

CPSC 3640/5400: Decentralized Payments, Contracts, and Finance for Humans and AI

Lecture 4: Transaction model & mechanism in Bitcoin

Instructor: Prof. Fan Zhang

Yale University

Fall 2026

OH updated

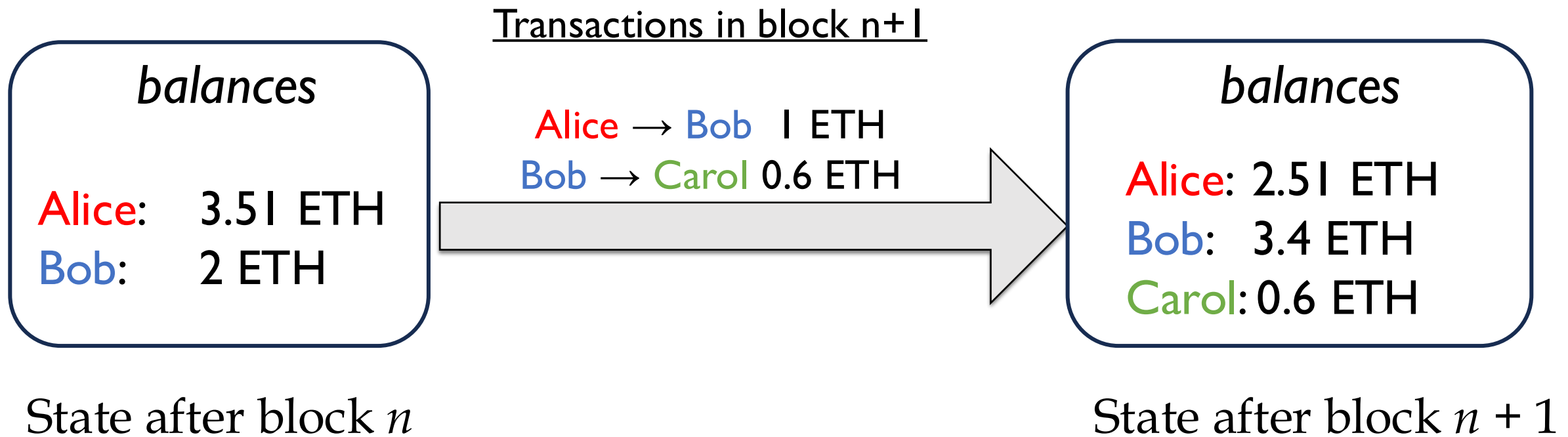
- Instructor
- Office hours: Thursday, 4:00pm in AWK 503
- Teaching Assistant Longfei Qiu
- Office hours: Tuesday, 4:00pm in AKW 313
- Info on <https://cpsc3640.netlify.app/>

Recap

- Last time: the Bitcoin ledger
- Mental model: public append-only ledger
 - What does “public” and “append-only” mean?
- What would happen if an attacker has 51% hash power or more?
- Even if there is no 51% attacker, what is the risk of accept transactions zero confirmations?
- Today: Bitcoin transaction model and mechanism.
 - i.e., how is the ledger used to create a cryptocurrency.

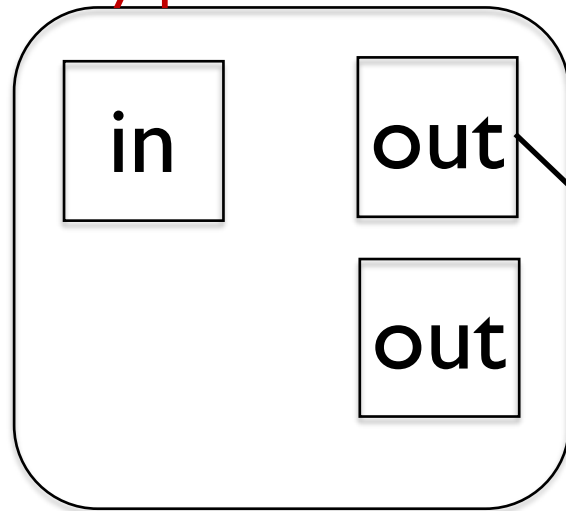
Transaction model: How to represent “coins”

