Blockchain: Smart Contracts

Lecture 4

Recap

(1) **SHA256**: a collision resistant hash function
that outputs 32-byte hash values

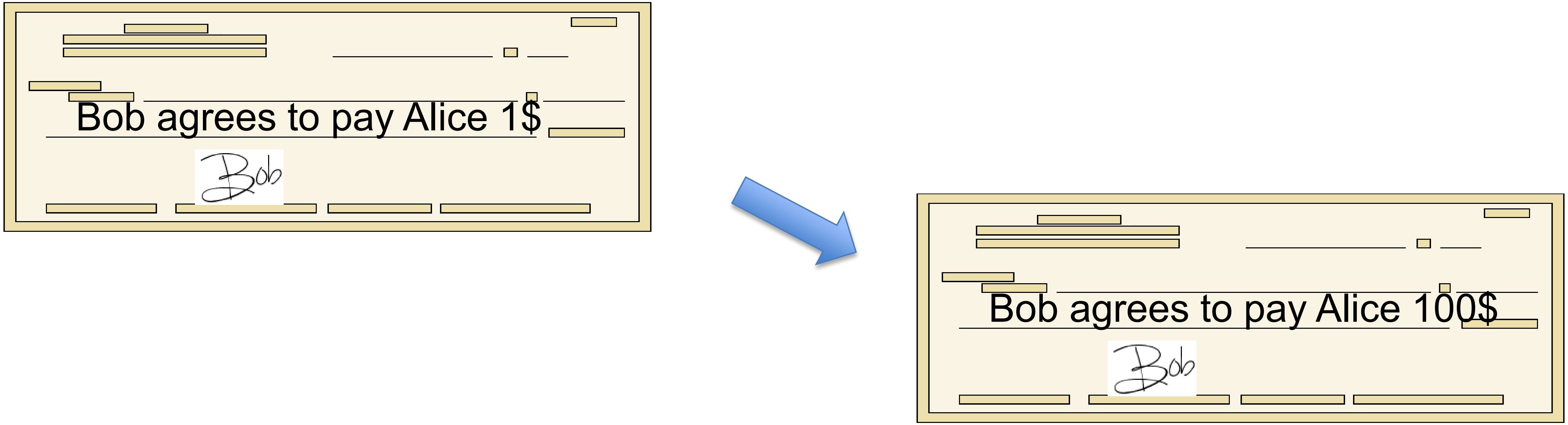
Applications:

- a binding commitment to one value: commit(m) $\rightarrow$ H(m)
  or to a list of values: commit($m_1, \ldots, m_n$) $\rightarrow$ Merkle($m_1, \ldots, m_n$)

- Proof of work with difficulty D:
  given x find y s.t. $H(x, y) < 2^{256}/D$ takes time $O(D)$

Digital Signatures

Physical signatures: bind transaction to author

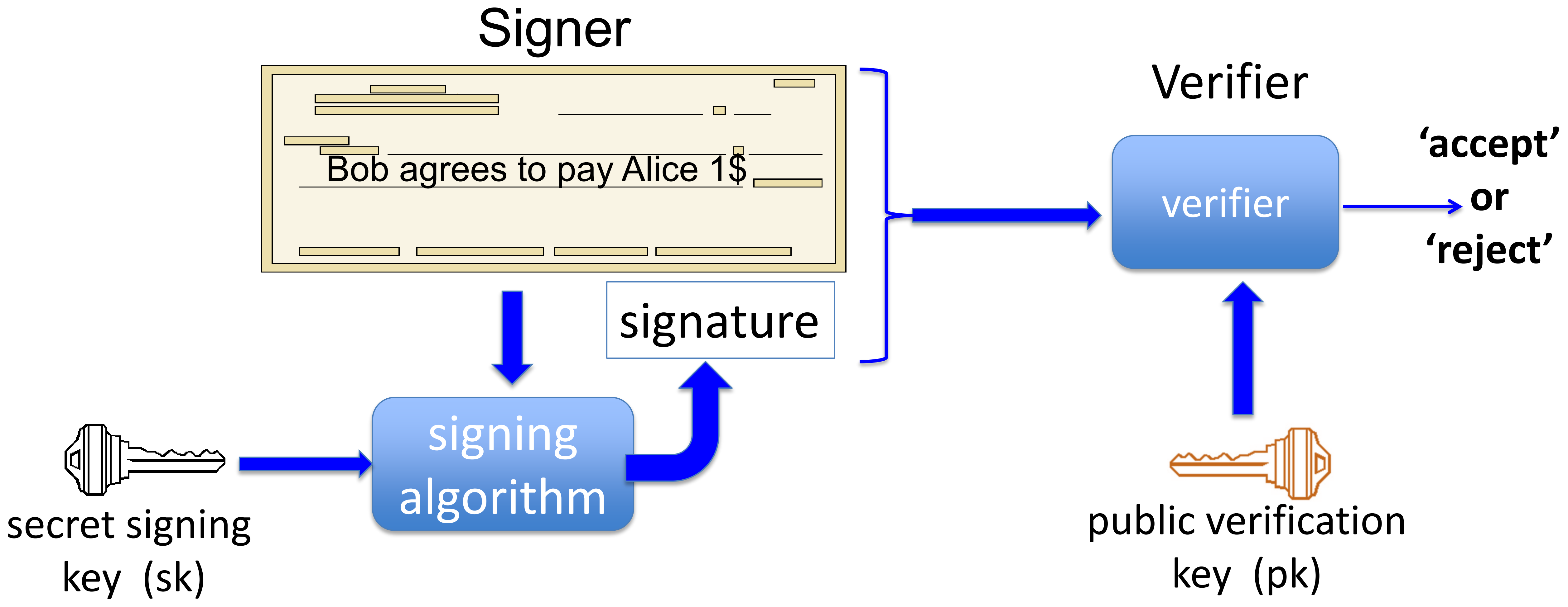


Problem in the digital world:

anyone can copy Bob's signature from one doc to another

Digital signatures

Solution: make signature depend on document



Digital signatures: syntax

Def: a signature scheme is a triple of algorithms:

- **Gen**(): outputs a key pair   (pk, sk)
- **Sign**(sk, msg) outputs sig. σ
- **Verify**(pk, msg, σ) outputs 'accept' or 'reject'

Secure signatures: (informal)

Adversary who sees signatures **on many messages** of his choice, cannot forge a signature on a new message.

Families of signature schemes

1. RSA signatures (old … not used in blockchains):
    - long sigs and public keys (≥256 bytes),   fast to verify

2. Discrete-log signatures: Schnorr and ECDSA   (Bitcoin, Ethereum)
    - short sigs (48 or 64 bytes) and public key (32 bytes)

3. BLS signatures: 48 bytes,   aggregatable,   easy threshold

    (Ethereum 2.0, Chia, Dfinity)

