Reminder: proj #1 is posted on the course web site. Due Oct. 4
Recap

(1) **SHA256**: a collision resistant hash function that outputs 32-byte hash values

**Applications:**

- a binding commitment to one value: $\text{commit}(m) \rightarrow H(m)$
  or to a list of values: $\text{commit}(m_1, \ldots, m_n) \rightarrow \text{Merkle}(m_1, \ldots, m_n)$

- Proof of work with difficulty $D$:
  
  given $x$ find $y$ s.t. $H(x, y) < 2^{256}/D$ takes time $O(D)$
Digital Signatures

Physical signatures: bind transaction to author

Problem in the digital world:

anyone can copy Bob’s signature from one doc to another
Digital signatures

Solution: make signature depend on document

Bob agrees to pay Alice 1$
**Def:** A signature scheme is a triple of algorithms:

- **Gen():** outputs a key pair $(pk, sk)$
- **Sign(sk, msg):** outputs sig. $\sigma$
- **Verify(pk, msg, $\sigma$):** outputs ‘accept’ or ‘reject’

**Secure signatures:** (informal)

Adversary who sees signatures on many messages of his choice, cannot forge a signature on a new message.
Families of signature schemes

1. RSA signatures (old ... not used in blockchains):
   - long sigs and public keys (≥256 bytes), fast to verify

2. Discrete-log signatures: Schnorr and ECDSA (Bitcoin, Ethereum)
   - short sigs (48 or 64 bytes) and public key (32 bytes)

3. BLS signatures: 48 bytes, aggregatable, easy threshold
   (Ethereum 2.0, Chia, Dfinity)

4. Post-quantum signatures: long (≥600 bytes)

   details in CS255
Signatures on the blockchain

Signatures are used everywhere:
• ensure Tx authorization,
• governance votes,
• consensus protocol votes.

\[ \text{sk}_1 \quad \text{data} \quad \text{signatures} \]

\[ \text{sk}_2 \quad \text{data} \quad \text{signatures} \]
In summary ...

Digital signatures: (Gen, Sign, Verify)

Gen() $\rightarrow$ (pk, sk),

Sign(sk, m) $\rightarrow$ $\sigma$, 

Verify(pk, m, $\sigma$) $\rightarrow$ accept/reject

signing key

verification key
Bitcoin mechanics
This lecture: Bitcoin mechanics

Total market value:

- Jan. 2009: Bitcoin network launched
- Sep. 2023: $528B
This lecture: Bitcoin mechanics

- user facing tools (cloud servers)
- applications (DAPPs, smart contracts)
- Execution engine (blockchain computer)
- Sequencer: orders transactions
- Data Availability / Consensus Layer

today

next week
First: overview of the Bitcoin consensus layer

Bitcoin P2P network

end users

signed Tx

\(\text{sk}_A\)

\(\text{sk}_B\)

\(\text{sk}_C\)

typically, miners are connected to eight other peers (anyone can join)
First: overview of the Bitcoin consensus layer

miners broadcast received Tx to the P2P network

every miner:
  validates received Tx and
  stores them in its **mempool** (unconfirmed Tx)

note: miners see all Tx before they are posted on chain
Every $\approx 10$ minutes:

- Each miner creates a candidate block from Tx in its mempool
- a “random” miner is selected (how: next week), and broadcasts its block to P2P network
- all miners validate new block
Selected miner is paid 6.25 BTC in coinbase Tx (first Tx in the block)

- only way new BTC is created
- block reward halves every four years
  \[\Rightarrow\text{ max } 21\text{M BTC } (\text{currently } 19.6\text{M BTC})\]

note: miner chooses order of Tx in block
Next week:

**Safety / Persistence:**
- to remove a block, need to convince 51% of mining power *

**Liveness:**
- to block a Tx from being posted, need to convince 51% of mining power **

(some sub 50% censorship attacks, such as feather forks)
Bitcoin blockchain: a sequence of block headers, 80 bytes each

