CS2361: Blockchain and Cryptocurrencies

Spring 2022

Mahavir Jhawar

Department of Computer Science
Ashoka University

Table of Contents

1. Lecture#1 (18/01) [Course Overview, Bitcoin Introduction]
2. Lecture#2 (21/01) [Hash Functions]
3. Lecture#3 (25/01) [Random Oracle Model, Iterated Hash Construction]
4. Lecture#4 (28/01) [Hash Puzzle, Signature Scheme]
5. Lecture#5 (01/02) [Group Theory]
6. Lecture#6 (04/02) [Discrete Logarithm Problem, DSA]
7. Lecture#7 (04/02) [Elliptic Curves, ECDSA]
8. Lecture#8 (11/02) [Bitcoin Address Generation]
9. Lecture#9 (15/02) [Bitcoin Transaction and its Validation]
10. Lecture#10 (18/02) [Bitcoin Consensus]
11. Lecture#11 (22/02) [Block Mining, txn Serialization]
12. Lecture#12 (25/02) [txn Serialization, PaytoScriptHash]
13. Lecture#13 (01/03) [More lockScripts, txn-malleability]
14. Lecture#14 (04/03) [SegWit transaction]
15. Lecture#15 (15/03) [Analysis of Bitcoin's Consensus]
16. Lecture#16 (21/03) [Hyperledger Fabric]
17. Lecture#17 (25/04) [Ethereum]
18. Lecture#18 (29/03) [Ethereum Contract Accounts]
19. Lecture#19 (05/04) [Payment Channels, Coin Mixers]
20. Lecture#20 (08/04) [Coin Mixers, MixEth, TumbleBit]
21. Lecture#21 (19/04) [Tokens]

ashoka UNIVERSITY

Course Overview

- Off-class communication Google Classroom
- Course Website https://sites.google.com/ashoka.edu.in/cs2361spring2022/home
- Grading Rubric

Evaluation Type	Weightage		Letter Grade	Percentage Bracket
Mid Term	15%		F	< 40
Assignments	40%		D-	$40 - 44$
Final Term	15%		D	$45 - 49$
Course Project	30%		D+	$50 - 54$
			C-	$55 - 59$
			C	$60 - 64$
			C+	$65 - 69$
			B-	$70 - 74$
			B	$75 - 79$
			B+	$80 - 84$
			A-	$85 - 95$
			A	> 95

- TA
  - Priyam Panda (priyam.panda_asp22@ashoka.edu.in)
  - Yash More (yash.more_asp22@ashoka.edu.in)
- References
  - Lecture Slides, Notes and Handouts.
  - Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction. Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, Steven Goldfeder. Princeton University Press.
  - Mastering Bitcoin: Unlocking Digital Cryptocurrencies, by Andreas M. Antonopoulos
  - Mastering Ethereum, by Andreas M. Antonopoulos and Gavin Wood

Course Overview

- Course Overview
- The course will provide a comprehensive introduction to principles and core topics in crypto-currencies and the wider blockchain space.
  - In reality, there exists significant confusion - about what Blockchain technology is and how it works? Here is a funny take on the hype/confusion: Presentation by Jon Callas at Crypto 2018 Rump Session
- The course will help cut through the hype (hopefully!) and get to the core of what makes Blockchain unique.
  - For that, it is important to identify and address the technical and theoretical foundations of Blockchain technology.
- **Foundations:** Cryptography; Peer-to-Peer Networking (basic familiarity); Distributed computing (basic familiarity); Solid programming skills; Full-stack development.
  - Blockchain technology and its applications make heavy use of cryptography.
  - It is an indispensable tool used to bootstrap security in Blockchain.
  - We require solid understanding of few specific cryptographic primitives: **hash functions, signature schemes, PKI**
1. We will begin with a fast-paced introduction to relevant concepts from cryptography.

Course Overview

2. The course will then focus extensively on Bitcoin (`https://bitcoin.org/en/`) and distributed consensus.
   - **In the process we will explore a list of important concepts and enablers relating to blockchain technology**.
   - The cryptocurrency Bitcoin remains Blockchain's most coveted application till date!
   - Bitcoin (฿) - is a new kind of money and an innovative payment network.
   - Bitcoin's market capitalisation as of today: `https://coinmarketcap.com/`
   - As Bitcoin and alternative cryptocurrencies (*Altcoins*) like Ethereum, Ripple (and several hundreds other) continue to grow and mature in market value, it is becoming increasingly likely that Blockchains as a means to facilitate financial transactions are here to stay.

---

3. We will next introduce Ethereum (`https://ethereum.org/en/`)
   - Unlike Bitcoin, Ethereum proposed to utilize blockchain technology not only for maintaining a decentralized payment network but also for storing computer code which can be used to power tamper-proof decentralized financial contracts and applications.

---

4. The course will finally introduce Hyperledger Fabric blockchain platform (`https://www.hyperledger.org/`)
   - Hyperledger Fabric is the modular blockchain framework that is very popular for enterprise blockchain platforms.

Bitcoin: Lets make an informal beginning!

- What is Bitcoin ? To start with, its a currency, like INR, USD etc
- How to get Bitcoin ? One way - get it from someone who already has it !!

- Currency Denomination

Currency	Unit	Subunit	
INR	1 INR (₹)	1 Paisa	1 INR = 100 Paisa
USD	1 USD ($)	1 Cent	1 USD = 100 Cents
Bitcoin	1 Bitcoin (₿)	1 Satoshi	1 Bitcoin = 10^8 Satoshi

- Exchange Rate

$$\begin{cases} 1₿ \approx 42,200.18\ \$ \approx ₹\ 3134701.11 \text{ (as on January 18, 2022 https://coinmarketcap.com/)} \\ 1 Satoshi = 0.00000001₿ \approx 0.0004220018\ \$ \approx ₹\ 0.031 \\ 0.01₿ \approx 422.0018\ \$ \approx ₹\ 31344.35 \end{cases}$$

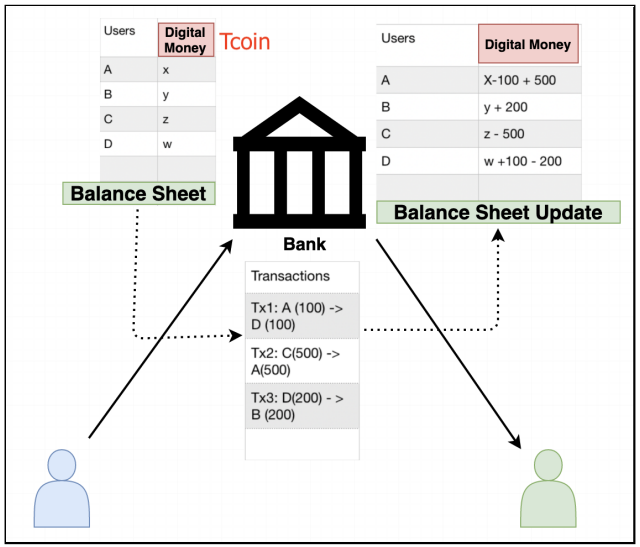
- Paper Money

Currency	Currency note
₹ 1	
1 $	
1 ₿	???

** Bitcoin is a virtual currency. It doesn't exist in the kind of physical form that the INR/USD exist in

Bank-issued Virtual Currency (Digital Money)

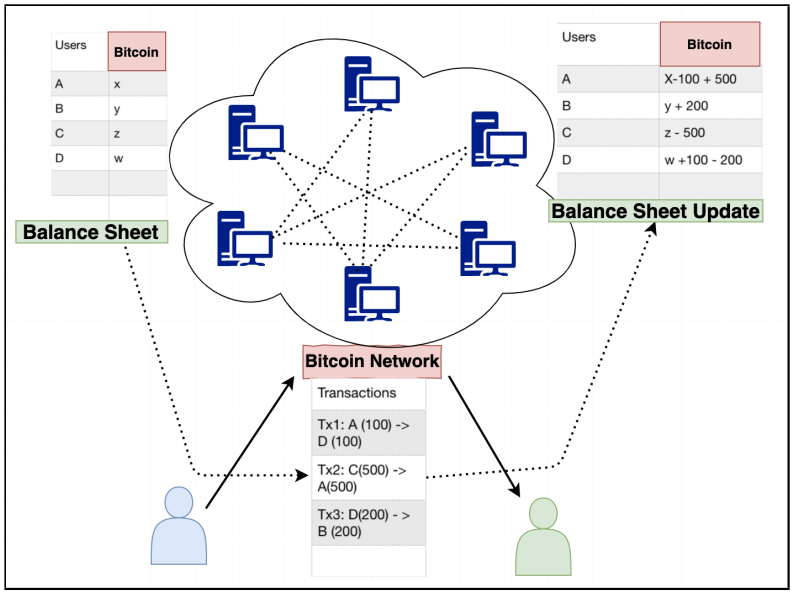
- Let's create a bank-issued virtual currency - $\mathcal{T}coin$
- Exchange rate: $\stackrel{\delta}{=} 1\ \mathcal{T}coin = ₹\ 10$
- How to get $\mathcal{T}coin$
  - Open a bank account that can receive $\mathcal{T}coin$s
  - Minting $\mathcal{T}coin$ Pay ₹ 10x to bank to receive x $\mathcal{T}coin$s to your account
  - You can request bank to transfer $\mathcal{T}coin$s to another account



- The bank is in complete charge of managing $\mathcal{T}coin$. All $\mathcal{T}coin$ transactions are routed through the bank!
- Reduce paper consumption

- A few concerns
- Anonymity: Bank has a record of every single transaction of yours
- Value of $\mathcal{T}coin$ currency $= 10 \times$ value of ₹

Bitcoin

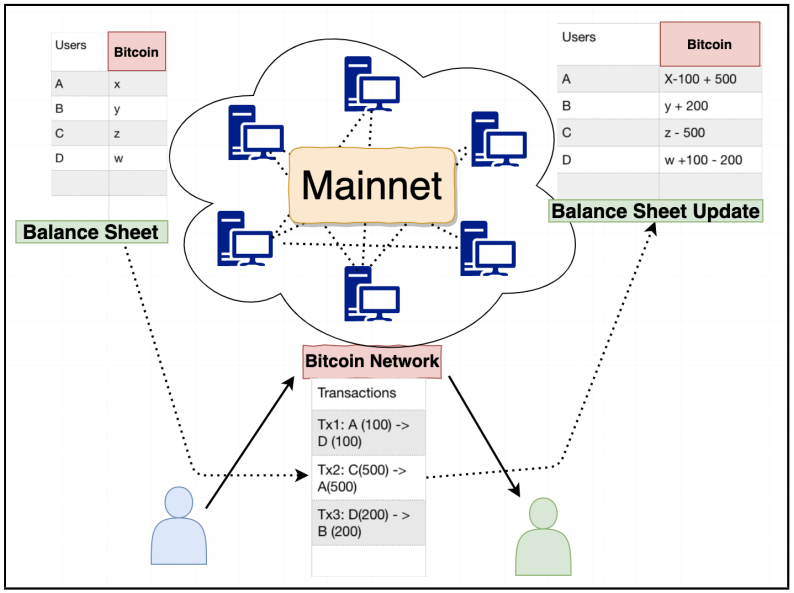


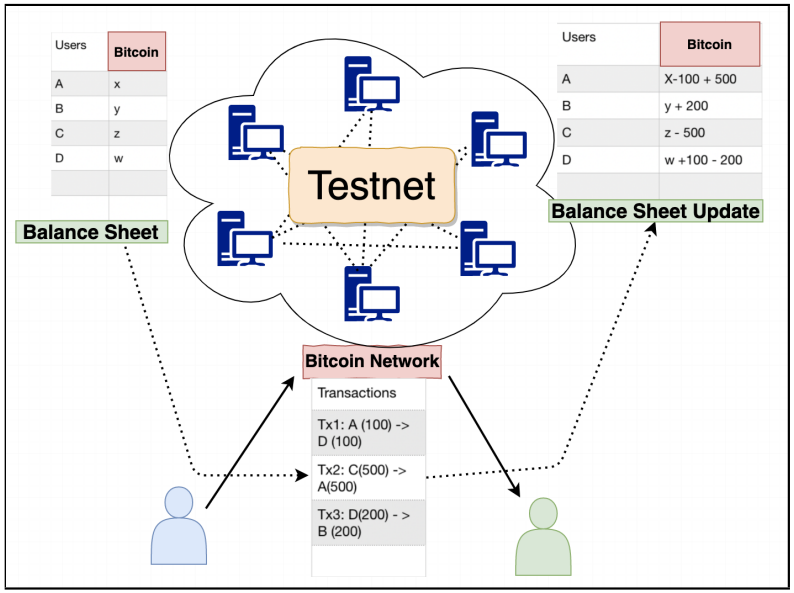
- Bitcoin is managed by Bitcoin network (no bank)
- Open a Bitcoin account address with the Bitcoin network to receive ₿s
- Minting Bitcoins ??
- For now, pay FIAT money to somebody who already has ₿s to receive some to your Bitcoin account !!

- Make Bitcoin transactions to transfer bitcoins from one account to another
- All transactions must be routed through and approved by the Bitcoin network !

- Unlike real banks, the Bitcoin network allows everyone an access to its ledger content !!
- Everyone have access to all transactions ever made, the process of approving transactions, account balance etc.

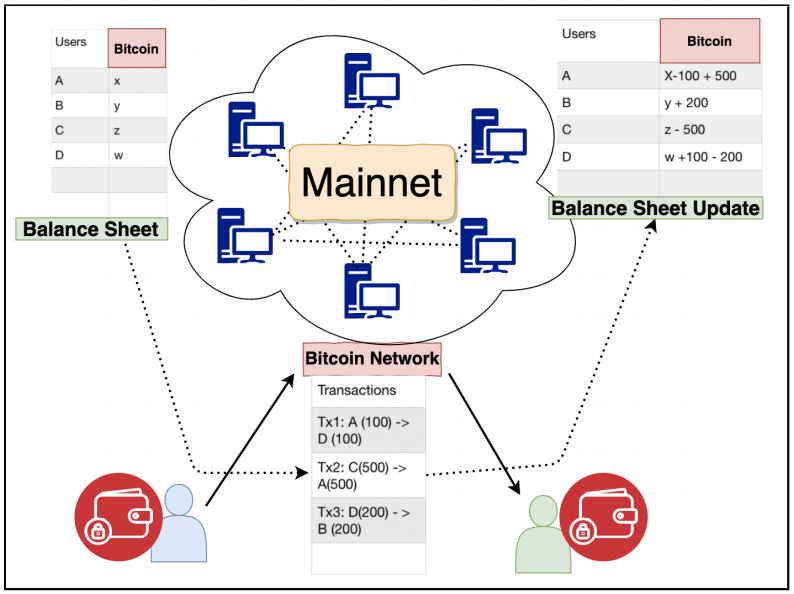
Bitcoin: Mainnet Vs Testnet

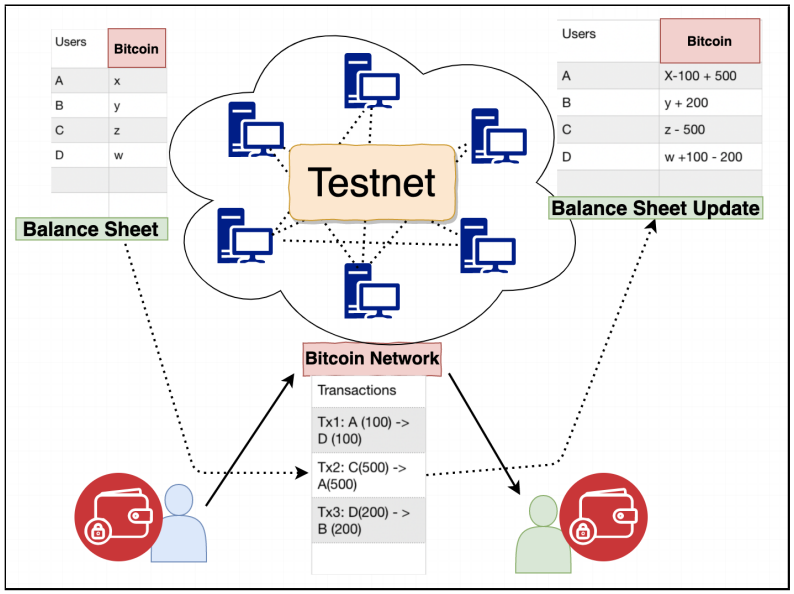




- Search engine for Bitcoin networks (both mainnet and testnet): Bitcoin Explorers
  - `https://www.blockchain.com/explorer`
  - `https://blockchair.com/bitcoin`
  - `https://live.blockcypher.com/btc/`

Bitcoin: Wallets





- Wallets
  - Use Web/Mobile Clients
  - Example - `https://www.blockchain.com/wallet`

Table of Contents

1 Lecture#1 (18/01) [Course Overview, Bitcoin Introduction]

2 Lecture#2 (21/01) [Hash Functions]

3 Lecture#3 (25/01) [Random Oracle Model, Iterated Hash Construction]

4 Lecture#4 (28/01) [Hash Puzzle, Signature Scheme]

5 Lecture#5 (01/02) [Group Theory]

6 Lecture#6 (04/02) [Discrete Logarithm Problem, DSA]

7 Lecture#7 (04/02) [Elliptic Curves, ECDSA]

8 Lecture#8 (11/02) [Bitcoin Address Generation]

9 Lecture#9 (15/02) [Bitcoin Transaction and its Validation]

10 Lecture#10 (18/02) [Bitcoin Consensus]

11 Lecture#11 (22/02) [Block Mining, txn Serialization]

12 Lecture#12 (25/02) [txn Serialization, PaytoScriptHash]

13 Lecture#13 (01/03) [More lockScripts, txn-malleability]

14 Lecture#14 (04/03) [SegWit transaction]

15 Lecture#15 (15/03) [Analysis of Bitcoin's Consensus]

16 Lecture#16 (21/03) [Hyperledger Fabric]

17 Lecture#17 (25/04) [Ethereum]

18 Lecture#18 (29/03) [Ethereum Contract Accounts]

19 Lecture#19 (05/04) [Payment Channels, Coin Mixers]

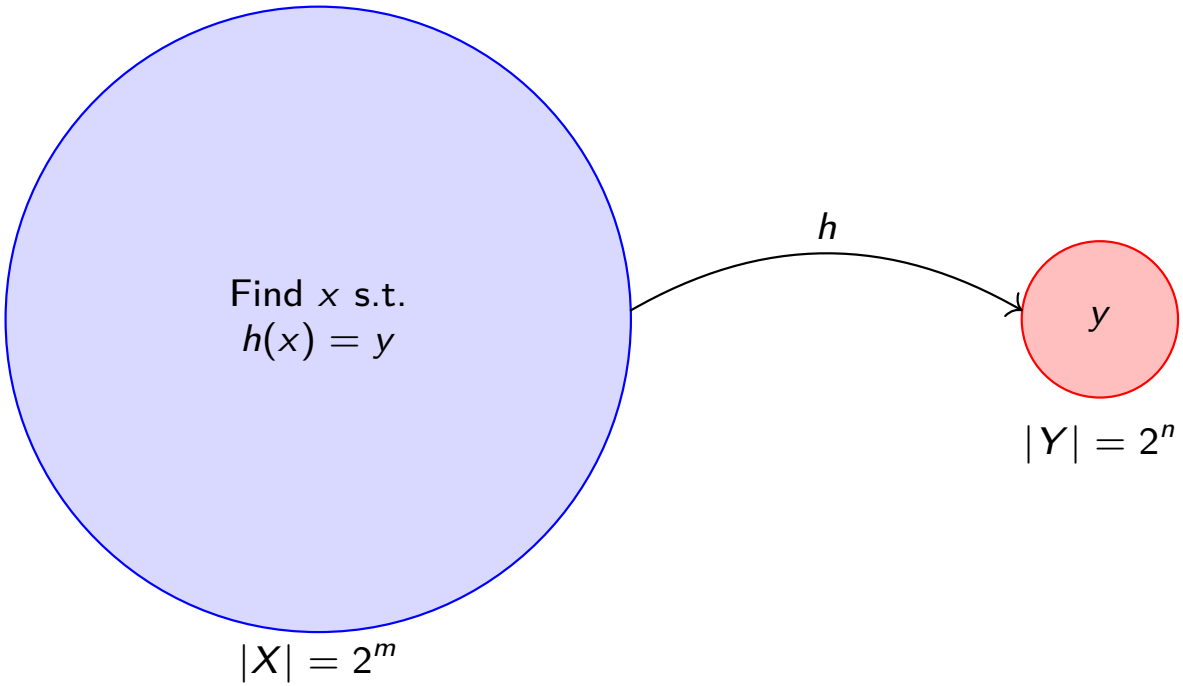
20 Lecture#20 (08/04) [Coin Mixers, MixEth, TumbleBit]

21 Lecture#21 (19/04) [Tokens]

ashoka UNIVERSITY

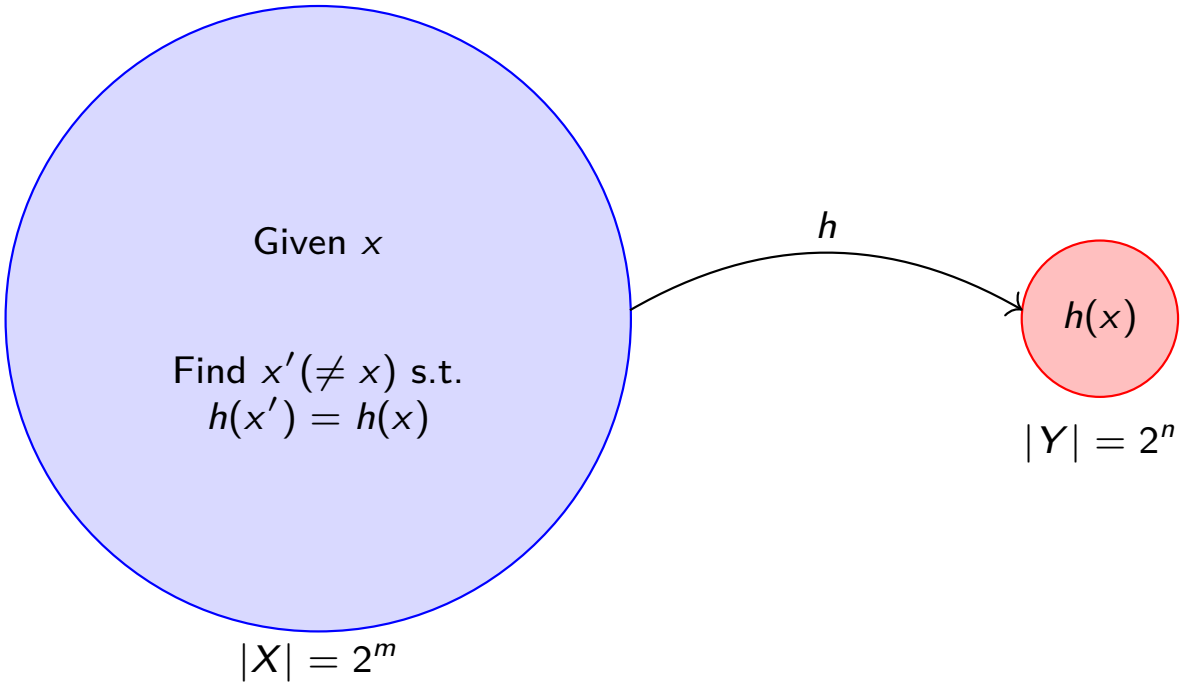
Hash Functions

- A Hash Function is a function $h : \{0,1\}^m \rightarrow \{0,1\}^n$ such that $m >> n$ (just functions that take arbitrary-length strings and compress them into shorter strings)
- The classic use of hash func tions is in data structures, where they can be used to achieve $O(1)$ insertion and lookup times for storing a set of elements.
- Security When considering cryptographic hash functions, there are typically three levels of security considered!
- Pre-image Resistant
- **Input:** $h : X \rightarrow Y$, and $y \in Y$
- **Output:** It is infeasible for a probabilistic polynomial-time adversary to output an $x \in X$ such that $h(x) = y$



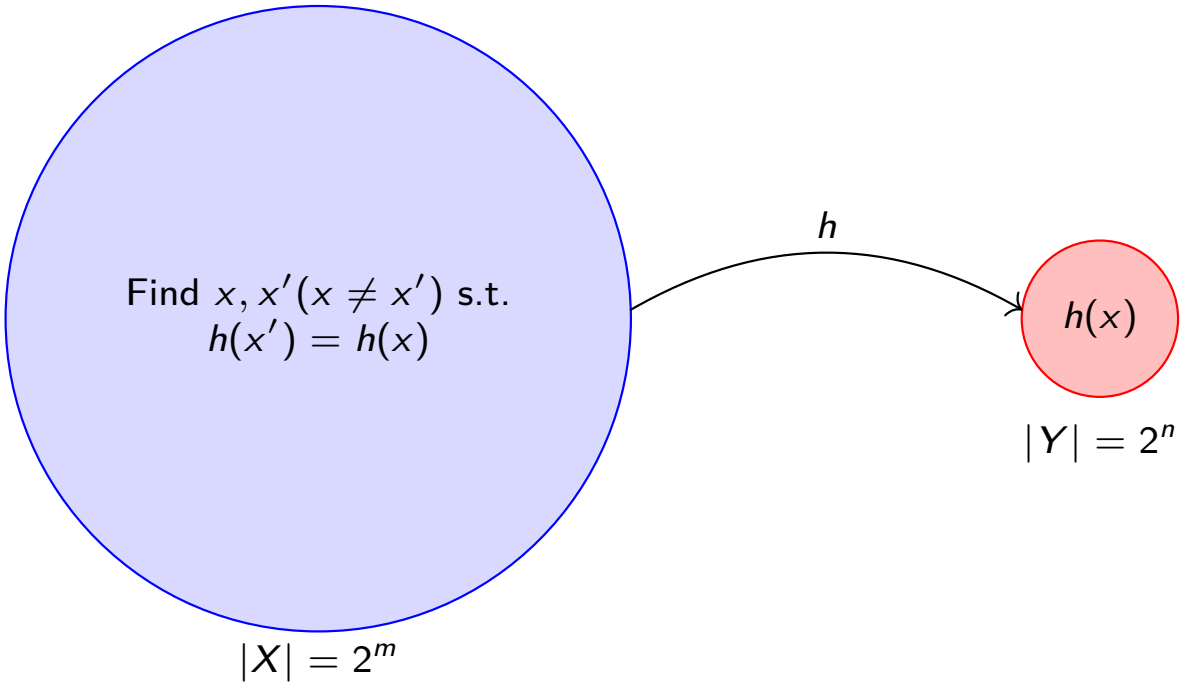
Hash Functions

- Second Pre-image Resistant
  - **Input:** $h : X \rightarrow Y$, and $x \in X$
  - **Output:** It is infeasible for a probabilistic polynomial-time adversary to output an $x' \in X$ such that $x' \neq x$ and $h(x') = h(x)$



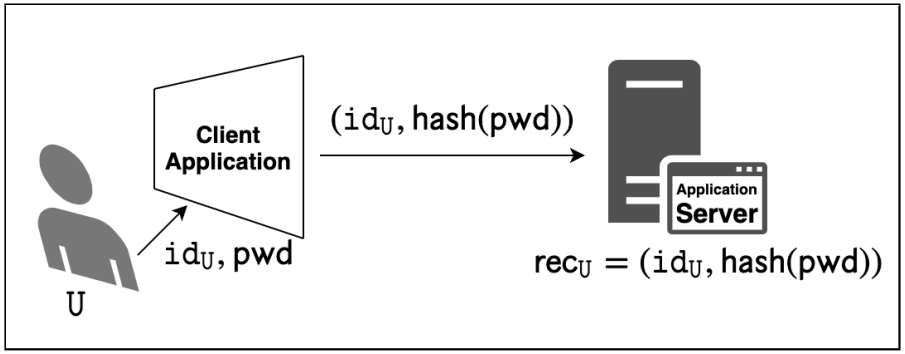
Hash Functions

- Collision Resistant
  - **Input:** $h : X \to Y$
  - **Output:** It is infeasible for a probabilistic polynomial-time adversary to output $x, x' \in X$ such that $x' \neq x$ and $h(x') = h(x)$



Hash Functions

- Password-based Authentication



- **Registration Phases**
  - A user U creates its user credential record — a user identity $\mathtt{id}_U$, and a hash of a secret password pwd, i.e. hash(pwd), with the application server
- **Login**
  - U provides $(\mathtt{id}_U, \mathtt{pwd})$ to its client which then computes hash(pwd), and sends $(\mathtt{id}_U, \text{hash}(\mathtt{pwd}))$ to the application server.
  - The server compares this to the stored record for the stated $\mathtt{id}_U$ and gives access if it matches.

