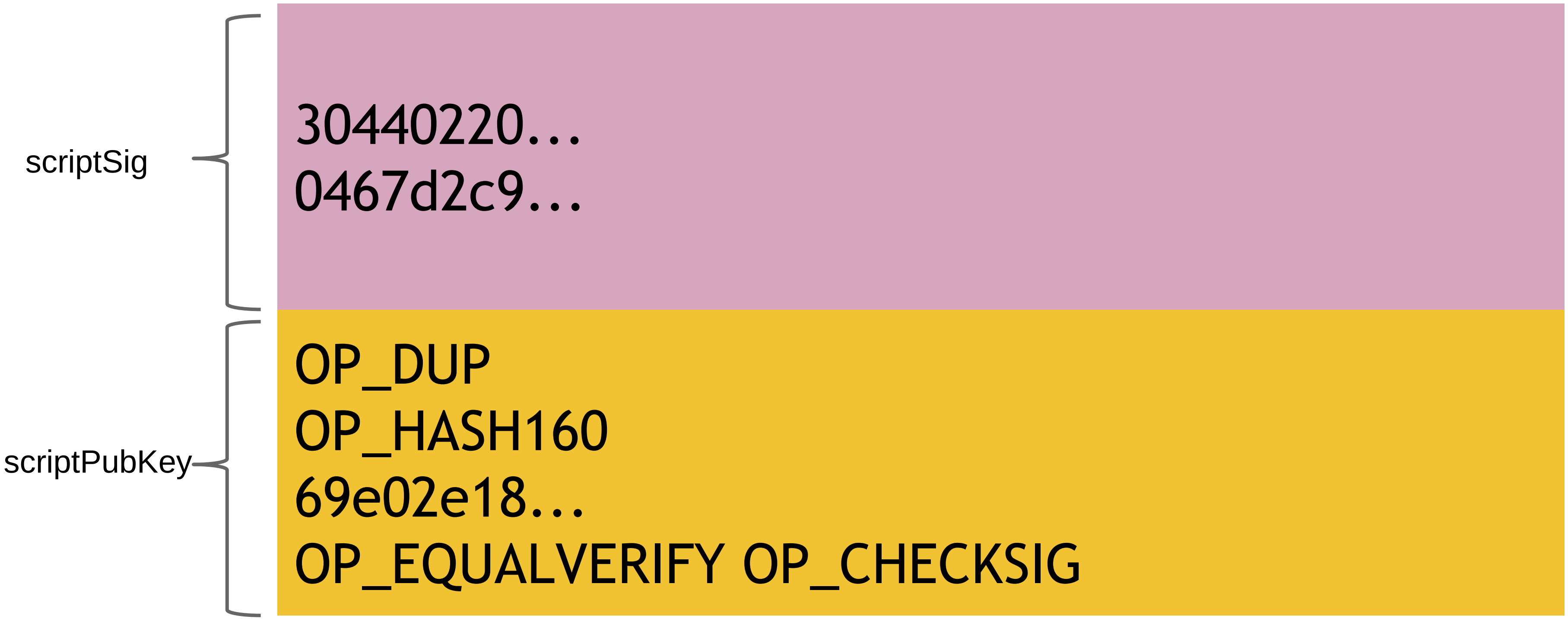
more on this soon...

Bitcoin scripts

Output “addresses” are really *scripts*

```
OP_DUP
OP_HASH160
69e02e18...
OP_EQUALVERIFY OP_CHECKSIG
```

Input “addresses” are *also* scripts



scriptSig

```
30440220...
0467d2c9...
```

scriptPubKey

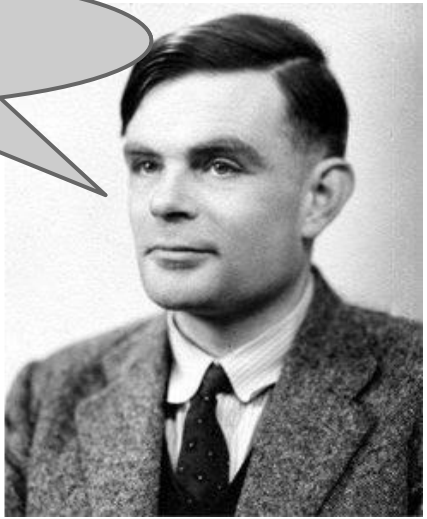
```
OP_DUP
OP_HASH160
69e02e18...
OP_EQUALVERIFY OP_CHECKSIG
```