- **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

- ...
Bitcoin blockchain: a sequence of block headers, 80 bytes each

**time**: time miner assembled the block. Self reported.
(block rejected if too far in past or future)

**bits**: proof of work difficulty

**nonce**: proof of work solution

**Merkle tree**: payer can give a short proof that Tx is in the block

new block every ≈10 minutes.
# An example

<table>
<thead>
<tr>
<th>Height</th>
<th>Mined</th>
<th>Miner</th>
<th>Size</th>
<th>#Tx</th>
</tr>
</thead>
<tbody>
<tr>
<td>648494</td>
<td>17 minutes</td>
<td>Unknown</td>
<td>1,308,663 bytes</td>
<td>1855</td>
</tr>
<tr>
<td>648493</td>
<td>20 minutes</td>
<td>SlushPool</td>
<td>1,317,436 bytes</td>
<td>2826</td>
</tr>
<tr>
<td>648492</td>
<td>59 minutes</td>
<td>Unknown</td>
<td>1,186,609 bytes</td>
<td>1128</td>
</tr>
<tr>
<td>648491</td>
<td>1 hour</td>
<td>Unknown</td>
<td>1,310,554 bytes</td>
<td>2774</td>
</tr>
<tr>
<td>648490</td>
<td>1 hour</td>
<td>Unknown</td>
<td>1,145,491 bytes</td>
<td>2075</td>
</tr>
<tr>
<td>648489</td>
<td>1 hour</td>
<td>Poolin</td>
<td>1,359,224 bytes</td>
<td>2622</td>
</tr>
</tbody>
</table>
# Block 648493

<table>
<thead>
<tr>
<th><strong>Timestamp</strong></th>
<th>2020-09-15 17:25</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Height</strong></td>
<td>648493</td>
</tr>
<tr>
<td><strong>Miner</strong></td>
<td>SlushPool</td>
</tr>
<tr>
<td></td>
<td><em>(from coinbase Tx)</em></td>
</tr>
<tr>
<td><strong>Number of Transactions</strong></td>
<td>2,826</td>
</tr>
<tr>
<td><strong>Difficulty</strong></td>
<td>17,345,997,805,929.09</td>
</tr>
<tr>
<td></td>
<td><em>(adjusts every two weeks)</em></td>
</tr>
<tr>
<td><strong>Merkle root</strong></td>
<td>350cbb917c918774c93e945b960a2b3ac1c8d448c2e67839223bbcf595baff89</td>
</tr>
<tr>
<td><strong>Transaction Volume</strong></td>
<td>11256.14250596 BTC</td>
</tr>
<tr>
<td><strong>Block Reward</strong></td>
<td>6.25000000 BTC</td>
</tr>
<tr>
<td><strong>Fee Reward</strong></td>
<td>0.89047154 BTC</td>
</tr>
<tr>
<td></td>
<td><em>(Tx fees given to miner in coinbase Tx)</em></td>
</tr>
</tbody>
</table>
View the blockchain as a sequence of Tx (append-only)
**Tx structure (non-coinbase)**

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

- **outputs**
  - output[0]
  - output[1]
  - witnesses (part of input)

- **locktime** (4 bytes)

- **segwit**
  - 32 byte hash
  - 4 byte index
  - program
  - ignore

- **ScriptSig**
  - 8 bytes
  - program

- **TxID**
  - 32 byte hash
  - out-index
  - seq

- **ScriptPK**
  - 8 bytes

- **ScriptSig**
  - value

- **ScriptPK**
  - #BTC = value/10^8

**Earliest block # that can include Tx**
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

null locktime

UTXO: unspent Tx output
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 from UTXO set.
An example (block 648493)

**COINBASE (Newly Generated Coins)**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00000000 BTC</td>
<td>6.25 + Tx fees = 7.14047154 BTC</td>
</tr>
</tbody>
</table>

| 3PuJbxJS1pKxf8EdVR18yBkD1fPAbgUtyw | input |
| 0.72333974 BTC (input UTXO value) | 1E5Ao1VUnA5BhffvXf2Xmd6avUgwkF3jv | 0.00917379 BTC |
| bc1qr8k3e0xv06lipu3j7m858pa2ak9yr56ttvvef | bc1qdrxve8kua3y5dgx6wf3u95nh0d3e648... | 0.09290152 BTC |
| 14ZhjuXpQ5jCDjtAy7ZMu3hfeQCWewzLw7 | 0.06616444 BTC |

| Tx0 | Tx1 | Tx2 |
| 1CK6KHY6MHgYvmRQ4PAafKYDrg1ejbH1cE | 1E5Ao1VUnA5BhffvXf2Xmd6avUgwkF3jv | 3G3C2RFQ8gsf77EQpdR4ZReChWFEHhxVU |
| 0.00000000 BTC OP_RETURN OP_RETURN | 0.05000000 BTC | 0.04808000 BTC |
| 0.00192000 BTC (Tx fee) | 0.04808000 BTC | 0.04808000 BTC |

sum of fees in block added to coinbase Tx
Bitcoin average Tx fees in USD (last 60 days, sep. 2023)