Hash Functions

- FIPS 180-4: SHA-1, SHA-2 family
- FIPS 202: SHA-3 family (KECCAK)

- An example hash function:
  $\text{SHA256} : \{0,1\}^* \rightarrow \{0,1\}^{256}$
  - **Input:** 
  - **Output:** $\text{SHA256}(logo) =$
    $d9d535a093b861e9858168f703d0a20b8$
    $a66763c76fd07a86c6a7dfc61b4096c$

Function	Output Size
SHA-1	160
SHA-224	224
SHA-512/224	224
SHA-256	256
SHA-512/256	256
SHA-384	384
SHA-512	512
SHA3-224	224
SHA3-256	256
SHA3-384	384
SHA3-512	512

Table of Contents

1 Lecture#1 (18/01) [Course Overview, Bitcoin Introduction]

2 Lecture#2 (21/01) [Hash Functions]

3 Lecture#3 (25/01) [Random Oracle Model, Iterated Hash Construction]

4 Lecture#4 (28/01) [Hash Puzzle, Signature Scheme]

5 Lecture#5 (01/02) [Group Theory]

6 Lecture#6 (04/02) [Discrete Logarithm Problem, DSA]

7 Lecture#7 (04/02) [Elliptic Curves, ECDSA]

8 Lecture#8 (11/02) [Bitcoin Address Generation]

9 Lecture#9 (15/02) [Bitcoin Transaction and its Validation]

10 Lecture#10 (18/02) [Bitcoin Consensus]

11 Lecture#11 (22/02) [Block Mining, txn Serialization]

12 Lecture#12 (25/02) [txn Serialization, PaytoScriptHash]

13 Lecture#13 (01/03) [More lockScripts, txn-malleability]

14 Lecture#14 (04/03) [SegWit transaction]

15 Lecture#15 (15/03) [Analysis of Bitcoin's Consensus]

16 Lecture#16 (21/03) [Hyperledger Fabric]

17 Lecture#17 (25/04) [Ethereum]

18 Lecture#18 (29/03) [Ethereum Contract Accounts]

19 Lecture#19 (05/04) [Payment Channels, Coin Mixers]

20 Lecture#20 (08/04) [Coin Mixers, MixEth, TumbleBit]

21 Lecture#21 (19/04) [Tokens]

ashoka UNIVERSITY

The Random Oracle Model

Defining RO : $\{0,1\}^* \rightarrow \{0,1\}^m$
Setup $L = \{\}$ **//An Empty Dictionary
Input: $x \in \{0,1\}^*$
Process
If the dictionary entry $L["x"]$ is not present
$r \xleftarrow{\$} \{0,1\}^m$
$\text{RO}(x) \leftarrow r$
$L["x"] \leftarrow r$ **//Updating L
Else
$\text{RO}(x) \leftarrow L["x"]$
Output: $\text{RO}(x)$

- The random oracle function $\text{RO} : \{0,1\}^* \rightarrow \{0,1\}^m$ is an ideal hash function !!
- **Fact** For any $x \in \{0,1\}^*$ and $y \in \{0,1\}^m$

$$\mathbb{P}[\text{RO}(x) = y] = \frac{1}{2^m}$$

- An algorithm that attempts to solve Pre-image resistance problem for RO by evaluating RO at k points.

Input: RO, $y \in \{0,1\}^m$, and $k \in \mathbb{N}$
Process
For $i = 1$ to k
$x \leftarrow \{0,1\}^*$
If $\text{RO}(x) == y$
break; return x
return Failure

- Fact The probability that the above algorithm returns a pre-image is $\approx 1 - (1 - \frac{1}{2^m})^k$
- If $k << 2^m$, then $1 - (1 - \frac{1}{2^m})^k \approx \frac{k}{2^m}$

- A well designed hash function should "behave" like a random oracle !!

Iterated Hash Function Construction

- Compression Function A hash function with finite domain.
- **Iterated Hash Function Construction:**

  Input: A collision resistant compression function $f : \{0,1\}^{m+t} \rightarrow \{0,1\}^m$
  Output: A collision resistant hash function $h_f : \cup_{i=m+t+1}^{\infty} \{0,1\}^i \rightarrow \{0,1\}^\ell$
  Defining h_f

  Input $x \in \cup_{i=m+t+1}^{\infty} \{0,1\}^i$
  Process
  $y \leftarrow \text{PublicAlgorithm}(x)$ such that $|y| \equiv 0 \pmod{t}$
  Parse y as $y = y_1 \| \ldots \| y_r$ with $|y_i| = t$, $1 \leq i \leq r$
  $\text{IV} \leftarrow \{0,1\}^m$ (IV - a public initial value)

  $$\begin{cases} z_0 \leftarrow \text{IV} \\ z_1 \leftarrow f(z_0 \| y_1) \\ z_2 \leftarrow f(z_1 \| y_2) \\ \vdots \\ z_r \leftarrow f(z_{r-1} \| y_r) \end{cases}$$

  Output $h_f(x) = g(z_r)$ where $g : \{0,1\}^m \rightarrow \{0,1\}^\ell$ is a public function.

The Merkle-Damgard Construction

- Compression Function A hash function with finite domain.
- **Iterated Hash Function Construction:**

  Input: A collision resistant compression function $f : \{0,1\}^{m+t} \rightarrow \{0,1\}^m$, $t \geq 2$
  Output: A collision resistant hash function $h_f : \cup_{i=m+t+1}^{\infty} \{0,1\}^i \rightarrow \{0,1\}^\ell$
  Defining h_f

  Input $x \in \cup_{i=m+t+1}^{\infty} \{0,1\}^i$
  Process
  $n \leftarrow |x|$
  $n = q(t-1) + r$
  $k \leftarrow \lceil \frac{n}{t-1} \rceil = \lceil q + \frac{r}{t-1} \rceil = q + 1$
  Parse x as $x_1 \| \ldots \| x_{k-1} \| x_k$ where $|x_i| = t - 1$ for $1 \leq i \leq k-1$ and $|x_k| = r$
  For $i = 1$ to $k - 1$
  $y_i \leftarrow x_i$
  $d \leftarrow (t-1) - r$; $y_k \leftarrow x_k \| 0^d$
  $y_{k+1} \leftarrow (t-1)$–bit binary representation of d
  $z_1 \leftarrow 0^{m+1} \| y_1$
  $g_1 \leftarrow f(z_1)$
  For $i = 1$ to k
  $z_{i+1} \leftarrow g_i \| 1 \| y_{i+1}$
  $g_{i+1} \leftarrow f(z_{i+1})$
  Output $h_f(x) = g_{k+1}$

Table of Contents

1 Lecture#1 (18/01) [Course Overview, Bitcoin Introduction]

2 Lecture#2 (21/01) [Hash Functions]

3 Lecture#3 (25/01) [Random Oracle Model, Iterated Hash Construction]

4 Lecture#4 (28/01) [Hash Puzzle, Signature Scheme]

5 Lecture#5 (01/02) [Group Theory]

6 Lecture#6 (04/02) [Discrete Logarithm Problem, DSA]

7 Lecture#7 (04/02) [Elliptic Curves, ECDSA]

8 Lecture#8 (11/02) [Bitcoin Address Generation]

9 Lecture#9 (15/02) [Bitcoin Transaction and its Validation]

10 Lecture#10 (18/02) [Bitcoin Consensus]

11 Lecture#11 (22/02) [Block Mining, txn Serialization]

12 Lecture#12 (25/02) [txn Serialization, PaytoScriptHash]

13 Lecture#13 (01/03) [More lockScripts, txn-malleability]

14 Lecture#14 (04/03) [SegWit transaction]

15 Lecture#15 (15/03) [Analysis of Bitcoin's Consensus]

16 Lecture#16 (21/03) [Hyperledger Fabric]

17 Lecture#17 (25/04) [Ethereum]

18 Lecture#18 (29/03) [Ethereum Contract Accounts]

19 Lecture#19 (05/04) [Payment Channels, Coin Mixers]

20 Lecture#20 (08/04) [Coin Mixers, MixEth, TumbleBit]

21 Lecture#21 (19/04) [Tokens]

ashoka UNIVERSITY

A Hash Puzzle

- The Problem P_T

  Input A hash function $\mathsf{H} : \{0,1\}^* \leftarrow \{0,1\}^{256}$; $\mathsf{T} \in \{0,1,\ldots,2^{256}-1\}$
  Output: Find $x \in \{0,1\}^*$ such that $\mathsf{H}(x) < T$

- Note that, every instance P_T of this problem comprises of T-many pre-image finding problems with respect to the underlying hash function H

- Recall: the pre-image finding problem, PF_y

$$\begin{cases} \textbf{Input: } H : \{0,1\}^* \rightarrow \{0,1\}^{256} \text{ and } y \in \{0,1\}^{256} \\ \textbf{Output: } x \in \{0,1\}^* \text{ such that } H(x) = y \end{cases}$$

- Therefore, for a $T \in \{0,1,\ldots,2^{256}-1\} = \{0,1\}^{256}$

$$\mathsf{P}_\mathsf{T} \equiv \{\mathsf{PF}_y \mid 0 \leq y \leq T\}$$

- If H is a secure hash function (behaves like a random oracle model), then the best algorithm to solve H_T would be

```
While H(x) ∉ {0, 1, ..., T − 1}
    x ←$ {0,1}*
    Compute H(x)
Output x
```

- The expected number of loops required before the above algorithm terminates with a pre-image

$$\approx \frac{2^{256}}{T}$$

- If $\lceil \log_2(T) \rceil = k$, i.e., $T \approx 2^k$, then the binary representation of any number which is $< T$ must have at lest $256 - k$ many leading 0's
- That is, for the correct solution x, the binary representation of $H(x)$ must have $256 - k$ many leading 0's

---

- *Miners* in Bitcoin currently solves a P_T, roughly every 10 seconds, where $T \approx 2^{180}$

  Therefore, the expected number of hash computations required $\approx \frac{2^{256}}{2^{180}} \approx 2^{76}$!!

Hash Libraries; A Simple Code for P_T

```
>>>
>>> from Crypto.Hash import SHA256
>>> h = SHA256.new()
>>> h.update(b"cs2361/Blockchain")
>>> h.hexdigest()
'544078da40a51fced780e3766183c099060acb138a70c73fb94df103c3d1f43d'
>>>
>>> import hashlib
>>> H = hashlib.sha256()
>>> H.update(b"cs2361/Blockchain")
>>> H.hexdigest()
'544078da40a51fced780e3766183c099060acb138a70c73fb94df103c3d1f43d'
>>>
```

```
import hashlib
k = int(input("Enter k for T=2**k: "))
T = 2**k
loop = 2**(256-k)
for i in range(loop):
    h = hashlib.sha256()
    h.update(i.to_bytes(2,'big'))
    a = h.hexdigest()
    print(a)
    if (int(str(a),16) < T):
        print("..................")
        print("....a solution is found and the number
        of iterations it took:", i)
        print("Number of leadig 0's in H(x):", 256-k)
        print(a)
        print("{0:0256b}".format(int(str(a), 16)))
        break
```

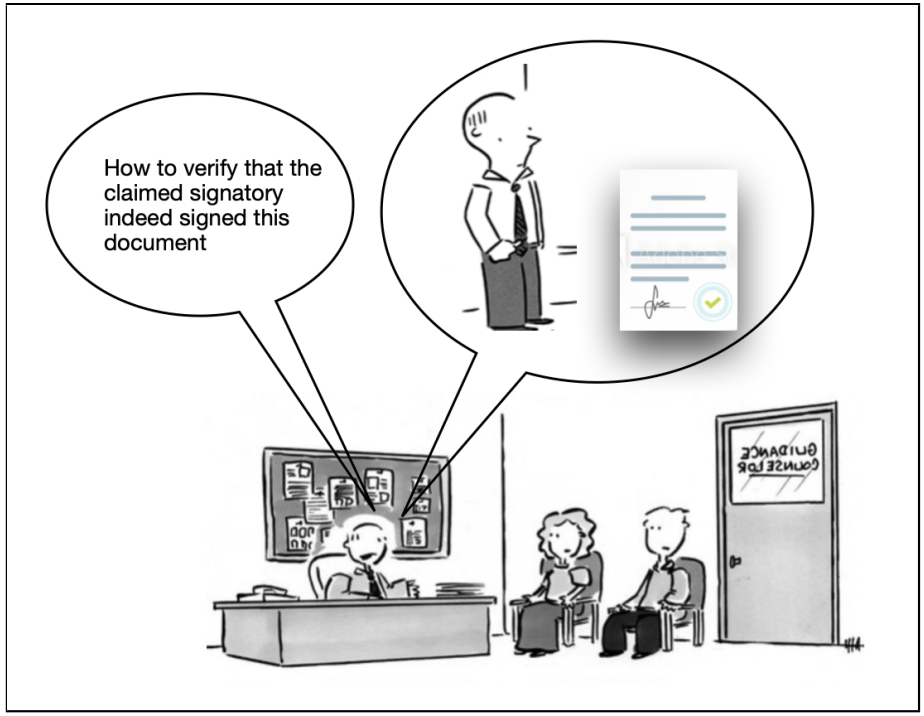
```
(base) Mahavirs-MacBook-Pro:Programs mahavirjhawar$ python3 hash.py
Enter k for T=2**k: 250
96a296d224f285c67bee93c30f8a309157f0daa35dc5b87e410b78630a09cfc7
b413f47d13ee2fe6c845b2ee141af81de858df4ec549a58b7970bb96645bc8d2
fcf0a6c700dd13e274b6fba8deea8dd9b26e4eedde3495717cac8408c9c5177f
583c7dfb7b3055d99465544032a571e10a134b1b6f769422bbb71fd7fa167a5d
4f35212d12f9ad2036492c95f1fe79baf4ec7bd9bef3dffa7579f2293ff546a4
9f1afa4dc124cba73134e82ff50f17c8f7164257c79fed9a13f5943a6acb8e3d
40d88127d4d31a3891f41598eeed41174e5bc89b1eb9bbd66a8cbfc09956a3fd
2ecd8a6b7d2845546659ad4cf443533cf921b19dc81fa83934e83821b4dfdcb7
b4c43b50bf245bd727623e3c775a8fcfb8d823d00b57dd65f7f79dd33f126315
c87479cd656e7e3ad6bd8db402e8027df454b2b0c42ff29e093458beb98a23d4
67ebbd370daa02ba9aadd05d8e091e862d0d8bcadafdf2a22360240a42fe922e
f09a7a12954169ae595d12d870e69a4c0092003157d72523d626d2a3990241e2
fe251e4dd034dcf589c84794120c85d6015d65ca7d9a2c2ec73c9ecb5e33d83b
a1f386a0ecb061b3c46a038616212779858ba7258b2eccb818a64986c97282da
9f4917386c45e2c0da0d9b475f1a19cf2db1e929195c6a9f4966ca0d2105b196
3b2b7c6ee25e2f28a6235e273eaf13f504bd445024147ebacb878262aae90509
0298d122906dcfc10892cb53a73992fc5b9f493ea4c9badb27b791b4127a7fe7
..................
....a solution is found and the number of iterations it took: 16
Number of leadig 0's in H(x): 6
0298d122906dcfc10892cb53a73992fc5b9f493ea4c9badb27b791b4127a7fe7
0000001010011000110100010010001010010000011011011100111111000001000010
1010100100110010011011101011011011001001111011011110010001101101000001
(base) Mahavirs-MacBook-Pro:Programs mahavirjhawar$
```

Handwritten Signatures

- A handwritten signature can be used to provide assurance that the claimed signatory signed a document.

- Suppose I am your faculty advisor and you require my approval to be able to audit a particular course. Unfortunately, I didn't approve your recent request.
- You thought of working around this by filling in the audit form and putting a random signature representing my name.
- What are the odds that this will go unnoticed at the Office of Academic Affairs?

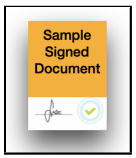
- For this to get detected, what measures must be taken? Who should be taking them?



Handwritten Signatures

- **Step** (1) Signature Style Generation
- I pick my signature style

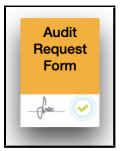
- Prepare a sample signed document



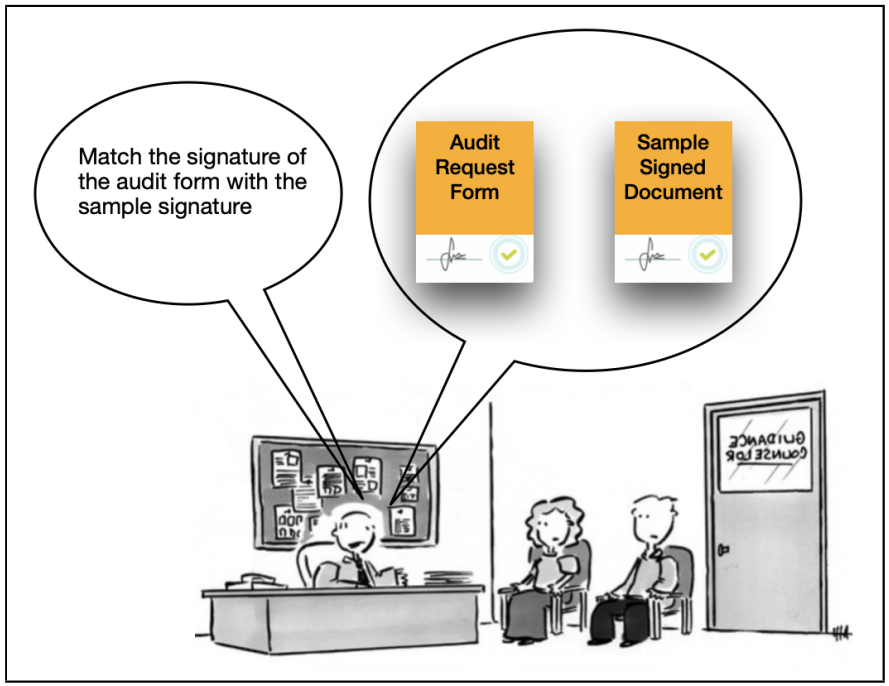
- Submit the sample signed document with the Office of Academic Affairs

---

- **Step** (2) Signing New Documents
- Given a audit request form, prepare a signed document



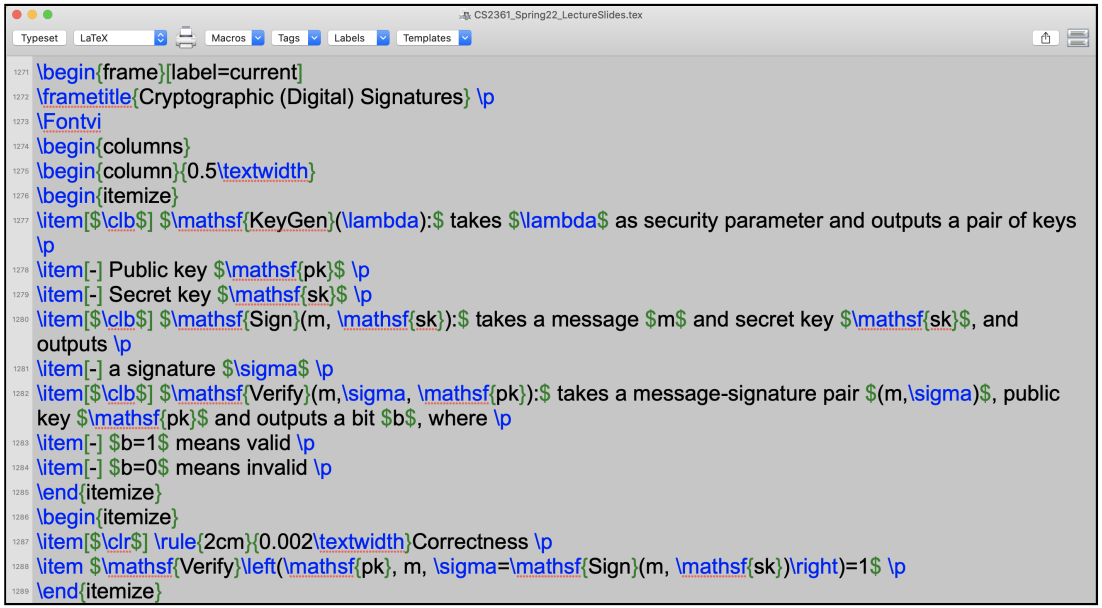
- **Step** (3) Signature Verification



Cryptographic (Digital) Signatures

- A digital signature is an electronic analogue of a hand-written signature

- **Digital message**




```
\begin{frame}[label=current]
\frametitle{Cryptographic (Digital) Signatures} \p
\Fontvi
\begin{columns}
\begin{column}{0.5\textwidth}
\begin{itemize}
\item[$\clb$] $\mathsf{KeyGen}(\lambda):$ takes $\lambda$ as security parameter and outputs a pair of keys \p
\item[-] Public key $\mathsf{pk}$ \p
\item[-] Secret key $\mathsf{sk}$ \p
\item[$\clb$] $\mathsf{Sign}(m, \mathsf{sk}):$ takes a message $m$ and secret key $\mathsf{sk}$, and outputs \p
\item[-] a signature $\sigma$ \p
\item[$\clb$] $\mathsf{Verify}(m,\sigma, \mathsf{pk}):$ takes a message-signature pair $(m,\sigma)$, public key $\mathsf{pk}$ and outputs a bit $b$, where \p
\item[-] $b=1$ means valid \p
\item[-] $b=0$ means invalid \p
\end{itemize}
\begin{itemize}
\item[$\clr$] \rule{2cm}{0.002\textwidth}Correctness \p
\item $\mathsf{Verify}\left(\mathsf{pk}, m, \sigma=\mathsf{Sign}(m, \mathsf{sk})\right)=1$ \p
\end{itemize}
```