TO VERIFY: Concatenated script must execute completely with no errors

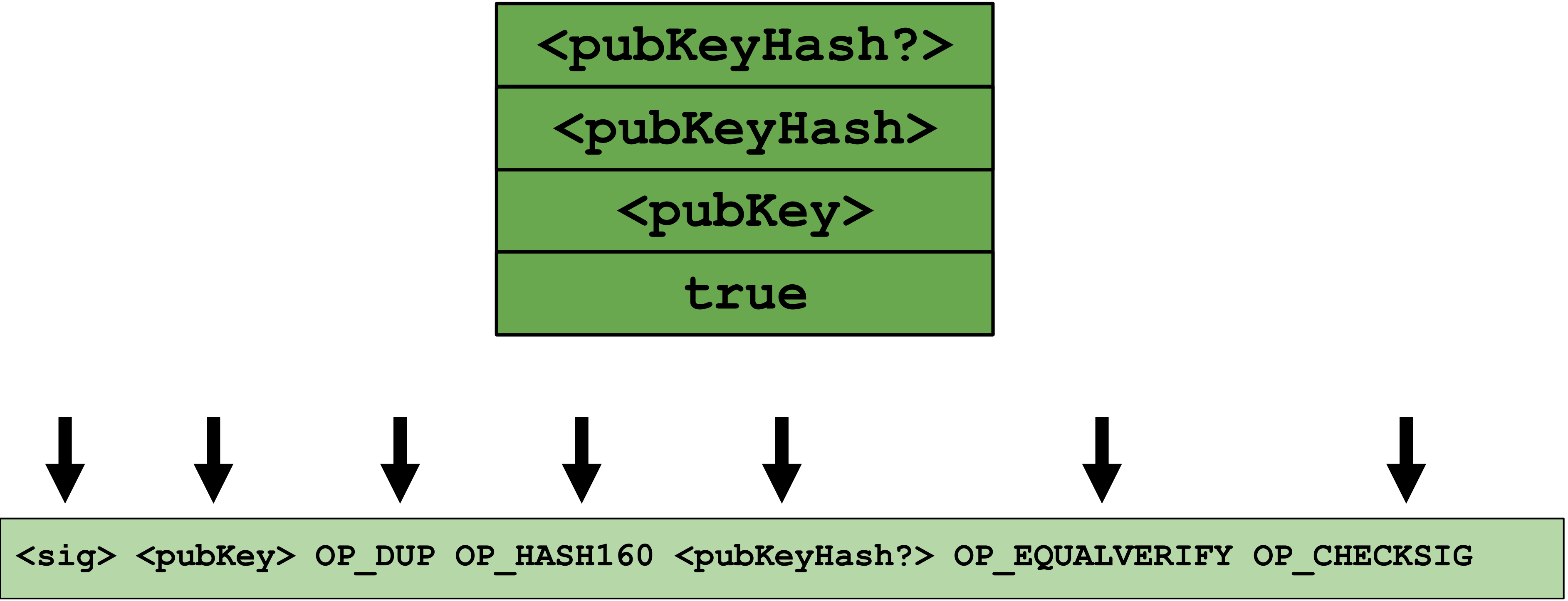
Bitcoin scripting language (“Script”)

Design goals

- Built for Bitcoin (inspired by Forth)
- Simple, compact
- Support for cryptography
- Stack-based
- Limits on time/memory
- No looping

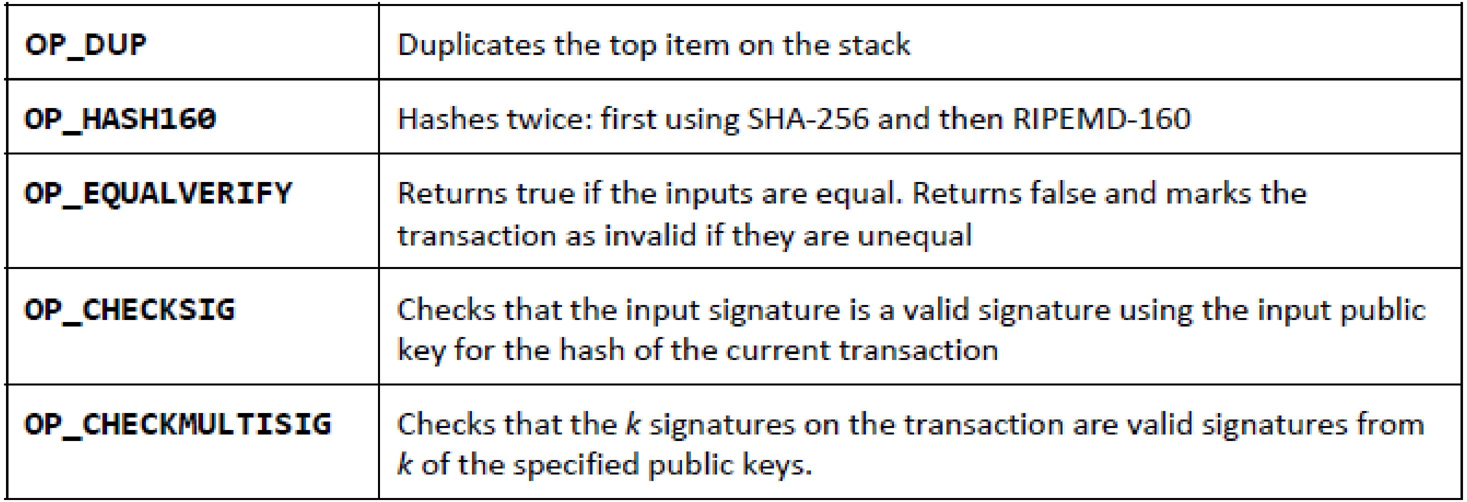


Bitcoin script execution example



```
<pubKeyHash?>
<pubKeyHash>
<pubKey>
true
```

```
<sig> <pubKey> OP_DUP OP_HASH160 <pubKeyHash?> OP_EQUALVERIFY OP_CHECKSIG
```



OP_DUP	Duplicates the top item on the stack
OP_HASH160	Hashes twice: first using SHA-256 and then RIPEMD-160
OP_EQUALVERIFY	Returns true if the inputs are equal. Returns false and marks the transaction as invalid if they are unequal
OP_CHECKSIG	Checks that the input signature is a valid signature using the input public key for the hash of the current transaction
OP_CHECKMULTISIG	Checks that the *k* signatures on the transaction are valid signatures from *k* of the specified public keys.

Figure 3.6 a list of common Script instructions and their functionality.