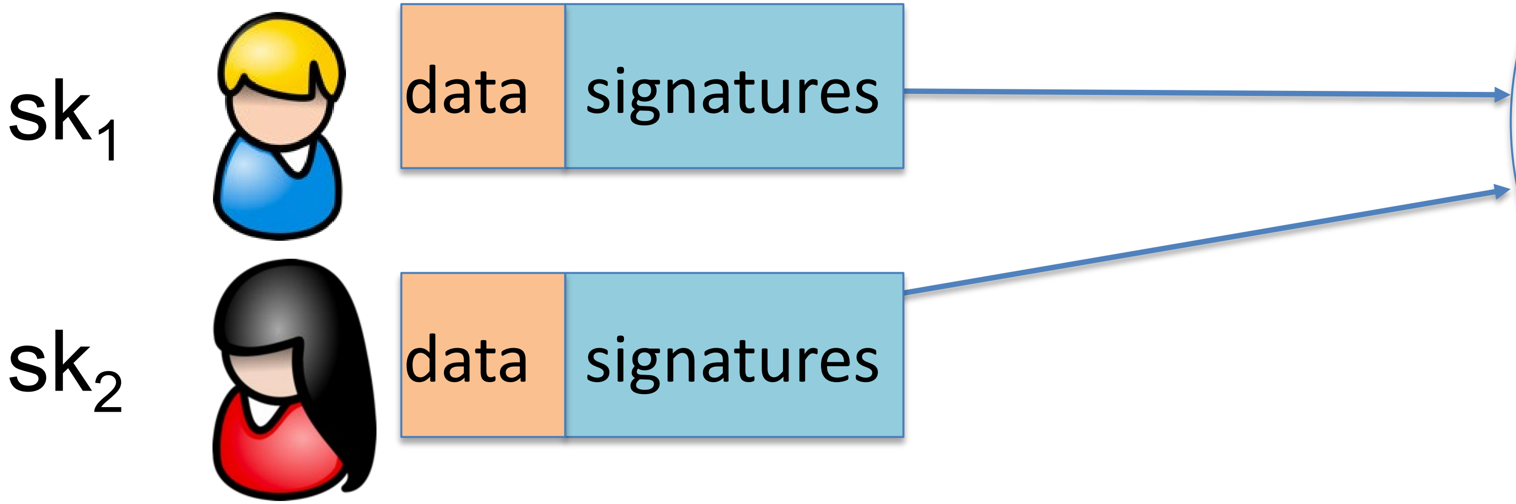
---

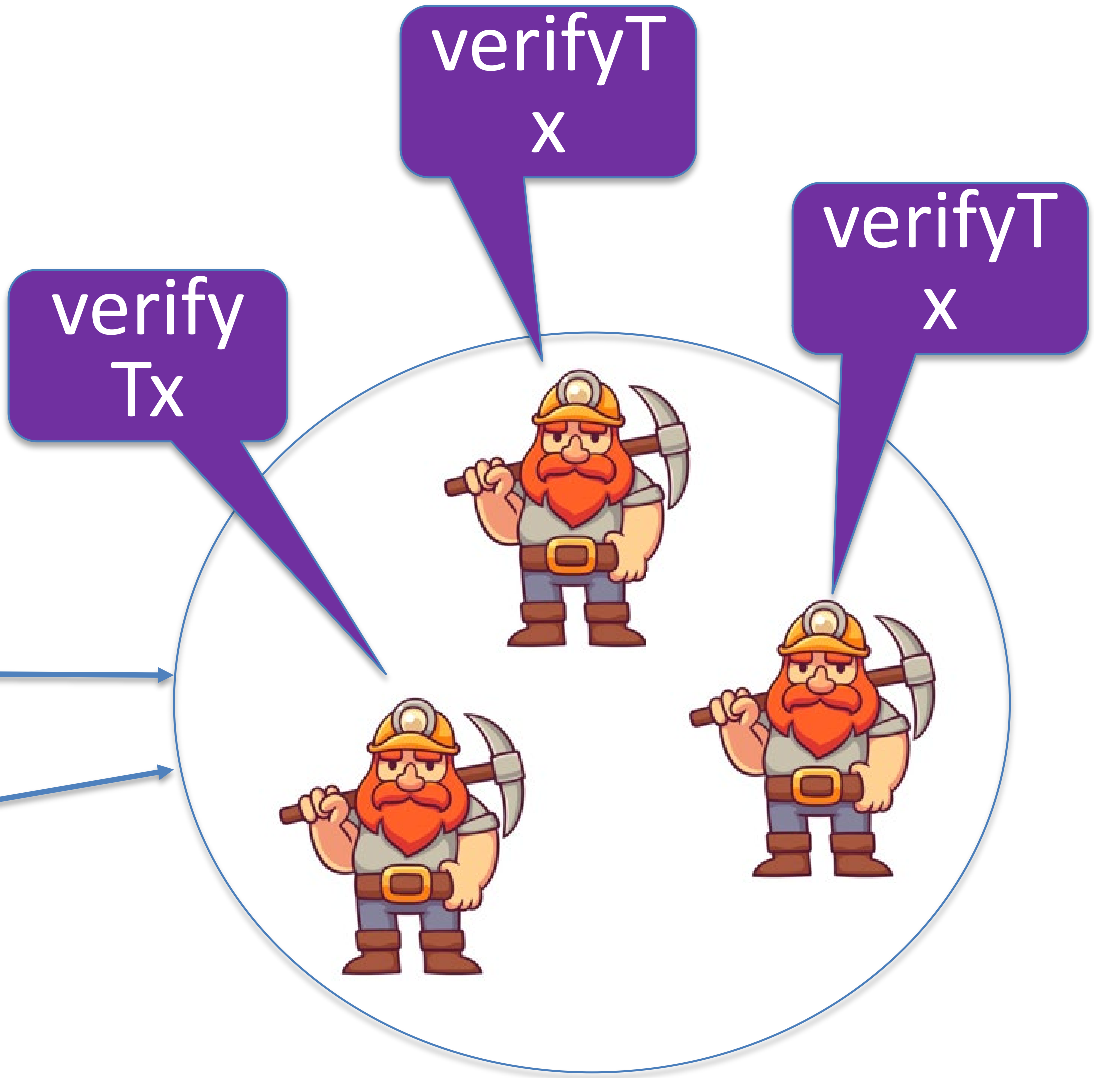
4. Post-quantum signatures: long (≥600 bytes)

Signatures on the blockchain

Signatures are used everywhere:

- ensure Tx authorization,
- governance votes,
- consensus protocol votes.





In summary …

Digital signatures: (Gen, Sign, Verify)

Gen() → (pk, sk),

Sign(sk, m) → σ, Verify(pk, m, σ) → accept/reject

signing key

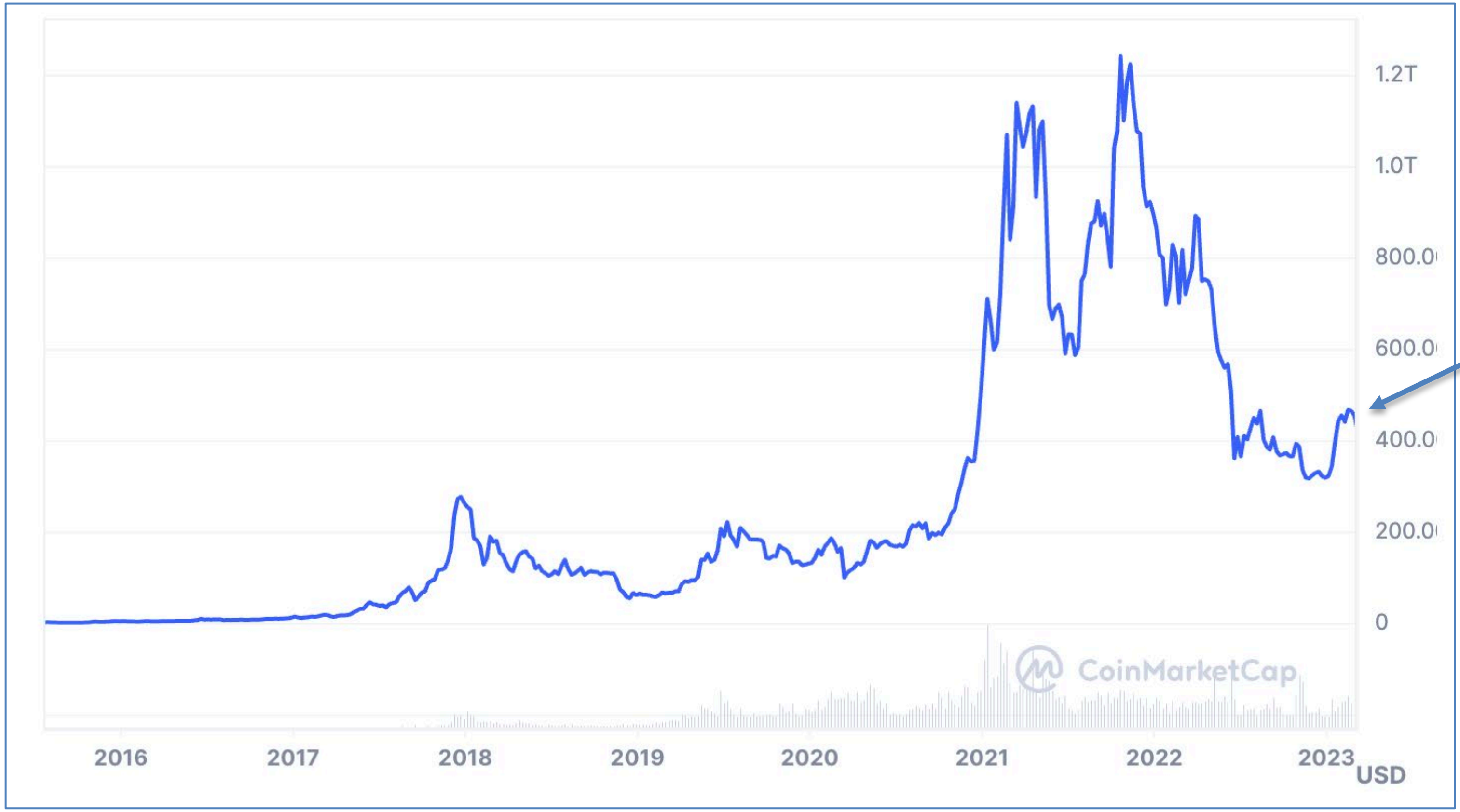
verification key

Bitcoin mechanics

This lecture: Bitcoin mechanics

Oct. 2008: paper by Satoshi Nakamoto
Jan. 2009: Bitcoin network launched

Total market value:



Sep. 2023: $528B

This lecture: Bitcoin mechanics

user facing tools (cloud servers)

applications (DAPPs, smart contracts)

Execution engine (blockchain computer)

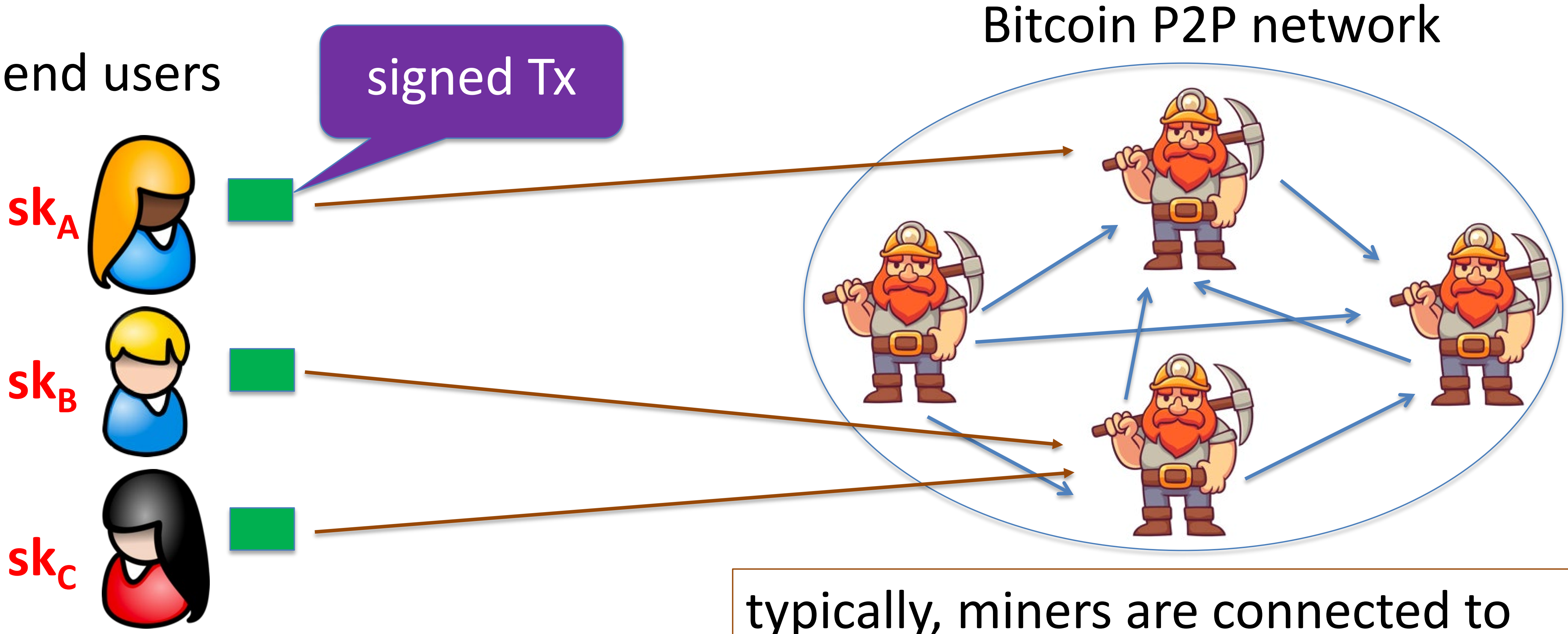
today

Sequencer: orders transactions

Data Availability / Consensus Layer

next week

First: overview of the Bitcoin consensus layer



typically, miners are connected to eight other peers (anyone can join)