- **Signature:**

MC4CFQDwxKQepzM24Odh
59e76FRjGRUs2AIVAMAcxe8
Digsnkv8FOWY3Qc31T32H

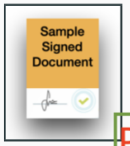
Cryptographic (Digital) Signatures

- **Step ①** Signature Style Generation
- I pick my signature style

Key Generation

Secret Key

- Prepare a sample signed document



Public Key

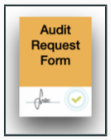
- Submit the sample signed document with the Office of Academic Affairs

---

- **Step ②** Signing New Documents

Sign

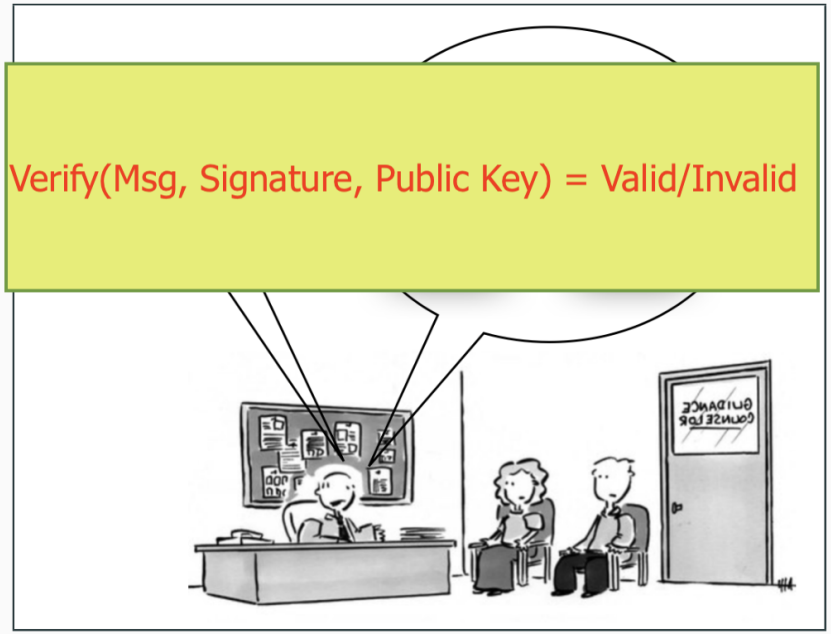
- Given a audit request form, prepared a signed document



Sign(Msg, Secret Key) = Signature

Verify

- **Step ③** Signature Verification



Cryptographic (Digital) Signatures

- KeyGen(λ) : takes λ as security parameter and outputs a pair of keys
  - Public key pk
  - Secret key sk
- Sign(m, sk) : takes a message m and secret key sk, and outputs
  - a signature σ
- Verify(m, σ, pk) : takes a message-signature pair (m, σ), public key pk and outputs a bit b, where
  - $b = 1$ means valid
  - $b = 0$ means invalid
- _____Correctness

  $\text{Verify}\,(\text{pk}, m, \sigma = \text{Sign}(m, \text{sk})) = 1$

- Security: **Existential Forgery**
  - An adversary $\mathcal{A}$'s Inputs
    - Public key pk
    - Adversary is allowed oracle access to Sign, i.e.,
    - $\mathcal{A}$ gets $\sigma_i \leftarrow \text{Sign}(m_i, \text{sk})$, $1 \leq i \leq k$, for m_i's of her choice
  - **Output:** (m^*, σ^*) such that
    - $\text{Verify}(m^*, \sigma^*, \text{pk}) = 1$, and
    - $m^* \neq m_i$, $1 \leq i \leq k$

- FIPS 186-4 Digital Signature Standard
  - DSA (Digital Signature Algorithm)
  - RSA Signature
  - ECDSA (Elliptic Curve DSA)

Table of Contents

1 Lecture#1 (18/01) [Course Overview, Bitcoin Introduction]

2 Lecture#2 (21/01) [Hash Functions]

3 Lecture#3 (25/01) [Random Oracle Model, Iterated Hash Construction]

4 Lecture#4 (28/01) [Hash Puzzle, Signature Scheme]

5 Lecture#5 (01/02) [Group Theory]

6 Lecture#6 (04/02) [Discrete Logarithm Problem, DSA]

7 Lecture#7 (04/02) [Elliptic Curves, ECDSA]

8 Lecture#8 (11/02) [Bitcoin Address Generation]

9 Lecture#9 (15/02) [Bitcoin Transaction and its Validation]

10 Lecture#10 (18/02) [Bitcoin Consensus]

11 Lecture#11 (22/02) [Block Mining, txn Serialization]

12 Lecture#12 (25/02) [txn Serialization, PaytoScriptHash]

13 Lecture#13 (01/03) [More lockScripts, txn-malleability]

14 Lecture#14 (04/03) [SegWit transaction]

15 Lecture#15 (15/03) [Analysis of Bitcoin's Consensus]

16 Lecture#16 (21/03) [Hyperledger Fabric]

17 Lecture#17 (25/04) [Ethereum]

18 Lecture#18 (29/03) [Ethereum Contract Accounts]

19 Lecture#19 (05/04) [Payment Channels, Coin Mixers]

20 Lecture#20 (08/04) [Coin Mixers, MixEth, TumbleBit]

21 Lecture#21 (19/04) [Tokens]

ashoka UNIVERSITY

Bitcoin Addresses

- To receive Bitcoins, one must have a Bitcoin address.
  - Bitcoin address $\equiv$ bank account numbers

- Users in banking system $\leftarrow$ (A/C No, Secret Pins)
- A/C No is used to receive money
- Secret Pins for establishing ownership of money
- Secret Pins for approving transactions on your money

- Bitcoin users get $(pk, sk) \leftarrow \Pi.\text{KeyGen}$ where Π is a signature scheme
- (A/C No, Secret Pins) $\equiv$ (Encode(pk), sk)
- Bitcoin address = Encode(pk)
- Bitcoin address - to receive money
- sk - to establish ownership of money
- sk - approve transactions on your money

Basics of Group Theory

- Groups A group is a tuple $(G, \circ)$, where G is a nonempty set equipped with a binary operation $\circ : G \times G \to G$ satisfying the following properties
  - **Associativity:** $a \circ (b \circ c) = (a \circ b) \circ c$ for all $a, b, c \in G$
  - **Identity:** there is an element, say $e \in G$, such that $a \circ e = e \circ a = a$ for all $a \in G$.
  - **Inverse:** For each element $a \in G$, there exists an element $a' \in G$ such that $a \circ a' = a' \circ a = e$. a' is called the inverse of a.

- Let $G = \{0, 1, 2, 3\}$ and $\circ = +$.
- Then "$\circ$" is not a binary operation on G

$\circ$	0	1	2	3
0	0	1	2	3
1	1	2	3	4
2	2	3	4	5
3	3	4	5	6

- $G = \{0, 1, 2, 3\}$ and $\circ = +$ (mod 4).
- Then $(G, \circ)$ forms a group.

$\circ$	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

- $e = 0$ and $0' = 0$
- $1' = 3$, $2' = 2$, and $3' = 1$

- $G = \{1, 2, 3, 4\}$ and $\circ = \times$ (mod 5).
- Then $(G, \circ)$ forms a group.

$\circ$	1	2	3	4
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

- $e = 1$ and $1' = 1$
- $2' = 3$, $3' = 2$, $4' = 4$

Basics of Group Theory

- $\boxed{\mathbb{Z}_n^+}$ For a positive integer n, by $\mathbb{Z}_n^+$ we denote the group $(G = \{0, 1, \ldots, n-1\}, \circ = + \pmod{n})$.
  - $e = 0$ in $\mathbb{Z}_n^+$; for any $a \in \mathbb{Z}_n^+$, $a^{-1} = n - a$ (We denote the inverse a' of a as a^{-1}).
- $\boxed{\mathbb{Z}_n^*}$ For a positive integer n, by $\mathbb{Z}_n^*$ we denote the group $(G = \{a \in \mathbb{Z}_n^+ \mid \gcd(a, n) = 1\}, \circ = \times \pmod{n})$.
  - $e = 1$ in $\mathbb{Z}_n^*$; for any $a \in \mathbb{Z}_n^*$ we have $\gcd(a, n) = 1$. This means, there exists s, r such that $as + rn = 1$ (Extended Euclidean theorem). Clearly, $as \bmod n = (1 - rn) \bmod n = 1$. Therefore, $a^{-1} = s$.
- Consider $\mathbb{Z}_{10}^*$

$\circ$	1	3	7	9
1	1	3	7	9
3	3	9	1	7
7	7	1	9	3
9	9	7	3	1

- $\boxed{\text{Group Order}}$ The order of a group $(G, \circ)$ is defined to be the number of elements in G.
  - Finite groups are groups with finite order
  - Order of $\mathbb{Z}_n^+ = n$
  - Order of $\mathbb{Z}_n^* = \phi(n)$, where ϕ denotes the Euler phi function.

- $\boxed{\text{Subgroups}}$ For a group $(G, \circ)$, a subgroup is a group $(H, \circ)$ such that $H \subseteq G$.
  - Let $G = \mathbb{Z}_7^* = \{1, 2, 3, 4, 5, 6\}$
  - Then $H = \{1, 2, 4\}$ is a subgroup of $\mathbb{Z}_7^*$

$\circ$	3	1	2	4	5	6
3	2	3	6	5	1	4
1	3	1	2	4	5	6
2	6	2	4	1	3	5
4	5	4	1	2	6	3
5	1	5	3	6	4	2
6	4	6	5	3	2	1

- $\boxed{\text{Group Exponents}}$ Let $(G, \circ)$ be a group and $g \in G$ be any group element. Then for any $k \in \{0, 1, 2, \ldots\}$ we define

$$g^k = \begin{cases} e & \text{if } k = 0 \\ \underbrace{g \circ \cdots \circ g}_{k \text{ times}} & \text{if } k \geq 1 \end{cases}$$

  - Take $G = \mathbb{Z}_7^*$, $g = 4$, and $k = 5$.
  - $g^k = 4^5 = 4 \times 4 \times 4 \times 4 \times 4 \pmod{7} = 2$

Basics of Group Theory

- For negative exponent $-k < 0$

$$g^{-k} = \underbrace{g^{-1} \circ \ldots \circ g^{-1}}_{k \text{ times}}$$

- $5^{-4} = 5^{-1} \times 5^{-1} \times 5^{-1} \times 5^{-1} \pmod 7$
  $= 3 \times 3 \times 3 \times 3 \pmod 7 = 4$

- Order of Group Elements For this definition we consider finite groups.
- Claim: Let $(G, \circ)$ be a finite group such that $|G| = k$ (order of G). Then, for every element g ($\neq e$) in G, there exists a r in $1 < r \leq k$ such that $g^r = e$.
- Consider the following list of $k+1$ elements: $g, g^2, g^3, \ldots, g^k, g^{k+1}$.
- As $g \in G$, the elements from $g^2, \ldots, g^{k+1}$ are all in G.
- But G has k elements. Therefore, at least 2 elements in the above list are equal.
- Say $g^i = g^j$, for some $1 \leq i < j \leq k+1$.
- This means, $g^{j-i} = e$, where $1 < j - i \leq k$.

- For $g \in G$, the order of g is the smallest positive integer r such that $g^r = e$. It is denoted as $o(g) = r$.
  - Let $G = \mathbb{Z}_7^* = \{1, 2, 3, 4, 5, 6\}$

g	g^2	g^3	g^4	g^5	g^6
1	1	1	1	1	1
2	4	1	2	4	1
3	2	6	4	5	1
4	2	1	4	2	1
5	4	6	2	3	1
6	1	6	1	6	1

  - $o(2) = 3$; $o(3) = 6$; $o(4) = 3$; $o(5) = 6$; $o(6) = 2$
- Cyclic Groups A group G is called cyclic if there exists a $g \in G$ such that $o(g) = |G|$.
  - The g is called a generator of G.
  - If $|G| = k$ then $G = \{g, g^2, \ldots, g^{k-1}, g^k = e\}$
  - $\mathbb{Z}_7^*$ is cyclic.
- Theorem $\mathbb{Z}_p^*$ is cycle for every prime p

Table of Contents

1. Lecture#1 (18/01) [Course Overview, Bitcoin Introduction]
2. Lecture#2 (21/01) [Hash Functions]
3. Lecture#3 (25/01) [Random Oracle Model, Iterated Hash Construction]
4. Lecture#4 (28/01) [Hash Puzzle, Signature Scheme]
5. Lecture#5 (01/02) [Group Theory]
6. Lecture#6 (04/02) [Discrete Logarithm Problem, DSA]
7. Lecture#7 (04/02) [Elliptic Curves, ECDSA]
8. Lecture#8 (11/02) [Bitcoin Address Generation]
9. Lecture#9 (15/02) [Bitcoin Transaction and its Validation]
10. Lecture#10 (18/02) [Bitcoin Consensus]
11. Lecture#11 (22/02) [Block Mining, txn Serialization]
12. Lecture#12 (25/02) [txn Serialization, PaytoScriptHash]
13. Lecture#13 (01/03) [More lockScripts, txn-malleability]
14. Lecture#14 (04/03) [SegWit transaction]
15. Lecture#15 (15/03) [Analysis of Bitcoin's Consensus]
16. Lecture#16 (21/03) [Hyperledger Fabric]
17. Lecture#17 (25/04) [Ethereum]
18. Lecture#18 (29/03) [Ethereum Contract Accounts]
19. Lecture#19 (05/04) [Payment Channels, Coin Mixers]
20. Lecture#20 (08/04) [Coin Mixers, MixEth, TumbleBit]
21. Lecture#21 (19/04) [Tokens]

ashoka UNIVERSITY

Discrete-logarithm Problem

- Discrete-logarithm Problem (DLP)
  - **Input:** (G, g, h) where G is a cyclic-group; g is a generator of G; and $h \in G$ is a random group element.
  - **Output:** Find $\alpha \in \{1, 2, \ldots, |G|\}$ such that $g^{\alpha} = h$.
  - α is called $\text{dlog}_g h$.

- We are interested in groups G where this problem is hard to solve
  - It is necessary that we take up groups with high order size.
  - DLP is not hard on all groups: e.g., easy in $(\mathbb{Z}_n, +)$!! (even if n is very big)
  - DLP is hard on $\mathbb{Z}_p^*$, where p is a large prime

---

- Another group where DLP is hard
  - Let p, q be two primes such that $q|p-1$
  - Choose $g \in \mathbb{Z}_p^*$ such that $o(g) = q$
  - Set $G = \langle g \rangle$
    - **Input:** (G, g, h) where $h \in G$
    - **Output:** Compute $\alpha \in \{1, \ldots, q\}$ such that $g^{\alpha} = h \pmod{p}$
- It is also presented as follows:
  - **Input:** $((G, q, p), g, h = g^{\alpha})$
  - **Output:** α

- Computational Diffie-Hellman (CDH) Problem
  - **Input:** $((G, q, p), g, g^{\alpha}, g^{\beta})$
  - **Output:** $g^{\alpha\beta}$

- Decision Diffie-Hellman (DDH) Problem
  - **Input:** $((G, q, p), g, g^{\alpha}, g^{\beta}, h)$
  - **Output:** $\begin{cases} 1 & \text{if } h = g^{\alpha\beta} \\ 0 & \text{otherwise} \end{cases}$

Group Theory: Sage Commands

SageMath is a free open-source mathematics software https://www.sagemath.org/ https://sagecell.sagemath.org/ https://cocalc.com/

```
In [5]: # Setting up a group G=Z_p/Z_p^* for some prime p
        p = 7
        G = Zmod(p)
        # listing all elements of Z_p
        for i in range(p+2):
            print(G(i))

In [22]: # Setting up a group G=Z_p/Z_p^* for some prime p
         p = 7
         G = Zmod(p)
         # Multiplying gruop elements in Z_p^*
         g = G(4)
         h = G(5)
         print(g*h)

In [23]: # Setting up a group G=Z_p/Z_p^* for some prime p
         p = next_prime(6)
         G = Zmod(p)
         # Computing group exponents in Z_p^*
         g = G(3)
         print(g**2)

In [24]: # Setting up a group G=Z_p/Z_p^* for some prime p
         p = 31
         G = Zmod(p)
         # Finding inverse of gruop elements in Z_p^*
         g = G(5)
         print(g^-1)
```

Group Theory: Sage Commands

```
In [25]: # Setting up a group G=Z_p/Z_p^* for some prime p
         p = 31
         G = Zmod(p)
         # Picking a generator in Z_p^*
         g = G.multiplicative_generator()
         # Iterating powers of a group element in Z_p^*
         for i in range(p):
             print(G(3)**i)
```

```
In [26]: # Setting up a group G=Z_p/Z_p^* for some prime p
         p = next_prime(452452435)
         G = Zmod(p)
         # Computing Discrete Log in Z_p^*
         g = G.multiplicative_generator()
         h = G(3143413)
         print(discrete_log(h,g))
```

Generating Large Primes

- How to generate primes ?
  - Miller-Rabin Algorithm
    - **Input:** n, Is n prime ?
    - **Output:** Yes/No
- Running time: $O((\log_2 n)^3)$!!
- If n has 90-decimal digits
- Then, $n \approx 10^{90}$
- Running time to test such a number:
- $(\log_2 n)^3 = (\log_2(10^{90}))^3 = (\log_2(10^{3\times 30}))^3 \approx (\log_2(2^{10\times 30}))^3 = (300)^3 \approx 2^{25}$

- Computing Power
  - A 1GHz CPU can make 10^9 cycles/sec.
  - Take a 4GHz machine.
  - It can compute, $4 \cdot 10^9 = 4 \cdot (10^3)^3 \overset{2^{10}\approx 10^3}{=} 4 \cdot (2^{10})^3 = 2^{32}$ cycles/sec.

It takes less than a second to generate a 90-decimal digit prime !!

Solving DLP: Brute-force (Exhaustive) Search

- Consider a DLP instance (G, g, h) where $G = \mathbb{Z}_p^*$ and p is a prime having roughly 90 decimal digits.
  - The order of $G \approx 10^{90} \approx 2^{300}$.
  - $\text{dlog}_g h \in \{1, 2, \ldots, 2^{300}\}$
- How big is 2^{230} ??
  - A 4GHz machine can execute 2^{32} cycles/sec.
    $\implies \approx 2^{26}$ ops/sec (30-60 cycle/op)
    $\implies \approx 2^{42}$ ops /day ($60 \times 60 \times 24 \approx 2^{16}$)
    $\implies \approx 2^{51}$ ops/year ($365 < 2^9$)
- Thus, it will take $\approx 2^{250}$ years in the worst case!!

Digital Signature Algorithm (DSA)

- DSA Parameter Generation
  - p be a L-bit prime s.t.
  - DLP is difficult in $\mathbb{G} = \mathbb{Z}_p^*$,
  - $L \equiv 0 \bmod 64$ and $512 \leq L \leq 1024$
  - Let q be a 160-bit prime s.t. $q|p-1$
  - Let $g \in \mathbb{Z}_p^*$ such that $o(g) = q$.
  - Set Params $= \langle p, q, g \rangle$

- **KeyGen**
  - Pick $a \xleftarrow{\$} \mathbb{Z}_q$.
  - Compute $h = g^a \bmod p$
  - Set $\begin{cases} \text{pk} = (p, q, g, h); \text{hash} : \{0,1\}^* \to \mathbb{Z}_p^* \\ \text{sk} = a \end{cases}$

- **Sign**(msg = msg.txt, sk)
  1. Compute $z = \text{hash}(\text{msg.txt}) \in \mathbb{Z}_p^*$
  2. Pick $r \xleftarrow{\$} \mathbb{Z}_q^*$.
  3. Compute $c_1 = (g^r \bmod p) \bmod q$. (If $c_1 = 0$, goto (2))
  4. $c_2 = r^{-1}(z + ac_1) \bmod q$ (If $c_2 = 0$, goto (2))
  5. Output signature $\sigma = (c_1, c_2)$

- **Verify**(msg.txt, $\sigma = (c_1, c_2)$, pk)
  - Compute $z = \text{hash}(\text{msg.txt})$
  - Compute $e_1 = zc_2^{-1} \bmod q$
  - Compute $e_2 = c_1 c_2^{-1} \bmod q$
  - Output **valid** $\iff (g^{e_1} h^{e_2} \bmod p) \bmod q = c_1$

- **Correctness**

$$(g^{e_1} h^{e_2} \bmod p) \bmod q = (g^{zc_2^{-1}} (g^a)^{c_1 c_2^{-1}} \bmod p) \bmod q$$

$$= (g^{zc_2^{-1} + ac_1 c_2^{-1}} \bmod p) \bmod q = (g^{c_2^{-1}(z+ac_1)} \bmod p) \bmod q$$

$$= (g^r \bmod p) \bmod q \quad // * c_2 = r^{-1}(z + ac_1) \bmod q$$

$$= c_1$$

DSA run using Openssl

- **Key Generation**
  - Generate DSA parameters
    - `openssl dsaparam -out dsaparam.pem 1024`
  - Check the parameters in text
    - `openssal dsaparam -in dsaparam.pem -text`
  - Generate DSA key pair
    - `openssl gendsa -out skey_A.pem dsaparam.pem`
  - Check the key pair in text
    - `openssl dsa -in skey_A.pem -text`
  - Extracting the public key from the secret key of A
    - `openssl dsa -in skey_A.pem -out pubkey_A.pem -pubout`
  - Checking the public key in text
    - `openssl dsa -pubin -in pubkey_A.pem -text`

- **Sign**
  - Sign the file sha1(msg.txt) using the secret key of A
    - `openssl dgst -sha1 -sign skey_A.pem -out sign_msg_A.bin msg.txt`

- **Verify**
  - Verify the signature signed by A using the public key of A
    - `openssl dgst -sha1 -verify pubkey_A.pem -signature sign_msg_A.bin rcv_msg.txt`

Table of Contents

1 Lecture#1 (18/01) [Course Overview, Bitcoin Introduction]

2 Lecture#2 (21/01) [Hash Functions]

3 Lecture#3 (25/01) [Random Oracle Model, Iterated Hash Construction]

4 Lecture#4 (28/01) [Hash Puzzle, Signature Scheme]

5 Lecture#5 (01/02) [Group Theory]

6 Lecture#6 (04/02) [Discrete Logarithm Problem, DSA]

7 Lecture#7 (04/02) [Elliptic Curves, ECDSA]

8 Lecture#8 (11/02) [Bitcoin Address Generation]

9 Lecture#9 (15/02) [Bitcoin Transaction and its Validation]

10 Lecture#10 (18/02) [Bitcoin Consensus]

11 Lecture#11 (22/02) [Block Mining, txn Serialization]

12 Lecture#12 (25/02) [txn Serialization, PaytoScriptHash]

13 Lecture#13 (01/03) [More lockScripts, txn-malleability]

14 Lecture#14 (04/03) [SegWit transaction]

15 Lecture#15 (15/03) [Analysis of Bitcoin's Consensus]

16 Lecture#16 (21/03) [Hyperledger Fabric]

17 Lecture#17 (25/04) [Ethereum]

18 Lecture#18 (29/03) [Ethereum Contract Accounts]

19 Lecture#19 (05/04) [Payment Channels, Coin Mixers]

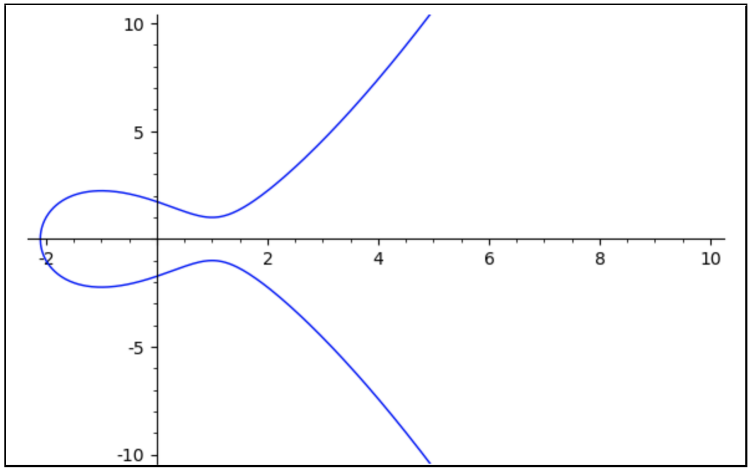
20 Lecture#20 (08/04) [Coin Mixers, MixEth, TumbleBit]

21 Lecture#21 (19/04) [Tokens]

ashoka UNIVERSITY

Elliptic Curve Cryptography (ECC)

- **Elliptic Curve:** An elliptic curve is the set of solutions to an equation of the form $E : y^2 = x^3 + ax + b$, where $a, b \in F$, a field.
- An Example: $E : y^2 = x^3 - 3x + 3$



  - An amazing feature of elliptic curves is that there is natural way to take two points on an elliptic curve and add (Elliptic curve addition) them to produce a third point on the curve.
- Define, $G_E = \{(x, y) \in F \times F \mid y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\}$
  - **Goal:** Show, G_E is a group with respect to this special elliptic curve addition

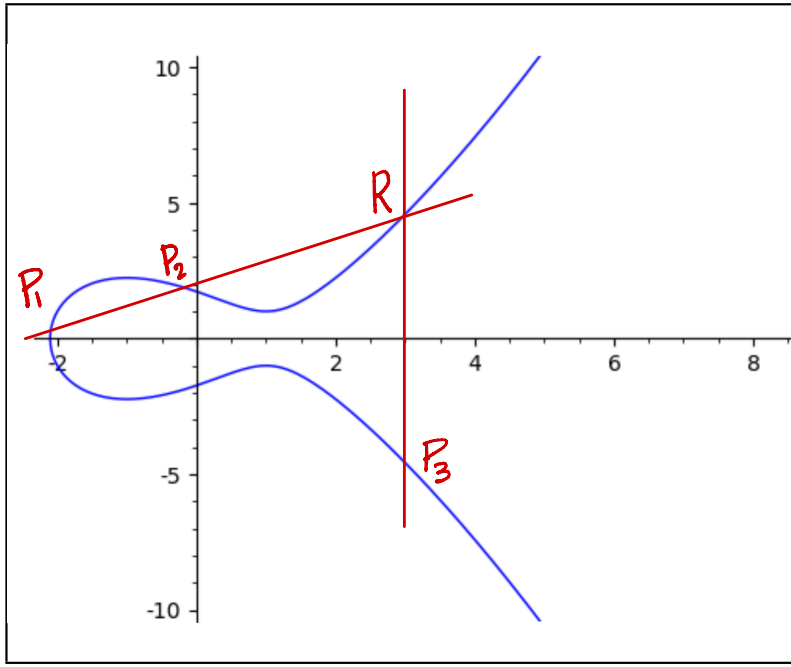
---

  - **Example:** For $E : y^2 = x^3 + x + 1$ over $\mathbb{Z}_3$, we have $G_E = \{(0, 1), (0, 2), (1, 0)\} \cup \{\mathcal{O}\}$
- The equation type ($y^2 = x^3 + ax + b$) is called *Weierstrass equations*.
- E is called non-singular if $4a^3 + 27b^2 \neq 0$ in F

Elliptic Curve Group

- Consider G_E, where E is a non-singular elliptic curve
- Define elliptic curve addition over G_E.