Bitcoin script instructions

256 opcodes total (15 disabled, 75 reserved)

- Arithmetic
- If/then
- Logic/data handling
- Crypto!
    - Hashes
    - Signature verification
    - Multi-signature verification

OP_CHECKMULTISIG

- Built-in support for joint signatures
- Specify *n* public keys
- Specify *t*
- Verification requires *t* signatures

BUG ALERT: Extra data value popped from the stack and ignored

Bitcoin scripts in practice (as of 2014)

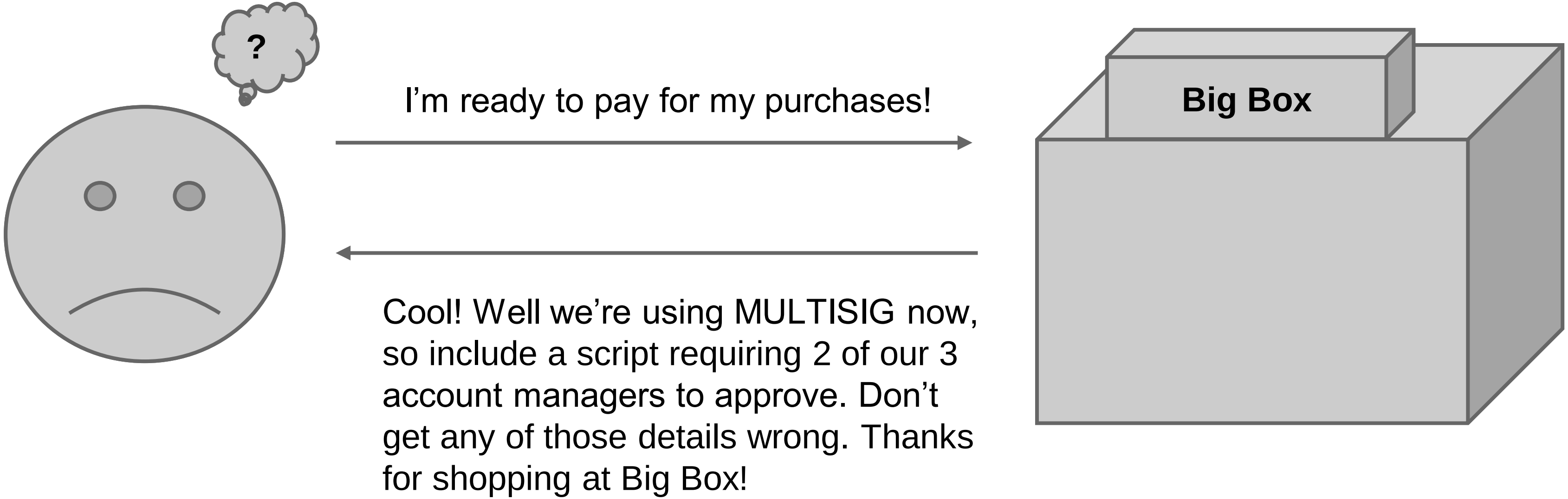
- Most nodes whitelist known scripts
- 99.9% are simple signature checks
- ~0.01% are MULTISIG
- ~0.01% are Pay-to-Script-Hash
- Remainder are errors, proof-of-burn

Proof-of-burn

nothing's going to redeem that ☹

OP_RETURN
<arbitrary data>

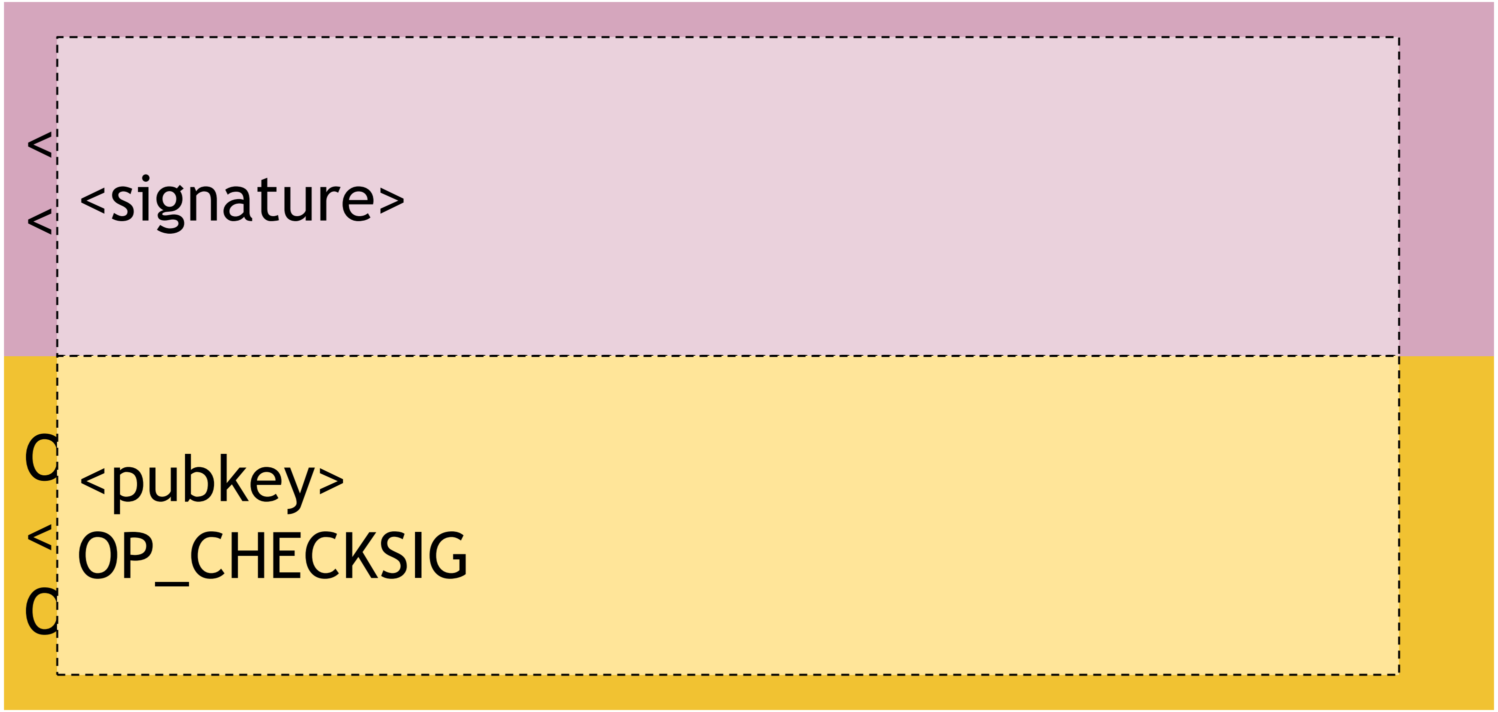
Should senders specify scripts?