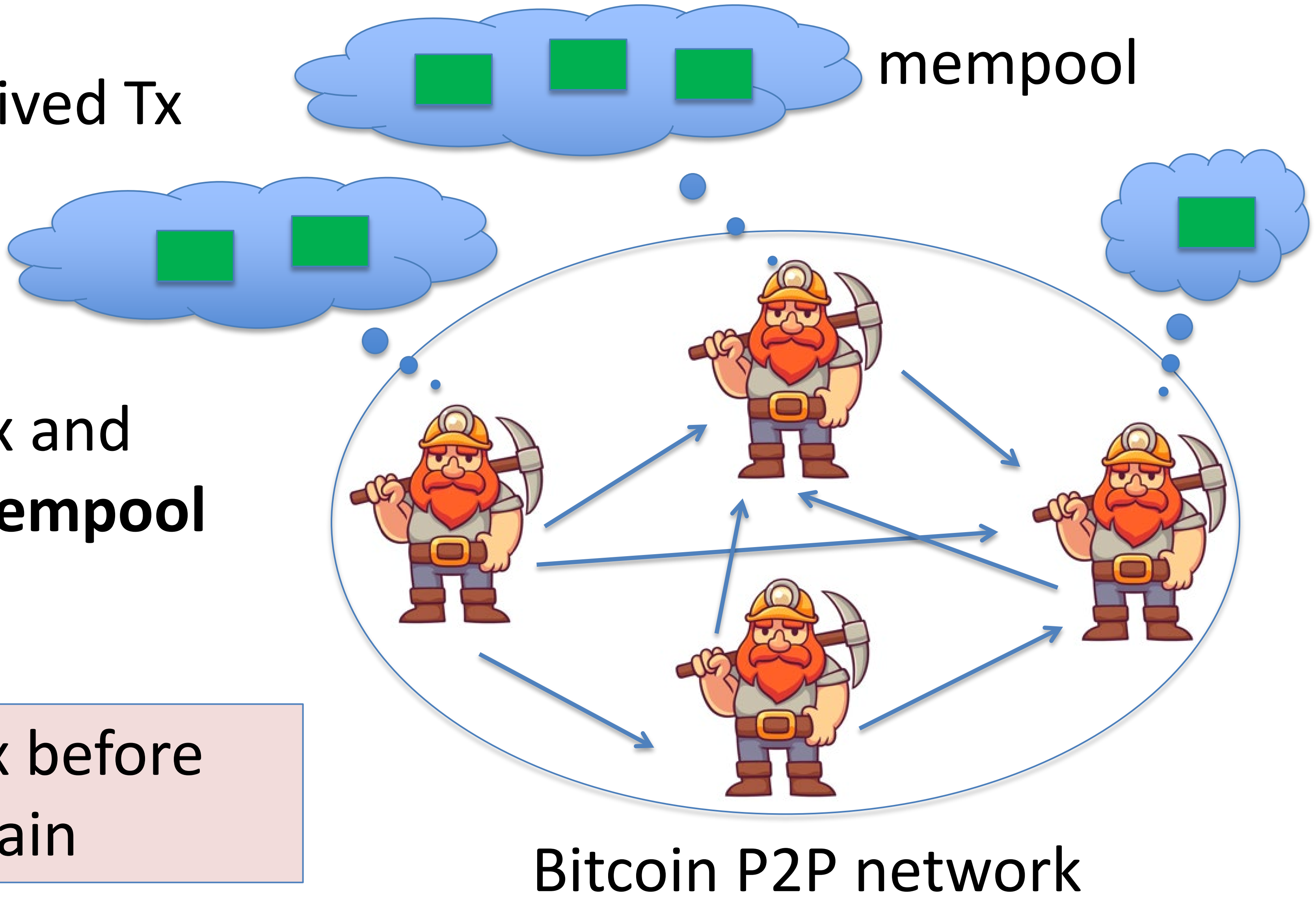
First: overview of the Bitcoin consensus layer

miners broadcast received Tx to the P2P network

every miner:

validates received Tx and stores them in its **mempool** (unconfirmed Tx)

note: miners see all Tx before they are posted on chain

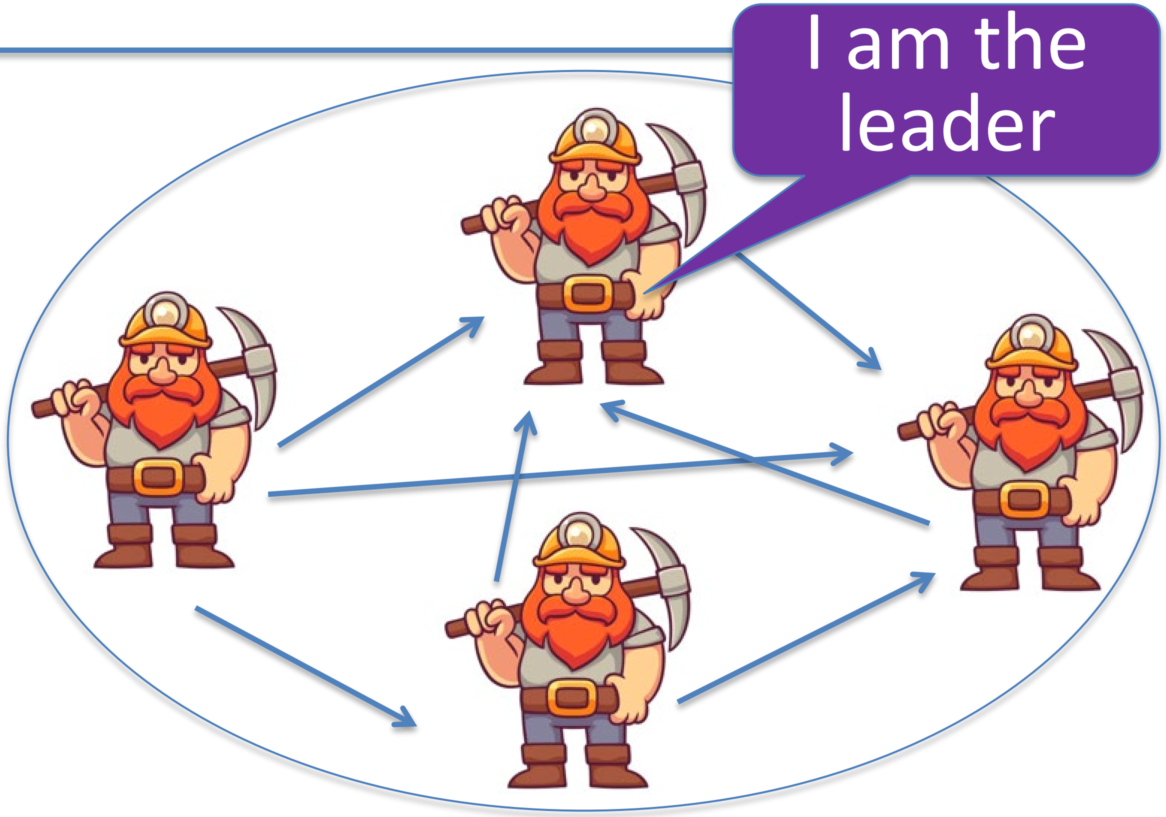


First: overview of the Bitcoin consensus layer

blockchain

<u>Every ≈**10 minutes**</u>:

- Each miner creates a candidate block from Tx in its mempool
- a “random” miner is selected (how: next week), and broadcasts its block to P2P network
- all miners validate new block



Bitcoin P2P network

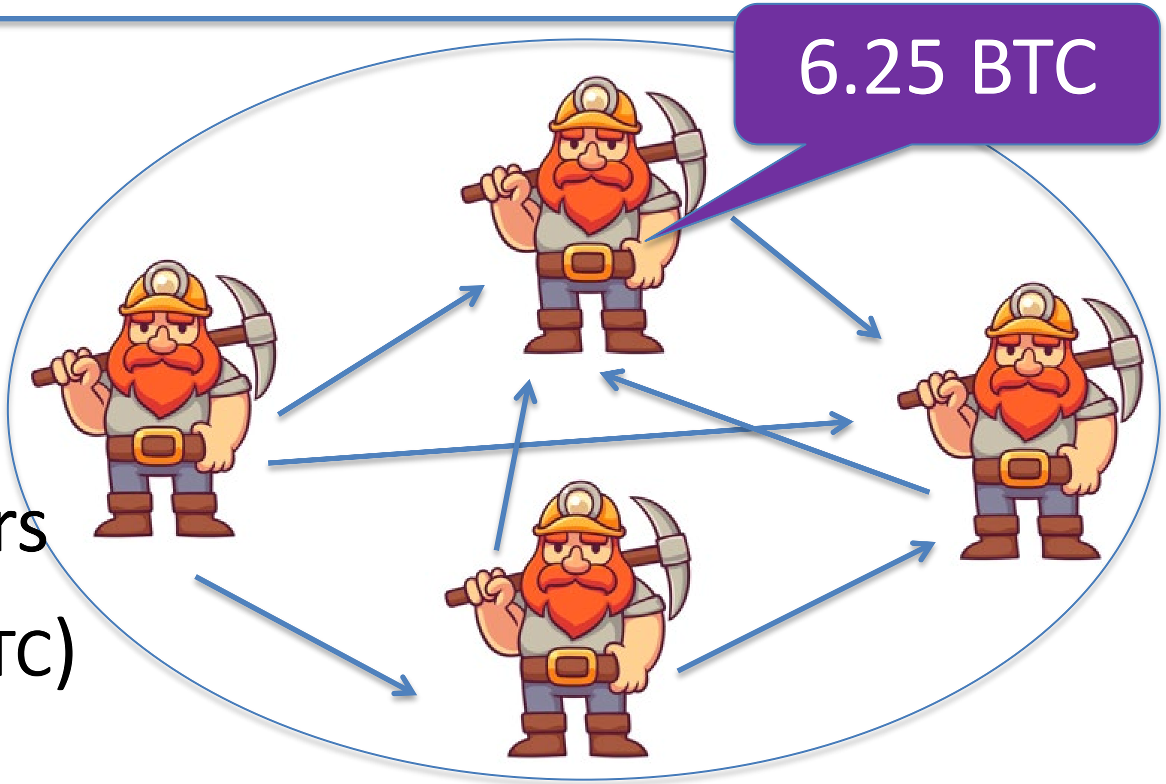
First: overview of the Bitcoin consensus layer

blockchain

Selected miner is paid 6.25 BTC
in **coinbase Tx** (first Tx in the block)

- only way new BTC is created
- block reward halves every four years

  ⇒ max 21M BTC (currently 19.6M BTC)



note: miner chooses order of Tx in block

Properties (very informal)

Safety / Persistence:

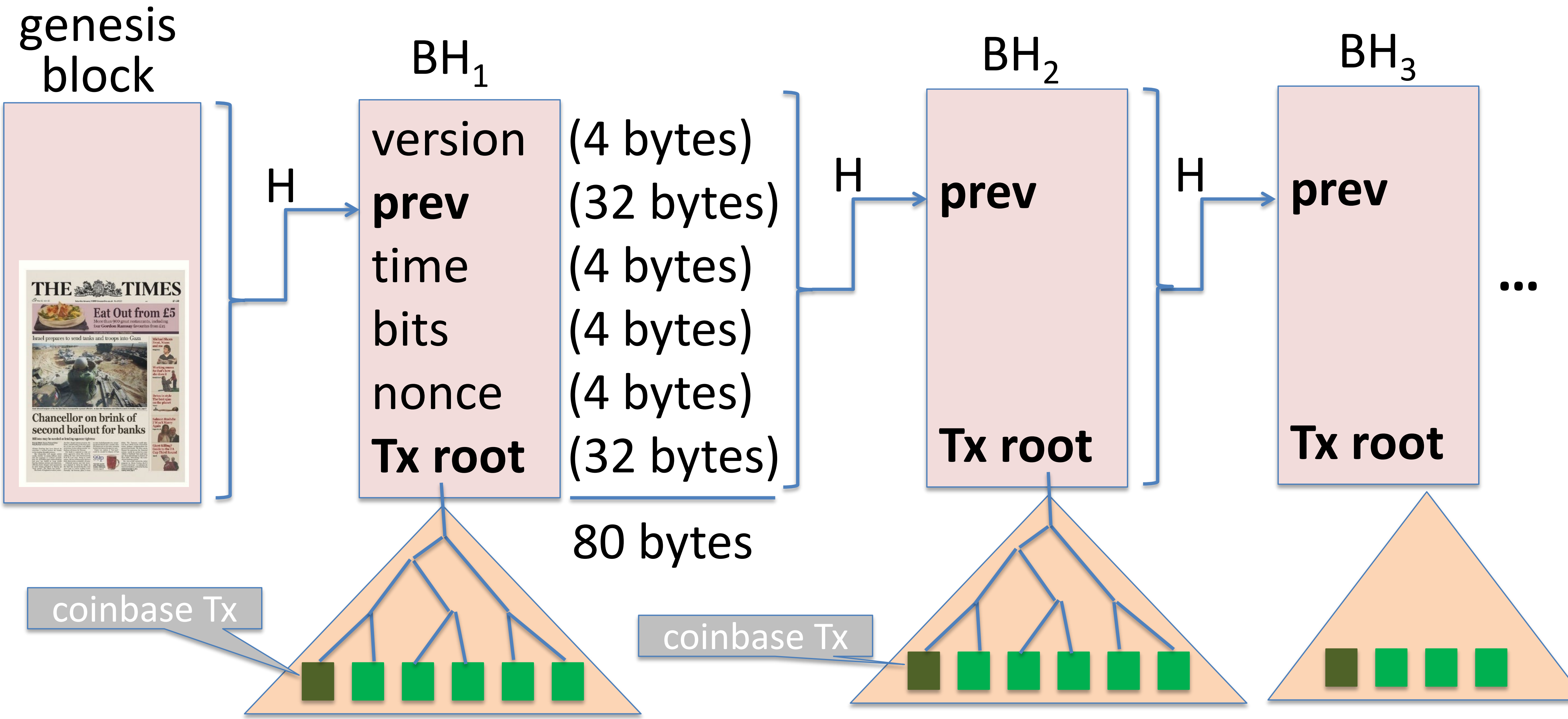
- to remove a block, need to convince 51% of mining power *

Liveness:

- to block a Tx from being posted, need to convince 51% of mining power **

(some sub 50% censorship attacks, such as feather forks)

Bitcoin blockchain: a sequence of block headers, 80 bytes each



Bitcoin blockchain: a sequence of block headers, 80 bytes each

time: time miner assembled the block. Self reported.
(block rejected if too far in past or future)

bits: proof of work difficulty
nonce: proof of work solution

for choosing a leader (next week)

Merkle tree: payer can give a short proof that Tx is in the block

new block every ≈10 minutes.

An example

Tx data

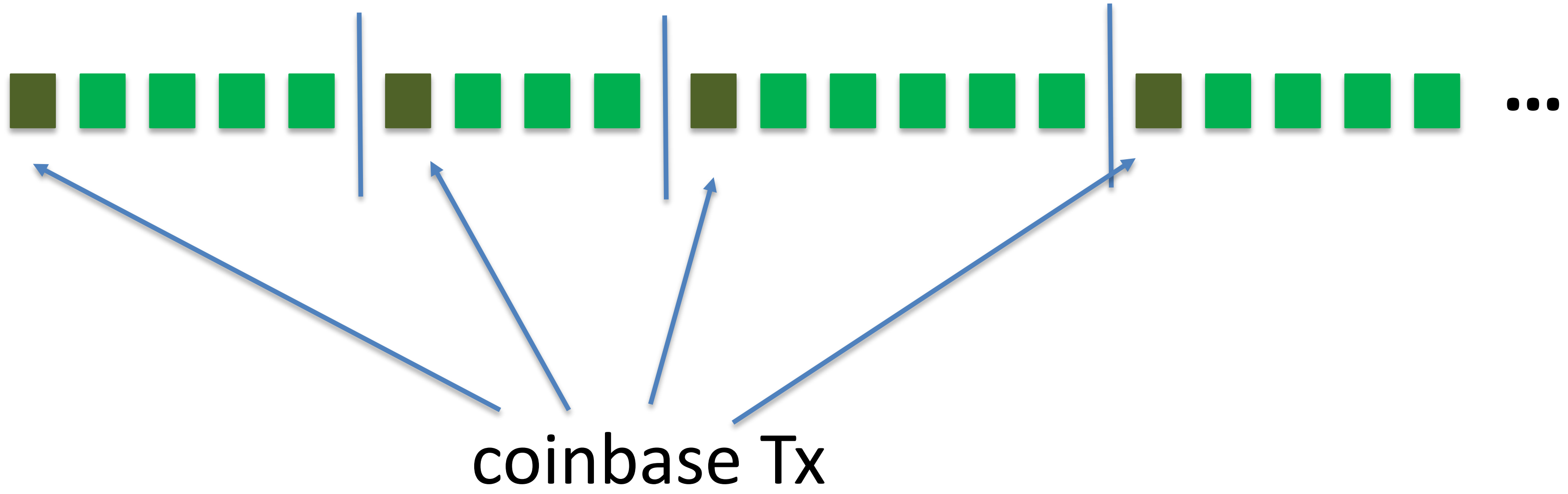
Height	Mined	Miner	Size	#Tx
648494	17 minutes	Unknown	1,308,663 bytes	1855
648493	20 minutes	SlushPool	1,317,436 bytes	2826
648492	59 minutes	Unknown	1,186,609 bytes	1128
648491	1 hour	Unknown	1,310,554 bytes	2774
648490	1 hour	Unknown	1,145,491 bytes	2075
648489	1 hour	Poolin	1,359,224 bytes	2622

Block 648493

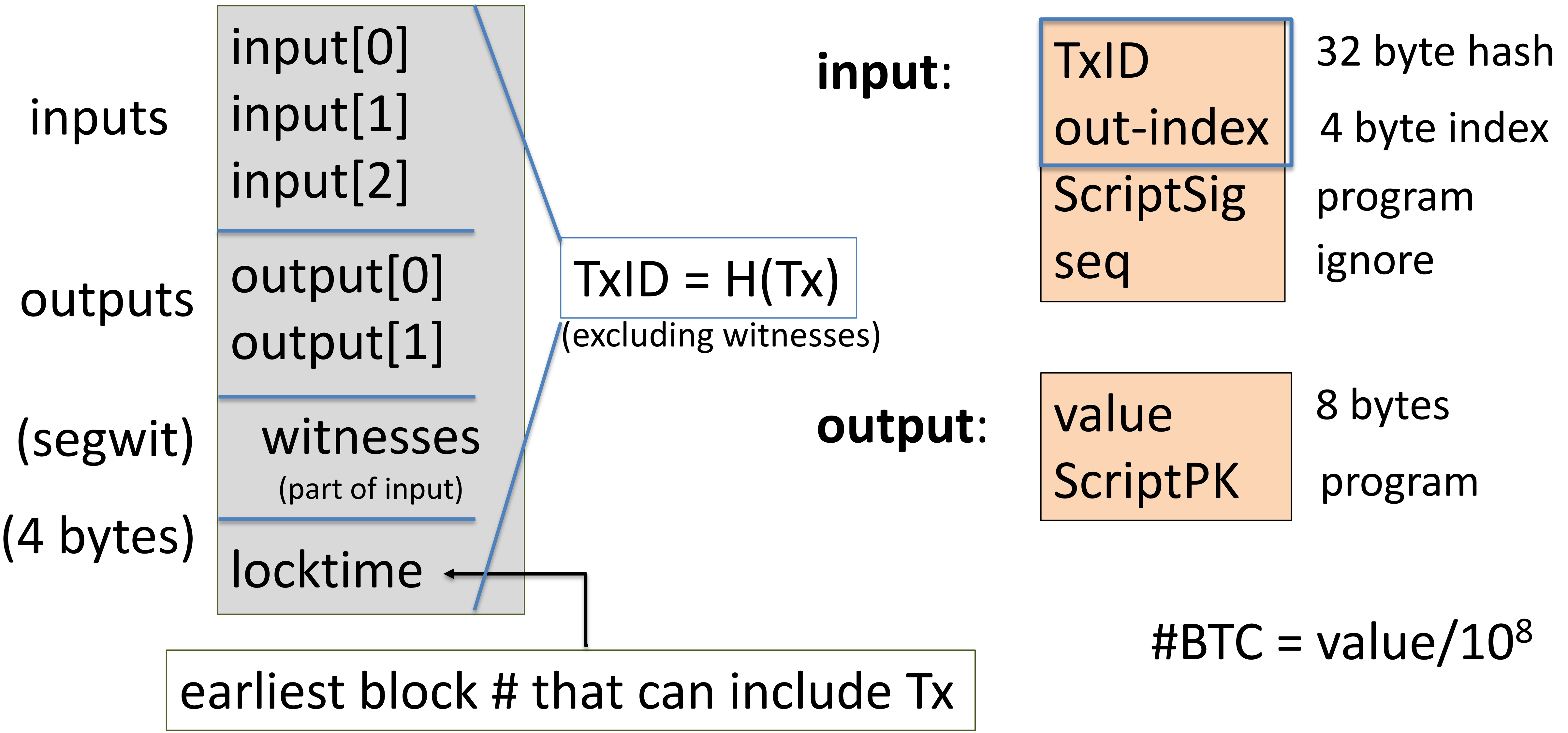
Timestamp	2020-09-15 17:25	
Height	648493	
Miner	SlushPool	(from coinbase Tx)
Number of Transactions	2,826	
Difficulty (D)	17,345,997,805,929.09	(adjusts every two weeks)
Merkle root	350cbb917c918774c93e945b960a2b3ac1c8d448c2e67839223bbcf595baff89	
Transaction Volume	11256.14250596 BTC	
Block Reward	6.25000000 BTC	
Fee Reward	0.89047154 BTC	(Tx fees given to miner in coinbase Tx)