- The intuitive way: state stores user balances
- Ethereum uses this “account model”

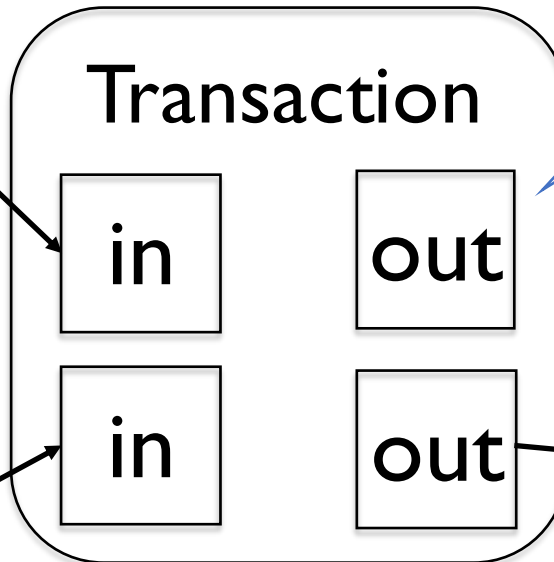


Bitcoin's UTXO model (Unspent Transaction Outputs)

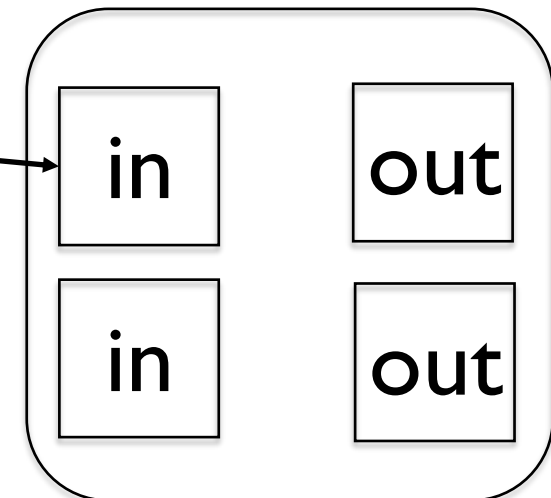
Inputs points to outputs
created by prior transactions



A Bitcoin transaction
consumes **inputs** and
create **outputs**.



Bitcoin state: Unspent TX
Output (UTXO)