I’m ready to pay for my purchases!

Cool! Well we’re using MULTISIG now, so include a script requiring 2 of our 3 account managers to approve. Don’t get any of those details wrong. Thanks for shopping at Big Box!

Idea: use the hash of redemption script

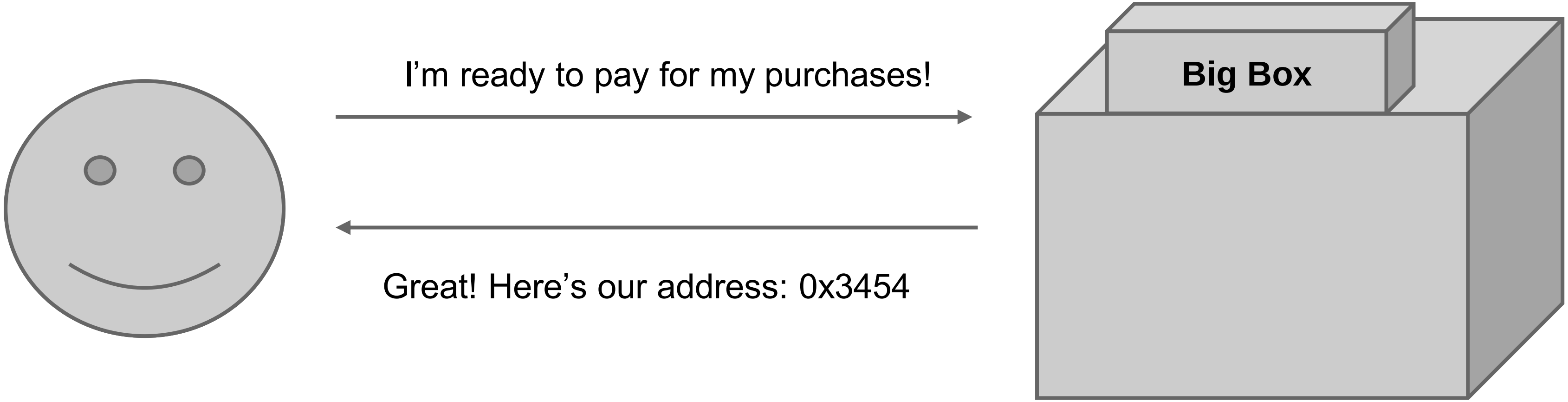


<
<signature>
<

O
<pubkey>
<
OP_CHECKSIG
O

“Pay to Script Hash”

Pay to script hash



I’m ready to pay for my purchases!