This lecture

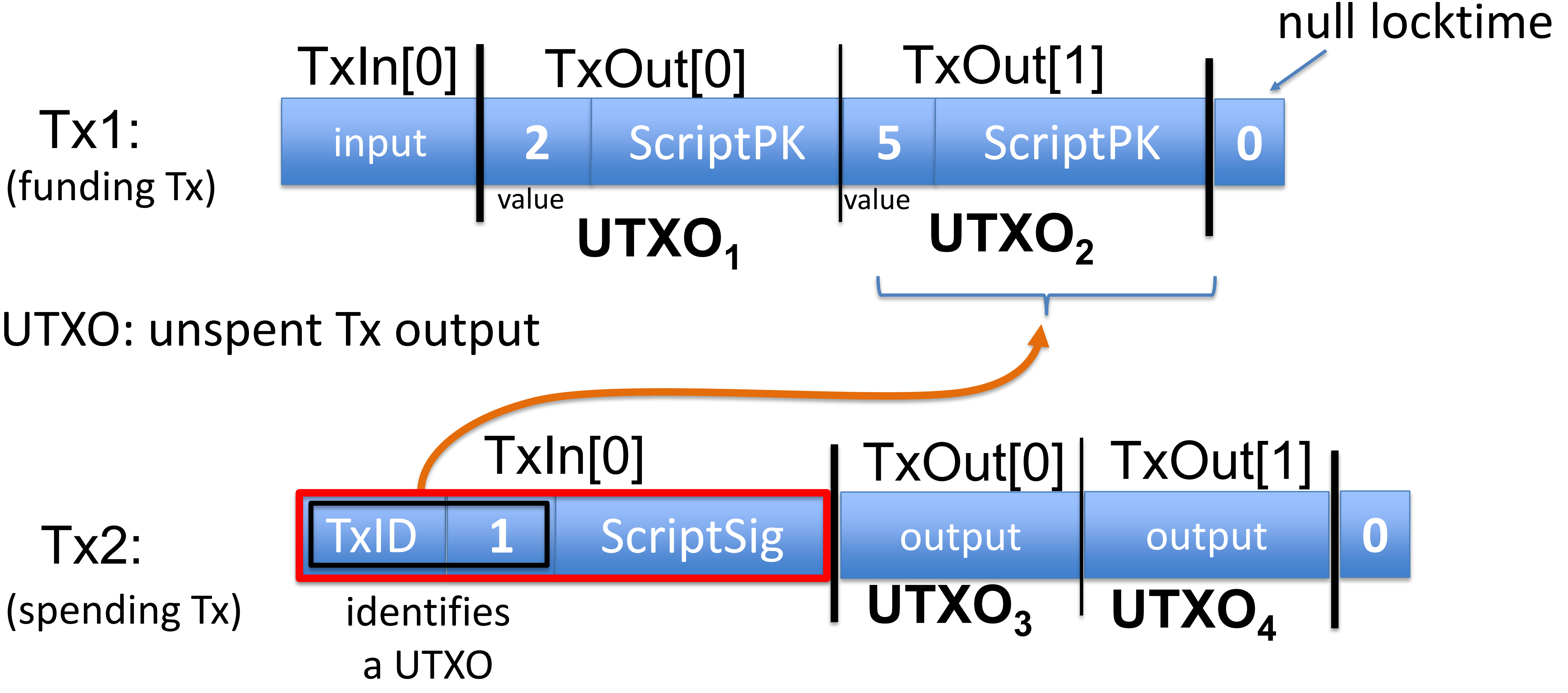
View the blockchain as a sequence of Tx (append-only)



Tx structure (non-coinbase)

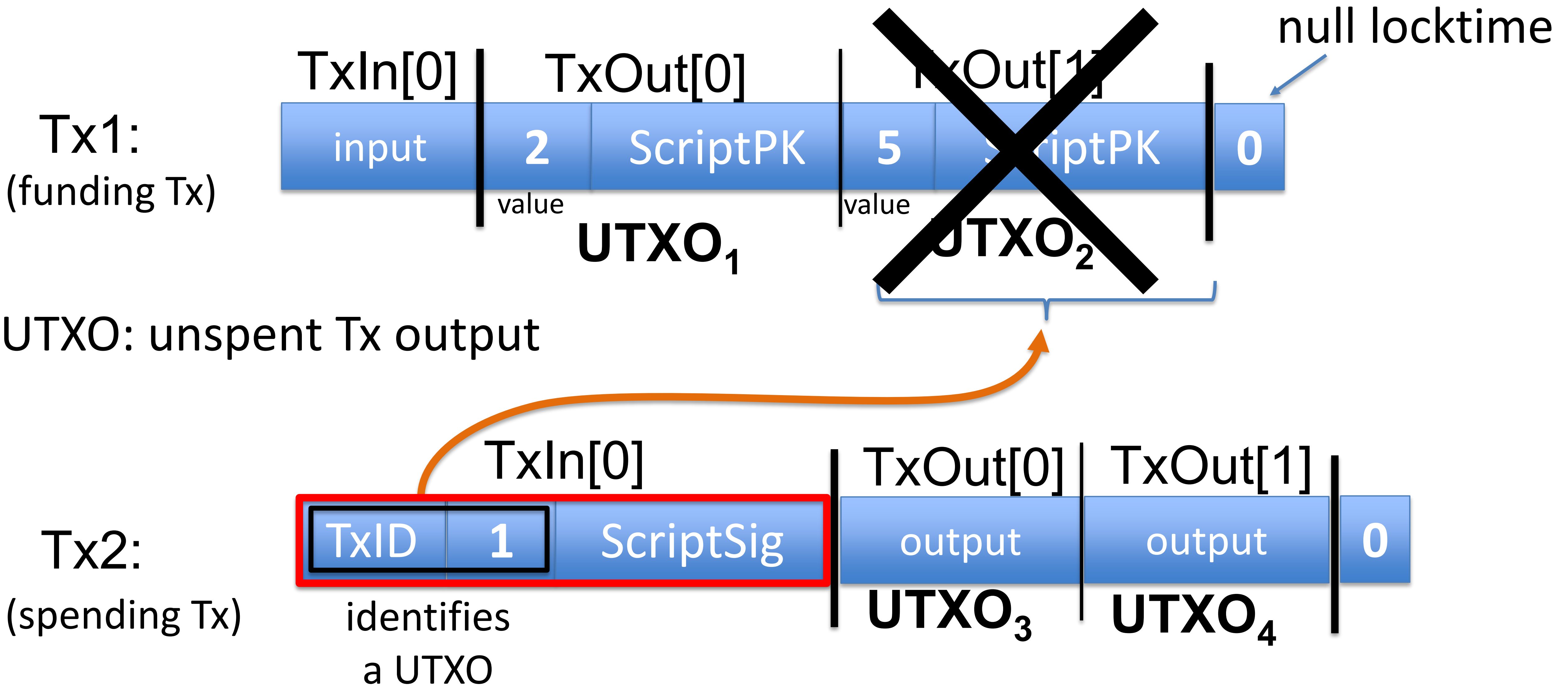


Example



null locktime
Tx1:
(funding Tx)
TxIn[0]
TxOut[0]
TxOut[1]
input
2
ScriptPK
5
ScriptPK
0
value
value
UTXO1
UTXO2
UTXO: unspent Tx output
Tx2:
(spending Tx)
TxIn[0]
TxOut[0]
TxOut[1]
TxID
1
ScriptSig
output
output
0
identifies
a UTXO
UTXO3
UTXO4

Example



null locktime
Tx1:
(funding Tx)
TxIn[0]
TxOut[0]
input
2
ScriptPK
5
0
value
value
UTXO$_1$
UTXO: unspent Tx output
Tx2:
(spending Tx)
TxIn[0]
TxOut[0]
TxOut[1]
TxID
1
ScriptSig
output
output
0
identifies
a UTXO
UTXO$_3$
UTXO$_4$

Validating Tx2

program from funding Tx:
under what conditions
can UTXO be spent

Miners check (for each input):

1. The program $\boxed{\text{ScriptSig | ScriptPK}}$ returns true

2. $\boxed{\text{TxID | index}}$ is in the current UTXO set

3. sum input values $\geq$ sum output values

After Tx2 is posted, miners remove $UTXO_2$ from UTXO set

An example (block 648493) [2826 Tx]

COINBASE (Newly Generated Coins)			1CK6KHY6MHgYvmRQ4PAafKYDrg1ejbH1cE	7.14047154 BTC
			OP_RETURN	0.00000000 BTC
Tx0			OP_RETURN	0.00000000 BTC
0.00000000 BTC			6.25 + Tx fees =	7.14047154 BTC

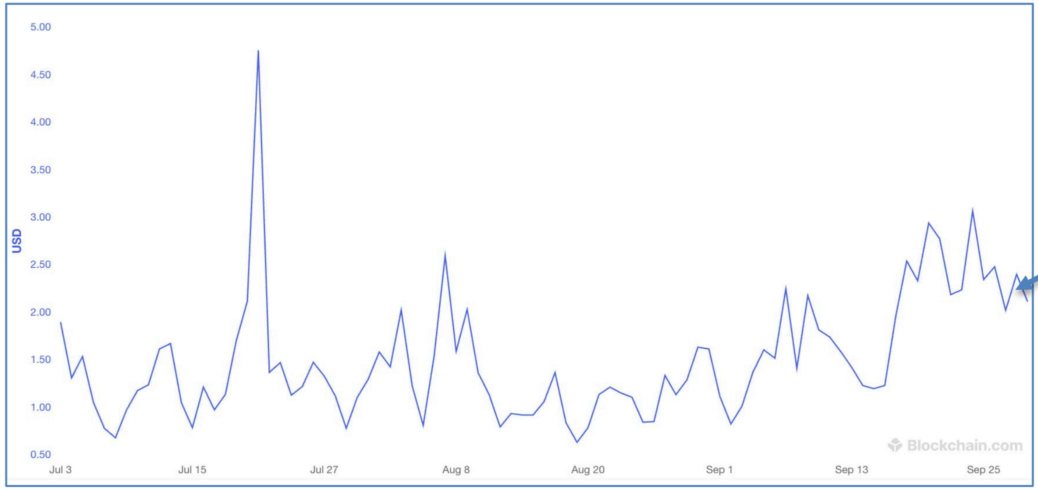
3PuJbxJS1pKxf8EdVR18yBkD1fPAbgUtyw	0.72333974 BTC		1E5Ao1VUnA5BhffvXf2Xmud6avUgwkFnJv	0.00917379 BTC
input	(input UTXO value)		bc1qr8k3e0vx06lpu3j7m858pa2ak9tyr56ttwvefk	0.61504199 BTC
			bc1qdrxve8kua3yz5dgx6wf3u95ngh0d3e648...	0.09290152 BTC
Tx1			14ZhjuXpQ5jCDjtAy7ZMu3hfEQCWewzLw7	0.00616444 BTC
0.00005800 BTC (Tx fee)			outputs	0.72328174 BTC