- Let $P_1 = (x_1, y_1), P_2 = (x_2, y_2) \in G_E$. Then
  - $P_1 + P_2 = \begin{cases} P_2 & \text{if } P_1 = \mathcal{O} \\ P_1 & \text{if } P_2 = \mathcal{O} \end{cases}$
- If $x_1 \neq x_2$. Then
  - $P_1 + P_2 = P_3 = (x_3, y_3)$, where $\begin{cases} x_3 = \lambda^2 - x_1 - x_2 \\ y_3 = \lambda(x_1 - x_3) - y_1 \\ \lambda = \frac{y_2 - y_1}{x_2 - x_1} \end{cases}$
- If $x_1 = x_2$ and $y_1 = y_2$.Then
  - $P_1 + P_2 = P_3 = (x_3, y_3)$, where
    $\begin{cases} x_3 = \lambda^2 - x_1 - x_2 \\ y_3 = \lambda(x_1 - x_3) - y_1 \\ \lambda = \frac{3x_1^2 + a}{2y_1} \end{cases}$
- If $x_1 = x_2$ and $y_1 = -y_2$. Then
  - $P_1 + P_2 = \mathcal{O}$



Nice demonstration available here:
`https://www.desmos.com/calculator/ialhd71we3`

Elliptic Curve Addition

- Draw a line L passing through the points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$.

  Equation of L : $y = \lambda(x - x_1) + y_1$ where $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$

  Any point on L has coordinates $(\alpha, (\alpha - x_1) + y_1)$

- Points of Intersection between L and E Let Q be a point of intersection between L and E.

- Q on $L \implies$ there exists an α such that $Q = (\alpha, (\alpha - x_1) + y_1)$

  But Q is also a point on $E : y^2 = x^3 + ax + b$

  Therefore

$$
\begin{aligned}
& (\lambda(\alpha - x_1) + y_1)^2 = \alpha^3 + a\alpha + b \\
\implies & \alpha^3 + a\alpha + b - (\lambda^2(\alpha - x_1)^2 + 2\lambda(\alpha - x_1)y_1 + y_1^2) = 0 \\
\implies & \alpha^3 + a\alpha + b - (\lambda^2(\alpha^2 - 2\alpha x_1 + x_1^2) + 2\lambda\alpha y_1 - 2x_1 y_1 \lambda + y_1^2) = 0 \\
\implies & \alpha^3 - \lambda^2\alpha^2 + (a + 2\lambda^2 x_1 - 2\lambda y_1)\alpha + (b - \lambda^2 x_1^2 + 2x_1 y_1 \lambda - y_1^2) = 0
\end{aligned}
$$

- Thus, if (α, β) is an intersecting point between L and E, then α is a root of the equation

$$X^3 - \lambda^2 X^2 + (a + 2\lambda^2 x_1 - 2\lambda y_1)X + (b - \lambda^2 x_1^2 + 2x_1 y_1 \lambda - y_1^2) = 0$$

- As $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ are intersecting points, x_1, x_2 are roots of the above equation.
- Let x_3 be the third root (the above equation has exactly three roots).
- Then $R = (x_3, (x_3 - x_1) + y_1)$ is the third point at which L intersect E.
- x_1, x_2, x_3 being the 3 roots of the above equation implies $x_1 + x_2 + x_3 = -\frac{\text{Coeff. of } X^2}{\text{Coeff. of } X} = -\frac{-\lambda^2}{1} = \lambda^2$
- Thus, $R = (x_3 = \lambda^2 - x_1 - x_2, \lambda(x_3 - x_1) + y_1)$
- This means, $P_3 = -R = (x_3, \lambda(x_1 - x_3) - y_1)$

ECDSA

- **secp256k1 curve parameters**
  - $p : 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$
  - $E : y^2 = x^3 + 7$, defined over $\mathbb{Z}_p$
  - $G = (x, y) \in \mathcal{G}_E$ is the base point, where $x =$

    79be667e f9dcbbac 55a06295 ce870b07
    029bfcdb 2dce28d9 59f2815b 16f81798

    and $y =$

    483ada77 26a3c465 5da4fbfc 0e1108a8
    fd17b448 a6855419 9c47d08f fb10d4b8
  - order$(G) = q$ is a prime, where $q =$

    FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
    BAAEDCE6 AF48A03B BFD25E8C D0364141
- **KeyGen**
  - Pick $a \xleftarrow{\$} \mathbb{Z}_q$. Compute $H = aG$
  - Set $\begin{cases} \text{pk} = (E, p, q, G, H) \\ \text{sk} = (a, \text{pk}) \end{cases}$
- **Sign**(msg.txt, sk)
1. Compute $h = \text{hash}(\text{msg.txt})$
2. Pick $r \xleftarrow{\$} \mathbb{Z}_q$. Compute rG.
3. Let $rG = (u, v)$, where $u, v \in \mathbb{Z}_p$
4. $c_1 = u \bmod q$ (If $c_1 = 0$, goto (2))
5. $c_2 = r^{-1}(h + ac_1) \bmod q$ (If $c_2 = 0$, goto (2))
6. Output signature $\sigma = (c_1, c_2)$

- **Verify**$(msg.txt, \sigma = (c_1, c_2), \text{pk})$
  - Compute $h = \text{hash}(\text{msg.txt})$
  - Compute $e_1 = hc_2^{-1} \bmod q$
  - Compute $e_2 = c_1 c_2^{-1} \bmod q$
  - Compute $J = e_1 G + e_2 H$
  - Let $J = (u, v)$
  - Output **valid** $\iff u \bmod q = c_1$

ECDSA run using Openssl

```
openssl ecparam -list_curves
```

- **Key Generation**

- Generate ECDSA secret-key
  - `openssl ecparam -name secp256k1 -genkey -out skey_A.pem`
- Check the secret-key in text
  - `openssl ec -in skey_A.pem -text`
- Extracting the public-key from the secret-key
  - `openssl ec -in skey_A.pem -pubout -out pubkey_A.pem`
- Checking the public-key in text
  - `openssl ec -pubin -in pubkey_A.pem -text`

- **Sign**

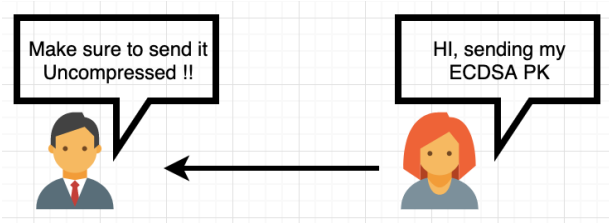
- Sign the file sha1(msg.txt) using the secret-key
  - `openssl dgst -sha1 -sign skey_A.pem -out sign_msg_A.bin msg.txt`

- **Verify**

- Verify the signature signed by A using the public-key of A
  - `openssl dgst -sha1 -verify pubkey_A.pem -signature sign_msg_A.bin msg.txt`

ECDSA pk Serialization

- Let $H = (z_1, z_2)$ be a ECDSA pk
- pk-**UnCompressed**:
- $\boxed{0x04 || \text{hexify}(z_1) || \text{hexify}(z_2)}$
- Example: 04B20F34AE . . . 42A3C4FE



- pk-**Compressed**
  - Key idea: Given z_1, it is easy to compute z_2.
  - $z_2 = \pm\sqrt{(z_1^3 + az_1 + b)} \pmod{p}$.
  - Therefore, pk $= H = (z_1, z_2)$ is completely recoverable just from the knowledge of z_1 and the sign of z_2 !
  - This is achieved as follows:
  - Note that, the square roots $\pm\sqrt{(z_1^3 + az_1 + b)}$ are of different parity- one is odd and the other is even.
  - If z_2 is a even number, then pk-Compressed $= \boxed{0x02 || \text{hexify}(z_1)}$
  - If z_2 is a odd number, then pk-Compressed $= \boxed{0x03 || \text{hexify}(z_1)}$

Table of Contents

1 Lecture#1 (18/01) [Course Overview, Bitcoin Introduction]

2 Lecture#2 (21/01) [Hash Functions]

3 Lecture#3 (25/01) [Random Oracle Model, Iterated Hash Construction]

4 Lecture#4 (28/01) [Hash Puzzle, Signature Scheme]

5 Lecture#5 (01/02) [Group Theory]

6 Lecture#6 (04/02) [Discrete Logarithm Problem, DSA]

7 Lecture#7 (04/02) [Elliptic Curves, ECDSA]

8 Lecture#8 (11/02) [Bitcoin Address Generation]

9 Lecture#9 (15/02) [Bitcoin Transaction and its Validation]

10 Lecture#10 (18/02) [Bitcoin Consensus]

11 Lecture#11 (22/02) [Block Mining, txn Serialization]

12 Lecture#12 (25/02) [txn Serialization, PaytoScriptHash]

13 Lecture#13 (01/03) [More lockScripts, txn-malleability]

14 Lecture#14 (04/03) [SegWit transaction]

15 Lecture#15 (15/03) [Analysis of Bitcoin's Consensus]

16 Lecture#16 (21/03) [Hyperledger Fabric]

17 Lecture#17 (25/04) [Ethereum]

18 Lecture#18 (29/03) [Ethereum Contract Accounts]

19 Lecture#19 (05/04) [Payment Channels, Coin Mixers]

20 Lecture#20 (08/04) [Coin Mixers, MixEth, TumbleBit]

21 Lecture#21 (19/04) [Tokens]

ashoka UNIVERSITY

Bitcoin Address Format

Bitcoin Address Generation

$(pk, sk) \leftarrow$ ECDSA.KeyGen
Bitcoin address $\leftarrow$ Encode(pk)

Bitcoin address needed to receive ฿
sk - to establish ownership of money

Bitcoin Encoding

Input: Public-key pk
Process:
$H1 \leftarrow$ SHA256(pk.uncompressed)
$H2 \leftarrow$ RIPEMD160($H1$)
$H3 \leftarrow A \| H_2$ where

$$A = \begin{cases} 0x00 & \text{if address is on main Bitcoin network} \\ 0x6f & \text{if address is on test network} \end{cases}$$

$H4 \leftarrow$ SHA256 (SHA256($H3$))
$H5 \leftarrow H3 \| H4[1:4] = A \| H2 \| H4[1:4]$
where $H4[1:4]$ denotes the first 4 bytes of $H4$.
Bitcoin Address $\leftarrow$ Base58Encoding($H5$)

Notations/Fact

- A Bitcoin address generated this way is called Pay to Public Key Hash (P2PKH) address
- Easy: RIPEMD160(SHA256(pk)) $\leftarrow$ P2PKH Address
- Difficult: $pk \leftarrow$ P2PKH Address
- HASH160(pk) $\stackrel{\text{def}}{=}$ RIPEMD160(SHA256(pk))

Base58Encoding

Input: $\boldsymbol{b} = \bar{b}_{n-1} \ldots \bar{b}_0$, $\bar{b}_i \in \{0,1\}^8$
Output: Base58 encoding of $\boldsymbol{b}$
Process:
Encode each leading zero byte (if any) as a 1

Let $\bar{b}_k$ be the first byte such that $\bar{b}_k \neq \bar{0}$

$S \leftarrow (\bar{b}_k)_{10} 256^k + \cdots + (\bar{b}_1)_{10} 256 + (\bar{b}_0)_{10}$
where $(\bar{b}_i)_{10}$ represents the decimal conversion.

Convert S to a base 58 representation $c_\ell \ldots c_1 c_0$, i.e., $S = c_\ell 58^\ell + \cdots + c_1 58 + c_0$.

Map c_i's as per the following table of Base58 characters

Append the resulting string with $n - k$ 1's (Step#1)

Int	Ch	Int	Ch	Int	Ch	Int	Ch	Int	Ch	Int	Ch	Int	Ch
0	1	9	A	18	K	27	U	36	d	44	m	53	v
1	2	10	B	19	L	28	V	37	e	45	n	54	w
2	3	11	C	20	M	29	W	38	f	46	o	55	x
3	4	12	D	21	N	30	X	39	g	47	p	56	y
4	5	13	E	22	P	31	Y	40	h	48	q	57	z
5	6	14	F	23	Q	32	Z	41	i	49	r		
6	7	15	G	24	R	33	a	42	j	50	s		
7	8	16	H	25	S	34	b	43	k	51	t		
8	9	17	J	26	T	35	c	44	m	52	u		

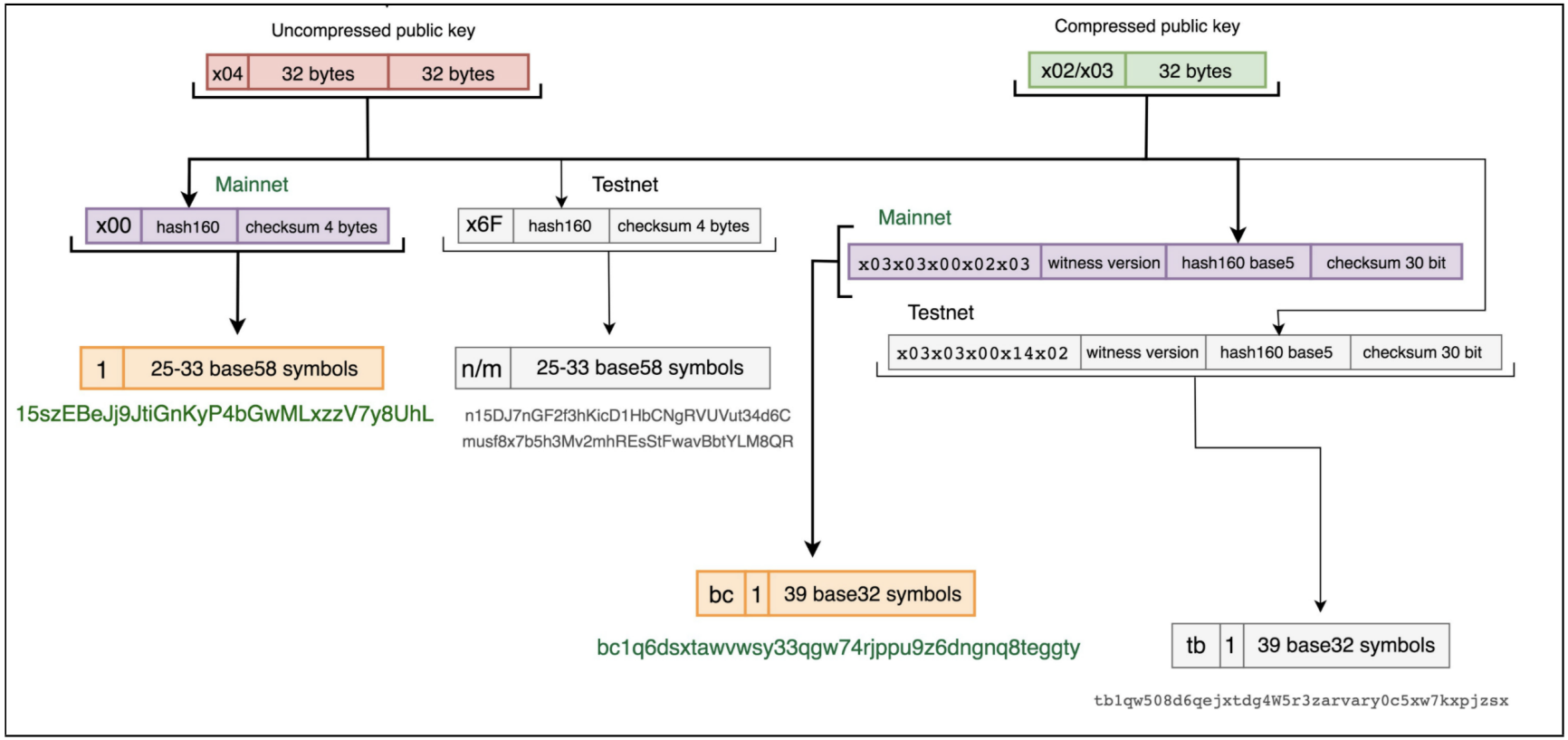
the number "0", the uppercase letter "O", the upper case letter "I" (of i) and the lower case letter "l" (ell) are excluded to avoid confusion.

Input: 0x000000261913

Leading zero bytes 0x00 00 00 26 19 13
$(S)_{256} = 38\ 25\ 19$
Thus $S = 2496787$
$(S)_{58} = 12\ 46\ 12\ 3$

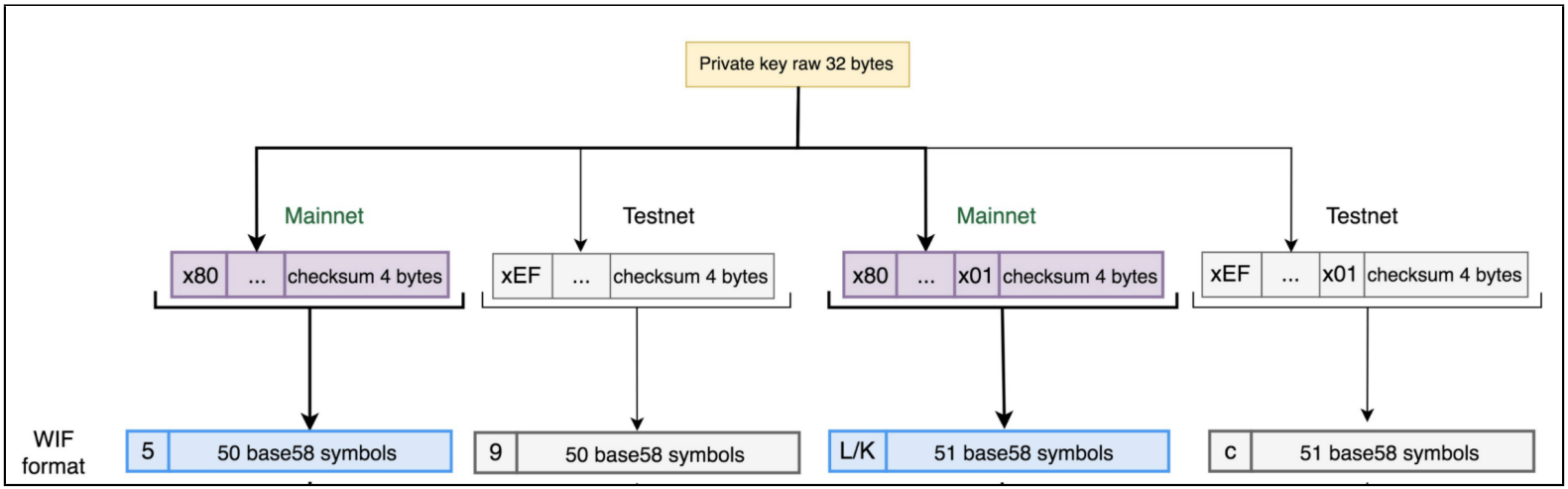
Output: 111DoD4

Bitcoin Address Prefix



Source: https://en.bitcoin.it/wiki/Invoice_address

Secret-key Prefix



Source: https://en.bitcoin.it/wiki/Invoice_address

Bitcoin Address: A Summary

Generate *sk*

Secret-key Serialization	
sk Format	Prefix
hex	
WIF	5
WIF-Comp.	K/L

pk $\leftarrow$ sk

Public-key Serialization	
pk Format	Prefix
Un-Comp.	04
Comp.	02/03

Bitcoin Address $\leftarrow$ pk

Address Serialization	
Address Format	Prefix
P2PK (mostly not used)	
P2PKH (Base58)	mainnet(1), testnet (m/n)
P2PKH (Bech32)	mainnet(bc1), testnet (tb1)
P2SH (Base58)	mainnet(3), testnet(2)
P2SH (Bech32)	mainnet(bc1), testnet(tb1)

- Few examples:
  - `1Dw1AaNiNVuCBL2nhJdE6tqRsWfg9W8GH8`
  - `bc1qzttylpu7m8gsaph3aj4fwquhjlgy34lmm2ysgm`
  - `34De3LyvhoJo5VvnRaMoyhQrutN6EGz5Ua`
- length(Bitcoin Address) ≤ 34
- Validation of Bitcoin Address

Ledger Accounting: An Alternative Way

txid	Sender Acc.	Receiver Acc.	Amount	Status
		...		
T_1	S	A	x_1	Unspent
		...		
A's Balance $= x_1$				

txid	Sender Acc.	Receiver Acc.	Amount	Status
		...		
T_1	S	A	x_1	Unspent
T_2	B	A	x_2	Unspent
		...		
A's Balance $= x_1 + x_2$				

txid	Sender Acc.	Receiver Acc.	Amount	Status
		...		
T_1	S	A	x_1	Unspent
T_2	B	A	x_2	Unspent
T_3	C	A	x_3	Unspent
		...		
A's Balance $= x_1 + x_2 + x_3$				

txid	Sender Acc.	Receiver Acc.	Amount	Status
		...		
T_1	S	A	x_1	Unspent
T_2	B	A	x_2	Spent
T_3	C	A	x_3	Unspent
T_4	spend T_2	D	x_2	Unspent
		...		
A's Balance $= x_1 + x_3$				

One must spend entire amount available at a txid

Balance of $A = \sum_{T_{X \to A}} T.\text{Unspent}$

txid	Sender Acc.	Receiver Acc.	Amount	Status
		...		
T_1	S	A	x_1	Unspent
T_2	B	A	x_2	Spent
T_3	C	A	x_3	Unspent
T_4	spend T_2	D	x_2	Unspent
T_5	spend some T of E	A	x_5	Unspent
		...		
A's Balance $= x_1 + x_3 + x_5$				

txid	Sender Acc.	Receiver Acc.	Amount	Status
		...		
T_1	S	A	x_1	Unspent
T_2	B	A	x_2	Spent
T_3	C	A	x_3	Spent
T_4	spend T_2	D	x_2	Unspent
T_5	spend some T of E	A	x_5	Unspent
T_6	spend T_3	F	x_6	Unspent
		A	$x_3 - x_6$	Unspent
		...		
A's Balance $= x_1 + x_5 + (x_3 - x_6)$				

- The Bitcoin network does the ledger accounting roughly in the above manner !!
- In the following, we see how users make Bitcoin transactions requesting bitcoin transfers and how these transactions get validated by the Bitcoin network !!

