Note: HW#1 is posted on the course web site. Due Oct. 4.
Recap: the Bitcoin blockchain

- **genesis block**
- **BH₁**: version (4 bytes), **prev** (32 bytes), time (4 bytes), bits (4 bytes), nonce (4 bytes), **Tx root** (32 bytes), 80 bytes
- **BH₂**: **prev**, **Tx root**
- **BH₃**: **prev**, **Tx root**

In the diagram:
- **H**: Hash function
- **coinbase Tx**: A specific type of transaction in the blockchain

The diagram shows the structure of the Bitcoin blockchain, starting with the genesis block and extending through subsequent blocks, each containing a hash of the previous block and a transaction root for all transactions in that block.
View the blockchain as a sequence of Tx (append-only)

Tx cannot be erased: mistaken Tx ⇒ locked or lost of funds
Tx structure (non-coinbase)

- **inputs**
  - input[0]
  - input[1]
  - input[2]
- **locktime**
- **outputs**
  - output[0]
  - output[1]
- **witnesses**
  - (part of input)
- **(segwit)**
- **(4 bytes)**

**input:**
- TxID
- out-index
- ScriptSig
- seq

**output:**
- value
- ScriptPK

TxID = H(Tx) (excluding witnesses)

Input:
- 32 byte hash
- 4 byte index
- program
- ignore

Output:
- 8 bytes
- program

#BTC = value/10^8

earliest block # that can include Tx
Example

Tx1: (funding Tx)

UTXO: unspent Tx output

Tx2: (spending Tx)

identifies a UTXO

null locktime
Example

Tx1: (funding Tx)

UTXO: unspent Tx output

Tx2: (spending Tx)

identifies a UTXO

null locktime
Validating Tx2

Miners check (for each input):

1. The program `ScriptSig | ScriptPK` returns true

2. `TxID | index` is in the current UTXO set

3. sum input values ≥ sum output values

After Tx2 is posted, miners remove UTXO from UTXO set
Transaction types: (1) P2PKH

Alice wants to pay Bob 5 BTC:

• step 1: Bob generates sig key pair \((pk_B, sk_B) \leftarrow \text{Gen}()\)
• 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:

\[
\text{ScriptPK}_B: \quad \text{DUP HASH256} <addr_B> \text{ EQVERIFY CHECKSIG}
\]

\[
\begin{array}{c|c|c|c}
\text{input} & \text{UTXO}_B \text{ for Bob} & \text{UTXO}_A \text{ for Alice (change)} \\
7 \text{ BTC} & 5 & 2 & 0 \\
\end{array}
\]
“input” contains ScriptSig that authorizes spending Alice’s UTXO
• example: ScriptSig contains Alice’s signature on Tx
  $\implies$ miners cannot change ScriptPK_B (will invalidate Alice’s signature)

Point to Alice’s UTXO

<table>
<thead>
<tr>
<th>input 7 BTC</th>
<th>UTXO_B for Bob</th>
<th>UTXO_A for Alice (change)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

ScriptPK_B:

DUP HASH256 <addr_B> EQVERIFY CHECKSIG
Later, when Bob wants to spend his UTXO:

create a $T_{x_{\text{spend}}}$

$T_{x_{\text{spend}}}$:

<table>
<thead>
<tr>
<th>TxID</th>
<th>0</th>
<th>ScriptSig$_B$</th>
</tr>
</thead>
</table>

points to UTXO$_B$

<sig> <pk$_B$>  

(authorizes spending UTXO$_B$)

<sig> = Sign(sk$_B$, Tx) where Tx = ($T_{x_{\text{spend}}}$ excluding all ScriptSigs) (SIGHASH_ALL)

Miners validate that ScriptSig$_B$ | ScriptPK$_B$ returns true
Transaction types: (2) P2SH: pay to script hash

(pre SegWit in 2017)

Payer specifies a redeem script (instead of just pkhash)

Usage: (1) Bob publishes hash(redeem script) ← Bitcoint addr.
(2) Alice sends funds to that address in funding Tx
(3) Bob can spend UTXO if he can satisfy the script

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

ScriptSig to spend: <sig₁> <sig₂> ... <sigₙ> <redeem script>

payer can specify complex conditions for when UTXO can be spent
Miner verifies:

(1) \texttt{<ScriptSig> ScriptPK = true} \quad \leftarrow \text{spending Tx gave correct script}

(2) \texttt{ScriptSig = true} \quad \leftarrow \text{script is satisfied}
Example P2SH: multisig

**Goal:** spending a UTXO requires $t$-out-of-$n$ signatures

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

$<2> \ <PK_1> \ <PK_2> \ <PK_3> \ <3> \ \text{CHECKMULTISIG}$

threshold

hash gives P2SH address

ScriptSig to spend: (by payee)

$<0> \ <\text{sig1}> \ <\text{sig3}> \ <\text{redeem script}>$

(in the clear)
Abstractly ...

