

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

Lecture 5: Contracts in Bitcoin

Instructor: Prof. Fan Zhang

Yale University

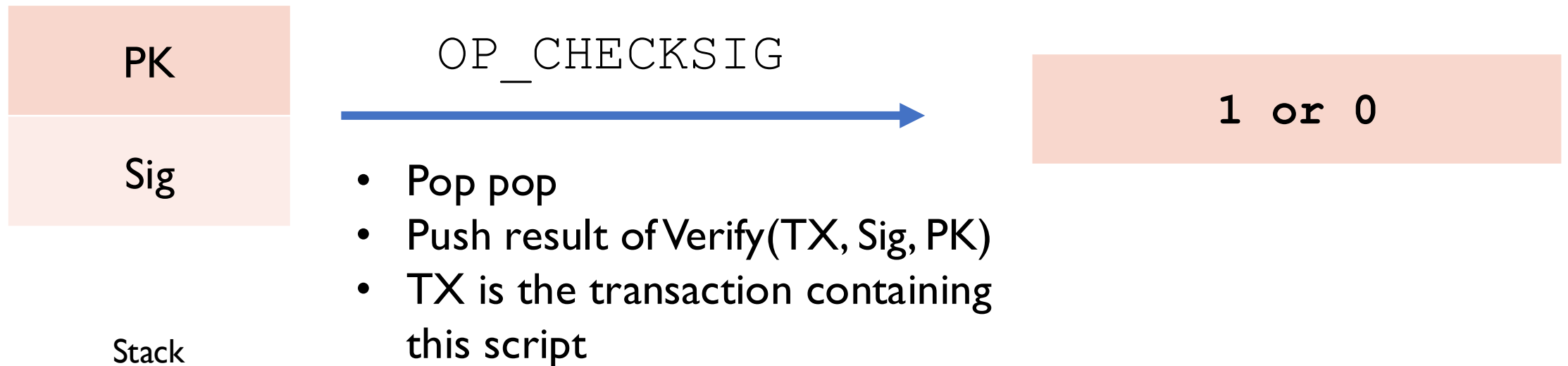
Fall 2026

Recall: Spend conditions & Bitcoin Scripts

- Account model v.s. UTXO model
- UTXO $\approx$ a “coin”
 - Each **UTXO** specifies an **amount** and a **ScriptPK**
 - ScriptPK is a Bitcoin Script that specifies **spend conditions**
- Each Bitcoin transaction **consumes** and **creates** UTXOs
- To spend an UTXO, a transaction input presents a satisfying ScriptSig
 - i.e., ScriptSig || ScriptPK runs to True.

OP codes

- A Script is a sequence of op codes
- We learn about a bunch of OP codes
- Let's look at an important example: OP_CHECKSIG



Simplest: Pay to PK (P2PK)

- To promote standardization, Bitcoin has a few **standard patterns**.
 - P2PK, P2PKH, P2SH, P2TR, etc (Pay-to-blah)
 - Called output **types**
- Let's look at the simplest: **Pay to PK (P2PK)**
- Satisfying ScriptSig is simply $\text{sig} = \text{Sign}(\text{TX}, \text{sk}_{\text{Bob}})$

Script PK

`<sig>`

`<pkBob>`

See previous slide for the execution of `<sig> <pk> CHECKSIG`

Question

- Mallory sees Bob's signature in the mempool. Why can't Mallory just copy it & spend Bob's outputs?