Receiving ₿: Bitcoin Transaction

- Suppose A $(\mathsf{pk}_A, \mathsf{sk}_A)$ and B $(\mathsf{pk}_B, \mathsf{sk}_B)$ are Bitcoin users.
  - A wishes to transfer x ₿ to B. For this, A must acquire $\mathsf{P2PKH}_{\mathsf{pk}_B}$
  - In the following we give two transactions (partial views) - the first one sending money to B; the second one describes how B can spend it subsequently

1. The transaction $T_{A \to B}$

txn	Input		Unspent txn Output (UTXO)	
	txid	scriptSig	Amount	scriptPubkey (LockScript)
$T_{A \to B}$			x ₿	OP_DUP OP_HASH160 $\langle \mathsf{HASH160}(\mathsf{pk}_B) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY

- Assuming $T_{A \to B}$'s approval by the Bitcoin network, the Bitcoin ledger (and the ledger state) gets updated as follows

Ledger	
txid	txn
...	
$H(T_{A \to B})$	$T_{A \to B}$
...	

Ledger State	
txid	txn-state
...	
$H(T_{A \to B})$	["0" : "(UTXO $= x$, Unspent)"]
...	

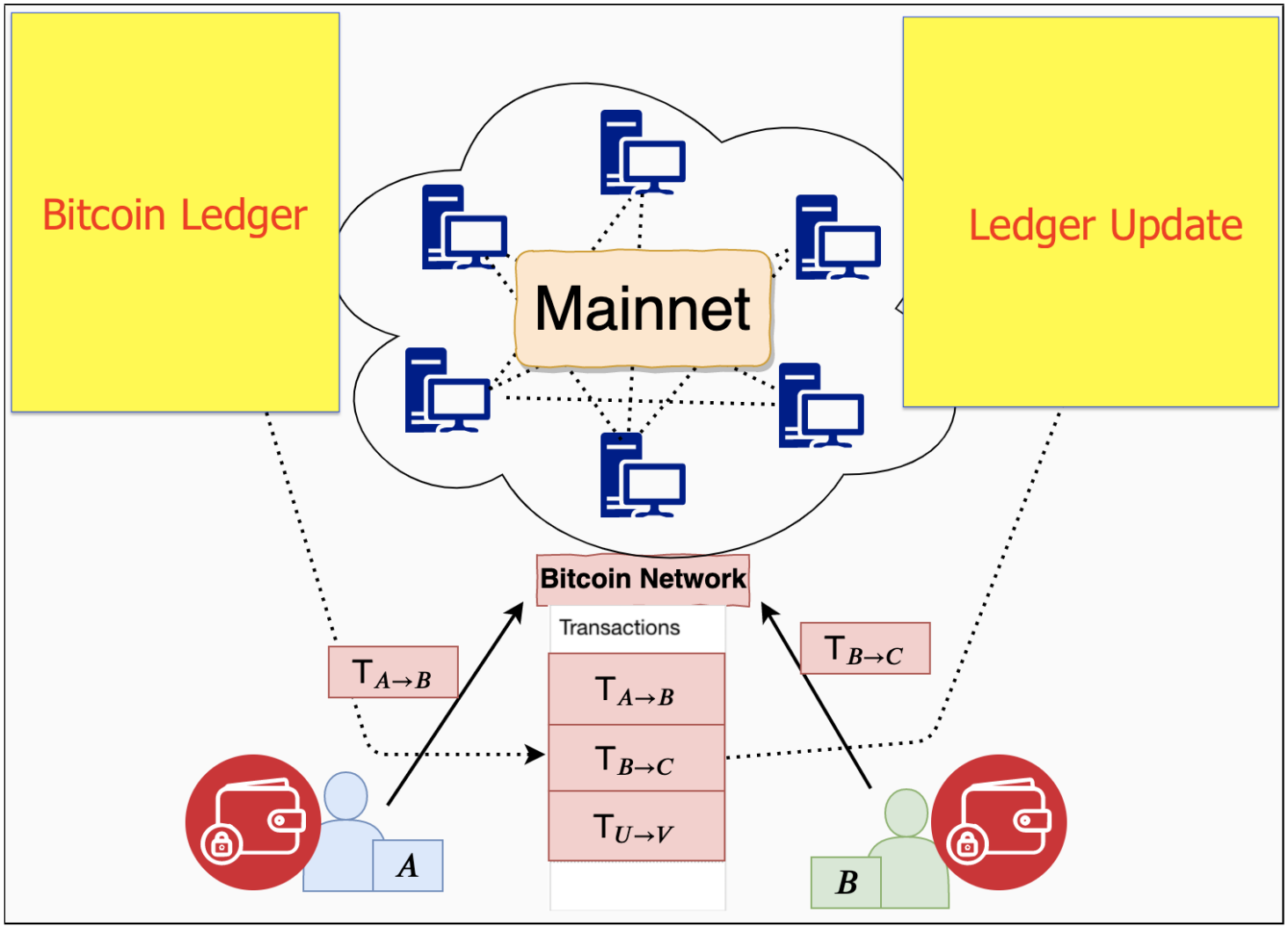
2. The transaction $T_{B \to C}$

txn	Input			Unspent txn Output (UTXO)
	txid	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{B \to C}$	$H(T_{A \to B})$	$\langle \mathsf{Sign}_{\mathsf{sk}_B}(T_{B \to C}) \rangle$ $\langle \mathsf{pk}_B \rangle$	x ₿	OP_DUP OP_HASH160 $\langle \mathsf{HASH160}(\mathsf{pk}_C) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY

Ledger	
txid	txn
...	
$H(T_{A \to B})$	$T_{A \to B}$
$H(T_{B \to C})$	$T_{B \to C}$
...	

Ledger State	
txid	txn-state
...	
$H(T_{A \to B})$	["0" : "(UTXO $= x$, Spent)"]
$H(T_{B \to C})$	["0" : "(UTXO $= x$, Unspent)"]
...	

Bitcoin Addresses, txns, Network: Piece Together



- **How does the network validate a Bitcoin transaction $T_{B\to C}$?**

Table of Contents

1 Lecture#1 (18/01) [Course Overview, Bitcoin Introduction]

2 Lecture#2 (21/01) [Hash Functions]

3 Lecture#3 (25/01) [Random Oracle Model, Iterated Hash Construction]

4 Lecture#4 (28/01) [Hash Puzzle, Signature Scheme]

5 Lecture#5 (01/02) [Group Theory]

6 Lecture#6 (04/02) [Discrete Logarithm Problem, DSA]

7 Lecture#7 (04/02) [Elliptic Curves, ECDSA]

8 Lecture#8 (11/02) [Bitcoin Address Generation]

9 Lecture#9 (15/02) [Bitcoin Transaction and its Validation]

10 Lecture#10 (18/02) [Bitcoin Consensus]

11 Lecture#11 (22/02) [Block Mining, txn Serialization]

12 Lecture#12 (25/02) [txn Serialization, PaytoScriptHash]

13 Lecture#13 (01/03) [More lockScripts, txn-malleability]

14 Lecture#14 (04/03) [SegWit transaction]

15 Lecture#15 (15/03) [Analysis of Bitcoin's Consensus]

16 Lecture#16 (21/03) [Hyperledger Fabric]

17 Lecture#17 (25/04) [Ethereum]

18 Lecture#18 (29/03) [Ethereum Contract Accounts]

19 Lecture#19 (05/04) [Payment Channels, Coin Mixers]

20 Lecture#20 (08/04) [Coin Mixers, MixEth, TumbleBit]

21 Lecture#21 (19/04) [Tokens]

ashoka UNIVERSITY

Transaction Validation by the Bitcoin Network

- Validating $T_{B \to C}$

txn	Input		Unspent txn Output (UTXO)	
	txid	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{B \to C}$	$H(T_{A \to B})$	$\langle \text{Sign}_{sk_B}(T_{B \to C}) \rangle \ \langle pk_B \rangle$	x ฿	$\begin{cases} \text{OP\_DUP OP\_HASH160 } \langle \text{HASH160}(pk_C) \rangle \\ \text{OP\_EQUALVERIFY OP\_CHECKSIGVERIFY} \end{cases}$

- **Validation Steps**

- **Step 1:** Use $H(T_{A \to B})$ to retrieve the transaction $T_{A \to B}$ which is validated and approved by the network earlier.

txn	Input		Unspent txn Output (UTXO)	
	txid	scriptSig	Amount	scriptPubkey (LockScript)
$T_{A \to B}$			x ฿	$\begin{cases} \text{OP\_DUP OP\_HASH160 } \langle \text{HASH160}(pk_B) \rangle \\ \text{OP\_EQUALVERIFY OP\_CHECKSIGVERIFY} \end{cases}$

- **Step 2:** Get the LockScript of $T_{A \to B}$

```
OP_DUP OP_HASH160 <HASH160(pk_B)> OP_EQUAL OP_CHECKSIG
```

- **Step 3:** Get the UnLockScript of $T_{B \to C}$

$$\text{Signature} = \langle \text{Sign}_{sk_B}(T_{B \to C}) \rangle \ \langle pk_B \rangle$$

- Feed ULScript and LScript in a special execution environment for $T_{B \to C}$'s validation!!

Special Execution Environment for txn Validation

- We assume at this point the Bitcoin Network as a single *node*
- LScript and ULScript are written using Bitcoin's programming language - "`Script`".
  - `Script` is a stack-based language. It avoids any mechanism for loops and is therefore not Turing complete! Turing completeness in a programming language essentially means that the program has the ability to loop.
  - But why pickup a language which is limited in its power??
  - There are a lot of reasons for this, but let's discuss this with respect to program execution.
  - For a moment assume `Script` supports "loops".
  - The transaction validation requires the Bitcoin Network to compile and execute both LScript and ULScript
  - A Bitcoin user knowingly/unknowingly may create a `Script` program (LSript/ULScript) having an infinite loop. This would cause Bitcoin network to enter and never leave that loop while executing the script.
  - This would be an easy way to attack the network through what would be called a denial-of-service (DoS) attack. A single `Script` program with an infinite loop could take down Bitcoin!
  - Protecting against this vulnerability is one of the major reasons why Turning completeness is avoided.
  - Also, LSript/ULScript written with Turning complete programming language are very difficult to analyze. Easy to create unintended behavior, causing bugs.
  - Bugs in a LSript/ULScript means that the coins are vulnerable to being unintentionally spent.

txn Validation: Executing ULScript and LScript in Sequence

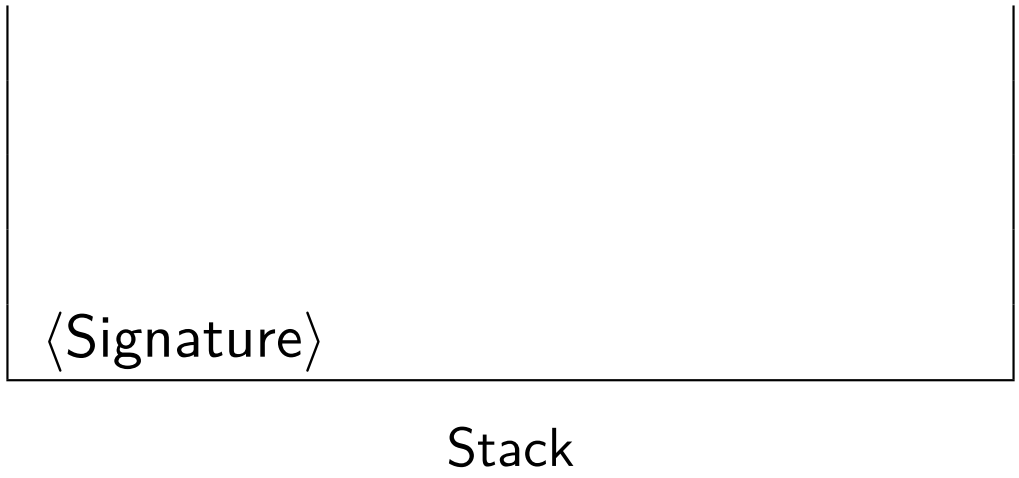
- ULScript||LSscript
- $\Downarrow$
  ⟨Signature⟩ ⟨Public Key⟩ op_dup op_hash160 ⟨Public Key Hash⟩ op_equal op_checksig

Stack

- $\Downarrow$ points to the next argument to be executed

txn Validation: Executing ULScript and LScript in Sequence

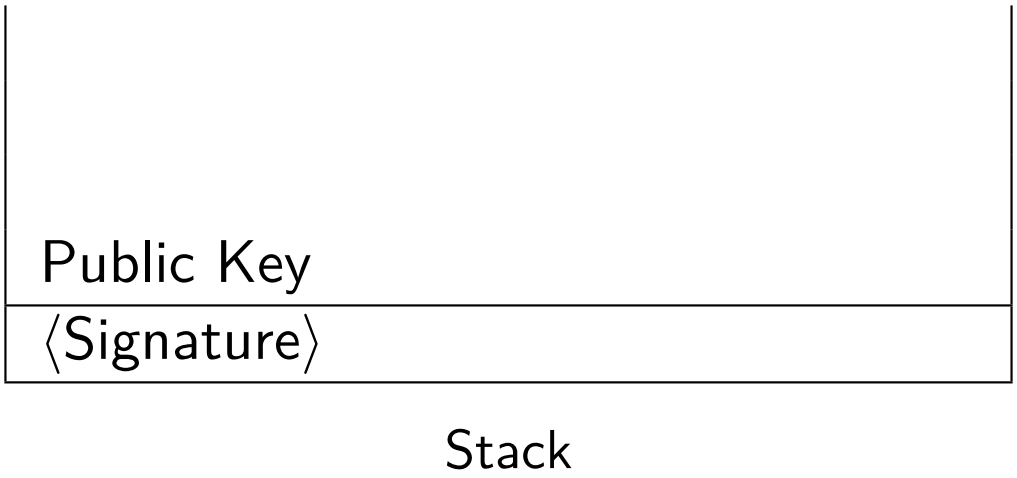
- $\text{ULS}_{\text{UTXO}} \| \text{LS}_{\text{UTXO}}$
- $\langle \text{Signature} \rangle \; \langle \text{Public Key} \rangle \Downarrow \; \text{op\_dup op\_hash160} \; \langle \text{Public Key Hash} \rangle \; \text{op\_equal op\_checksig}$



- $\Downarrow$ points to the next argument to be executed

txn Validation: Executing ULScript and LScript in Sequence

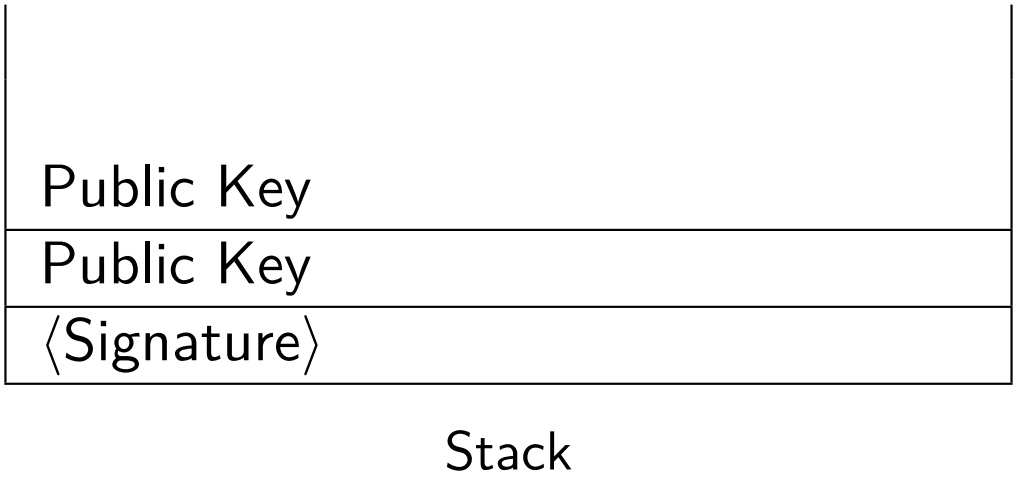
- $\text{ULS}_{\text{UTXO}} \| \text{LS}_{\text{UTXO}}$
- $\langle \text{Signature} \rangle \ \langle \text{Public Key} \rangle \ \underset{\Downarrow}{\text{op\_dup}} \ \text{op\_hash160} \ \langle \text{Public Key Hash} \rangle \ \text{op\_equal} \ \text{op\_checksig}$



- $\Downarrow$ points to the next argument to be executed

txn Validation: Executing ULScript and LScript in Sequence

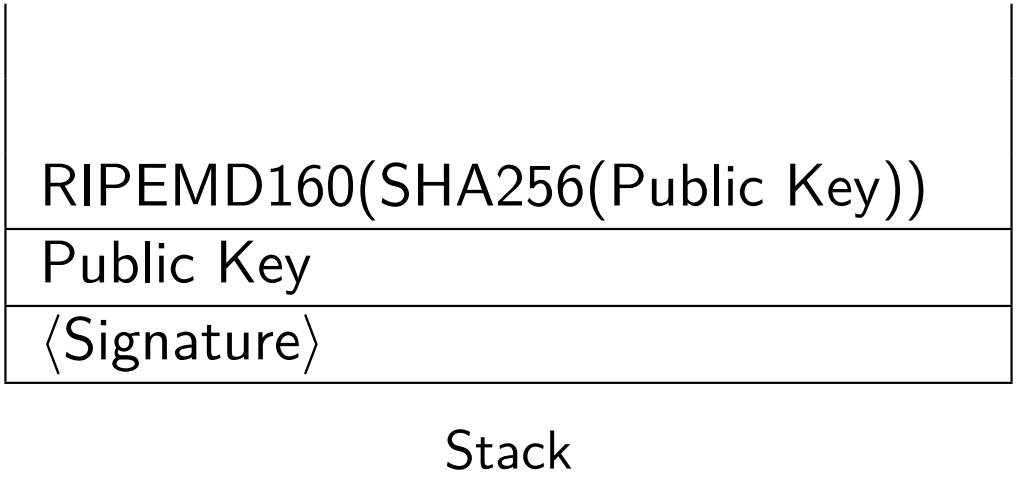
- $\text{ULS}_{\text{UTXO}} \| \text{LS}_{\text{UTXO}}$
- $\langle \text{Signature} \rangle \ \langle \text{Public Key} \rangle \ \text{op\_dup} \ \text{op\_hash160} \overset{\Downarrow}{} \ \langle \text{Public Key Hash} \rangle \ \text{op\_equal} \ \text{op\_checksig}$



- $\Downarrow$ points to the next argument to be executed

txn Validation: Executing ULScript and LScript in Sequence

- $\text{ULS}_{\text{UTXO}} \| \text{LS}_{\text{UTXO}}$
- $\langle \text{Signature} \rangle \ \langle \text{Public Key} \rangle \ \text{op\_dup} \ \text{op\_hash160} \ \langle \text{Public Key Hash} \rangle \overset{\Downarrow}{} \ \text{op\_equal} \ \text{op\_checksig}$



- $\Downarrow$ points to the next argument to be executed

txn Validation: Executing ULScript and LScript in Sequence

- $\text{ULS}_{\text{UTXO}} \| \text{LS}_{\text{UTXO}}$
- $\langle \text{Signature} \rangle \ \langle \text{Public Key} \rangle \ \text{op\_dup op\_hash160} \ \langle \text{Public Key Hash} \rangle \ \overset{\Downarrow}{\text{op\_equal}} \ \text{op\_checksig}$

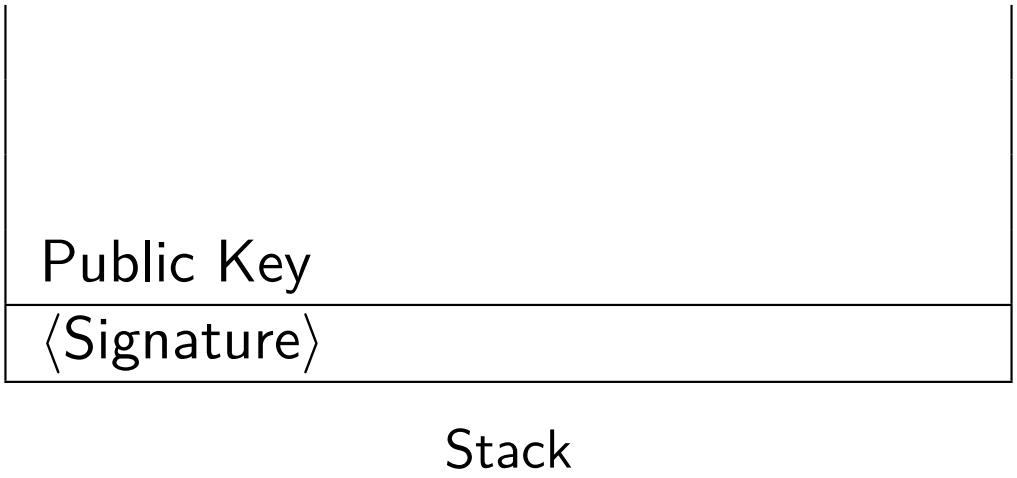
$\langle \text{Public Key Hash} \rangle$
RIPEMD160(SHA256(Public Key))
Public Key
$\langle \text{Signature} \rangle$

Stack

- $\Downarrow$ points to the next argument to be executed

txn Validation: Executing ULScript and LScript in Sequence

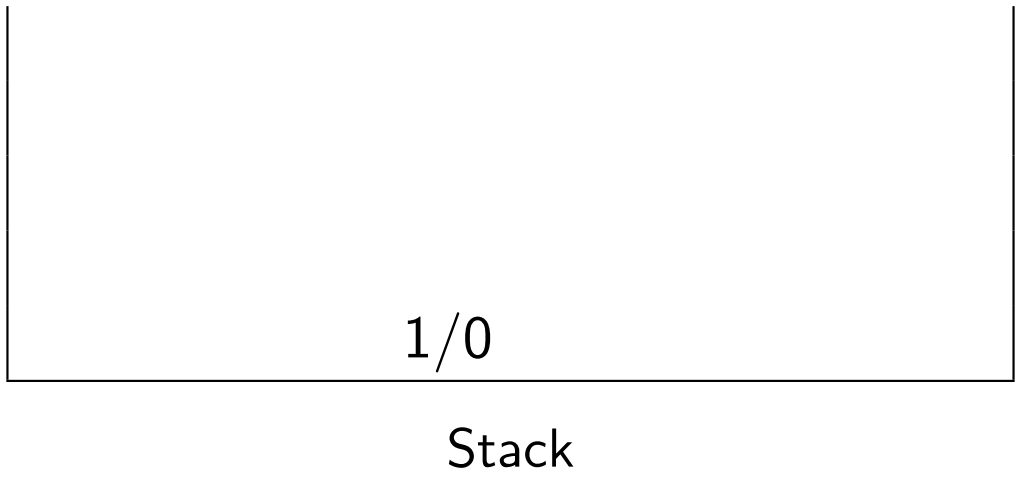
- $\text{ULS}_{\text{UTXO}} \| \text{LS}_{\text{UTXO}}$
- $\langle \text{Signature} \rangle \ \langle \text{Public Key} \rangle \ \text{op\_dup op\_hash160} \ \langle \text{Public Key Hash} \rangle \ \text{op\_equal op\_checksig}$ (⇓ over checksig)



- op_checksig consumes the top two elements from the stack, the first being the pubkey and the second being a signature.
  - It pushes $\begin{cases} 1 & \text{if Verify(signature, pubkey)} = \text{valid} \\ 0 & \text{otherwise} \end{cases}$ to the stack!

txn Validation: Executing ULScript and LScript in Sequence

- $\text{ULS}_{\text{UTXO}} || \text{LS}_{\text{UTXO}}$
- ⟨Signature⟩ ⟨Public Key⟩ op_dup op_hash160 ⟨Public Key Hash⟩ op_equal op_checksig



- After all the commands are evaluated, the top element of the stack must be nonzero for the combine script to resolve as valid
  - Having no elements in the stack or the top element being 0 would resolve as invalid
  - Resolving as invalid means that the txn is not accepted on the network.

Transaction: Multiple Inputs, Multiple Outputs

- **Multiple Inputs, Single Output**

txn	Input		Unspent txn Output (UTXO)	
	txid	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{B_{123} \to C}$	$H(T_{A_1 \to B_1})$ $H(T_{A_2 \to B_2})$ $H(T_{A_3 \to B_3})$	$\langle \text{Sign}_{sk_{B_1}}(T_{B_{123} \to C}) \rangle \langle pk_{B_1} \rangle$ $\langle \text{Sign}_{sk_{B_2}}(T_{B_{123} \to C}) \rangle \langle pk_{B_2} \rangle$ $\langle \text{Sign}_{sk_{B_3}}(T_{B_{123} \to C}) \rangle \langle pk_{B_3} \rangle$	x ฿	OP_DUP OP_HASH160 $\langle \text{HASH160}(pk_C) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY

Ledger

txid	txn
...	
$H(T_{B_{123} \to C})$	$T_{B_{123} \to C}$
...	

Ledger State

txid	txn-state
...	
$H(T_{B_{123} \to C})$	["0" : "(UTXO = x, Unspent)"]
...	