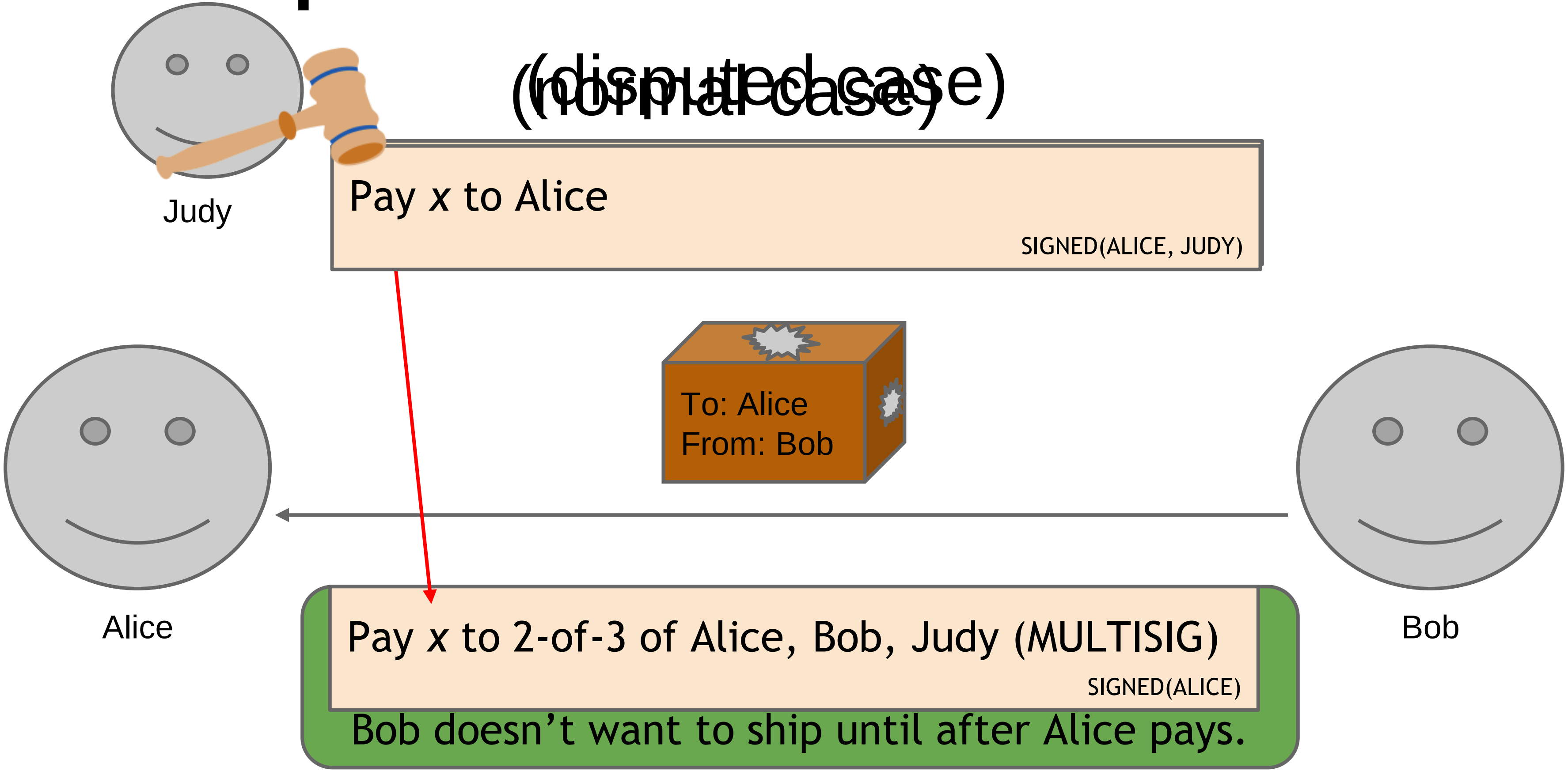
Great! Here’s our address: 0x3454

Applications of Bitcoin scripts

Example 1: Escrow transactions

(disputed case)

(normal case)



Pay *x* to Alice

SIGNED(ALICE, JUDY)

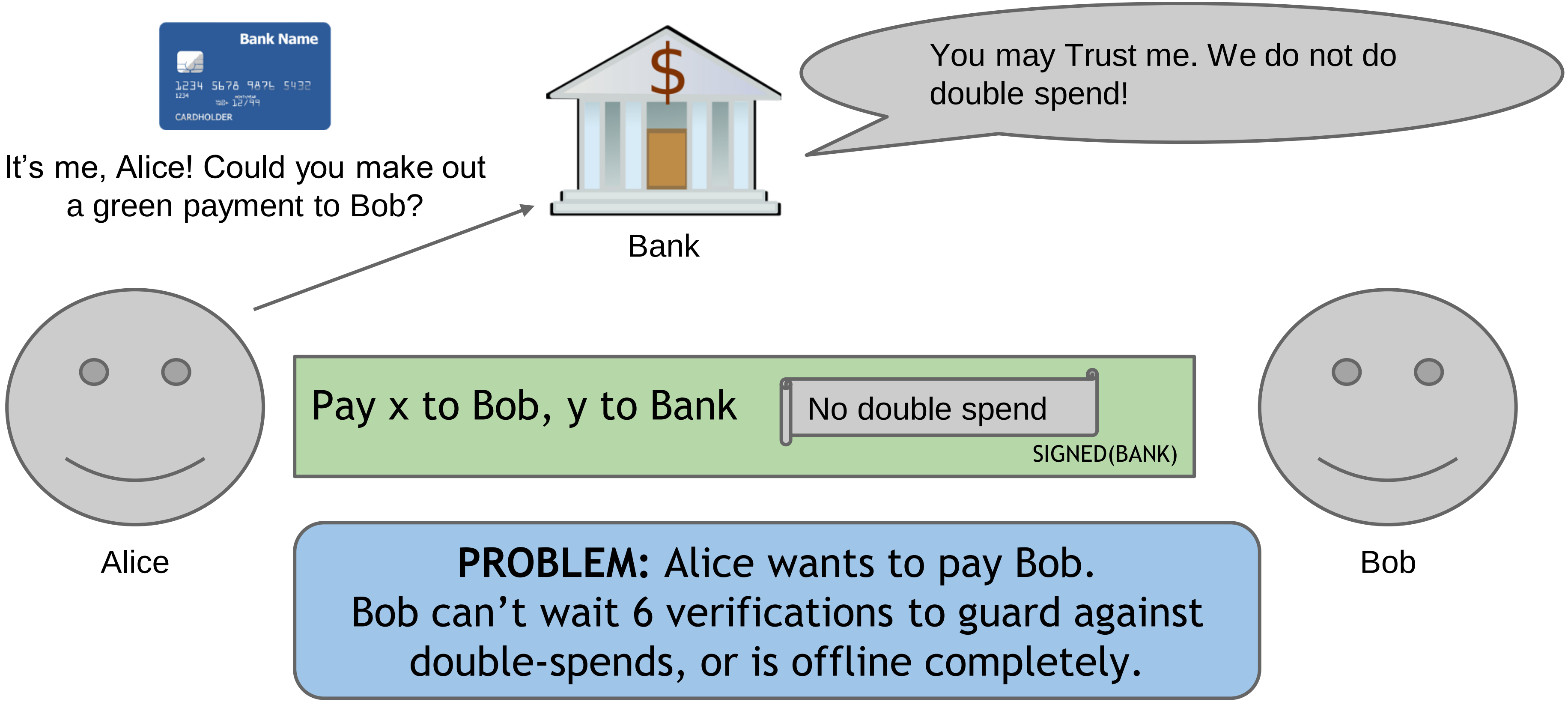
Pay *x* to 2-of-3 of Alice, Bob, Judy (MULTISIG)

SIGNED(ALICE)

Bob doesn’t want to ship until after Alice pays.

Example 2: Green addresses

InstaWallet, Mt. Gox Collapsed!



It’s me, Alice! Could you make out a green payment to Bob?

Bank

You may Trust me. We do not do double spend!

