Note: HW#1 is posted on the course web site. Due Sep. 28.
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

---

**coinbase Tx**
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]

- **Outputs**
  - output[0]
  - output[1]
  - witnesses
  - locktime

- **TxID (4 bytes)**
  - (excluding witnesses)
  - \( \text{TxID} = H(Tx) \)

- **ScriptSig (8 bytes)**
  - program

- **ScriptPK (8 bytes)**
  - program

- **value (8 bytes)**
  - \( \text{value} = \#\text{BTC}/10^8 \)

- **Lang (4 bytes)**
  - \( \text{lang} \)

---

- **Value**
  - 32 byte hash
  - 4 byte index
  - program

- **Ignore**
  - 8 bytes

- **ScriptSig**
  - seq

- **Output**
  - value

- **Input**
  - ScriptPK
Example

Tx1: (funding Tx)

UTXO: unspent Tx output

Tx2: (spending Tx)

identifies a UTXO
Example

Tx1: (funding Tx)

UTXO: unspent Tx output

Tx2: (spending Tx)

identifies a UTXO
Validating Tx2

Miners check (for each input):

1. The program \( \text{ScriptSig} \mid \text{ScriptPK} \) returns true.
2. \( \text{TxID} \mid \text{index} \) 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.
Transaction types: (1) P2PKH

Alice want 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 \(Addr_B \leftarrow \text{H}(pk_B)\)
- step 3: Bob sends \(Addr_B\) to Alice
- step 4: Alice creates Tx:

\[
\text{input:
  \[\begin{array}{c}
  5 \\
  \text{7 BTC}
  \end{array}\]
}

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

\[
\text{2 UTXO}_B\text{ for Bob}
\]

\[
\text{0 UTXO}_A\text{ for Alice (change)}
\]
Later, when Bob wants to spend his UTXO:

\[ \text{create a } T_{\text{spend}} \]

\[ T_{\text{spend}}: \]

\[ \text{TxID} | 0 \]

\[ \text{ScriptSig}_B: \]

\[ <\text{sig}> <\text{pk}_B> \]

\[ <\text{sig}> = \text{Sign}(sk_B, T) \quad \text{where} \quad T = (T_{\text{spend}} \text{ excluding all ScriptSigs}) \quad (\text{SIGHASH}\_\text{ALL}) \]

Miners validate that \[ \text{ScriptSig}_B | \text{ScriptPK}_B \] returns true
Segregated Witness

ECDSA malleability:

• given \( (m, \text{sig}) \) anyone can create \( (m, \text{sig}') \) with \( \text{sig} \neq \text{sig}' \)

⇒ miner can change sig in Tx, and change \( \text{TxID} = H(\text{Tx}) \)
⇒ 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
\( \text{TxID} = \text{Hash( Tx without witnesses }) \)
Transaction types: (2) P2SH: pay to script hash

(pre SegWit in 2017)

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

Usage:
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: 

\[
\text{HASH160 } \langle H(\text{redeem script}) \rangle \text{ EQUAL }
\]

ScriptSig to spend:

\[
\langle \text{sig}_1 \rangle \langle \text{sig}_2 \rangle \ldots \langle \text{sig}_n \rangle \langle \text{redeem script} \rangle
\]

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

(1) \(<\text{ScriptSig}>\) $\text{ScriptPK} = \text{true}$ ← spending Tx gave correct script

(2) $\text{ScriptSig} = \text{true}$ ← 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)

\[
<2> \ <\text{PK}_1> \ <\text{PK}_2> \ <\text{PK}_3> \ <3> \ \text{CHECKMULTISIG}
\]

hash gives P2SH address

ScriptSig to spend: (by payee)

\[
<0> \ <\text{sig}_1> \ <\text{sig}_3> \ <\text{redeem script}>
\]

(threshold)
Abstractly ...

Multisig address: $\textit{addr} = H(\text{PK}_1, \text{PK}_2, \text{PK}_3, \text{2-of-3})$

Tx1: (funding Tx)

7 BTC  
UTXO$_B$ for Bob  
UTXO$_A$ for Alice (change)

Tx2: (spending Tx)

input: UTXO, sig$_1$, sig$_3$, PK$_1$, PK$_2$, PK$_3$, 2-of-3  
output
Example Bitcoin scripts
Protecting assets with a co-signatory

Alice stores her funds in UTXOs for

\[ \text{addr} = \text{2-of-2}(\text{PK}_A, \text{PK}_S) \]

Alice stores her funds in UTXOs for

\[ \text{addr} = \text{2-of-2}(\text{PK}_A, \text{PK}_S) \]

Alice

PK\(_A\)

spending Tx

is this Alice

PK\(_S\)

BitGo

yep, it’s me

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

\(<\text{sig}_S> \text{ on Tx} \)

⇒ theft of Alice’s SK\(_A\) 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

\[ addr = 2-of-3(PK_A, PK_B, PK_J) \]

post payment of 0.11₿ to \( addr \)

(U TXO \(_A\))

want backpack for 0.1₿

once see Tx on chain

mail backpack

backpack arrives

send \(<\text{sig}_A>\) on Tx:

\[ \text{UTXO}_A \rightarrow (PK_B:0.1, PK_A:0.01) \]

redeem using \(<\text{sig}_A> <\text{sig}_B>\) on Tx

Judge

\( PK_J \)
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 \langle\text{sig}_{\text{Judge}}\rangle 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 (may pay interest)
- **laptop/phone**: Electrum, MetaMask, …
- **hardware**: Trezor, Ledger, …
- **paper**: print all sk on paper
- **brain**: memorize sk (bad idea)

Lost key $\Rightarrow$ lost funds
How does a wallet display Alice’s current balances?

• Laptop/phone wallet needs to verify an incoming payment

• **Goal**: do so w/o downloading entire blockchain (300 GB)

**SPV**: (1) download all block headers (52 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
Lose hardware wallet ⇒ loss of funds. What to do?

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

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

$p_{k1}, p_{k2}, p_{k3}, ...$: 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 \((24 \times 11 = 268\) bits\)
  - list of 2048 words in different languages (BIP 39)
Example: English word list

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>abandon</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ability</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>able</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>about</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>above</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>absent</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>absorb</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>abstract</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>absurd</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>abuse</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2046</td>
<td>zero</td>
<td></td>
</tr>
<tr>
<td>2047</td>
<td>zone</td>
<td></td>
</tr>
<tr>
<td>2048</td>
<td>zoo</td>
<td></td>
</tr>
</tbody>
</table>

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 \(i=1,2,\ldots:\)

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

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

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

\(k_0\): on ledger
Paper wallet (be careful when generating)

Bitcoin address = base58(hash(PK))

base58 = a-zA-Z0-9 without {0,O,I,1}
Managing crypto assets: Exchanges
Hot/cold storage

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

$$k_0^{(1)}$$
$$k_0^{(2)}$$
$$k_0^{(3)}$$

Cold storage (98%)

$$k_0$$

Hot wallet (2%)

$$h, k_2$$ used to verify cold storage balances

$$SK_{hot}$$ 2% of assets

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

Customers
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