- **Multiple Inputs, Multiple Outputs**

txn	Input		Unspent txn Output (UTXO)	
	txid	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{B_{123} \to C_{12}}$	$H(T_{A_1 \to B_1})$	$\langle \text{Sign}_{sk_{B_1}}(T_{B_{123} \to C_{12}}) \rangle \langle pk_{B_1} \rangle$	x_1 ฿	OP_DUP OP_HASH160 $\langle \text{HASH160}(pk_{C_1}) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
	$H(T_{A_2 \to B_2})$	$\langle \text{Sign}_{sk_{B_2}}(T_{B_{123} \to C_{12}}) \rangle \langle pk_{B_2} \rangle$		
	$H(T_{A_3 \to B_3})$	$\langle \text{Sign}_{sk_{B_3}}(T_{B_{123} \to C_{12}}) \rangle \langle pk_{B_3} \rangle$	x_2 ฿	OP_DUP OP_HASH160 $\langle \text{HASH160}(pk_{C_2}) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY

- The ledger gets updated as follows post $T_{B_{123} \to C_{12}}$'s approval

Ledger

txid	txn
...	
$H(T_{B_{123} \to C_{12}})$	$T_{B_{123} \to C_{12}}$
...	

Ledger State

txid	txn-state
...	
$H(T_{B_{123} \to C_{12}})$	["0" : "(UTXO = x_1, Unspent)", "1" : "(UTXO = x_2, Unspent)"]
...	

Transaction: Spending Selective Outputs

- **Current View**

Ledger

txid	txn
...	
$H(T_{B_{123} \to C_{12}})$	$T_{B_{123} \to C_{12}}$
...	

Ledger State

txid	txn-state
...	
$H(T_{B_{123} \to C_{12}})$	["0" : "(UTXO $= x_1$, Unspent)", "1" : "(UTXO $= x_2$, Unspent)"]
...	

txn	Input		Unspent txn Output (UTXO)	
	txid	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{B_{123} \to C_{12}}$	$H(T_{A_1 \to B_1})$	$\langle \text{Sign}_{sk_{B_1}}(T_{B_{123} \to C_{12}}) \rangle \ \langle pk_{B_1} \rangle$	x_1 ฿	OP_DUP OP_HASH160 $\langle$HASH160(pk_{C_1})$\rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
	$H(T_{A_2 \to B_2})$	$\langle \text{Sign}_{sk_{B_2}}(T_{B_{123} \to C_{12}}) \rangle \ \langle pk_{B_2} \rangle$		
	$H(T_{A_3 \to B_3})$	$\langle \text{Sign}_{sk_{B_3}}(T_{B_{123} \to C_{12}}) \rangle \ \langle pk_{B_3} \rangle$	x_2 ฿	OP_DUP OP_HASH160 $\langle$HASH160(pk_{C_2})$\rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY

- **A New Transaction, Subsequent Ledger Update**

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{C_1 \to D}$	$H(T_{B_{123} \to C_{12}})$	0	$\langle \text{Sign}_{sk_{C_1}}(T_{C_1 \to D}) \rangle \ \langle pk_{C_1} \rangle$	x_1 ฿	OP_DUP OP_HASH160 $\langle$HASH160(pk_D)$\rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY

Ledger

txid	txn
...	
$H(T_{B_{123} \to C_{12}})$	$T_{B_{123} \to C_{12}}$
$H(T_{C_1 \to D})$	$T_{C_1 \to D}$
...	

Ledger State

txid	txn-state
...	
$H(T_{B_{123} \to C_{12}})$	["0" : "(UTXO $= x_1$, Spent)", "1" : "(UTXO $= x_2$, Unspent)"]
$H(T_{C_1 \to D})$	["0" : "(UTXO $= x_1$, Unspent)"]
...	

Transaction: Consuming UTXO Amounts in its Entirety

- **Current View**

Ledger	
txid	txn
...	
$H(T_{B_{123} \to C_{12}})$	$T_{B_{123} \to C_{12}}$
...	

Ledger State	
txid	txn-state
...	
$H(T_{B_{123} \to C_{12}})$	["0" : "(UTXO = x_1, Unspent)", "1" : "(UTXO = x_2, Unspent)"]
...	

txn	Input		Unspent txn Output (UTXO)	
	txid	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{B_{123} \to C_{12}}$	$H(T_{A_1 \to B_1})$	$\langle \text{Sign}_{sk_{B_1}}(T_{B_{123} \to C_{12}}) \rangle \ \langle pk_{B_1} \rangle$	x_1 ฿	OP_DUP OP_HASH160 $\langle$HASH160(pk_{C_1})$\rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
	$H(T_{A_2 \to B_2})$	$\langle \text{Sign}_{sk_{B_2}}(T_{B_{123} \to C_{12}}) \rangle \ \langle pk_{B_2} \rangle$		
	$H(T_{A_3 \to B_3})$	$\langle \text{Sign}_{sk_{B_3}}(T_{B_{123} \to C_{12}}) \rangle \ \langle pk_{B_3} \rangle$	x_2 ฿	OP_DUP OP_HASH160 $\langle$HASH160(pk_{C_2})$\rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY

- **A New Transaction, Subsequent Ledger Update**

- Suppose C_1 wants to transfer $x < x_1$ ฿ !!
- **Rule** A transaction output must be consumed completely by another transaction. So the spending transaction must create a new output where $(x_1 - x)$ ฿ are sent back to C_1 by locking it to pk_{C_1} or some other address controlled by C_1.

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{C_1 \to \{D, C_1\}}$	$H(T_{B_{123} \to C_{12}})$	0	$\langle \text{Sign}_{sk_{C_1}}(T_{C_1 \to \{D, C_1\}}) \rangle \ \langle pk_{C_1} \rangle$	x ฿	OP_DUP OP_HASH160 $\langle$HASH160(pk_D)$\rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
				$(x_1 - x)$ ฿	OP_DUP OP_HASH160 $\langle$HASH160(pk_{C_1})$\rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY

Transaction Fee

- Current View

Ledger	
txid	txn
...	
$H(T_{B_{123}\to C_{12}})$	$T_{B_{123}\to C_{12}}$
...	

Ledger State	
txid	txn-state
...	
$H(T_{B_{123}\to C_{12}})$	["0" : "(UTXO $= x_1$, Unspent)", "1" : "(UTXO $= x_2$, Unspent)"]
...	

txn	Input		Unspent txn Output (UTXO)	
	txid	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{B_{123}\to C_{12}}$	$H(T_{A_1\to B_1})$	$\langle \text{Sign}_{sk_{B_1}}(T_{B_{123}\to C_{12}})\rangle\ \langle pk_{B_1}\rangle$	x_1 ฿	OP_DUP OP_HASH160 $\langle$HASH160(pk_{C_1})$\rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
	$H(T_{A_2\to B_2})$	$\langle \text{Sign}_{sk_{B_2}}(T_{B_{123}\to C_{12}})\rangle\ \langle pk_{B_2}\rangle$		
	$H(T_{A_3\to B_3})$	$\langle \text{Sign}_{sk_{B_3}}(T_{B_{123}\to C_{12}})\rangle\ \langle pk_{B_3}\rangle$	x_2 ฿	OP_DUP OP_HASH160 $\langle$HASH160(pk_{C_2})$\rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY

- A New Transaction, Subsequent Ledger Update

- Bitcoin network charges a transaction fee for validating transactions
- Rule

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{C_1\to\{D,C_1\}}$	$H(T_{B_{123}\to C_{12}})$	0	$\langle \text{Sign}_{sk_{C_1}}(T_{C_1\to\{D,C_1\}})\rangle\ \langle pk_{C_1}\rangle$	x ฿	OP_DUP OP_HASH160 $\langle$HASH160(pk_D)$\rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
				y ฿	OP_DUP OP_HASH160 $\langle$HASH160(pk_{C_1})$\rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY

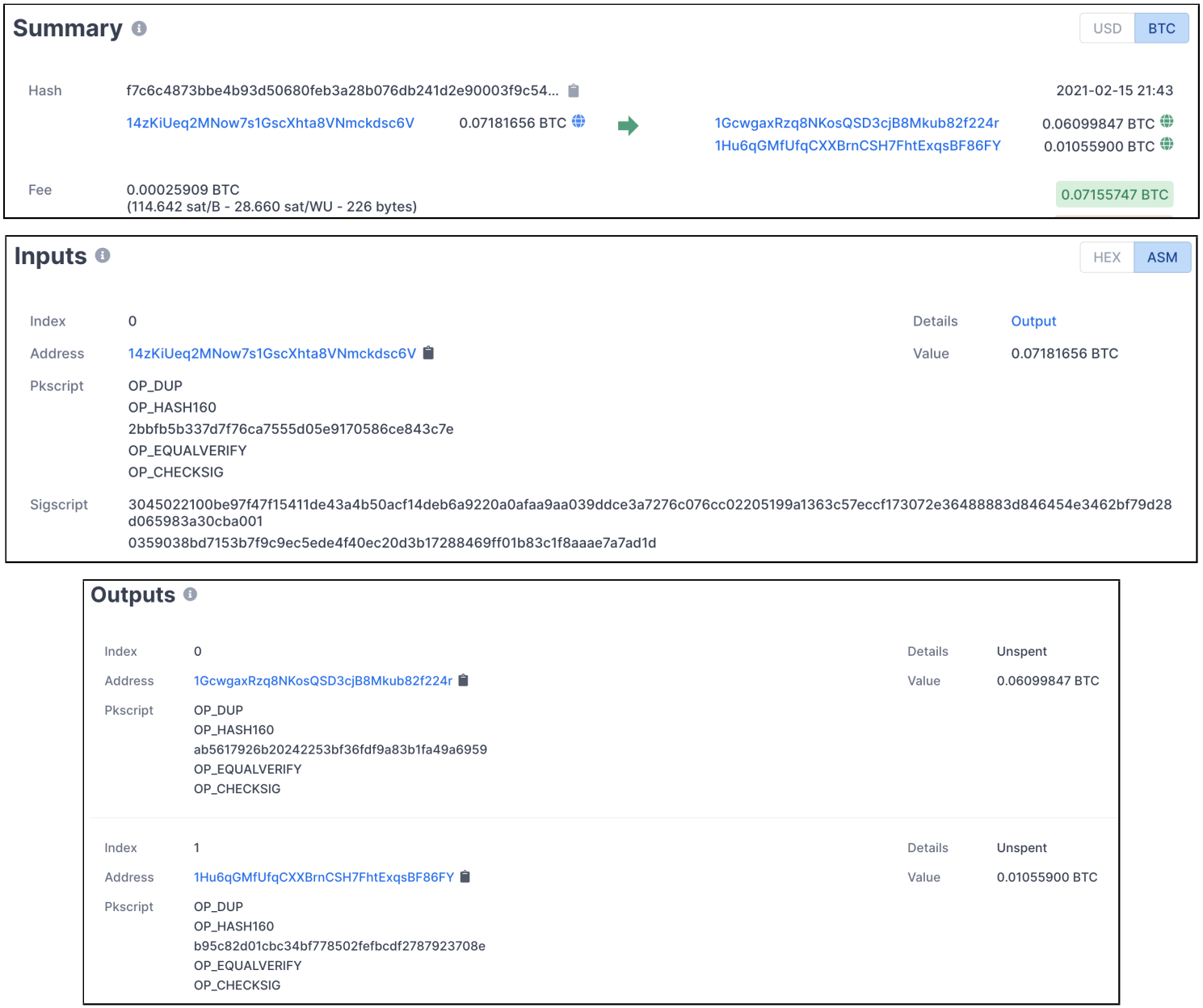
where $y = x_1 - x - \text{fee}$

Ledger	
txid	txn
...	
$H(T_{B_{123}\to C_{12}})$	$T_{B_{123}\to C_{12}}$
$H(T_{C_1\to\{D,C_1\}})$	$T_{C_1\to\{D,C_1\}}$
...	

Ledger State	
txid	txn-state
...	
$H(T_{B\to\{C_1,C_2\}})$	["0" : "(UTXO $= x_1$, Spent)", "1" : "(UTXO $= x_2$, Unspent)"]
$H(T_{C_1\to\{D,C_1\}})$	["0" : "(UTXO $= x$, Unspent)", "1" : "(UTXO $= y$, Unspent)"]
...	

- Who gets the fee ? How it gets reflected in the ledger ?

A Real Transaction



Summary

USD | BTC

Hash	f7c6c4873bbe4b93d50680feb3a28b076db241d2e90003f9c54...			2021-02-15 21:43
	14zKiUeq2MNow7s1GscXhta8VNmckdsc6V	0.07181656 BTC	1GcwgaxRzq8NKosQSD3cjB8Mkub82f224r 1Hu6qGMfUfqCXXBrnCSH7FhtExqsBF86FY	0.06099847 BTC 0.01055900 BTC
Fee	0.00025909 BTC (114.642 sat/B - 28.660 sat/WU - 226 bytes)			0.07155747 BTC

Inputs

HEX | ASM

Index	0	Details	Output
Address	14zKiUeq2MNow7s1GscXhta8VNmckdsc6V	Value	0.07181656 BTC
Pkscript	OP_DUP OP_HASH160 2bbfb5b337d7f76ca7555d05e9170586ce843c7e OP_EQUALVERIFY OP_CHECKSIG		
Sigscript	3045022100be97f47f15411de43a4b50acf14deb6a9220a0afaa9aa039ddce3a7276c076cc02205199a1363c57eccf173072e36488883d846454e3462bf79d28d065983a30cba001 0359038bd7153b7f9c9ec5ede4f40ec20d3b17288469ff01b83c1f8aaae7a7ad1d		

Outputs

Index	0	Details	Unspent
Address	1GcwgaxRzq8NKosQSD3cjB8Mkub82f224r	Value	0.06099847 BTC
Pkscript	OP_DUP OP_HASH160 ab5617926b20242253bf36fdf9a83b1fa49a6959 OP_EQUALVERIFY OP_CHECKSIG		
Index	1	Details	Unspent
Address	1Hu6qGMfUfqCXXBrnCSH7FhtExqsBF86FY	Value	0.01055900 BTC
Pkscript	OP_DUP OP_HASH160 b95c82d01cbc34bf778502fefbcdf2787923708e OP_EQUALVERIFY OP_CHECKSIG		

Bitcoin Ledger and Worldstate

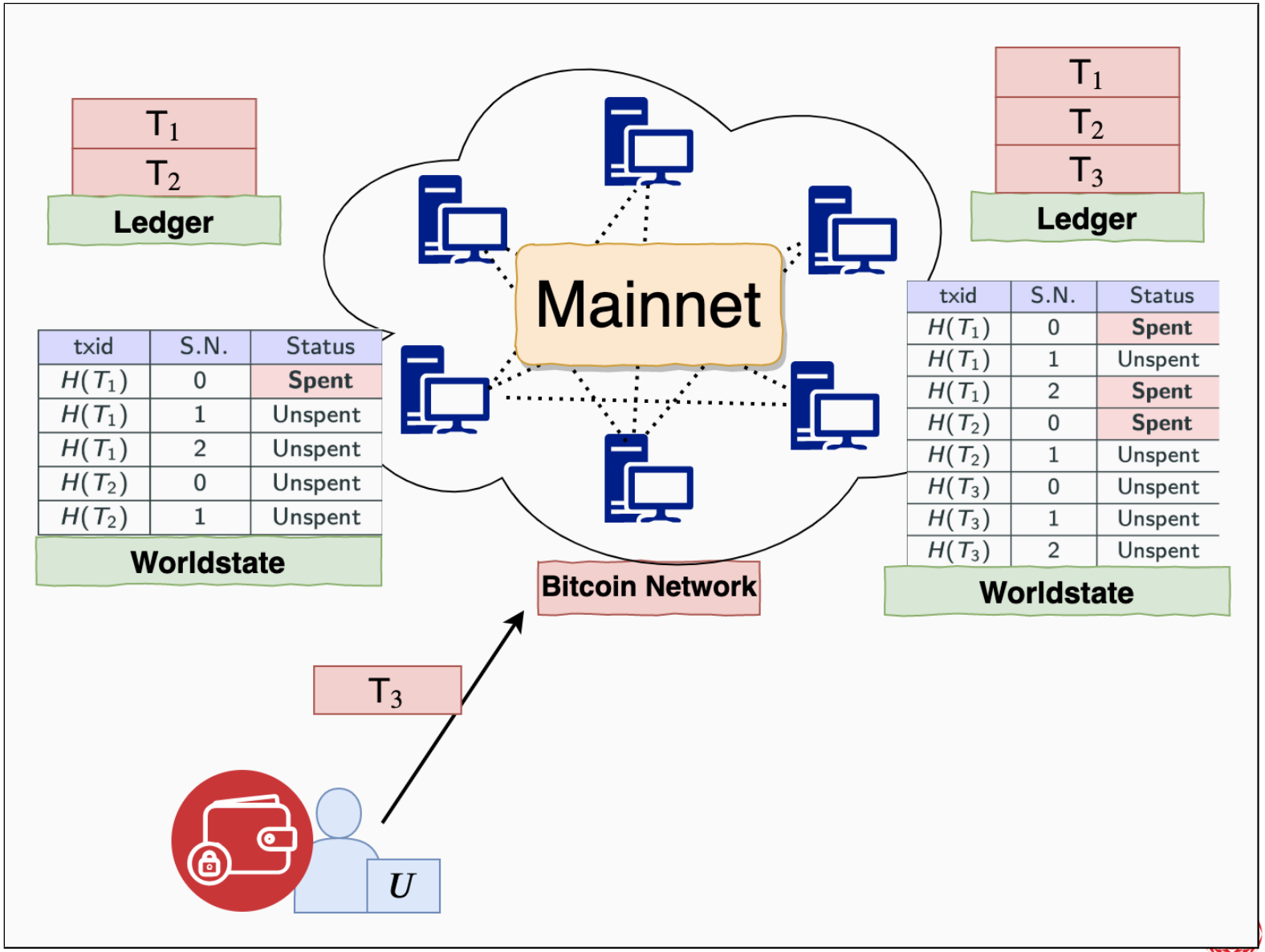


Table of Contents

1 Lecture#1 (18/01) [Course Overview, Bitcoin Introduction]

2 Lecture#2 (21/01) [Hash Functions]

3 Lecture#3 (25/01) [Random Oracle Model, Iterated Hash Construction]

4 Lecture#4 (28/01) [Hash Puzzle, Signature Scheme]

5 Lecture#5 (01/02) [Group Theory]

6 Lecture#6 (04/02) [Discrete Logarithm Problem, DSA]

7 Lecture#7 (04/02) [Elliptic Curves, ECDSA]

8 Lecture#8 (11/02) [Bitcoin Address Generation]

9 Lecture#9 (15/02) [Bitcoin Transaction and its Validation]

10 Lecture#10 (18/02) [Bitcoin Consensus]

11 Lecture#11 (22/02) [Block Mining, txn Serialization]

12 Lecture#12 (25/02) [txn Serialization, PaytoScriptHash]

13 Lecture#13 (01/03) [More lockScripts, txn-malleability]

14 Lecture#14 (04/03) [SegWit transaction]

15 Lecture#15 (15/03) [Analysis of Bitcoin's Consensus]

16 Lecture#16 (21/03) [Hyperledger Fabric]

17 Lecture#17 (25/04) [Ethereum]

18 Lecture#18 (29/03) [Ethereum Contract Accounts]

19 Lecture#19 (05/04) [Payment Channels, Coin Mixers]

20 Lecture#20 (08/04) [Coin Mixers, MixEth, TumbleBit]

21 Lecture#21 (19/04) [Tokens]

ashoka UNIVERSITY

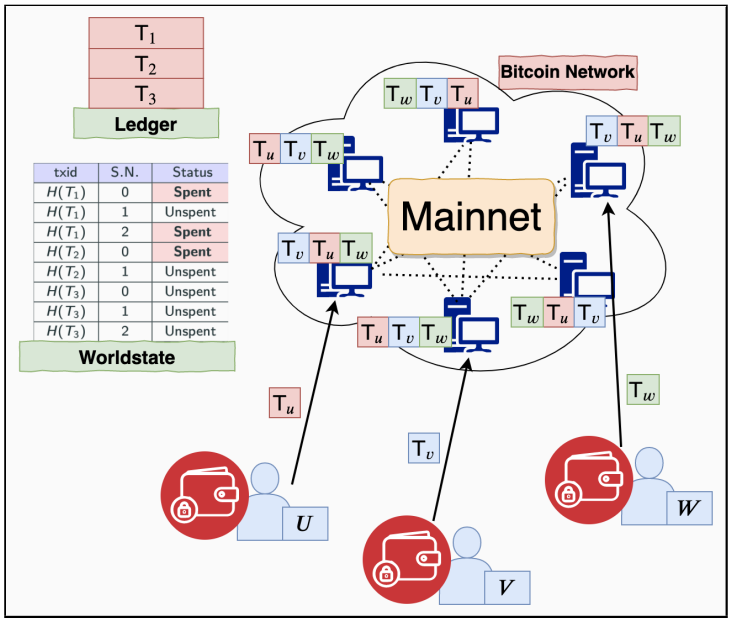
Working of the Bitcoin Network

- How does the Bitcoin Network, a peer-to-peer network, work ?

1. Who maintains the ledger of transactions?
2. Who has authority over which transactions are valid?
3. Who creates new bitcoins?
4. Who determines how the rules of the system change?
5. How do bitcoins acquire exchange value?

- Our focus: 1 – 4

Working of the Bitcoin Network



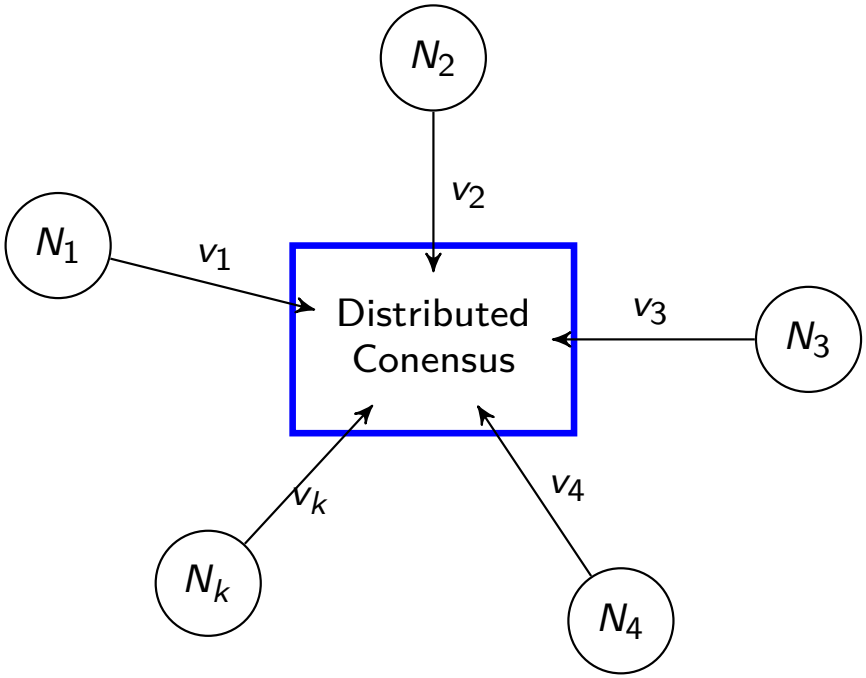
- When a user A wants to pay user B, it broadcasts a transaction to the peer-to-peer network, thus reaching all nodes.
- Given that a variety of users are broadcasting these transactions to the network, the nodes must agree on all transactions that were broadcasted and the order in which those transactions happened.
- This will result in a single, global ledger for the system

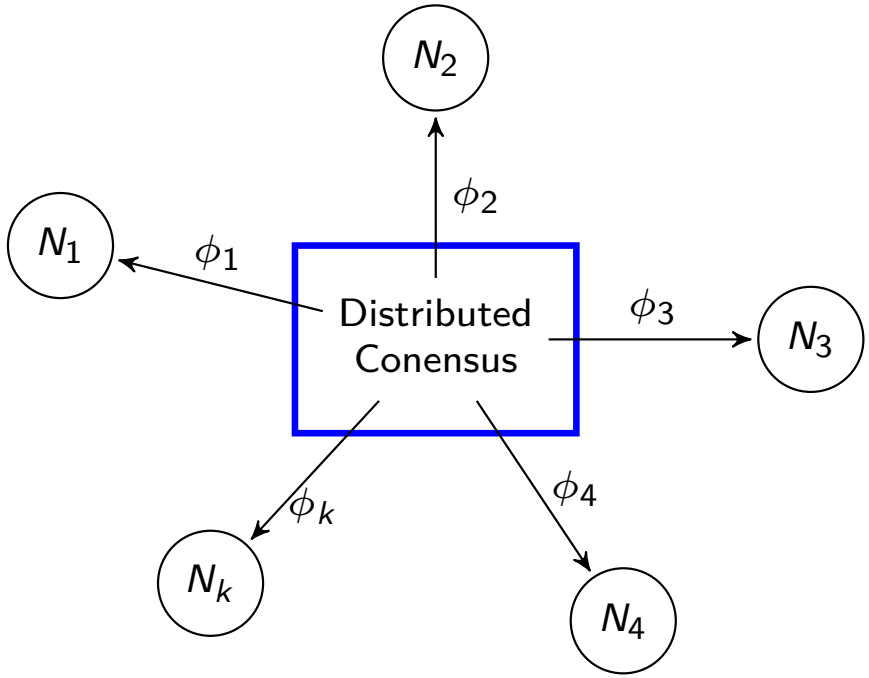
- The nodes in the Bitcoin network must reach a consensus on all those transactions proposed and the order in which those transactions happened !!
- How difficult is this to achieve??