Pay x to Bob, y to Bank

No double spend

SIGNED(BANK)

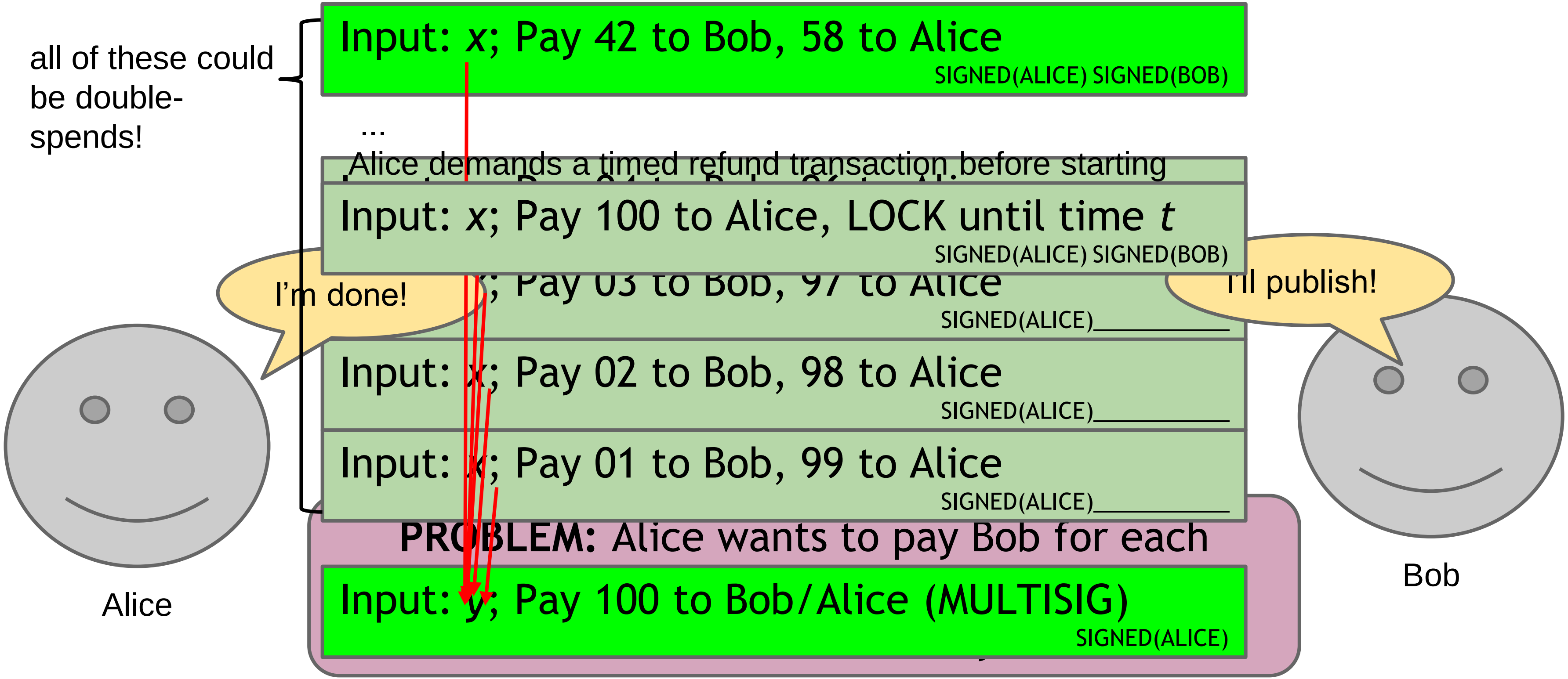
Alice

Bob

PROBLEM: Alice wants to pay Bob.
Bob can’t wait 6 verifications to guard against double-spends, or is offline completely.

Example 3: Efficient micro-payments

What if Bob never signs??



all of these could be double-spends!

Input: *x*; Pay 42 to Bob, 58 to Alice
SIGNED(ALICE) SIGNED(BOB)

...

Alice demands a timed refund transaction before starting

Input: *x*; Pay 100 to Alice, LOCK until time *t*
SIGNED(ALICE) SIGNED(BOB)

Input: *x*; Pay 03 to Bob, 97 to Alice
SIGNED(ALICE)__________

Input: *x*; Pay 02 to Bob, 98 to Alice
SIGNED(ALICE)__________

Input: *x*; Pay 01 to Bob, 99 to Alice
SIGNED(ALICE)__________

PROBLEM: Alice wants to pay Bob for each

Input: *y*; Pay 100 to Bob/Alice (MULTISIG)
SIGNED(ALICE)

lock_time

```
{
    "hash":"5a42590...b8b6b",
      "ver":1,
      "vin_sz":2,
      "vout_sz":1,
      "lock_time":315415,
      "size":404,
...
}
```

Block index or real-world timestamp before which this transaction can’t be published

