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)

- **BH₂**
  - prev
  - Tx root

- **BH₃**
  - prev
  - Tx root

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

Tx cannot be erased: mistaken Tx $\Rightarrow$ loss of funds
**Tx structure (non-coinbase)**

**Inputs**
- `input[0]`
- `input[1]`
- `input[2]`
- `locktime` (4 bytes)

**Outputs**
- `output[0]`
- `output[1]`
- `witnesses` (segwit)
- `TxID = H(Tx)` (excluding witnesses)

**Input**
- `TxID`
- `out-index` (4 byte index)
- `ScriptSig`
- `seq`

**Output**
- `value`
- `ScriptPK`

**ScriptSig**
- Program

**Value**
- `value = #BTC/10^8`

**Earliest block # that can include Tx**
**Example**

**Tx1:** (funding Tx)

UTXO: unspent Tx output

- **UTXO₁**: input
  - value
  - ScriptPK
  - locktime

- **UTXO₂**: output
  - null locktime

**Tx2:** (spending Tx)

- **UTXO₃**: output
- **UTXO₄**: output

**Identifies a UTXO**
Example

Tx1: (funding Tx)

TxO: 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 wants 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 creates Tx:

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

\[
\begin{align*}
\text{input} & \quad 5 \quad \text{ScriptPK}_B & \quad 2 \quad \text{ScriptPK}_A & \quad 0 \\
7 \text{ BTC} & \quad \text{UTXO}_B \text{ for Bob} & \quad \text{UTXO}_A \text{ for Alice (change)}
\end{align*}
\]
Later, when Bob wants to spend his UTXO:

\[ \text{create a Tx}_{\text{spend}} \]

Tx$_{\text{spend}}$:  

\[
\begin{array}{c|c}
\text{TxID} & 0 \\
\hline
\text{ScriptSig}_B
\end{array}
\]

points to UTXO$_B$

ScriptSig$_B$:  

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

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

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

ECDSA malleability:

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

\[ \Rightarrow \] miner can change sig in Tx and change \( \text{TxID} = \text{SHA256}(\text{Tx}) \)

\[ \Rightarrow \] Tx issuer cannot tell what TxID is, until Tx is posted

\[ \Rightarrow \] leads to problems and attacks

**Segregated witness:** signature is moved to witness field in Tx

\( \text{TxID} = \text{Hash(\text{Tx without witnesses})} \)
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: \[
\text{HASH160 } <\text{H(redeem script)}> \text{ EQUAL}
\]

**ScriptSig** to spend: \[
<\text{sig}_1> <\text{sig}_2> ... <\text{sig}_n> <\text{redeem script}>
\]

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

1. `<ScriptSig> ScriptPK = true` ← spending Tx gave correct script

2. `ScriptSig = 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)

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

Threshold

Hash gives P2SH address

ScriptSig to spend: (by payee)

\[<0> \ <\text{sig1}> \ <\text{sig3}> \ <\text{redeem script}>\]
Abstractly ...

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

**Tx1:**
(funding Tx)

- **input:** 7 BTC
- **addr:**
- **UTXO\(_B\) for Bob**
- **ScriptPK\(_A\)**
- **UTXO\(_A\) for Alice (change)**

**Tx2:**
(spending Tx)

- **input:** UTXO, \( \text{sig}_1, \text{sig}_3, PK_1, PK_2, PK_3, 2\text{-of-3} \)
- **output:** 0
Interesting Bitcoin scripts
Protecting assets with a co-signatory

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

Post Tx with $$\langle \text{sig}_A \rangle < \text{sig}_S \rangle$$

⇒ 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

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

Alice

PK_A

post payment of 0.11₿ to \( addr \)

(UTXO_A)

want backpack for 0.1₿

once see Tx on chain

mail backpack

backpack arrives

send <sig_A> on Tx:

\[ UTXO_A \rightarrow (PK_B:0.1, PK_A:0.01) \]

Bob

PK_B

Judge

PK_J

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:

\[ \text{Tx: } ( \text{UTXO}_A \rightarrow \text{PK}_A, \text{sig}_A, \text{sig}_{\text{Judge}} ) \]  [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.
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: prog. 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**: Electrum, MetaMask, ...
- **hardware**: Trezor, Ledger
- **paper**: print all sk on paper
- **brain**: memorize seed (bad idea)

Lost key ⇒ lost funds
Simplified Payment Verification (SPV)

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
Hardware wallet: backup

Lose hardware wallet ⇒ loss of funds. What to do?

**Idea 1:** generate a secret seed $k_0 \in \{0,1\}^{256}$
for $i=1,2,\ldots:$ \[ sk_i \leftarrow \text{HMAC}(k_0, i) \, , \, \, \, pk_i \leftarrow g^{sk_i} \]

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

$k_0$ is stored on HW device and in offline storage (as 24 words)
⇒ 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 \((24 \times 11 = 268 \text{ bits})\)
  • list of 2048 words in different languages (BIP 39)
### Example: English word list

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

Save list of 24 words
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 \((\text{BIP 39})\)

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 \( i=1,2,...: \)
\[
\begin{align*}
\text{sk}_i & \leftarrow k_1 + \text{HMAC}(k_2, i) \\
\text{pk}_i & \leftarrow g^{\text{sk}_i} = g^{k_1} \cdot g^{\text{HMAC}(k_2,i)} = h \cdot g^{\text{HMAC}(k_2,i)}
\end{align*}
\]

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

\( k_{\text{pub}} \): on laptop/phone, generates unlinkable addresses \( pk_1, pk_2, ... \)
\( k_0 \): on ledger

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

Bitcoin address = base58(hash(PK))

base58 = a-zA-Z0-9 without {0,0,L,1}
Managing crypto assets: Exchanges
Coinbase: holds customer assets
Design: 98% of assets (SK) are held in cold storage

Cold storage (98%)

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

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

Hot wallet (2%)

$$h, k_2$$

used to verify cold storage balances

$$SK_{hot}$$

2% of assets

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