- Abstracting out the problem:
  - **Input**
    - A peer-to-peer decentralised network comprises of n **untrusted** nodes $N_1, \ldots, N_k$.
    - Some of these nodes are faulty or malicious.
    - Each of these nodes have an input value v_i.
  - **Output**
    - All honest nodes must agree on a value v_j for some $1 \leq j \leq k$.
    - N_j, who has the value v_j, must be honest.

- This is called: "Consensus Problem"!
- Algorithms for solving a consensus problem are called "consensus protocols"
- A consensus protocol is called "distributed" if the underlying peer-to-peer network is decentralised.
- Do we have good distributed consensus protocols ??

General Distributed Consensus Setup





- The consensus requires that
  - $\phi_{i_1} = \cdots = \phi_{i_r}$ where $\mathcal{H} = \{N_{i_1}, \ldots, N_{i_r}\}$ is the set of all honest nodes, and
  - $\phi_{i_1} = \cdots = \phi_{i_r} = v_{i_s}$ where v_{i_s} is an honest N_{i_s}'s input

Distributed Consensus Protocol

- Distributed consensus been studied for decades in computer science.
- It has various applications.
- The traditional motivating application is reliability in distributed systems.
- Imagine you're in charge of the backend for a large social networking company.
- Systems of this sort typically have thousands of servers, which together form a massive distributed database that records all of the actions that happen in the system.
- Each piece of information must be recorded on several different nodes in this backend, and the nodes must be in sync about the overall state of the system.

▶ Distributed Consensus: (Im)Possibility Results

1. It is impossible to achieve distributed consensus if one-third or more of the total number of participating nodes are dishonest.
2. Distributed consensus achievable under certain limitations (Popular protocol - **Paxos**)
   - Under certain conditions, the protocol can get stuck and fail to make any progress
3. Many other impossibility results proven in specific models

---

- Breakthrough: Bitcoin's Distributed Consensus Protcol

The Flow

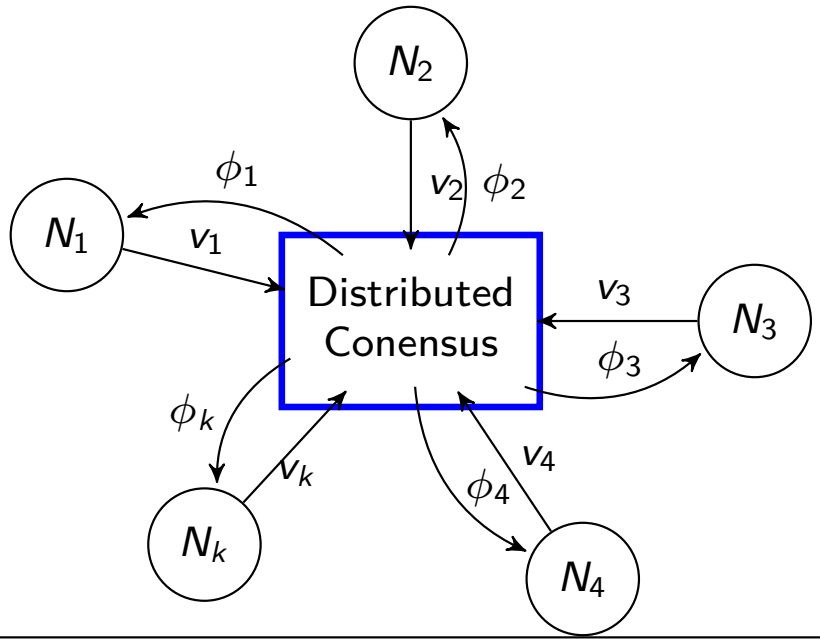
- txn Creation
  - Bitcoin transactions are created by wallets and pushed into the network

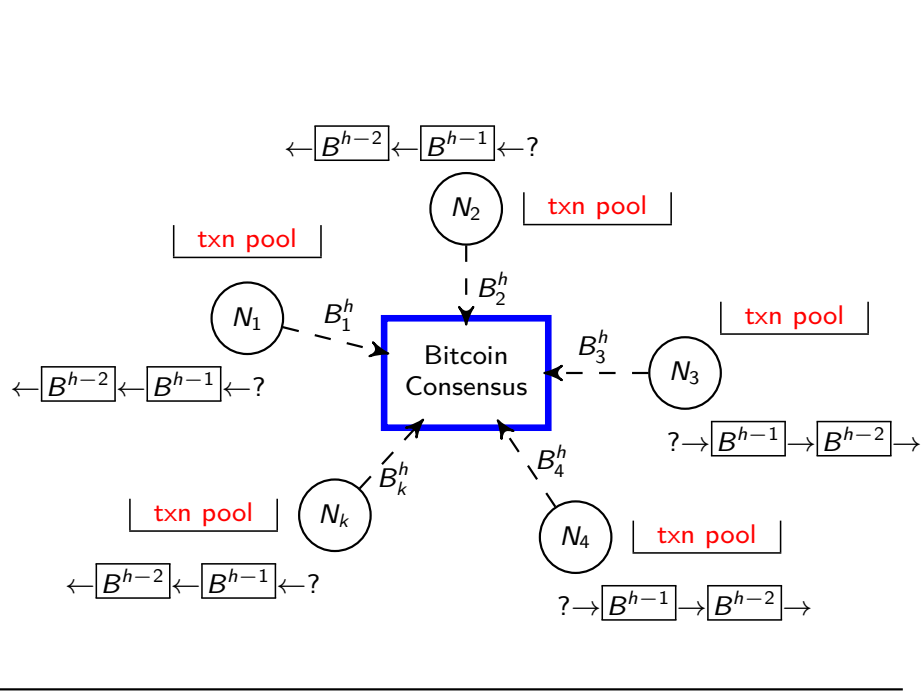
---

- txn Validation
  - Transactions soon reach all full nodes in the network
  - Every node would validate received transactions
  - Validation requires each node to use `Bitcoin Core` software, and the special execution environment therein
  - The validated transactions are put into each node's `transaction pool`
  - Unconfirmed `txn` The `txns` that are present in the transaction pool of a node, but not in the network's ledger.

- txn Ordering $\equiv$ Bitcoin's Distributed Consensus
  - The network now get together to reach a consensus on a set of unconfirmed `txns` next.
  - To this, every node prepares a valid block containing a set of unconfirmed `txns` from its pool.
  - Confirmed `txn` A `txn` is tagged as confirmed if it is included into a valid block.
  - Network is said to reach a consensus if all nodes agree on the same block.

General Vs Bitcoin - Distributed Consensus

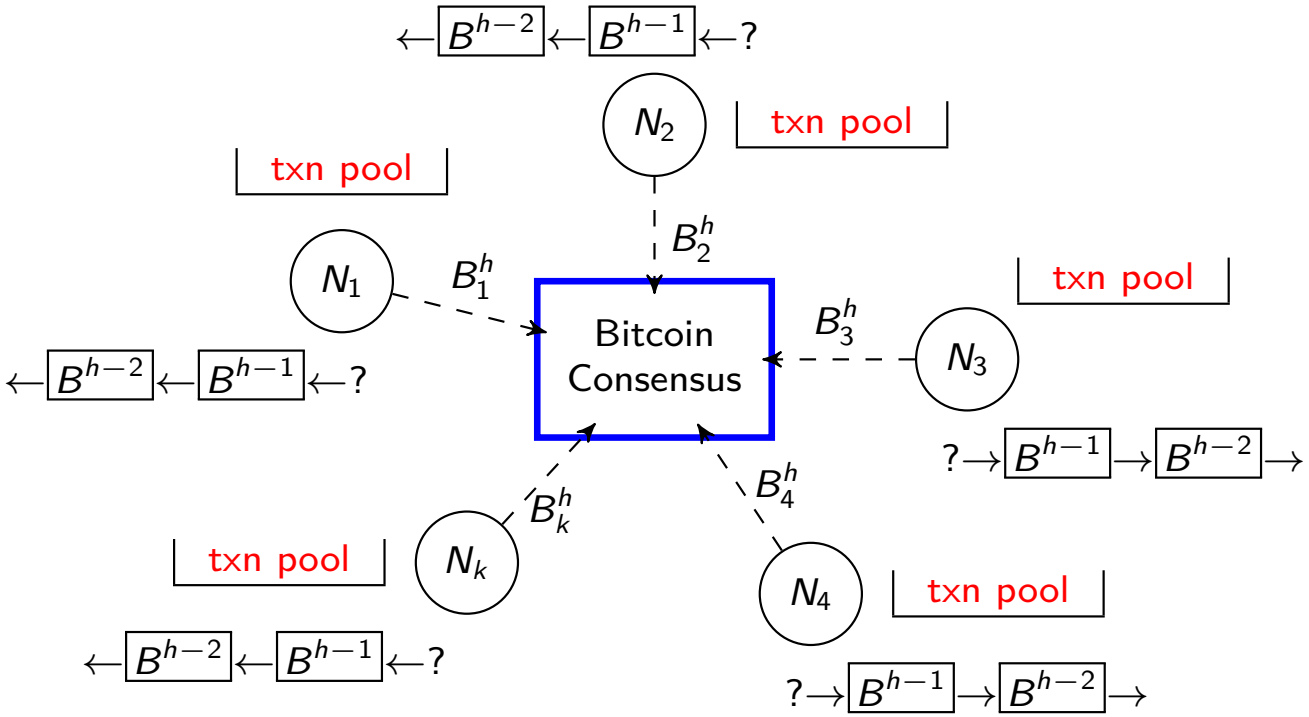




- Important observations about existing distributed consensus algorithms !!

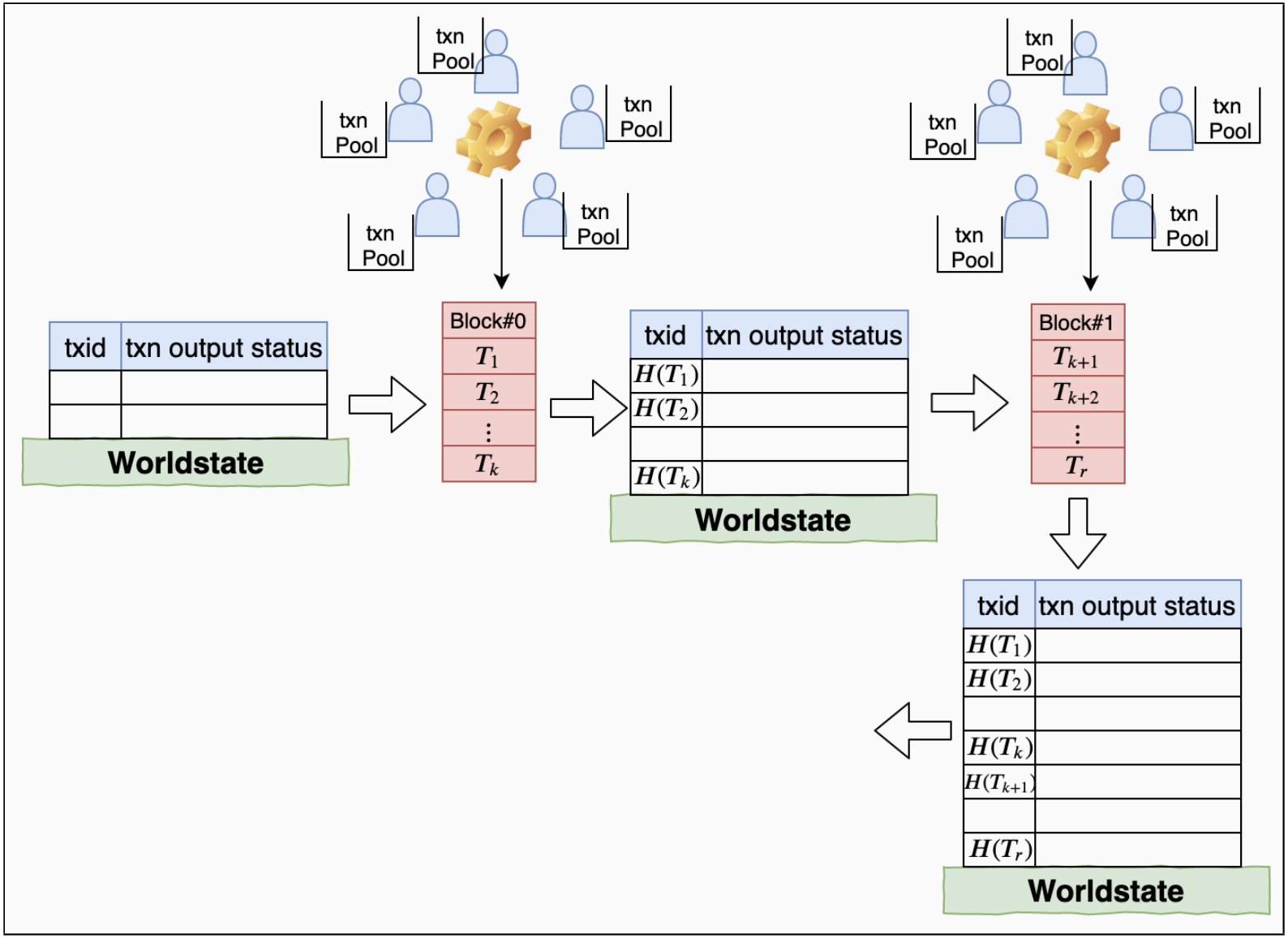
1. The traditional algorithms have a starting point (when participating nodes provide their inputs) and an ending point (when the algorithm converges to a consensus and output a single value to every participating node).
2. They do not, mostly, have any mechanism to incentivise participants to act honestly.
3. The existing algorithms are mostly deterministic.

Bitcoin's Distributed Consensus



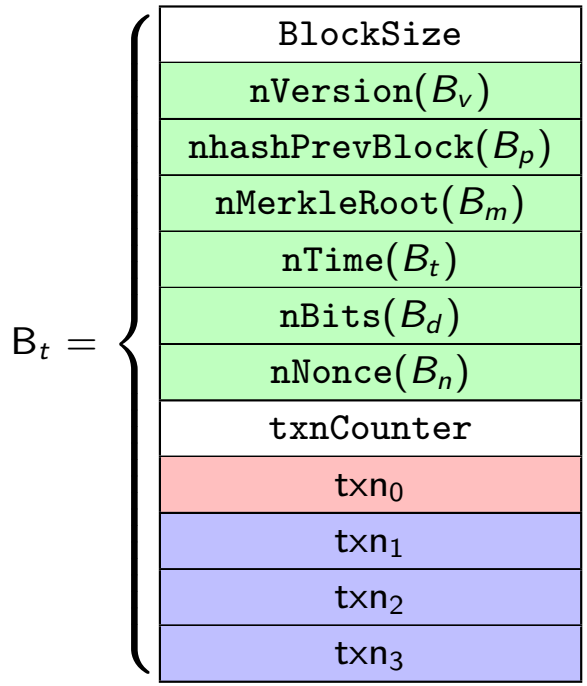
- The Bitcoin consensus mechanism does away with the notion of a specific starting point and ending point! Instead, consensus takes place over a long time, about an hour in the practice. But even at the end of that time, participants can't be certain about it
- The Bitcoin consensus incorporates a natural mechanism of incentivising the participating nodes to act honestly.

The Working of the Network



How to prepare a block? - Mining

- Assume, so far $t-1$ blocks were accepted by the network
- Now is the turn for a t-th block to be prepared and accepted by the network.
- Suppose, a node wants to prepare the tth block using three txns: $\text{txn}_1, \text{txn}_2$ and txn_3.
- The node also puts a special transactions txn_0 into the block!!

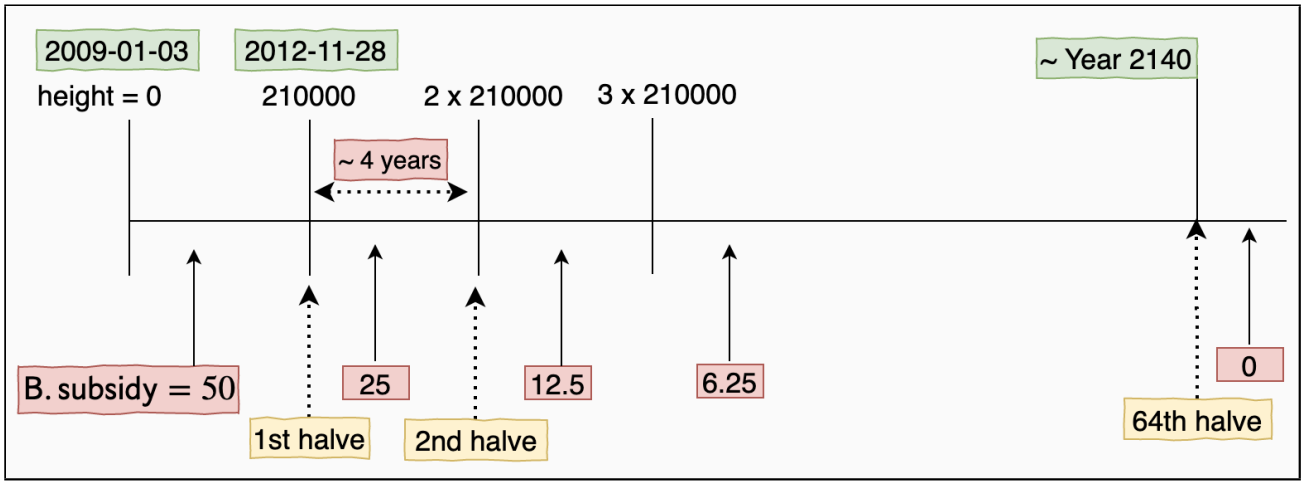


txn_0 - The Coinbase Transaction

- The first transaction in any block B is a special transaction, called a **coinbase transaction**.
  - A coinbase transaction is constructed by a miner who is preparing the block.
  - The transaction contains miner's reward for the mining effort.
  - A block reward is computed as follows:
- $\boxed{\text{B.reward}} = \boxed{\text{B.subsidy}} + \boxed{\sum_{\text{txn} \in \text{B}} \text{txn.fee}}$, where B.subsidy is computed as follows:

$$\text{B.subsidy} = \frac{50}{2^{\ell}} \text{ ฿ , where } \ell = \left\lfloor \frac{\text{height(B)}}{210000} \right\rfloor$$

  - The block B_{722784} was mined on 2022-02-11 20:52. Therefore, $\ell = \left\lfloor \frac{722784}{210000} \right\rfloor = \lfloor 3.44 \rfloor = 3$.
  - This means, $B_{722784}.\text{subsidy} = \frac{50}{2^3} = \frac{50}{8} = 6.25$ ฿



How to prepare a block? - Mining

- Assume, so far $t-1$ blocks were accepted by the network
  - This is race for tth block !!
- Suppose, a node wants to prepare the tth block using three txns: $\text{txn}_1, \text{txn}_2$ and txn_3.
- The block structure

$\mathrm{B}_t =$

BlockSize
nVersion(B_v)
nhashPrevBlock(B_p)
nMerkleRoot(B_m)
nTime(B_t)
nBits(B_d)
nNonce(B_n)
txnCounter
txn_0
txn_1
txn_2
txn_3

- BlockSize: number of bytes following up to end of block.
  - Uses 4 bytes to represent Blocksize
- **Block Header**: $\mathrm{B}_t^h = [\mathrm{B}_v \| \mathrm{B}_p \| \mathrm{B}_m \| \mathrm{B}_t \| \mathrm{B}_d \| \mathrm{B}_n]$

- nVersion(B_v): 4-byte field, specifies the version of the block.
- nhashPrevBlock(B_p): $H(\mathrm{B}_{t-1}^h)$

Notation: $H(\cdot) = \text{SHA256}(\text{SHA256}(\cdot))$

Block Preparation: Mining

$B_t =$

BlockSize
nVersion(B_v)
nhashPrevBlock(B_p)
nMerkleRoot(B_m)
nTime(B_t)
nBits(B_d)
nNonce(B_n)
txnCounter
txn_0
txn_1
txn_2
txn_3

- nMerkleRoot $= H\Big(H(\text{txid}_0 \| \text{txid}_1) \| H(\text{txid}_2 \| \text{txid}_3)\Big)$
  - And it is computed as follows

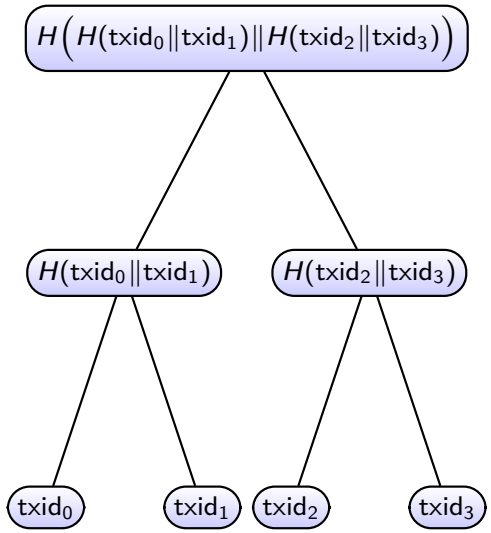


Table of Contents

1 Lecture#1 (18/01) [Course Overview, Bitcoin Introduction]

2 Lecture#2 (21/01) [Hash Functions]

3 Lecture#3 (25/01) [Random Oracle Model, Iterated Hash Construction]

4 Lecture#4 (28/01) [Hash Puzzle, Signature Scheme]

5 Lecture#5 (01/02) [Group Theory]

6 Lecture#6 (04/02) [Discrete Logarithm Problem, DSA]

7 Lecture#7 (04/02) [Elliptic Curves, ECDSA]

8 Lecture#8 (11/02) [Bitcoin Address Generation]

9 Lecture#9 (15/02) [Bitcoin Transaction and its Validation]

10 Lecture#10 (18/02) [Bitcoin Consensus]

11 Lecture#11 (22/02) [Block Mining, txn Serialization]

12 Lecture#12 (25/02) [txn Serialization, PaytoScriptHash]

13 Lecture#13 (01/03) [More lockScripts, txn-malleability]

14 Lecture#14 (04/03) [SegWit transaction]

15 Lecture#15 (15/03) [Analysis of Bitcoin's Consensus]

16 Lecture#16 (21/03) [Hyperledger Fabric]

17 Lecture#17 (25/04) [Ethereum]

18 Lecture#18 (29/03) [Ethereum Contract Accounts]

19 Lecture#19 (05/04) [Payment Channels, Coin Mixers]

20 Lecture#20 (08/04) [Coin Mixers, MixEth, TumbleBit]

21 Lecture#21 (19/04) [Tokens]

ashoka UNIVERSITY

Block Preparation: Mining

$B_t =$

BlockSize
nVersion(B_v)
nhashPrevBlock(B_p)
nMerkleRoot(B_m)
nTime(B_{nt})
nBits(B_d)
nNonce(B_n)
txnCounter
txn_0
txn_1
txn_2
txn_3

- `BlockSize`: number of bytes following up to end of block.
  - Uses 4 bytes to represent Blocksize
- **Block Header**: $B_t^h = [B_v \| B_p \| B_m \| B_t \| B_d \| B_n]$

- `nVersion`(B_v): 4-byte field, specifies the version of the block.
- `nhashPrevBlock`(B_p): $H(B_{t-1}^h)$
- `nMerkleRoot` (B_m):
- `nTime` (B_{nt}): is a timestamp in Unix time format to record the time of candidate block creation.

- `nBits` (B_d), and `nNonce` (B_n) ??

Block Preparation: Mining

$B_t =$

BlockSize
nVersion(B_v)
nhashPrevBlock(B_p)
nMerkleRoot(B_m)
nTime(B_{nt})
nBits(B_d)
nNonce(B_n)
txnCounter
txn_0
txn_1
txn_2
txn_3

- In order to prepare a valid block, a miner must solve a difficult computational problem.
  - Every miner gets the same problem
- A miner puts the encoded representation of the problem into the nBits field in the header.
- After solving the problem, a miner puts the solution into the nNonce field in the header.
  - Consequently, the block is now ready (mined successfully) as a valid block

We now discuss,

1. The problem description
2. The encoded representation of the problem
3. How difficult is the problem?
4. How is the problem constructed/determined?