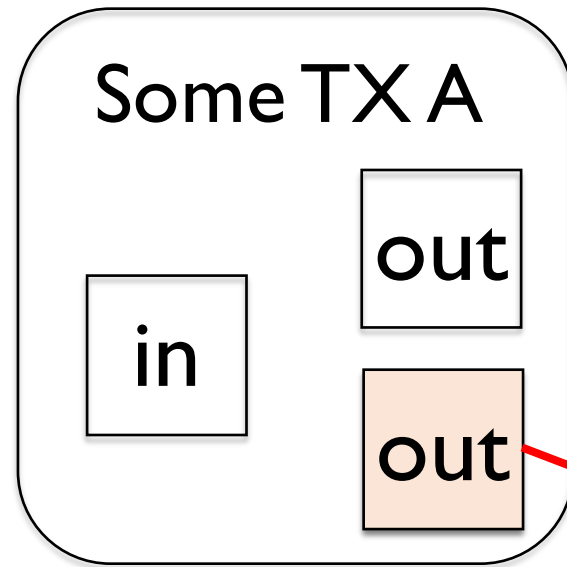


Outputs can be
consumed in whole
by future transactions

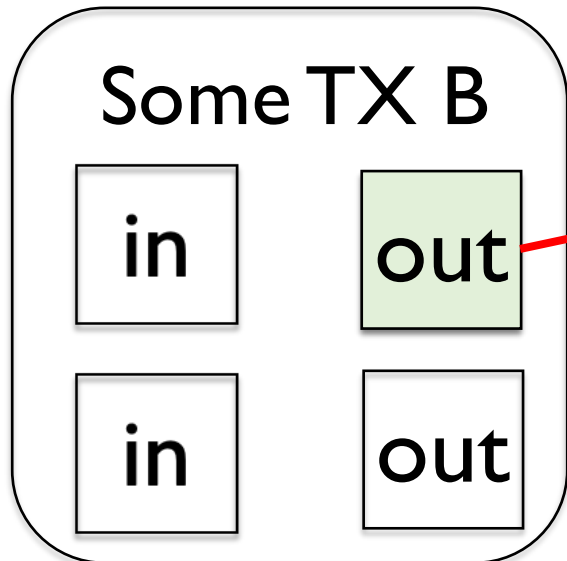
UTXO model

- **Output** = (amount, spending condition)
 - programmed via Bitcoin Scripts (more on this later)
 - E.g., “spendable by PK”
- **Input** = (one unspent output, proof to show spending condition is met)
 - E.g., signature [on the txn] that verifies under PK