More advanced scripts

- Multiplayer lotteries
- Hash pre-image challenges
- Coin-swapping protocols

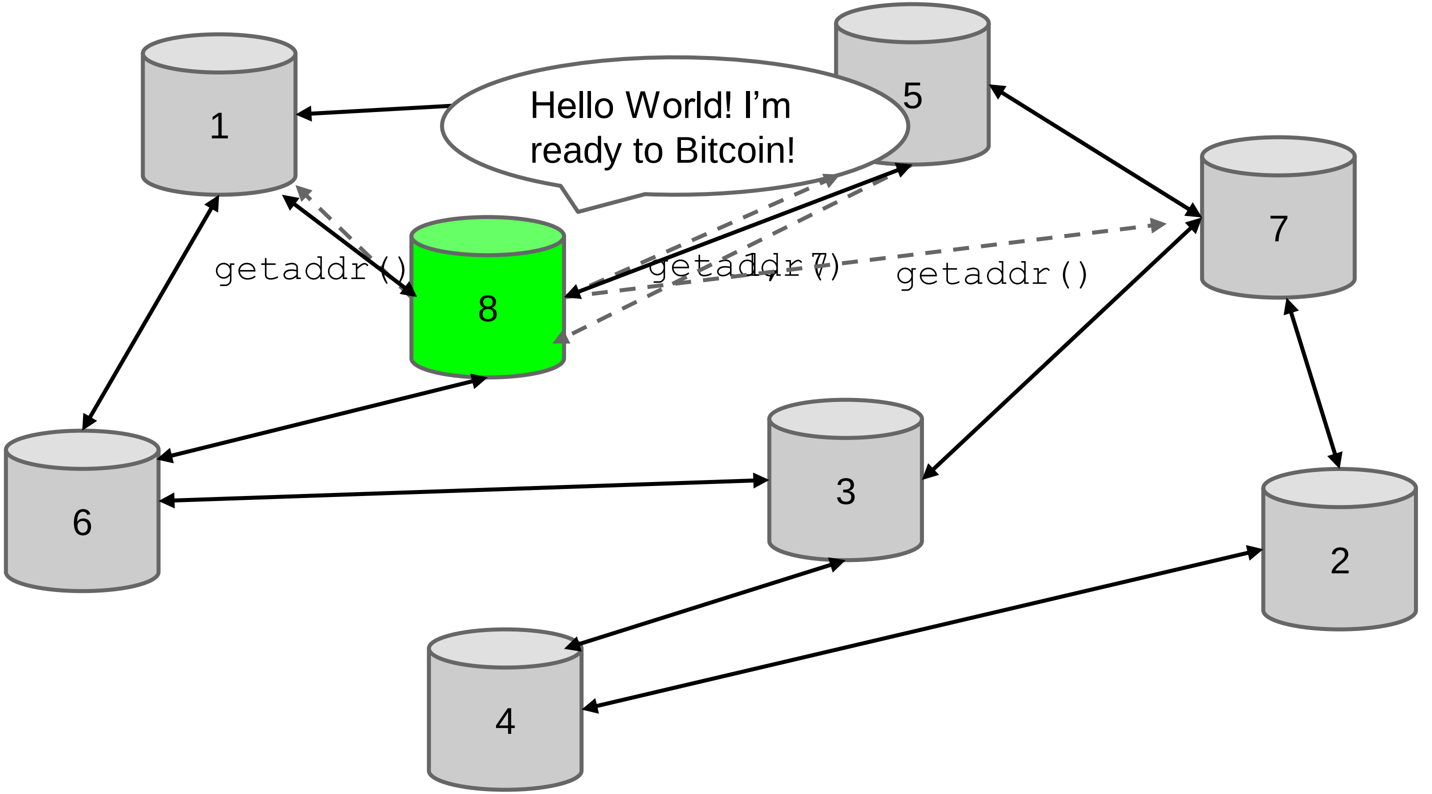
“Smart contracts”