Multisig address: \[ addr = H(PK_1, PK_2, PK_3, 2-of-3) \]

Tx1: (funding Tx)
- Input: 7 BTC
- Output: UTXO_B for Bob, UTXO_A for Alice (change)

Tx2: (spending Tx)
- Input: UTXO, \( \sigma_1, \sigma_3, PK_1, PK_2, PK_3, 2-of-3 \)
- Output: 0
Example Bitcoin scripts
Protecting assets with a co-signatory

Alice stores her funds in UTXOs for \( addr = 2\text{-of-}2(PK_A, PK_S) \)

\[ \text{PK}_A \quad \text{Alice} \quad \text{spending Tx} \quad \text{PK}_S \quad \text{custody server} \]

Post Tx with \(<\text{sig}_A> <\text{sig}_S>\)

\[ \Rightarrow \text{theft of Alice’s SK}_A \text{ does not compromise BTC} \]
Alice wants to buy a backpack for 0.1₿ from merchant Bob

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

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

---

**Diagram**:

- Alice wants backpack for 0.1₿
- Post payment of 0.11₿ to $$addr$$ (UTXO_A)
- Judge
  - PK_J
- Once see Tx on chain
  - Mail backpack
  - Backpack arrives
  - Send $$\langle \text{sig}_A \rangle$$ on Tx:
    $$\text{UTXO}_A \rightarrow (PK_B:0.1, PK_A:0.01)$$
  - Redeem using $$\langle \text{sig}_A \rangle \langle \text{sig}_B \rangle$$ 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:
   \[ \text{Tx: } ( \text{UTXO}_A \rightarrow \text{PK}_A , \text{sig}_A, \text{sig}_{\text{Judge}} ) \]  [2-out-of-3]

(2) Alice never sends \( \text{sig}_A \): (Alice at fault)
   Bob gets paid with help of Judge as a Tx:
   \[ \text{Tx: } ( \text{UTXO}_A \rightarrow \text{PK}_B , \text{sig}_B, \text{sig}_{\text{Judge}} ) \]  [2-out-of-3]

(3) Both are at fault: Judge publishes \( <\text{sig}_{\text{Judge}}> \) on Tx:
   \[ \text{Tx: } ( \text{UTXO}_A \rightarrow \text{PK}_A: 0.05, \text{PK}_B: 0.05, \text{PK}_J: 0.01 ) \]
   Now either Alice or Bob can execute this Tx.
Cross Chain Atomic Swap

Alice has 5 BTC, Bob has 2 LTC (LiteCoin). They want to swap.

Want a sequence of Tx on the Bitcoin and Litecoin chains s.t.:
• either success: Alice has 2 LTC and Bob has 5 BTX,
• or failure: no funds move.

Swap cannot get stuck halfway.

Goal: design a sequence of Tx to do this.

solution: programming proj #1 ex 4.
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, ...

• **paper**: print all sk on paper

• **brain**: memorize sk (bad idea)

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

Not your keys, not your coins ... but lose key ⇒ 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 (56 MB)

(2) Tx download:

- wallet → server: list of my wallet addrs (Bloom filter)
- server → wallet: Tx involving addresses + Merkle proof to block header.
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)
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
Lose hardware wallet ⇒ loss of funds. What to do?

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

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

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

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

⇒ in case of loss, buy new device, restore $k_0$, recompute keys
When initializing ledger:

- user asked to write down the 24 words
- each word encodes 11 bits \((24 \times 11 = 268 \text{ bits})\)
- list of 2048 words in different languages \((\text{BIP 39})\)
**Example: English word list**

| 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         |

2048 lines (2048 sloc) | 12.8 KB

**save list of 24 words**
Careful with unused letters ...
On Ledger

When initializing ledger:

• user asked to write down the 24 words
• each word encodes 11 bits \( (24 \times 11 = 268 \text{ bits}) \)
  • 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 ($pk_1, pk_2, pk_3, ...$)

  ... but $k_0$ can also be used to spend funds

• Can we check balances without the spending key ??

Goal: 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: \( k_{\text{pub}} = (k_2, \ h = g^{k_1}) \)

for all \( i=1,2,... \):

\[
\begin{aligned}
&\text{sk}_i \leftarrow k_1 + \text{HMAC}(k_2, i) \\
&\text{pk}_i \leftarrow g^{sk_i} = g^{k_1} \cdot g^{\text{HMAC}(k_2,i)} = h \cdot g^{\text{HMAC}(k_2,i)}
\end{aligned}
\]

\( k_{\text{pub}} \) does not reveal \( \text{sk}_1, \text{sk}_2, \ldots \)

\( k_{\text{pub}} \): on laptop/phone, generates unlinkable addresses \( \text{pk}_1, \text{pk}_2, \ldots \)

\( k_0 \): on ledger

computed from \( k_{\text{pub}} \)
Paper wallet (be careful when generating)

Bitcoin address = \text{base58}(\text{hash}(PK))

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

signing key (cleartext)
Managing crypto assets in the cloud

How exchanges store assets
Coinbase: holds customer assets

Design: 98% of assets (SK) are held in cold storage

- **Cold storage (98%)**:
  - $k_0^{(1)}$
  - $k_0^{(2)}$
  - $k_0^{(3)}$

- **Hot wallet (2%)**: $h, k_2$

- **t-out-of-n secret sharing of $k_0$**

- **$k_0$** used to verify cold storage balances

- **$SK_{\text{hot}}$**: 2% of assets
Problems

Can’t prove ownership of assets in cold storage, without accessing cold storage:
• To prove ownership (e.g., in audit or in a proof of solvency)
• To participate in proof-of-stake consensus

Solutions:
• Keep everything in hot wallet (e.g., Anchorage)
• Proxy keys: keys that prove ownership of assets, but cannot spend assets
END OF LECTURE

Next lecture: consensus