17MWze4Z1uP1jnvqvj7SAnGtxcoVq11H8A	0.05000000 BTC		3G3C2RFQ8gsf77EQpdR4ZReChWFKEHhxVU	0.04808000 BTC
Tx2				
0.00192000 BTC (Tx fee)				0.04808000 BTC

sum of fees in block added to coinbase Tx

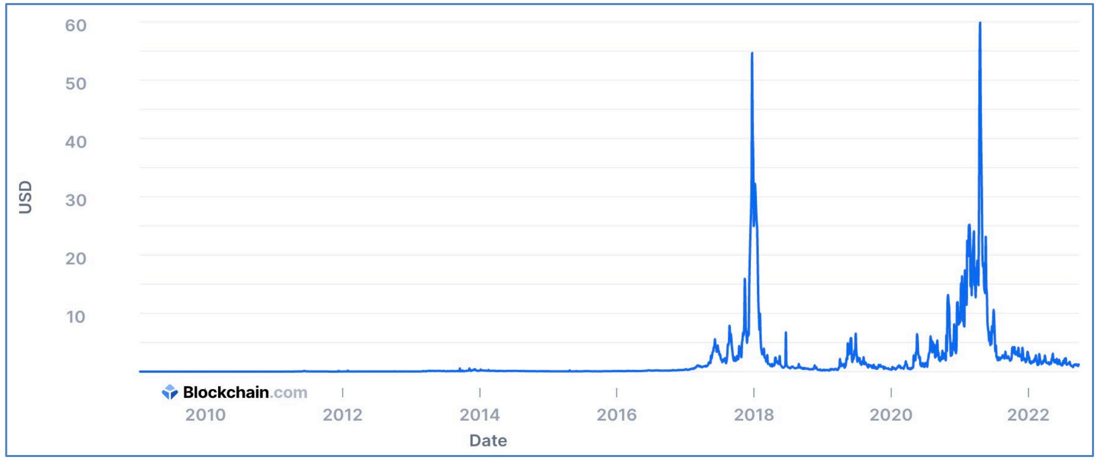
Tx fees

Bitcoin average Tx fees in USD (last 6 months, sep. 2023)

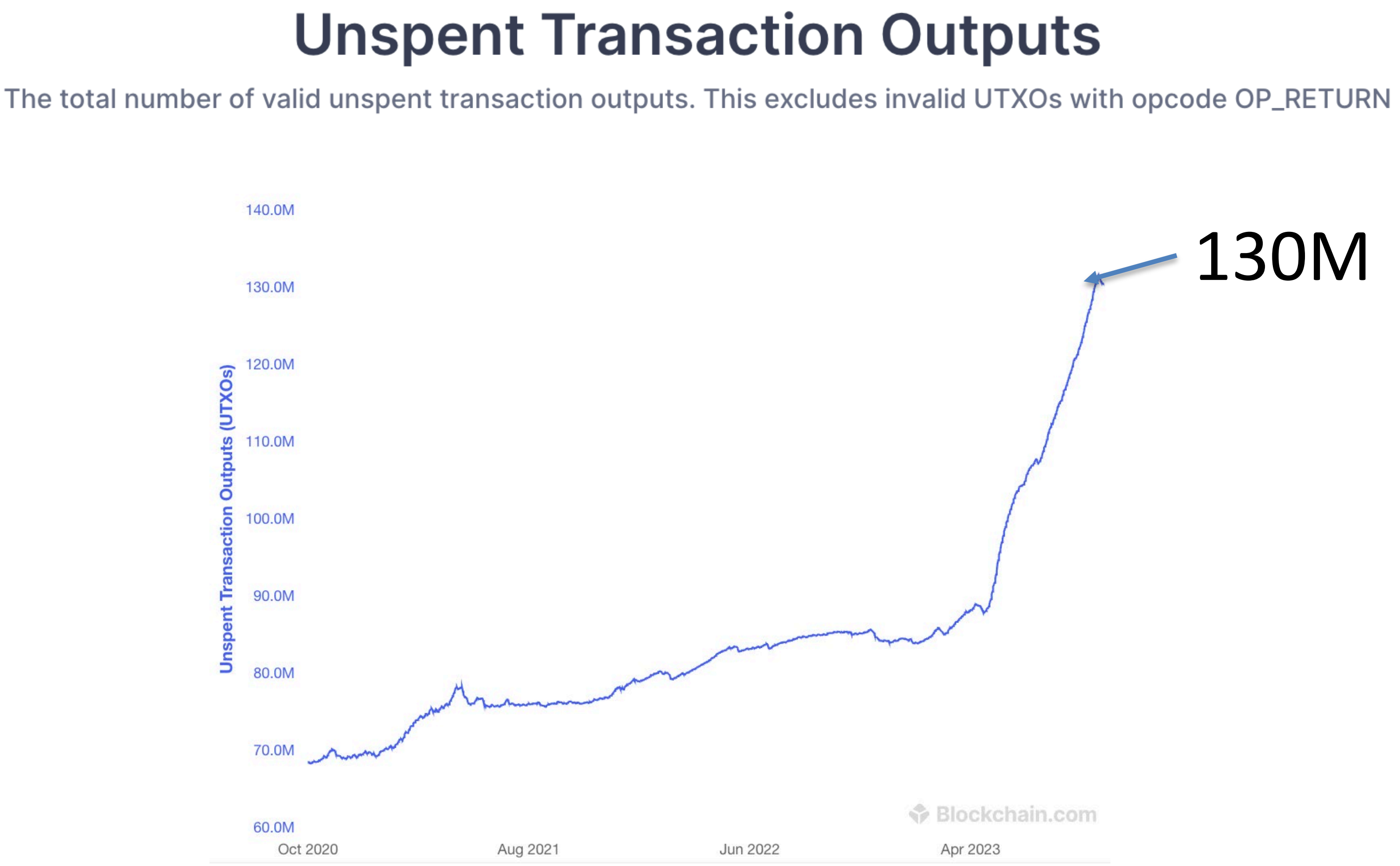


$2.11

Bitcoin average Tx fees in USD (all time)



All value in Bitcoin is held in UTXOs



Sep. 2023: miners need to store ≈130M UTXOs in memory

Focusing on Tx2: TxInp[0]

from UTXO
(Bitcoin script)

Value	0.05000000 BTC
Pkscript	OP_DUP OP_HASH160 45b21c8a0cb687d563342b6c729d31dab58e3a4e OP_EQUALVERIFY OP_CHECKSIG
Sigscript	304402205846cace0d73de82dfbdeba4d65b9856d7c1b1730eb401cf4906b2401a69bdc90220589d36d36be64e774c8796b96c011f29768191abeb7f56ba20ffb0351280860c01 03557c228b080703d52d72ead1bd93fc72f45c4ddb4c2b7a20c458e2d069c8dd9e

from TxInp[0]

Bitcoin Script

A stack machine. Not Turing Complete: no loops.

Quick survey of op codes:

1. **OP_TRUE** (OP_1), **OP_2**, …, **OP_16**: push value onto stack

   81 82 96

2. **OP_DUP**: push top of stack onto stack

   118

Bitcoin Script

3. control:

99 **OP_IF** <statements> **OP_ELSE** <statements> **OP_ENDIF**

105 **OP_VERIFY**: abort fail if top = false

106 **OP_RETURN**: abort and fail

what is this for? ScriptPK = [OP_RETURN, <data>]

136 **OP_EQVERIFY**: pop, pop, abort fail if not equal

Bitcoin Script

4. arithmetic:

   OP_ADD, **OP_SUB**, **OP_AND**, …: pop two items, add, push

5. crypto:

   OP_SHA256: pop, hash, push

   OP_CHECKSIG: pop pk, pop sig, verify sig. on Tx, push 0 or 1

6. Time: **OP_CheckLockTimeVerify** (CLTV):

   fail if value at the top of stack > Tx locktime value.

   usage: UTXO can specify min-time when it can be spent

Example: a common script

<sig> <pk>	**DUP HASH256** <pkhash> **EQVERIFY CHECKSIG**

stack:	empty	init
	<sig> <pk>	push values
	<sig> <pk> <pk>	**DUP**
	<sig> <pk> <hash>	**HASH256**
	<sig> <pk> <hash> <pkhash>	push value
	<sig> <pk>	**EQVERIFY**
	1	**CHECKSIG**
		verify(pk, Tx, sig)

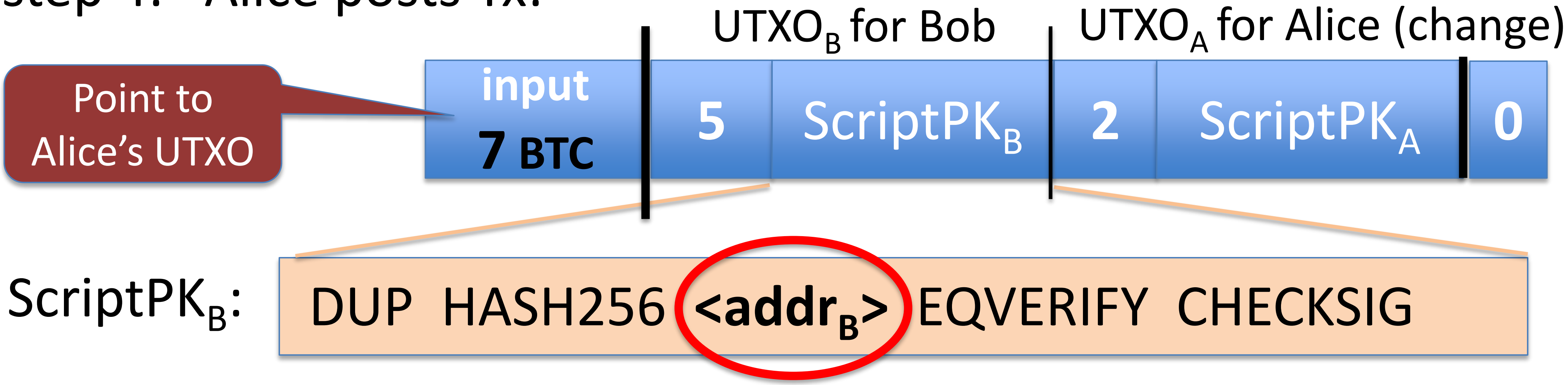
⇒ successful termination

Transaction types: (1) P2PKH

pay to public key hash

Alice want to pay Bob 5 BTC:

- step 1: Bob generates sig key pair $(pk_B, sk_B) \leftarrow Gen()$
- step 2: Bob computes his Bitcoin address as $addr_B \leftarrow H(pk_B)$
- step 3: Bob sends $addr_B$ to Alice
- step 4: Alice posts Tx:



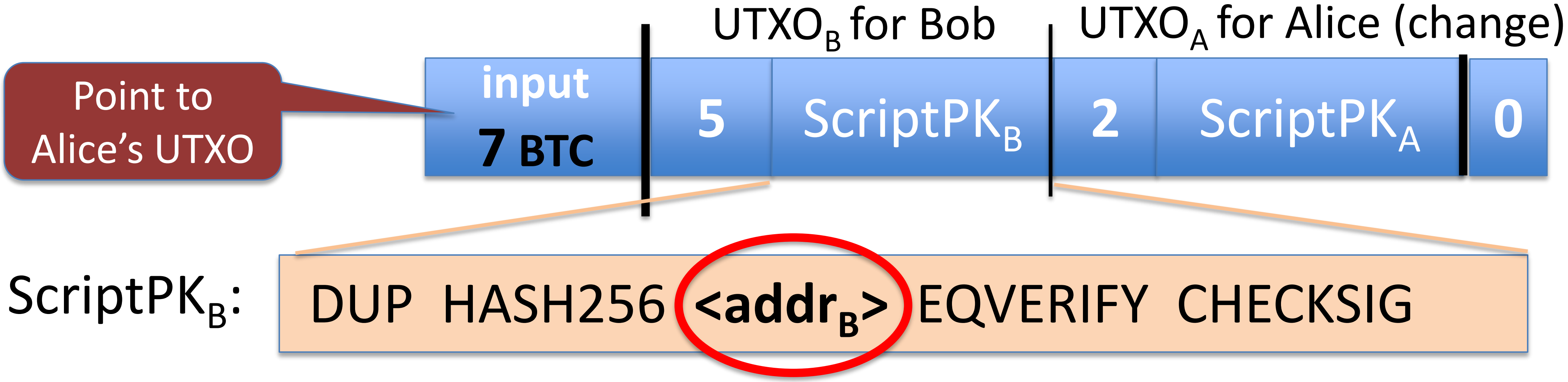
Transaction types: (1) P2PKH

pay to public key hash

"input" contains ScriptSig that authorizes spending Alice's UTXO

- example: ScriptSig contains Alice's signature on Tx

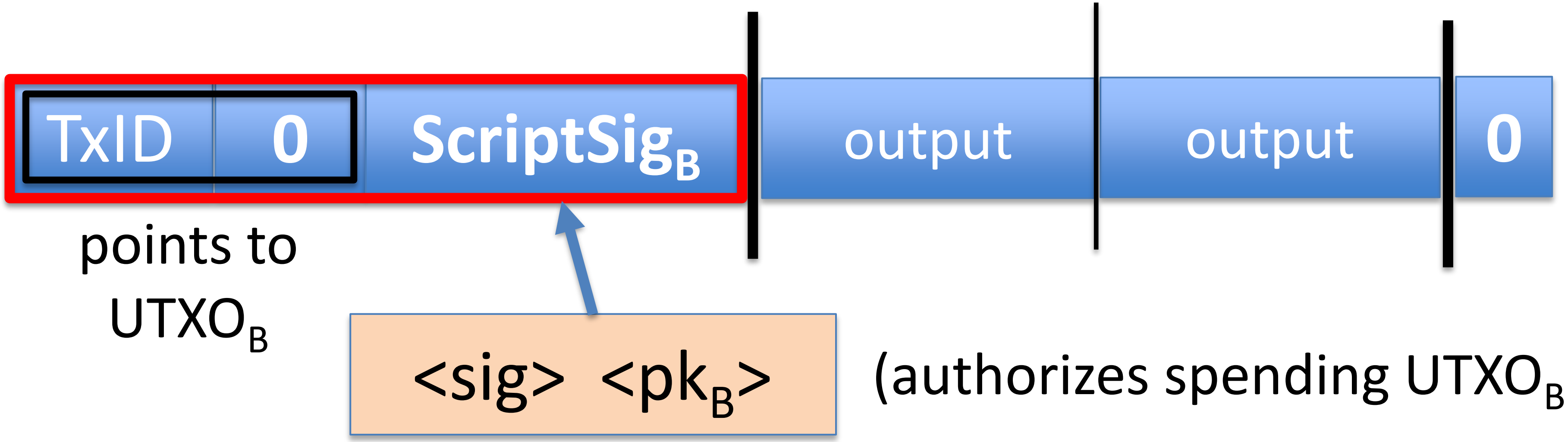
  $\Longrightarrow$ miners cannot change ScriptPK$_B$ (will invalidate Alice's signature)



Transaction types: (1) P2PKH

Later, when Bob wants to spend his UTXO: create a Tx_{spend}

Tx_{spend}:



(authorizes spending $UTXO_B$)

<sig> = Sign(sk_B, Tx) where Tx = (Tx_{spend} excluding all ScriptSigs) (SIGHASH_ALL)

Miners validate that $ScriptSig_B$ | $ScriptPK_B$ returns true

P2PKH: comments

- Alice specifies recipient’s pk in $UTXO_B$

- Recipient’s pk is not revealed until UTXO is spent

  (some security against attacks on pk)

- Miner cannot change <$Addr_B$> and steal funds:

  invalidates Alice’s signature that created $UTXO_B$

Segregated Witness

ECDSA malleability:

Given (m, sig) anyone can create (m, sig’) with sig ≠ sig’

⇒ miner can change sig in Tx and change TxID = SHA256(Tx)

⇒ Tx issuer cannot tell what TxID is, until Tx is posted

⇒ leads to problems and attacks

Segregated witness: signature is moved to witness field in Tx

TxID = Hash(Tx without witnesses)

Transaction types: (2) P2SH: pay to script hash

(pre SegWit in 2017)

Let's payer specify a redeem script (instead of just pkhash)

Usage: payee publishes hash(redeem script) ⟵ Bitcoint addr.

payer sends funds to that address

ScriptPK in UTXO: HASH160 <H(redeem script)> EQUAL

ScriptSig to spend: <sig_1> <sig_2> … <sig_n> <redeem script>

payer can specify complex conditions for when UTXO can be spent

P2SH

Miner verifies:

(1) <ScriptSig> ScriptPK = true ⟵ payee gave correct script

(2) ScriptSig = true ⟵ script is satisfied

Example P2SH: multisig

<u>Goal</u>: spending a UTXO requires t-out-of-n signatures

Redeem script for 2-out-of-3: (set by payer)

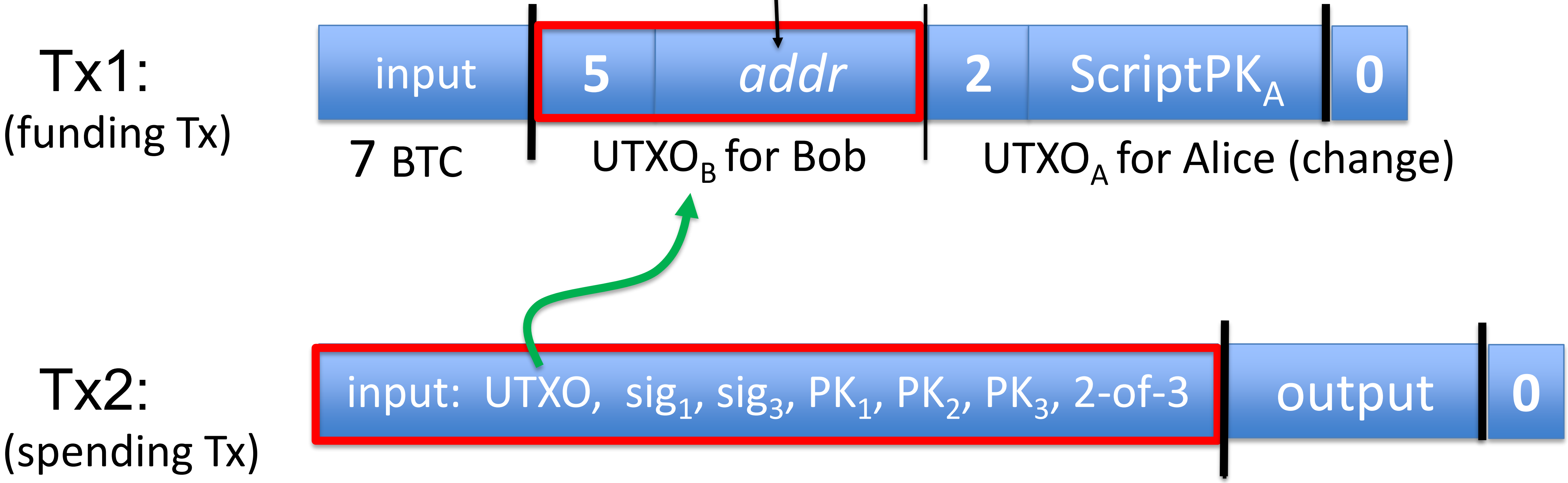
<2> <PK$_1$> <PK$_2$> <PK$_3$> <3> CHECKMULTISIG

hash gives P2SH address

ScriptSig to spend: (by payee) <0> <sig1> <sig3> <redeem script>

Abstractly …

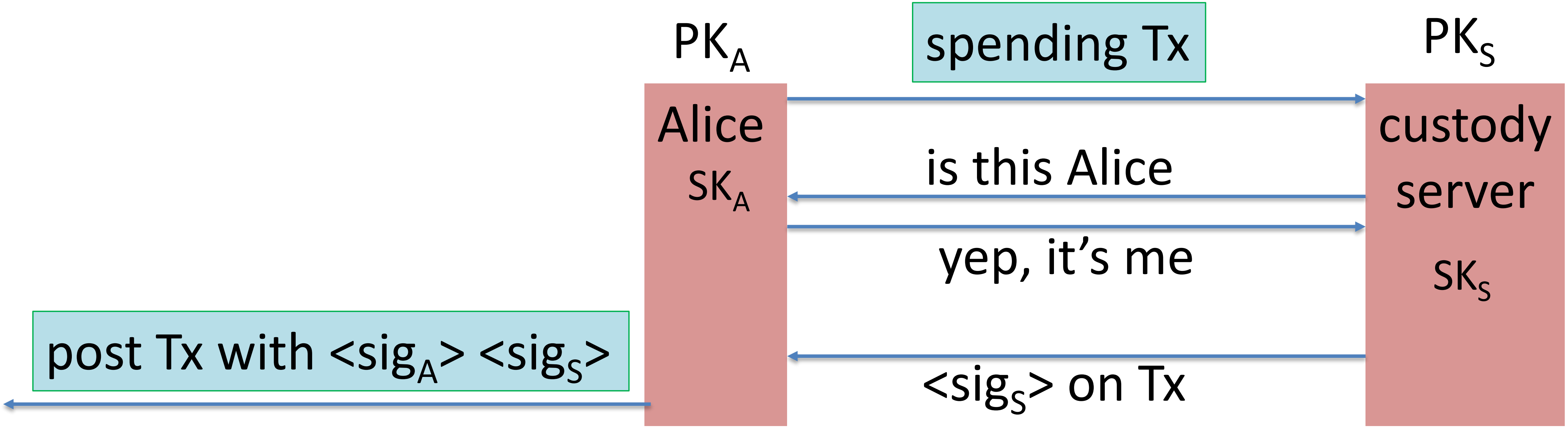
Multisig address: $addr = H(PK_1, PK_2, PK_3, \text{2-of-3})$



Example Bitcoin scripts

Protecting assets with a co-signatory

Alice stores her funds in UTXOs for *addr* = **2-of-2(PK_A, PK_S)**



⇒ theft of Alice's SK_A does not compromise BTC

Escrow service

Alice wants to buy a backpack for 0.1Ƀ from merchant Bob

Goal: Alice only pays after backpack arrives, but can't not pay

$$addr = \textbf{2-of-3}(\mathbf{PK_A}, \mathbf{PK_B}, \mathbf{PK_J})$$

Alice PK_A | Bob PK_B | Judge PK_J

post payment of 0.11Ƀ to *addr*

(creates $UTXO_A$)

want backpack for 0.1Ƀ

once see Tx on chain

mail backpack

backpack arrives

send $<sig_A>$ on Tx:

$UTXO_A \rightarrow (PK_B{:}0.1,\ PK_A{:}0.01)$

redeem using $<sig_A>$ $<sig_B>$ on Tx

Escrow service: a dispute

(1) Backpack never arrives: (Bob at fault)

Alice gets her funds back with help of Judge and a Tx:

Tx: $(\mathbf{UTXO_A} \rightarrow \mathbf{PK_A} \;,\; \mathbf{sig_A},\; \mathbf{sig_{Judge}})$ [2-out-of-3]

---

(2) Alice never sends sig_A: (Alice at fault)

Bob gets paid with help of Judge and a Tx:

Tx: $(\mathbf{UTXO_A} \rightarrow \mathbf{PK_B} \;,\; \mathbf{sig_B},\; \mathbf{sig_{Judge}})$ [2-out-of-3]

---

(3) Both are at fault: Judge publishes $<sig_{Judge}>$ on Tx:

Tx: $(\mathbf{UTXO_A} \rightarrow \mathbf{PK_A}: \mathbf{0.05},\; \mathbf{PK_B}: \mathbf{0.05},\; \mathbf{PK_J}: \mathbf{0.01})$

Now either Alice or Bob can execute this Tx.