The Bitcoin network

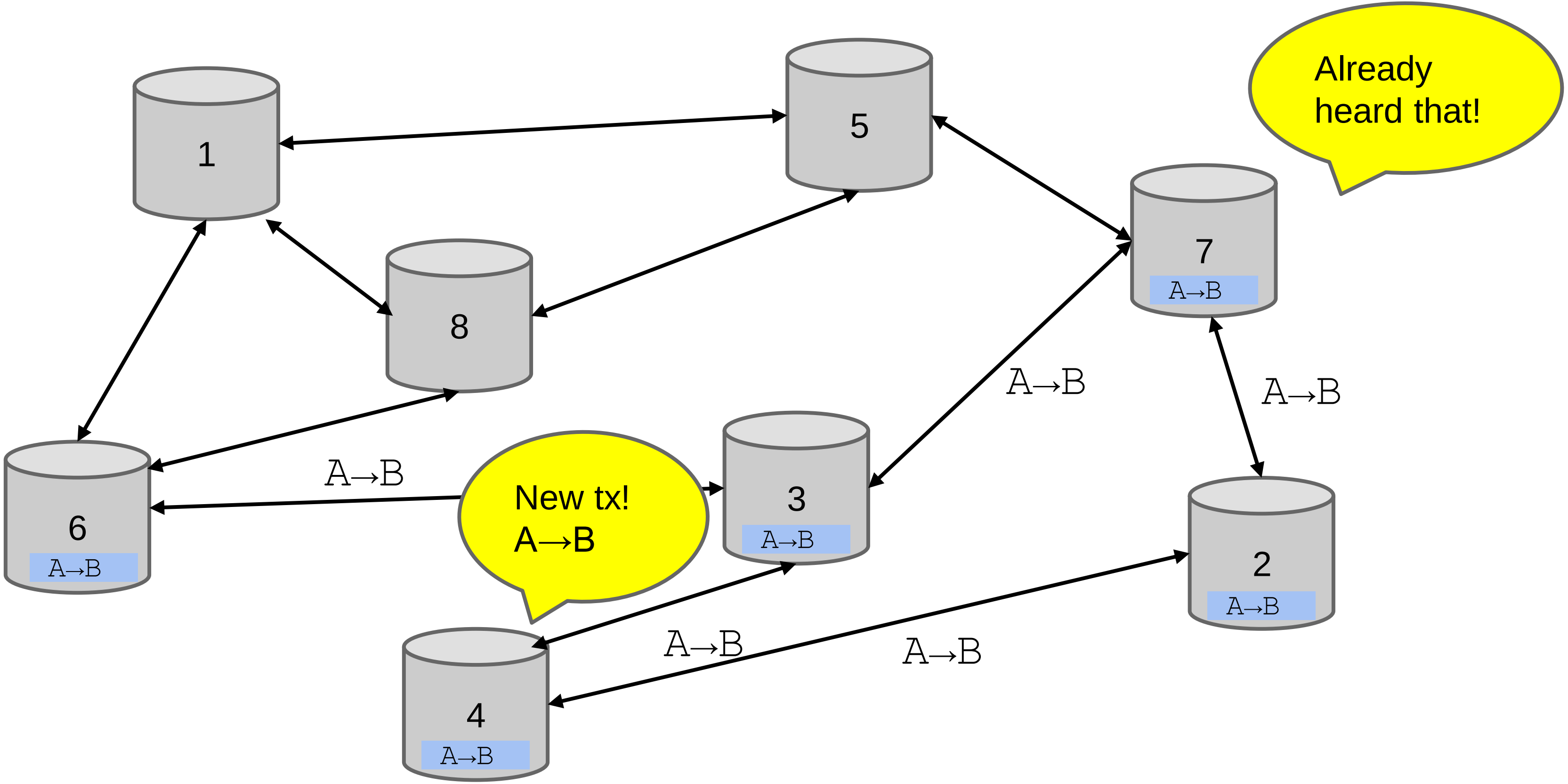
Bitcoin P2P network

- Ad-hoc protocol (runs on TCP port 8333)
- Ad-hoc network with random topology
- All nodes are equal
- New nodes can join at any time
  - Network Changes over time - dynamic
- No explict way to leave network
  - Forget non-responding nodes after 3 hr

Joining the Bitcoin P2P network



Transaction propagation (flooding)

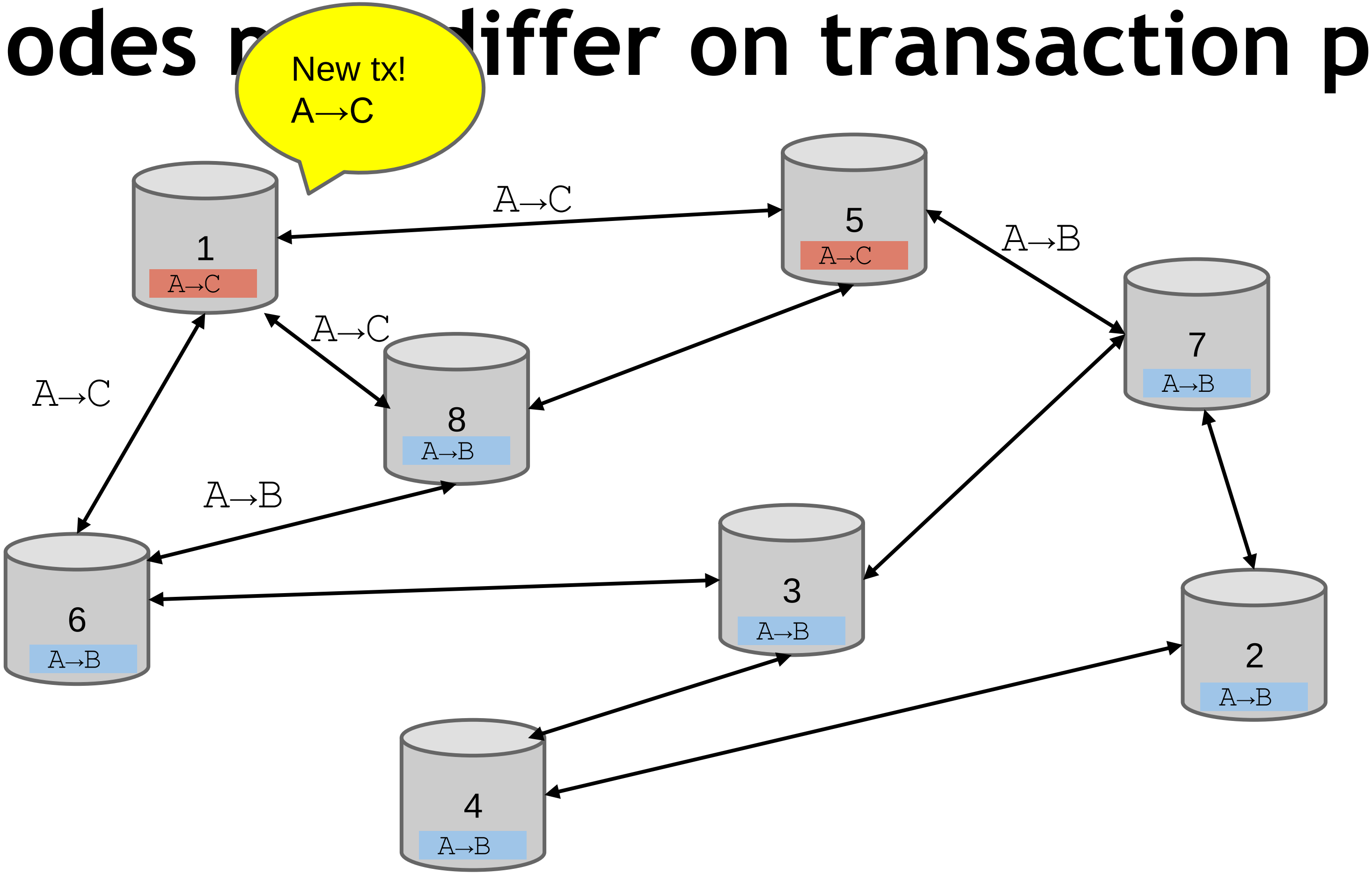


Should I relay a proposed transaction?

- Transaction valid with current block chain
- (default) script matches a whitelist
  - Avoid unusual scripts
- Haven’t seen before
  - Avoid infinite loops
- Doesn’t conflict with others I’ve relayed
  - Avoid double-spends

Sanity checks only...
Some nodes may ignore them!

Nodes m differ on transaction pool



Race conditions

Transactions or blocks may *conflict*

- Default behavior: accept what you hear first
- Network position matters
- Miners may implement other logic!