Pay 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 $hpk_B \leftarrow H(pk_B)$
- 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

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

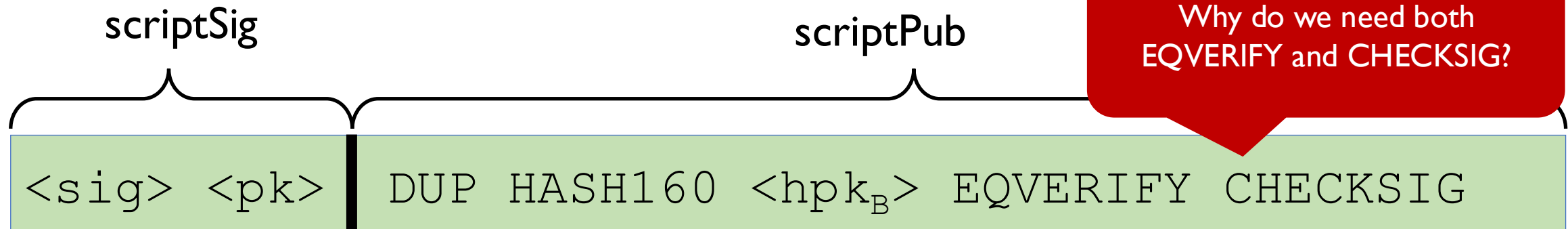
Tx2 (Bob spending)

Input:

$\langle \text{sig}_B \rangle \langle \text{pk}_B \rangle$

new out

new out



stack: empty

`<sig> <pk>`

`<sig> <pk> <pk>`

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

`<sig> <pk> <H(pk)> <hpkB`

`<sig> <pk>`

|

⇒ successful termination

init

push values

DUP

HASH160

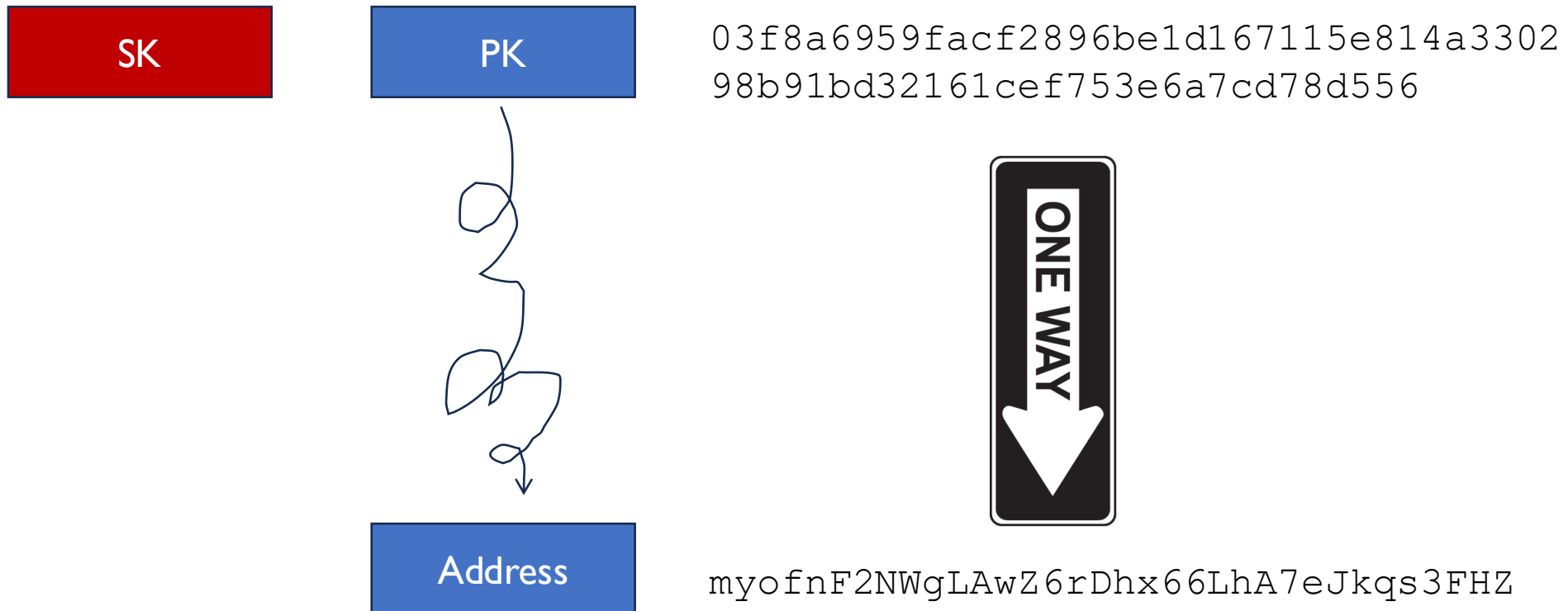
push value

EQVERIFY

CHECKSIG `verify(pk, Tx, sig)`

Hash of PK is a P2PKH Address

- Bitcoin address is what payee needs to construct ScriptPK
- In P2PKH, it's some encoding of $\text{HASH160}(\text{PK})$



Legacy address calculator

Public key (hex)

03f8a6959facf2896be1d167115e814a330298b91bd32161cef753e6a7cd78d556

Construct address

1. Let pk be the public key.

03f8a6959facf2896be1d167115e814a330298b91bd32161cef753e6a7cd78d556

2. Compute $h = \text{SHA-256}(pk)$.

545cbc45b604ab716f3cbc2c4a0f717e9a0b55c1c8d459ac7253d355d5a4c15b

3. Compute $pkh = \text{RIPEMD-160}(h)$. This is the public-key hash, or $\text{HASH160}(pk)$.

c89b8cdebea7eb29407a827665799cf7e37e02b3

4. Prepend the Testnet4 version byte: $p = 6f \parallel pkh$.

6fc89b8cdebea7eb29407a827665799cf7e37e02b3

5. Compute $c = \text{SHA-256}(\text{SHA-256}(p))$.

First SHA-256:

ca9e37a50e273c2c14d63638f75e3492e72278d1150297bcd55d67120e9bf84

Second SHA-256:

cab07f68d4a97798a913052e1f38f6730aa1a7abc916d79d58c56957e9088208

6. Take the first four bytes of c as the checksum, then compute $b = p \parallel c[0:4]$.

cab07f68

Versioned payload + checksum:

6fc89b8cdebea7eb29407a827665799cf7e37e02b3cab07f68

7. Compute $address = \text{Base58}(b)$.

myofnF2NWgLAwZ6rDhx66LhA7eJkqs3FHZ

- If you are interested in detail, the course website has a calculator <https://cpsc3640.netlify.app/hands-on/constructing-bitcoin-address/>

- $\text{HASH160}(PK)$ is in fact $\text{RIPEMD-160}(\text{SHA-256}(PK))$

where RIPEMD and SHA are two cryptographic hash functions designed by EU and US NIST, respectively.

Why hashing?

- Address is shorter, so UTXO is smaller! 65 byte vs 20 byte
- Payer spends less on tx fees (payee spends more)
- Protection against **unforeseen ways of reconstructing private keys from public keys**
 - E.g., bugs in ECDSA signature
 - E.g., **quantum computers**

Let's see P2PKH in action

- `getaddressinfo myofnF2NWgLAwZ6rDhx66LhA7eJkqs3FHZ`
- `Decodescript`
- Examine a transaction from one of you

HoC submission

- Created an assignment on Canvas
- Submit your P2PKH address (a testnet 4 address that starts with m or n) + public key, by today
- I will send you coins for Lab 1
- I will check if you sent me 0.0001 coins
- We will allocate 10 minutes at the end of lecture.

Contracts on Bitcoin via P2SH

Contracts here mean “autonomous programs”

Pay to Script Hash (P2SH)

- P2SH is a generalization of P2PKH
- Output specifies a **redeem script**
- To spend, present data such that
 <data> <redeem script> evaluates to true.
- Conceptually:
 - ScriptPK = <redeem script>
 - ScriptSig = <data>
- Problem: <redeem script> can be long (big UTXO, high fees).

Pay to Script Hash (P2SH)

- Bitcoin uses a hash of <redeem script> in ScriptPK
 - So UTXO has constant size
- **ScriptPK:** `HASH160 <H (redeem script)> EQUAL`
- **ScriptSig:** `<data> <redeem script>`
- Miner runs two scripts:
 - (1) `<ScriptSig> <ScriptPK>`
 - (2) `<data> <redeem script>`
- What do each verify?