- Balance is not tracked explicitly. Your Bitcoin balance = **sum of all UTXOs** you can spend.

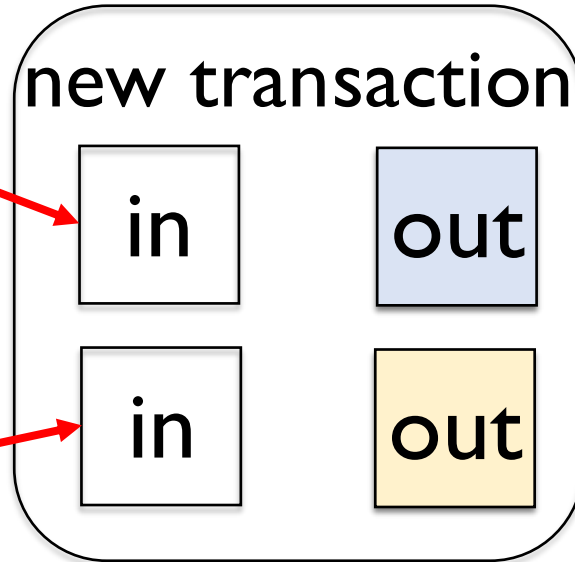
Alice paying Carol ₿3



UTXO of ₿1.01
spendable by **Alice**



UTXO of ₿2.5
spendable by **Alice**



UTXO of ₿3
spendable by **Carol**

Fee: 0.01



UTXO of ₿0.5 spendable
by **Alice** ("change")

Pros and cons of the UTXO model

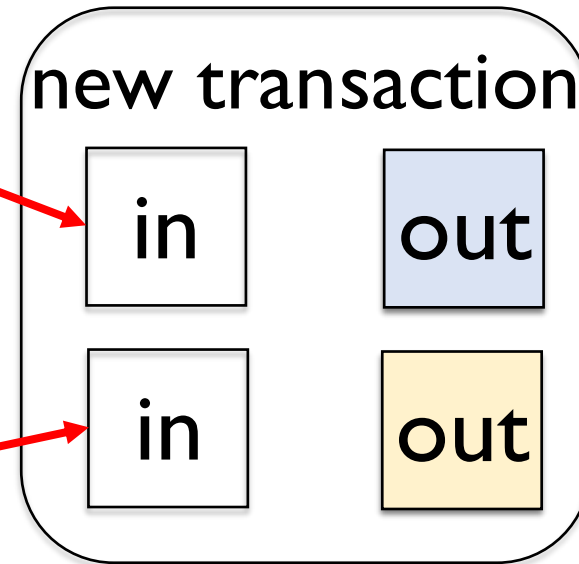
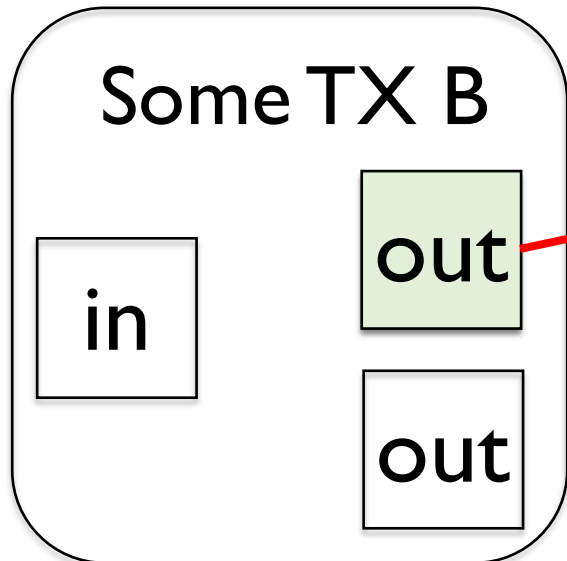
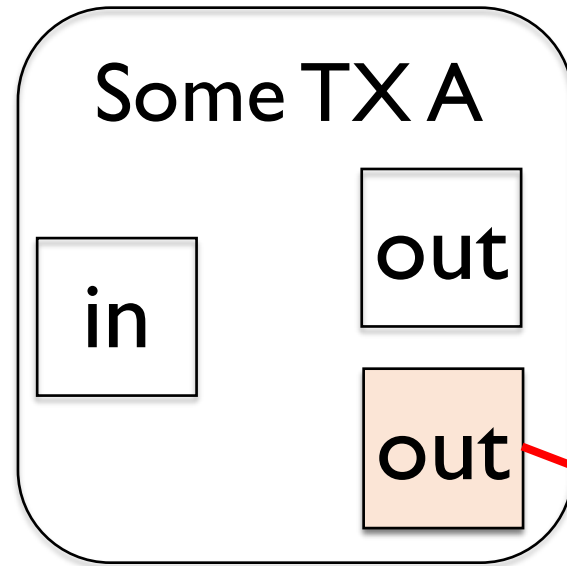
Pros

