Note: HW#1 is posted on the course web site. Due Tue, Oct. 10.
Recap: the Bitcoin blockchain
View the blockchain as a sequence of Tx (append-only)

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

- **Inputs**
  - input[0]
  - input[1]
  - input[2]

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

- **Witnesses** (part of input)

- **Locktime** (4 bytes)

- **TxID** = H(Tx) (excluding witnesses)

**Input:**
- 32 byte hash
- 4 byte index
- Program

**Output:**
- Value
- 8 bytes
- ScriptPK

**Program**

#BTC = value/10^8
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)

null locktime

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. $\text{sum input values} \geq \text{sum output values}$

After Tx2 is posted, miners remove UTXO 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 \(addr_B \leftarrow H(pk_B)\)
- step 3: Bob sends \(addr_B\) to Alice
- step 4: Alice posts Tx:
Transaction types: (1) P2PKH

Pay to public key hash

“input” contains ScriptSig that authorizes spending Alice’s UTXO

- example: ScriptSig contains Alice’s signature on Tx

  ⇒ miners cannot change ScriptPK_B (will invalidate Alice’s signature)

```
input 7 BTC  UTXO_B for Bob  UTXO_A for Alice (change)
```

ScriptPK_B:

```
DUP HASH256 <addr_B> EQVERIFY CHECKSIG
```
Transaction types: (1) P2PKH

Later, when Bob wants to spend his UTXO:

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

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

- **TxID** 0
- **ScriptSig$_B$**

points to UTXO$_B$

$<\text{sig}> <\text{pk}_B>$

(authorizes spending UTXO$_B$)

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

Miners validate that $\text{ScriptSig}_B \mid \text{ScriptPK}_B$ returns true
**P2PKH: comments**

- Alice specifies recipient’s pk in UTXO$_B$

- Recipient’s pk is not revealed until UTXO is spent  
  (some security against attacks on pk)

- Miner cannot change $<\text{Addr}_B>$ and steal funds:  
  invalidates Alice’s signature that created UTXO$_B$
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 \(\text{TxID} = \text{SHA256}(Tx)\)
\[\Rightarrow\] Tx issuer cannot tell what \(\text{TxID}\) 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)

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. `<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)

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

hash gives P2SH address

ScriptSig to spend: (by payee)

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

(threshold)
Abstractly ...

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

Tx1: (funding Tx)
- Input: 7 BTC
- Input UTXO for Bob
- Input UTXO for Alice (change)

Tx2: (spending Tx)
- Input: UTXO, \( \text{sig}_1, \text{sig}_3, PK_1, PK_2, PK_3, 2-of-3 \)
- Output
Example Bitcoin scripts
Protecting assets with a co-signatory

Alice stores her funds in UTXOs for $\text{addr} = 2\text{-of}-2(\text{PK}_A, \text{PK}_S)$

Alice stores her funds in UTXOs for $\text{addr} = 2\text{-of}-2(\text{PK}_A, \text{PK}_S)$

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

$\Rightarrow$ theft of Alice’s $\text{SK}_A$ 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.

```plaintext
addr = 2-of-3(PK_A, PK_B, PK_J)
```

Alice

PK_A

Bob

PK_B

Judge

PK_J

post payment of 0.11₿ to addr (creates 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)

redeem using <sig_A> <sig_B> on Tx

mail backpack
(1) Backpack never arrives: (Bob at fault)

Alice gets her funds back with help of Judge and a Tx:

\[ \text{Tx: } ( \text{UTXO}_A \rightarrow \text{PK}_A, \text{sig}_A, \text{sig}_{\text{Judge}} ) \quad \text{[2-out-of-3]} \]

(2) Alice never sends \( \text{sig}_A \): (Alice at fault)

Bob gets paid with help of Judge and a Tx:

\[ \text{Tx: } ( \text{UTXO}_A \rightarrow \text{PK}_B, \text{sig}_B, \text{sig}_{\text{Judge}} ) \quad \text{[2-out-of-3]} \]

(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, Keystone, ...

• **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 (60 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)
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 $i=1,2,...$: $sk_i \leftarrow \text{HMAC}(k_0, i)$, $pk_i \leftarrow g^{sk_i}$

$pk_1, pk_2, pk_3, ...$: 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 (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

save list of 24 words
Careful with unused letters ...
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 \( p_{k_1}, p_{k_2}, \ldots \)

\( k_0 \): on ledger

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

Bitcoin address = base58(hash(PK))

base58 = a-zA-Z0-9 without {0,O,l,1}
Managing crypto assets in the cloud

How exchanges store assets
Hot/cold storage

Coinbase: holds customer assets

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

cold storage (98%)

hot wallet (2%)

\[ k_0^{(1)} \]
\[ k_0^{(2)} \]
\[ k_0^{(3)} \]

\( k_0 \)

t-out-of-n secret sharing of \( k_0 \)

\( h, k_2 \) used to verify cold storage balances

\( \text{SK}_{\text{hot}} \) 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
Next lecture: consensus