Managing crypto assets: Wallets

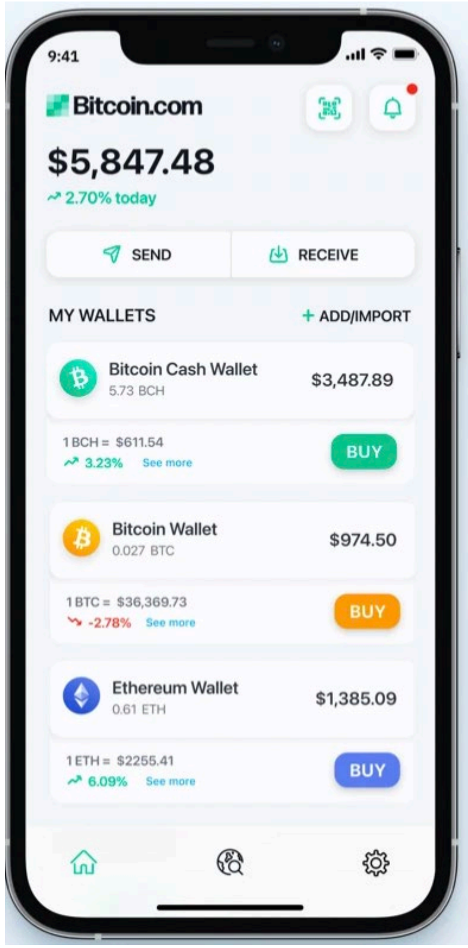
Managing secret keys

Users can have many PK/SK:

- one per Bitcoin address, Ethereum address, …

Wallets:

- Generates PK/SK, and stores SK,
- Post and verify Tx,
- Show balances



Managing lots of secret keys

Types of wallets:

- **cloud** (e.g., Coinbase): cloud holds secret keys … like a bank.
- **laptop/phone**: Electrum, MetaMask, …
- **hardware**: Trezor, Ledger, Keystone, …
- **paper**: print all sk on paper
- **brain**: memorize sk (bad idea)

client stores secret keys

- **Hybrid**: non-custodial cloud wallet (using threshold signatures)

Not your keys, not your coins … but lose key $\Rightarrow$ lose funds

Simplified Payment Verification (SPV)

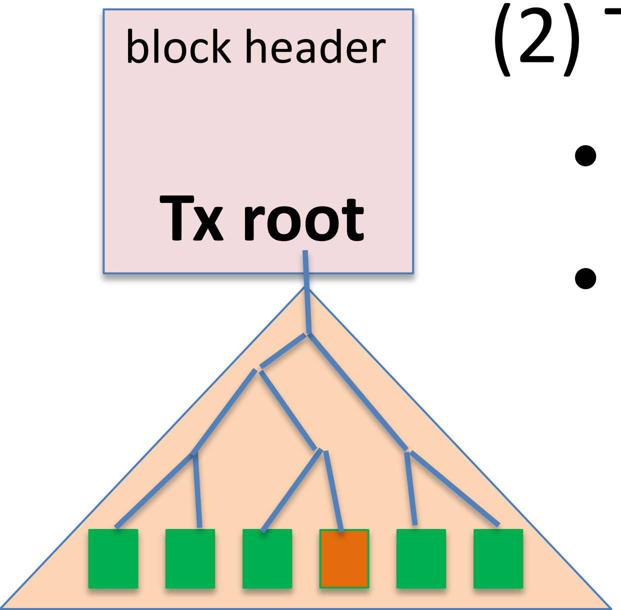
How does a client wallet display Alice's current balances?

- Laptop/phone wallet needs to verify an incoming payment
- **Goal**: do so w/o downloading entire blockchain (366 GB)

SPV: (1) download all block headers (60 MB)

(2) Tx download:

- wallet → server: list of my wallet addrs (Bloom filter)
- server → wallet: Tx involving addresses + Merkle proof to block header.



Simplified Payment Verification (SPV)

Problems:

(1) **Security**: are BH the ones on the blockchain? Can server omit Tx?

- Electrum: download block headers from ten random servers, optionally, also from a trusted full node.

List of servers: electrum.org/#community

(2) **Privacy**: remote server can test if an *addr* belongs to wallet

We will see better light client designs later in the course (e.g. Celo)

Hardware wallet: Ledger, Trezor, …

End user can have lots of secret keys. How to store them ???

Hardware wallet (e.g., Ledger Nano X)

- connects to laptop or phone wallet using Bluetooth or USB
- manages many secret keys
  - Bolos OS: each coin type is an app on top of OS
- PIN to unlock HW (up to 48 digits)
- screen and buttons to verify and confirm Tx

Hardware wallet: backup

Lose hardware wallet $\Rightarrow$ loss of funds. What to do?

ECDSA public key

Idea 1: generate a secret seed $k_0 \in \{0,1\}^{256}$

for i=1,2,…: $sk_i \leftarrow HMAC(k_0, i)$, $pk_i \leftarrow g^{sk_i}$

$pk_1, pk_2, pk_3, \ldots$: random unlinkable addresses (without k_0)

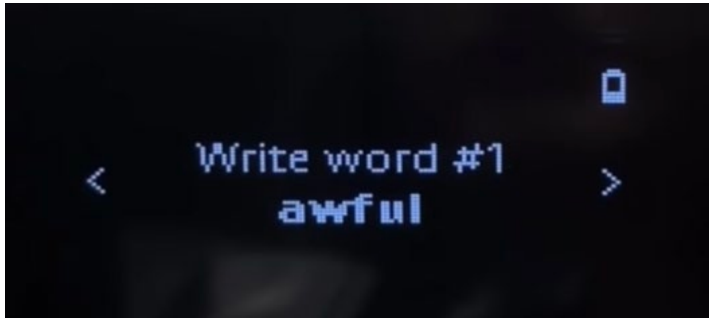
k_0 is stored on HW device and in offline storage (as 24 words)

$\Rightarrow$ in case of loss, buy new device, restore k_0, recompute keys

On Ledger

When initializing ledger:

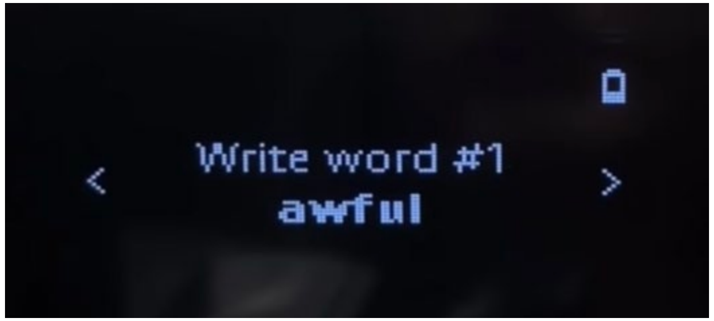
- user asked to write down the 24 words
- each word encodes 11 bits (24 $\times$ 11 = 268 bits)
  - list of 2048 words in different languages (BIP 39)



Example: English word list

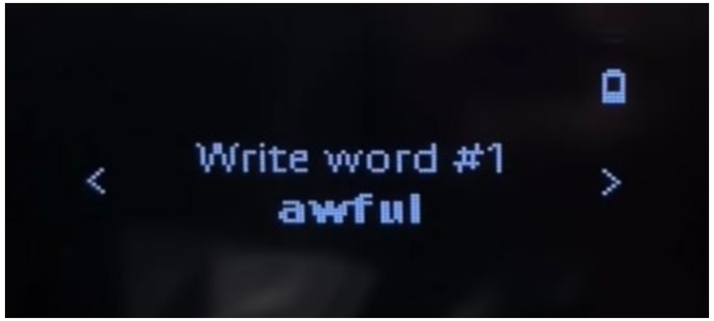
2048 lines (2048 sloc) | 12.8 KB

1	abandon
2	ability
3	able
4	about
5	above
6	absent
7	absorb
8	abstract
9	absurd
10	abuse
	⋮
2046	zero
2047	zone
2048	zoo



save list of 24 words

On Ledger



When initializing ledger:

- user asked to write down the 24 words
- each word encodes 11 bits (24 $\times$ 11 = 268 bits)
  - list of 2048 words in different languages (BIP 39)

Beware of “pre-initialized HW wallet”

- 2018: funds transferred to wallet promptly stolen

How to securely check balances?

With Idea1: need k_0 just to check my balance:

- k_0 needed to generate my addresses $(pk_1, pk_2, pk_3, \dots)$

  … but k_0 can also be used to spend funds

- Can we check balances without the spending key ??

<u>Goal:</u> two seeds

- k_0 lives on Ledger: can generate all secret keys (and addresses)
- k_{pub}: lives on laptop/phone wallet: can only generate addresses (for checking balance)

Idea 2: (used in HD wallets)

secret seed: $k_0 \in \{0,1\}^{256}$; $(k_1, k_2) \leftarrow \text{HMAC}(k_0, \text{"init"})$

balance seed: $k_{pub} = (k_2,\ h = g^{k_1})$

for all i=1,2,…: $\begin{cases} sk_i \leftarrow k_1 + \text{HMAC}(k_2, i) \\ pk_i \leftarrow g^{sk_i} = g^{k_1} \cdot g^{HMAC(k_2,i)} = \underbrace{h \cdot g^{HMAC(k_2,i)}}_{\text{computed from } k_{pub}} \end{cases}$

$\boxed{k_{pub} \text{ does not reveal } sk_1, sk_2, \ldots}$

k_{pub}: on laptop/phone, generates unlinkable addresses $pk_1, pk_2, \ldots$

k_0: on ledger

Paper wallet (be careful when generating)

signing key (cleartext)

Bitcoin address = base58(hash(PK))

base58 = a-zA-Z0-9 without {0,O,l,1}