Some cute examples, then a not-so-cute one.

Example: Multisig

- E.g., Three people managing a treasury fund only spendable if *at least two people sign*.
- Bitcoin has op code CHECKMULTISIG
- Redeem script for 2-out-of-3

```
<2> <PK1> <PK2> <PK3> <3> CHECKMULTISIG
```

ScriptSig to spend by (say) signers 1 and 3

```
<0> <sig1> <sig3> <redeem script>
```

Example: retractable coins

- Retractable coins: Merchant gives away free Bitcoin for promotion, but wants to *retracts coins* if they are not used promptly.
- `OP_CHECKLOCKTIMEVERIFY` (CLTV) locks an UTXO until `<time>`
- Redeem script:

```
OP_IF <receiverkey>  
OP_ELSE <time> OP_CLTV OP_DROP <senderkey>  
OP_ENDIF  
OP_CHECKSIG
```

Example: retractable coins

Lab 1 (to be released today):

- I'll send you some retractable (testnet4) BTC! Spend them before I take them back at deadline.
- Figure out what <data> satisfies the redeem script
- You can use my open-source tool `timelock.py` and Bitcoin Core to build and send transactions.
- You can collaborate with another student, but everyone must submit their own results.

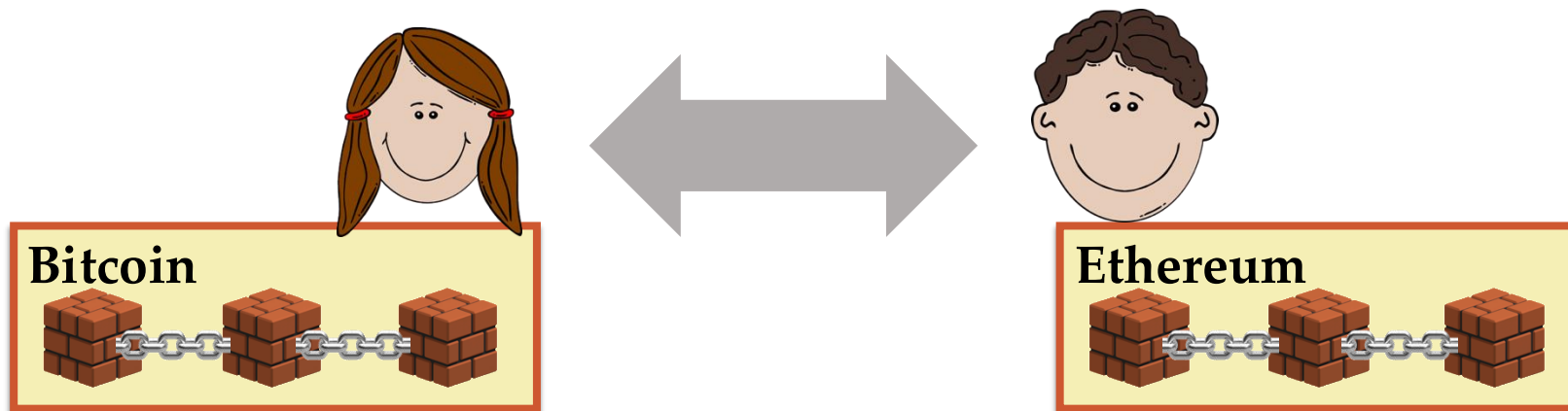
Example: Atomic Swaps

Alice has ETH and wants BTC

Bob has BTC and wants ETH

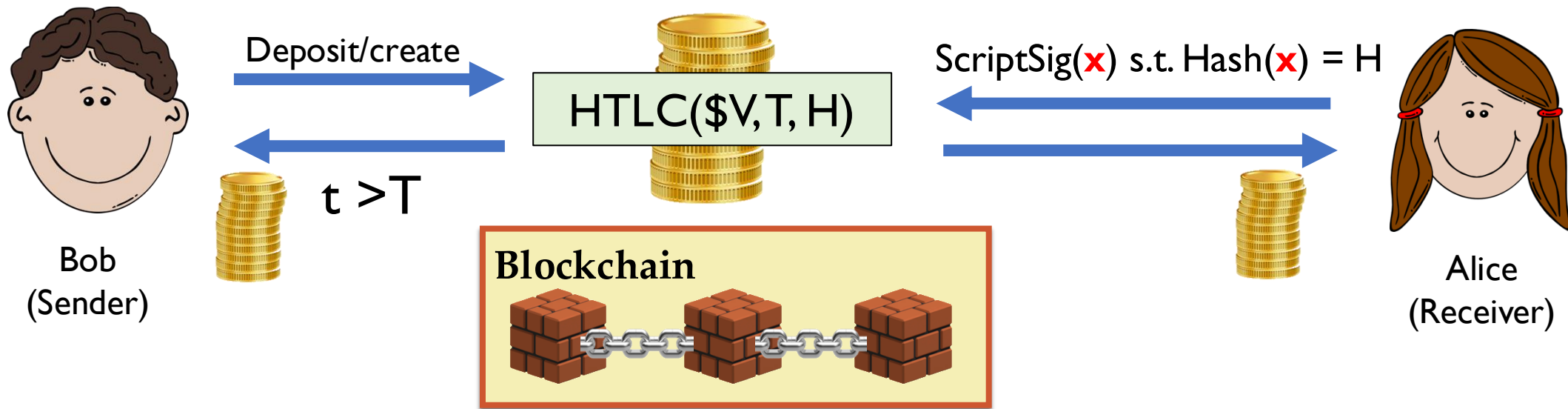
Challenge: If Alice sends ETH to Bob first, then Bob may disappear

Goal: **Atomicity** (“all or nothing”)



New Ingredient: Hash Time Locked Contract (HTLC)

- $\text{HTLC}(v, T, H)$: a fancier version of retractable v coins