$2.11

Bitcoin average Tx fees in USD (all time)
All value in Bitcoin is held in UTXOs

Unspent Transaction Outputs
The total number of valid unspent transaction outputs. This excludes invalid UTXOs with opcode OP_RETURN

Sep. 2023: miners need to store ≈130M UTXOs in memory
### Focusing on Tx2: TxInp[0]

#### Value
0.05000000 BTC

#### Pkscript
- OP_DUP
- OP_HASH160
- 45b21c8a0cb687d563342b6c729d31dab58e3a4e
- OP_EQUALVERIFY
- OP_CHECKSIG

#### Sigscript
- 304402205846cace0d73de82dfbdeba4d65b9856d7c1b1730eb401cf4906b2401a69b
dc90220589d36d36be64e774c8796b96c011f29768191abebed7f56ba20ff0351280860c01
- 03557c228b080703d52d72ead1bd93fc72f45c4dddb4c2b7a20c458e2d069c8dd9e

from UTXO (Bitcoin script)

from TxInp[0]
A stack machine. Not Turing Complete: no loops.

Quick survey of op codes:

1. **OP_TRUE** (OP_1), **OP_2**, ..., **OP_16**: push value onto stack
   - 81 82 96

2. **OP_DUP**: push top of stack onto stack
   - 118
3. control:

- **OP_IF** <statements> **OP_ELSE** <statements> **OP_ENDIF**

- **OP_VERIFY**: abort fail if top = false

- **OP_RETURN**: abort and fail

  what is this for? ScriptPK = [OP_RETURN, <data>]

- **OP_EQVERIFY**: pop, pop, abort fail if not equal
4. arithmetic:
   \texttt{OP\_ADD}, \texttt{OP\_SUB}, \texttt{OP\_AND}, ...: pop two items, add, push

5. crypto:
   \texttt{OP\_SHA256}: pop, hash, push

   \texttt{OP\_CHECKSIG}: pop pk, pop sig, verify sig. on Tx, push 0 or 1

6. Time: \texttt{OP\_CheckLockTimeVerify} (CLTV):
   fail if value at the top of stack > Tx locktime value.
   usage: UTXO can specify min-time when it can be spent
Example: a common script

| <sig> | <pk> | DUP HASH256 <pkhash> | EQVERIFY CHECKSIG |

**stack:**
- empty
- <sig> <pk>
- <sig> <pk> <pk>
- <sig> <pk> <hash>
- <sig> <pk> <hash> <pkhash>
- <sig> <pk>
- 1

⇒ successful termination

verify(pk, Tx, sig)
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:

\[
\begin{align*}
\text{input} & \quad 5 & \quad \text{UTXO}_B \text{ for Bob} & \quad \text{UTXO}_A \text{ for Alice (change)} \\
7 \text{ BTC} & \quad \text{ScriptPK}_B & \quad 2 & \quad \text{ScriptPK}_A & \quad 0
\end{align*}
\]

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

Point to Alice’s UTXO
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

  $$\implies$$ miners cannot change ScriptPK_B (will invalidate Alice’s signature)

```
input
7 BTC

UTXO_B for Bob
5 ScriptPK_B
2 UTXO_A for Alice (change)

ScriptPK_B:
DUP HASH256 <addr_B> EQVERIFY CHECKSIG
```

Point to Alice’s UTXO

0
Transaction types: (1) P2PKH

Later, when Bob wants to spend his UTXO:

create a $\text{Tx}_{\text{spend}}$

$\text{Tx}_{\text{spend}}$: $\text{TxID} | 0$ $\text{ScriptSig}_B$

points to $\text{UTXO}_B$

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

(authorizes spending $\text{UTXO}_B$)

$\langle \text{sig} \rangle = \text{Sign}(\text{sk}_B, \text{Tx})$ \ where \ $\text{Tx} = (\text{Tx}_{\text{spend}} \text{excluding all ScriptSigs})$ \ (SIGHASH_ALL)

Miners validate that $\text{ScriptSig}_B \ | \ \text{ScriptPK}_B$ 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 $<$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 TxID = SHA256(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

TxID = 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: payee publishes hash(redeem script) ← Bitcoint addr.
payer sends funds to that address

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

ScriptSig to spend: <sig_1> <sig_2> ... <sig_n> <redeem script>

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

(1) <ScriptSig> ScriptPK = true ← payee 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: (set by payer)

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

hash gives P2SH address

ScriptSig to spend: (by payee)

\[
<0> \text{ } <\text{sig1}> \text{ } <\text{sig3}> \text{ } <\text{redeem script}>
\]
Next lecture: interesting scripts, wallets, and how to manage crypto assets