- Small (50x–100x smaller than blockchain, and doesn't grow indefinitely)
 - After sync, a bitcoin node can prune old blocks and only keep UTXOs.
- Some **privacy benefits**: a user's many UTXO can use different public keys so they are unrelated.
- Simple parallel processing: most txns are independent.

Cons

- Unintuitive (it's helpful to think of UTXOs as bank notes)
- Hard to express some spending conditions (more on this later)
- Privacy is weak (next slide)

Some of these 4 UTXOs are likely to be the same owner. Do you see why?



Fee



- **Linkability** of UTXOs significantly weakens privacy.
- it's like the kind of privacy you get on Reddit.

Warning: weak privacy

- But building privacy coins in the UTXO model is easier
- Zcash has the strongest privacy of all privacy focused coins



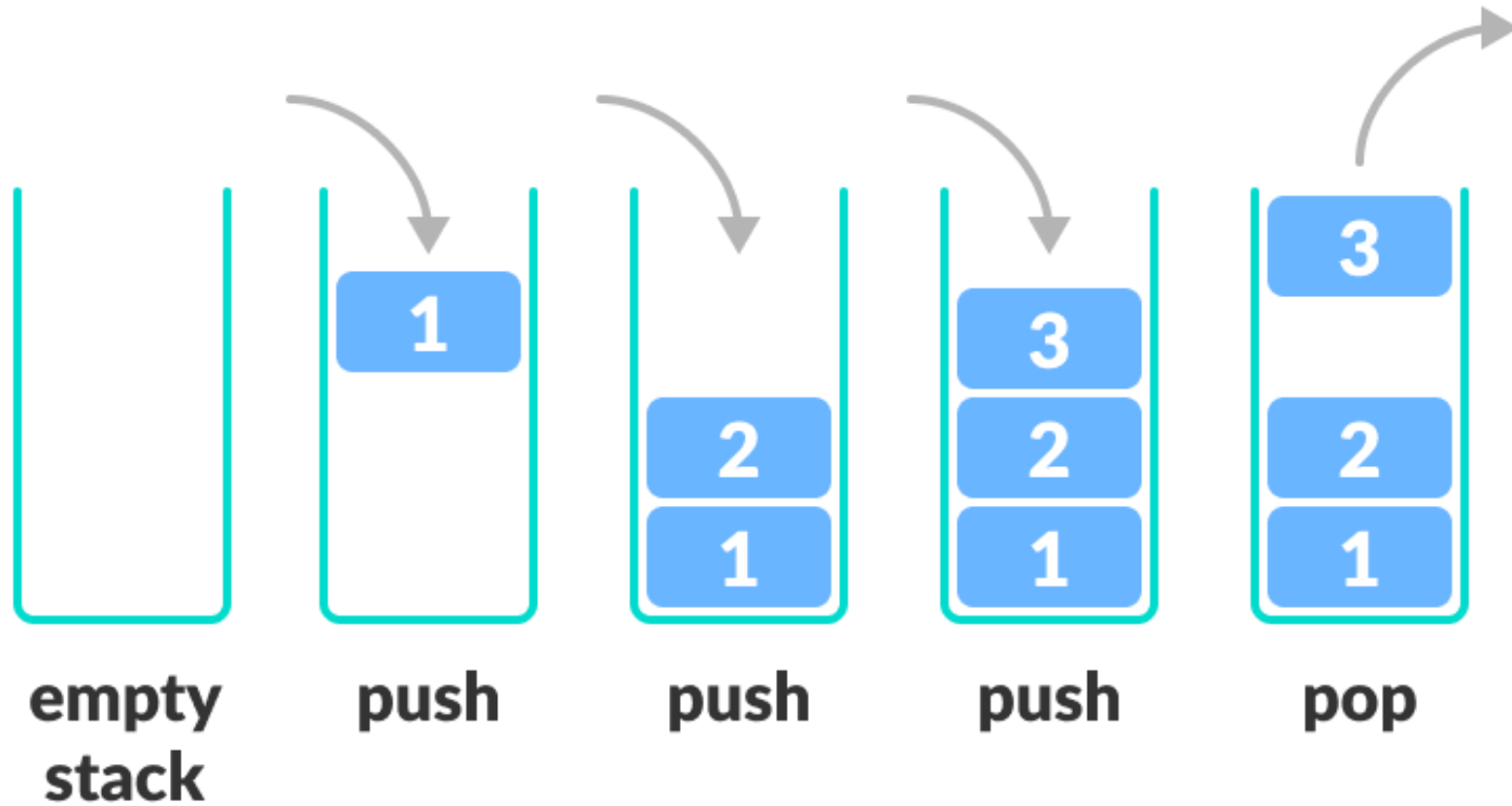
- Achieving similar privacy on Ethereum (which uses account model) is harder (lecture 8)

Spend conditions & Bitcoin Scripts

Bitcoin Scripts is a simple language

- **Scripts:** sequence of OP codes
 - Op codes are defined in <https://en.bitcoin.it/wiki/Script>
- **Example:** DUP HASH256 <pkhash> EQVERIFY CHECKSIG
 - We will decipher this shortly
- **High level idea**
 - Start with an empty execution [stack](#)
 - Execute each op code from left to right (OP codes modify the stack)
 - **Success** if no error is triggered during execution, and the top item is True (non-zero) when the script exits.
- We will see a bunch of examples.

Stack (data structure)



- Last-in-first-out (LIFO)

Bitcoin Scripts: data

- Push numbers
 - OP_TRUE (OP_1),
 - OP_2, ..., OP_16
- Push data/bytes
 - we simply write as `<data>`
- OP_DUP: duplicate the top of stack

Constants

When talking about scripts, these value-pushing words are usually

Word	Opcode	Hex	Input	Output
OP_0, OP_FALSE	0	0x00	Nothing.	(empty value)
N/A	1-75	0x01- 0x4b	(special)	data
OP_PUSHDATA1	76	0x4c	(special)	data
OP_PUSHDATA2	77	0x4d	(special)	data
OP_PUSHDATA4	78	0x4e	(special)	data
OP_1NEGATE	79	0x4f	Nothing.	-1
OP_1, OP_TRUE	81	0x51	Nothing.	1
OP_2-OP_16	82-96	0x52- 0x60	Nothing.	2-16

Example

```
OP_TRUE  <data>  OP_DUP
```

stack: empty

|

| <data>

| <data> <data>

init

OP_TRUE pushes |

<data> pushes data

duplicate top element

Bitcoin Scripts: control

- Control flow
- **OP_IF** <code1> **OP_ELSE** <code2> **OP_ENDIF**
 - if top != false, run <code1>
 - Else, run <code2>
 - remove top value
- **OP_VERIFY**: fail if top = false
- **OP_EQVERIFY**: pop, pop, abort fail if not equal
- **OP_RETURN**: fail immediately
 - Why might this be useful?