- Before time T , coins spendable by receiver by revealing *a preimage of H*
- After time T , sender can spend the UTXO (retracting)



HTLC part I: Hash lock on outputs

TxId	
INPUTS	OUTPUTS
Some UTXO	HashLock(H) ₿8

We will use
this notation
for UTXO

```
Redeem Script: OP_SHA256 <H> OP_EQVERIFY
```

```
ScriptSig: <x>
```

HTLC part 2: *time lock*

TxId	
INPUTS	OUTPUTS
Some UTXO	Alice & “in 3 days” ₿8

- OP_CLTV (CheckLockTimeVerify) T
 - Output becomes spendable after T (absolute time: e.g., **Feb 28th**)
- OP_CSV (Check sequence Verify) N
 - Output becomes spendable after N confirmation (relative time: **in 3 days**)

Combining hash and time lock (HTLC)

TxId	
INPUTS	OUTPUTS
Some UTXO	(Alice in 3 days) OR (HashLock(H) & Bob) 8 coins

- Alice can spend in 3 days
- Bob can spend if he knows the pre-image of H.

Redeem Script

```
OP_IF OP_CSV 3days <pk_alice>  
OP_ELSE OP_SHA256 <x> OP_EQVERIFY <pk_bob>  
OP_ENDIF  
OP_CHECKSIG
```

ScriptSig: <sig alice> OP_1

ScriptSig: <sig Bob> <x> OP_0

Atomic Swaps

- This is the first DeFi *interoperability* protocol we will learn
- Timing difference matters here: $T_1 > T_2$ or $T_2 > T_1$?

1. Alice and Bob agree on the exchange rate (e.g., 30 ETH for 1 BTC)
2. Bob picks a random secret x and compute $H = \text{SHA256}(x)$
3. Bob creates HTLC on Bitcoin, Alice creates HTLC on Ethereum
4. Bob can spend Alice's HTLC first with x , then Alice can get BTC with x