Word	Opcode	Hex	Input	Output
OP_NOP	97	0x61	Nothing	Nothing
OP_IF	99	0x63	<expression> if [statements] [else [statements]]* endif	
OP_NOTIF	100	0x64	<expression> notif [statements] [else [statements]]* endif	
OP_ELSE	103	0x67	<expression> if [statements] [else [statements]]* endif	
OP_ENDIF	104	0x68	<expression> if [statements] [else [statements]]* endif	
OP_VERIFY	105	0x69	True / false	Nothing / <i>fail</i>
OP_RETURN	106	0x6a	Nothing	<i>fail</i>

OP_RETURN

- What does an output with this script accomplish?

```
OP_RETURN  
OP_PUSHDATA <some data>
```

- Answer: it stores <some data>
- E.g., Bitcoin NFT
- E.g., **checkpoints** in Proof-of-Stake protocols
 - Babylon <https://arxiv.org/pdf/2207.08392.pdf>
- It's very expensive so only suitable for small data.
- **What might this be better than just OP_pushdata?**



This 374420 byte MP4 video is embedded in Bitcoin block #802522

Bitcoin Scripts (cont.)

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 T: fail before T

Small example (cont.)

```
OP_TRUE  <data>  OP_DUP  HASH256
```

stack: empty

|

| <data>

| <data> <data>

| <data> H(data)

init

OP_TRUE pushes |

<data> pushes data

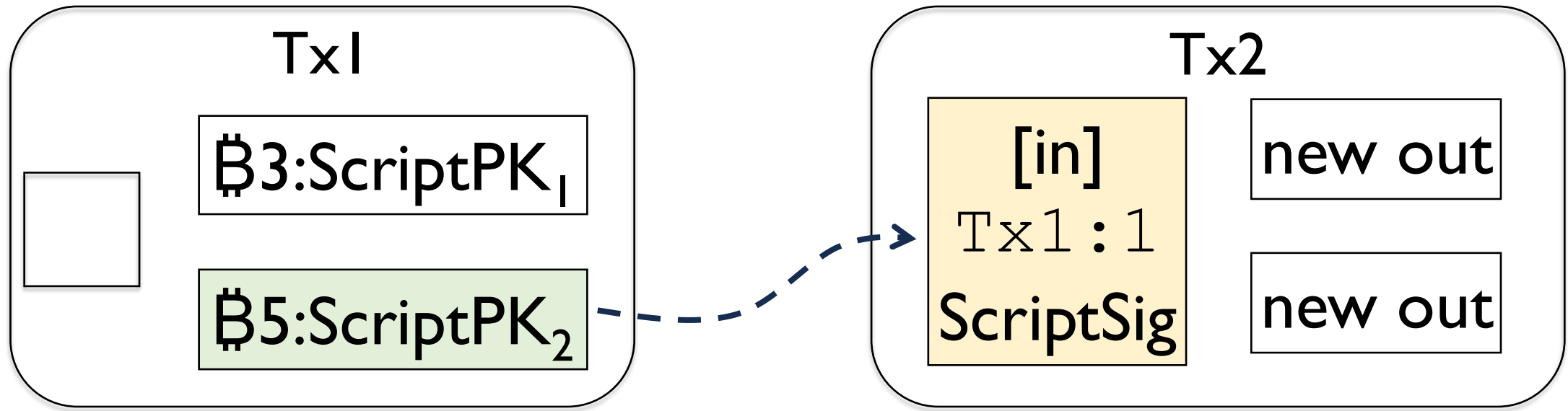
duplicate top element

compute the hash and replace

Bitcoin script is intentionally limited

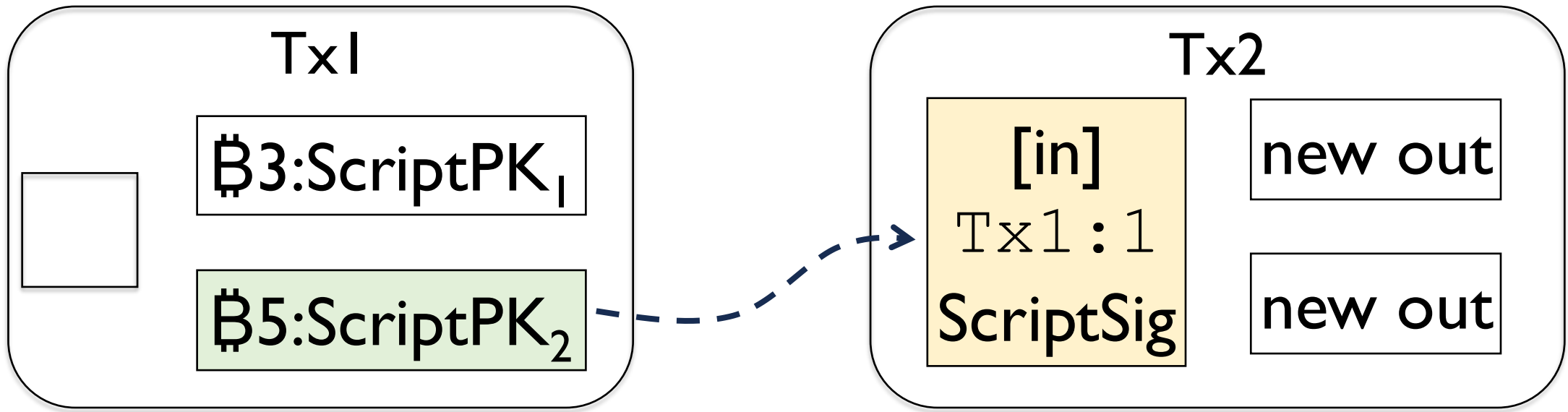
- E.g., Bitcoin Script doesn't support loops.
- Fact: Bitcoin charges tx fee according to **the size of tx**, whereas Ethereum charges tx fee for **each step of computation**.
- **Based on these facts, what would go wrong if loops were supported in Bitcoin?**
- Answer: an attacker can write a dead loop and halt the entire Bitcoin network.

Spend conditions



- $[\text{out}]$ specifies a “**lock script**” ScriptPK
- $[\text{in}]$
 - points to an $[\text{out}]$: “ $\text{Tx1}:1$ ” means the 2nd output of tx1
 - presents a satisfying “unlocking script” ScriptSig

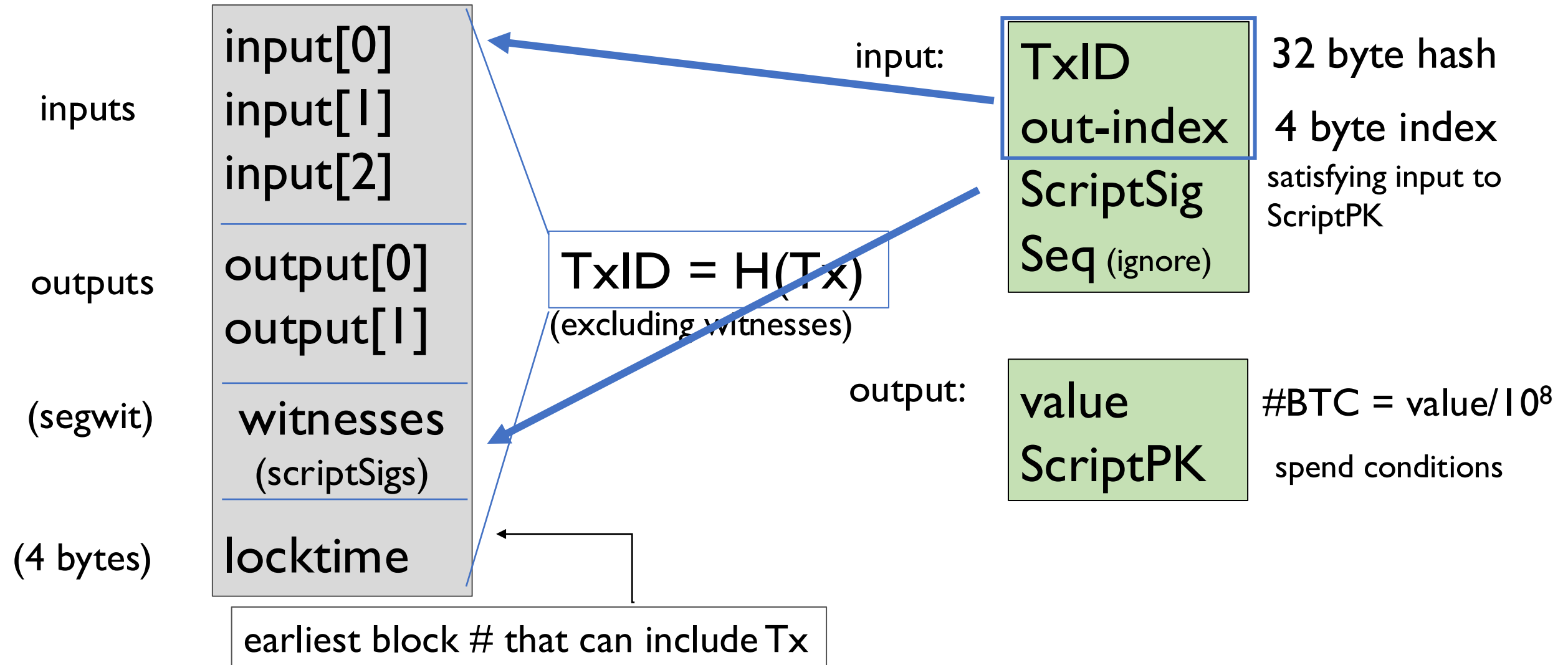
Flexible spending conditions



To validate Tx2, miners check (for each input):

1. Output $(\text{TxID}, \text{out-index})$ is in UTXO set
2. $\text{sum input values} \geq \text{sum output values}$
3. The script $\text{ScriptSig} \parallel \text{ScriptPK}$ evaluates to true

Transaction data structure



Common pattern: Payment to PK Hash (P2PKH)

Alice wants to pay Bob ₿5:

- step 1: Bob generates sig key pair $(pk_B, sk_B) \leftarrow \text{KeyGen}()$
- step 2: Bob sends Alice the address $hpk_B \leftarrow \text{HASH160}(pk_B)$
- step 3: Alice posts the following output:

Tx1 (paying Bob)

Inputs
(details
omitted)

Output:
₿5

DUP HASH160 <hpk_B> EQVERIFY CHECKSIG

Tx1 (paying Bob)

Input:
One of Alice's
UTXOs

Output:
₿5

DUP HASH160 **<hpk_B>** EQVERIFY CHECKSIG

To spend this P2PKH output

$$\langle \text{sig}_B \rangle = \text{Sign}(\text{sk}_B, \text{Tx2})$$

[sighash_all]

Tx2 (Bob spending)

Input:

<sig_B> **<pk_B>**

new out

new out

Why do we need both
EQVERIFY and CHECKSIG?

scriptSig scriptPub

<sig> <pk> | DUP HASH160 <hpk_B> EQVERIFY CHECKSIG

stack: empty

<sig> <pk>

<sig> <pk> <pk>

<sig> <pk> <H(pk)>

<sig> <pk> <H(pk)> <hpk_B>

<sig> <pk>

|

⇒ successful termination

init

push values

DUP

HASH160

push value

EQVERIFY

CHECKSIG verify(pk, Tx, sig)

Hands-on component:

Bitcoin Core and Bitcoin Addresses

HoC

- Install Bitcoin Core and configure it to use Testnet4.
- Create a wallet and a legacy receiving address.
- Receive and send coins and inspect a transaction.
- **Go to Course website → Lecture 4 → Hands-on component**