Block Preparation: Mining

1. The problem description: It is the hash puzzle problem - P_T

- **Input:** $\mathrm{T} \in \{0, 1, \ldots, 2^{256} - 1\}$
- **Output:** Prepare B_t^h such that $\mathrm{H}(\mathrm{B}_t^h) < T$

- The values in the following fields in B_t^h are not in a miner's control:

$$\mathrm{B}_t^h = [\mathrm{B}_v \| \mathrm{B}_p \| \mathrm{B}_m \| \mathrm{B}_{nt} \| \mathrm{B}_d \| \mathrm{B}_n]$$

- $\mathrm{B}_m = \mathrm{MerkleRoot}(\mathrm{txn}_0[[r]], \mathrm{tx}_1, \mathrm{tx}_2, \mathrm{tx}_3)$, where $r \in \{0, 1\}^{\leq 768}$ and $\mathrm{txn}_0[[r]]$ is a function of r
- And, $\mathrm{B}_n \in \{0, 1\}^{32}$

- Therefore, **Output:** Find B_m and B_n such that $\mathrm{H}(\mathrm{B}_t^h) < T$
- Equivalently, the problem is described as follows

$$\mathrm{P}_T = \begin{cases} \textbf{Input: } X \text{ and } \mathrm{T} \in \{0, 1, \ldots, 2^{256} - 1\} \\ \textbf{Output: } \text{Find } Y \text{ such that } \mathrm{H}(X \| Y) < T \end{cases}$$

where $X = [\mathrm{B}_v \| \mathrm{B}_p \| \mathrm{B}_{nt} \| \mathrm{B}_d]$ and $Y = [\mathrm{B}_m \| \mathrm{B}_n]$

Block Preparation: Mining

2. The encoded representation of the problem.

- **Input** $T \in \{0, 1, \ldots, 2^{256} - 1\}$
- **Output** $T_{e\|c} \in \{0, 1\}^{32}$ where

$$T_{e\|c} = 0x d_1 d_2 \ldots d_7 d_8$$

and d_i's are hex numbers.

- T can be recovered from $T_{e\|c}$ as follows

$$T = c \times (256)^{(e-3)}$$

where $e = (0x d_1 d_2)_{10}$ and $c = (0x d_3 \ldots d_8)_{10}$ are decimal numbers.

- **An example**

- The first Block, i.e., B_0, was mined by "**Satoshi**" on 2009-01-03 23:45 [See - Blockchain.com explorer]
- The nBits value given in its header = 486604799
- Thus, $T_{e\|c} = \text{hex}(486604799) = 0x1d00ffff$
- This means,

$$\begin{aligned} T &= c \times (256)^{(e-3)} \\ &= (0x00ffff)_{10} \times (256)^{((0x1d)_{10}-3)} \\ &= 65535 \times (256)^{(29-3)} \\ &= 26959535291011309493156476344723991336010898738574164086137773096960 \end{aligned}$$

Block Preparation: Problem Difficulty

- The Problem P_T

  Input A hash function $\mathsf{H} : \{0,1\}^* \leftarrow \{0,1\}^{256}$; $\mathsf{T} \in \{0,1,\ldots,2^{256}-1\}$
  Output: Find $x \in \{0,1\}^*$ such that $\mathsf{H}(x) < T$

- Note that, every instance P_T of this problem comprises of T-many pre-image finding problems with respect to the underlying hash function H

- Recall: the pre-image finding problem, PF_y

$$\begin{cases} \textbf{Input: } H : \{0,1\}^* \rightarrow \{0,1\}^{256} \text{ and } y \in \{0,1\}^{256} \\ \textbf{Output: } x \in \{0,1\}^* \text{ such that } H(x) = y \end{cases}$$

- Therefore, for a $T \in \{0,1,\ldots,2^{256}-1\} = \{0,1\}^{256}$

$$\mathsf{P}_\mathsf{T} \equiv \{\mathsf{PF}_y \mid 0 \leq y \leq T\}$$

- If H is a secure hash function (behaves like a random oracle model), then the best algorithm to solve H_T would be

```
While H(x) ∉ {0, 1, ..., T − 1}
    x ←$ {0,1}*
    Compute H(x)
Output x
```

- The expected number of loops required before the above algorithm terminates with a pre-image

$$\approx \frac{2^{256}}{T}$$

- If $\lceil \log_2(T) \rceil = k$, i.e., $T \approx 2^k$, then the binary representation of any number which is $< T$ must have at lest $256 - k$ many leading 0's
- That is, for the correct solution x, the binary representation of $H(x)$ must have $256 - k$ many leading 0's

---

- *Miners* in Bitcoin currently solves a P_T, roughly every 10 minutes, where $T \approx 2^{180}$

  Therefore, the expected number of hash computations required $\approx \frac{2^{256}}{2^{180}} \approx 2^{76}$!!

Block Preparation: Mining

An Example

- The Block at height 670000, i.e., B_{670000} was mined on 2021-02-10 18:39 [See Blockchain.com explorer]
- The nBits value given in its header = 386736569
- Thus, $T_{e\|c} = \text{hex}(386736569) = 0x170d21b9$
- This means,

$$
\begin{aligned}
T &= c \times (256)^{(e-3)} \\
&= (0x0d21b9)_{10} \times (256)^{((0x17)_{10}-3)} \\
&= 860601 \times (256)^{(23-3)} \\
&= 1257769770588612382309009370720465882998915202417688576
\end{aligned}
$$

- As $\log_2(T) \approx 180$, i.e., $T \approx 2^{180}$, the expected number of hash computations required before one finds a solution to P_T is

$$\approx \frac{2^{256}}{T} \approx \frac{2^{256}}{2^{180}} \approx 2^{76}$$

Block Preparation: Mining

- **Fact** A characterisation of $H(X \| Y)$ when $H(X \| Y) < T$ and $T \in \{0,1\}^{256}$
  - If $\lceil \log_2(T) \rceil = k$, i.e., $T \approx 2^k$, then the binary representation of any number which is $< T$ must have at lest $256 - k$ many leading 0's
  - That is, for the correct solution Y, the binary representation of $H(X \| Y)$ must have $256 - k$ many leading 0's

---

- The above observation implies that, the problem P_T can also be represented as follows:

$$\mathsf{P}_{n\text{-bits}} = \begin{cases} \textbf{Input: } X \text{ and } n \in \{1, \ldots, 256\} \\ \textbf{Output: } \text{Find } Y \text{ such that the binary representation of} \\ \mathrm{H}(X \| Y) \text{ must have } n \text{ leading } 0's \end{cases}$$

- Clearly, for a $n \in \{1, \ldots, 256\}$, if you set $k = 256 - n$, then

$$\mathsf{P}_{n\text{-bits}} \equiv \mathsf{P}_\mathsf{T} \text{ , where } \mathsf{T} = 2^k$$

  - Indeed, binary representation of $H(X \| Y)$ having n many leading 0's implies

$$\begin{aligned} & H(X \| Y) < 2^{256-n} \\ \implies & H(X \| Y) < 2^k \\ \implies & H(X \| Y) < T \end{aligned}$$

Target Adjustment

- **Average Mining Time**
- Bitcoin protocol specifies that the average time required to mine a valid block should be about 10 mins
  - How does network decides on the difficulty target to ensure the above ?
  - A solution (Assuming average HashRate = # hashes/second of the network, say $\mathcal{H}$, is know):
  - Set the difficulty target T by solving the following

$$\frac{2^{256}}{T} = 600 \times \mathcal{H}$$

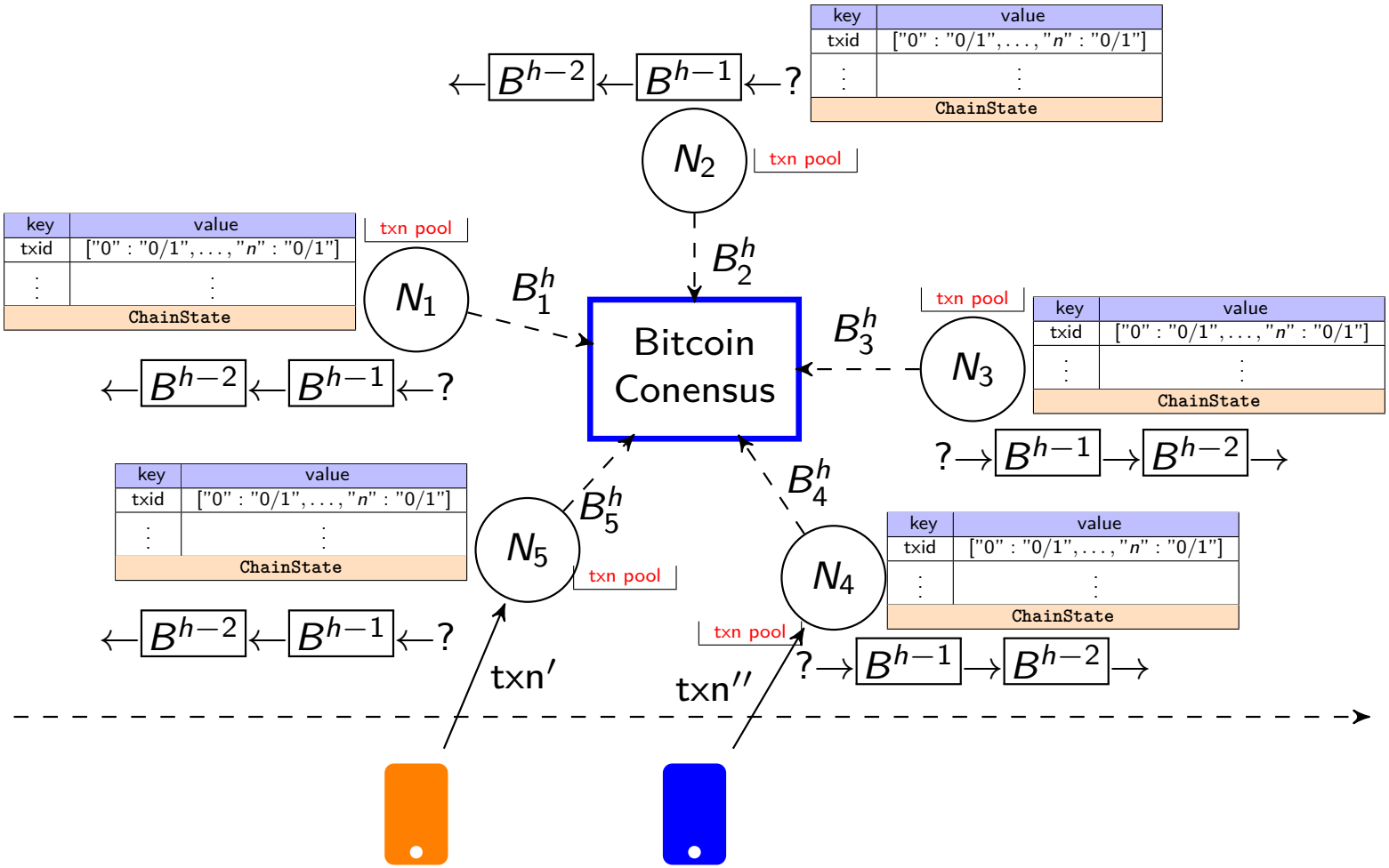
  - But there is no direct mean available to compute $\mathcal{H}$

- How it is done currently?
  - Measure the time which was spent in finding the previous 2016 blocks - say K minutes
  - Ideally, this time should be $2016 \times 10 = 20160$ minutes.
  - If $K < 20160$, decrease the target
  - If $K > 20160$, increase the target
- The formula used for increase/decrease is

$$T_{\text{new}} = T_{\text{old}} \times \frac{K}{20160}$$

- The target is updated every 2016 blocks (every 2 weeks) by all the nodes independently.

The Bitcoin Architecture



1. Bitcoin Network: Miners - $N_1, \ldots, N_5$
2. Wallets:
3. Distributed Ledger: `Blockchain` + `ChainState`

Revisiting Bitcoin txns: Serialization Structure

- An example of a raw Bitcoin transaction (in hex)

```
01000000011935b41d12936df99d322ac8972b74ecff7b79408bbccaf1b2eb8015228beac8000000006b
483045022100921fc36b911094280f07d8504a80fbab9b823a25f102e2bc69b14bcd369dfc7902200d070
67d47f040e724b556e5bc3061af132d5a47bd96e901429d53c41e0f8cca012102152e2bb5b273561ece7
bbe8b1df51a4c44f5ab0bc940c105045e2cc77e618044ffffffff0240420f00000000001976a9145fb1af31ed
d2aa5a2bbaa24f6043d6ec31f7e63288ac20da3c00000000001976a914efec6de6c253e657a9d5506a78
ee48d89762fb3188ac00000000
```

txn Serialization Structure

txn	Input		Unspent txn Output (UTXO)	
	txid	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{B_{123} \to C_{12}}$	$H(T_{A_1 \to B_1})$	$\langle \text{Sign}_{sk_{B_1}}(T_{B_{123} \to C_{12}}) \rangle \ \langle pk_{B_1} \rangle$	x_1 ฿	OP_DUP OP_HASH160 $\langle \text{HASH160}(pk_{C_1}) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
	$H(T_{A_2 \to B_2})$	$\langle \text{Sign}_{sk_{B_2}}(T_{B_{123} \to C_{12}}) \rangle \ \langle pk_{B_2} \rangle$		
	$H(T_{A_3 \to B_3})$	$\langle \text{Sign}_{sk_{B_3}}(T_{B_{123} \to C_{12}}) \rangle \ \langle pk_{B_3} \rangle$	x_2 ฿	OP_DUP OP_HASH160 $\langle \text{HASH160}(pk_{C_2}) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY

Field Type	Size (bytes)
nVersion	4
Number of Inputs N	1/3/5/9
Input_0	
⋮	
Input_{N-1}	
Number of Outputs M	1/3/5/9
Output_0	
⋮	
Output_{M-1}	
nLockTime	4

txn **Structure**

Field Type	Size (bytes)
hash	32
n	4
ScriptSigLen	
ScriptSig	
nSequence	4

txn **Input Structure**

Field Type	Size (bytes)
nValue	
ScriptPubKeyLen	
ScriptPubKey	

txn **Output Structure**

Var Int Encoding

- Consider the following encoding scheme Π for representing integers in range 0 to $2^{64} - 1$.
- For an integer x, we use $[x]^k$ to denote k-bit unsigned representation of x.

Integer	Encoding
$0 \leq x \leq 252$	$\Pi(x) = [x]^8$
$253 \leq x \leq 2^{16} - 1$	$\Pi(x) = [253]^8 \| [x]^{16}$
$2^{16} \leq x \leq 2^{32} - 1$	$\Pi(x) = [254]^8 \| [x]^{32}$
$2^{32} \leq x \leq 2^{64} - 1$	$\Pi(x) = [255]^8 \| [x]^{64}$

An example

- For some positive integer n, numbers $x_1, \ldots, x_n$ are chosen in $\{0, \ldots, 2^{64-1}\}$, and you are given $\Pi(x_1) \| \cdots \| \Pi(x_n)$ as below:

1110101111111101111111101111111111111111111101011111111011111111101111111111111111111111111111111111

- Determine k and $x_1, \ldots, x_k$

Timelock: nLocktime

- Timelocks are restrictions on **transactions or outputs** that only allow spending after a point in time, i.e., locking funds to a date in the future.
- nLocktime - a txn-level timelock
  - A nLocktime value defines the earliest time that a txn is ready for execution by the network. It is set to "0" in most txn's to indicate immediate execution as is the following txn

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{B\to\{C_1,C_2\}}$	$H(T_{A_1\to B})$	n_1	$\langle \text{Sign}_{sk_B}(T_{B\to\{C_1,C_2\}})\rangle\ \langle pk_B\rangle$	x_1 ฿	OP_DUP OP_HASH160 $\langle$HASH160(pk_{C_1})$\rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
	$H(T_{A_2\to B})$	n_2	$\langle \text{Sign}_{sk_B}(T_{B\to\{C_1,C_2\}})\rangle\ \langle pk_B\rangle$		
	$H(T_{A_3\to B})$	n_3	$\langle \text{Sign}_{sk_B}(T_{B\to\{C_1,C_2\}})\rangle\ \langle pk_B\rangle$	x_2 ฿	OP_DUP OP_HASH160 $\langle$HASH160(pk_{C_2})$\rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
nLockTime = 0					

- An nLocktime value is encoded using a block-height/Unix-time.
  - It is interpreted as block-height if nLockTime $< 5 \times 10^8$
  - It is Interpreted as Unix-time if nLockTime $\geq 5 \times 10^8$
- The following txn $T_{C_1\to D}$ with nLockTime = 674000 will not be picked up for execution in a valid block whose height is less than 674000.
- At the time of writing $T_{C_1\to D}$, the current time is 2021-03-01 07:30 and the block minded by the network is of height 672638

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{C_1\to D}$	$H(T_{B\to\{C_1,C_2\}})$	0	$\langle \text{Sign}_{sk_{C_1}}(T_{C_1\to D})\rangle\ \langle pk_{C_1}\rangle$	x_1 ฿	OP_DUP OP_HASH160 $\langle$HASH160(pk_D)$\rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
nLockTime = 674000					

Timelock: nLocktime

- What about this txn?

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{C_1 \to D}$	$H(T_{B \to \{C_1, C_2\}})$	0	$\langle \text{Sign}_{\text{sk}_{C_1}}(T_{C_1 \to D}) \rangle \ \langle \text{pk}_{C_1} \rangle$	x_1 ฿	$\begin{cases} \text{OP\_DUP OP\_HASH160} \ \langle \text{HASH160}(\text{pk}_D) \rangle \\ \text{OP\_EQUALVERIFY OP\_CHECKSIGVERIFY} \end{cases}$
nLockTime = 1615102200					

- As nLockTime $= 1615102200 > 5 \times 10^8$, its value must be interpreted as Unix timestamp.
- One may check that 1615102200 represents "Sun Mar 07 2021 13:00:00 GMT+0530 (India Standard Time) "
- As per the rule, $T_{C_1 \to D}$ will not be picked up for execution in a valid block unless the median-time-past of this block is beyond `Sun Mar 07 2021 13:00:00 GMT+0530`

Median-time-past The median-time-past of a block at height h is the median of $\{\texttt{nTime}^{block}_{h-11}, \texttt{nTime}^{block}_{h-10}, \ldots, \texttt{nTime}^{block}_{h-1}\}$, where $\texttt{nTime}^{block}_{j}$ denotes the `nTime` of the block in the blockchain at height j.

- The use of nLocktime is equivalent to postdating a paper check.
- Indeed, after $T_{C_1 \to D}$ is prepared by C_1 and is handed over to D, it resembles like a post dated check
- D can relay $T_{C_1 \to D}$ to the network only at a future point of time in order to receive the money.

Table of Contents

1 Lecture#1 (18/01) [Course Overview, Bitcoin Introduction]

2 Lecture#2 (21/01) [Hash Functions]

3 Lecture#3 (25/01) [Random Oracle Model, Iterated Hash Construction]

4 Lecture#4 (28/01) [Hash Puzzle, Signature Scheme]

5 Lecture#5 (01/02) [Group Theory]

6 Lecture#6 (04/02) [Discrete Logarithm Problem, DSA]

7 Lecture#7 (04/02) [Elliptic Curves, ECDSA]

8 Lecture#8 (11/02) [Bitcoin Address Generation]

9 Lecture#9 (15/02) [Bitcoin Transaction and its Validation]

10 Lecture#10 (18/02) [Bitcoin Consensus]

11 Lecture#11 (22/02) [Block Mining, txn Serialization]

12 Lecture#12 (25/02) [txn Serialization, PaytoScriptHash]

13 Lecture#13 (01/03) [More lockScripts, txn-malleability]

14 Lecture#14 (04/03) [SegWit transaction]

15 Lecture#15 (15/03) [Analysis of Bitcoin's Consensus]

16 Lecture#16 (21/03) [Hyperledger Fabric]

17 Lecture#17 (25/04) [Ethereum]

18 Lecture#18 (29/03) [Ethereum Contract Accounts]

19 Lecture#19 (05/04) [Payment Channels, Coin Mixers]

20 Lecture#20 (08/04) [Coin Mixers, MixEth, TumbleBit]

21 Lecture#21 (19/04) [Tokens]

ashoka UNIVERSITY

txn Input Structure

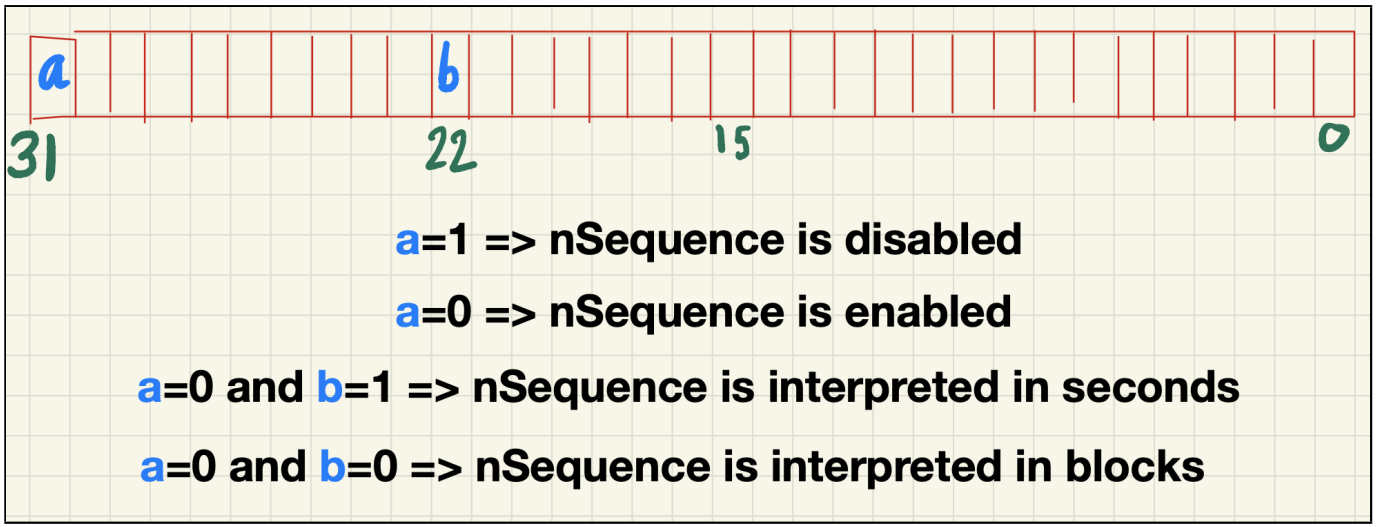
Field Type	Size
hash	32
n	4
ScriptSigLen	
ScriptSig	
nSequence	4

txn **Input Structure**

- `hash` Contains the transaction identifier (TXID) of an earlier `txn` containing the UTXO that this input tries to unlock.
- `n` Index of the UTXO in the earlier txn
- `scriptSigLen` contains `VarInt` encoding of the length of `scriptSig`
- `scriptSig` contains the unlock script - `scriptSig`
- nSequence ??

Relative Timelock: nSquence

- nLocktime implements an absolute timelock in that it specifies an absolute future time.
- nSequence implements a relative timelock in that it specifies, as a condition of spending an output, an elapsed time from the confirmation of the output in the blockchain
- Transaction inputs with nSequence enabled are interpreted as having a relative timelock. Such a transaction is only valid once the input has aged by the relative timelock amount.
- nSequence is applied to each **input** of a transaction.
- Like nLocktime, it also possible to disable nSequence
- The nSequence value is specified in either blocks or seconds, but in a slightly different format than we saw used in nLocktime.



Relative Timelock: nSquence

- **An Example**

txn	Input				Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	nSequence	Amount	scriptPubkey
$T_{C_1 \to D}$	$H(T_{B \to \{C_1, C_2\}})$	0	$\langle \text{Sign}_{sk_{C_1}}(T_{C_1 \to D}) \rangle$ $\langle pk_{C_1} \rangle$	0x00000014	x_1 ฿	$\begin{cases} \text{OP\_DUP OP\_HASH160 } \langle \text{HASH160}(pk_D) \rangle \\ \text{OP\_EQUALVERIFY OP\_CHECKSIGVERIFY} \end{cases}$

- For 0x00400014, $a = b = 0$.
- nSequence is interpreted as 20 blocks.

- This means, $T_{C_1 \to D}$ is ready for execution when at least 20 blocks have elapsed from the time the reference txn $T_{B \to \{C_1, C_2\}}$ was accepted into a valid block by the network!

txn Output Structure

Field Type	Size
nValue	
ScriptPubKeyLen	
ScriptPubKey	

txn **Output Structure**

`nValue` contains the number of satoshis as UTXO.

Coinbase txn Serialization

Field Type	Size (bytes)
nVersion	4
Number of Inputs = 1	1
Dummy Input	
Number of Outputs M	1/3/5/9
Output_0	
⋮	
Output_{M-1}	
nLockTime	4

Coinbase txn Structure

Field Type	Size (bytes)
hash	32
n	4
ScriptSigLen	
ScriptSig	
nSequence	4

Coinbase txn Input Structure

nValue	
ScriptPubKeyLen	
ScriptPubKey	

Coinbase txn Output Structure

Coinbase txn Serialization

Field Type	Size (bytes)
hash	32
n	4
ScriptSigLen	1/3/5/9
ScriptSig	
nSequence	4

- `hash` Set to 0x000 . . . 00
- `n` Set to 0xFFFFFFFF
- `scriptSigLen` contains the length of `scriptSig`
- `scriptSig` Set to X||Y, where $|X| = 4$ bytes and $|Y| \leq 96$ bytes. X encodes the height of the block. The first byte of X is 0x03 - signaling next 3 bytes encodes the block height
- `nSequence` is ignored

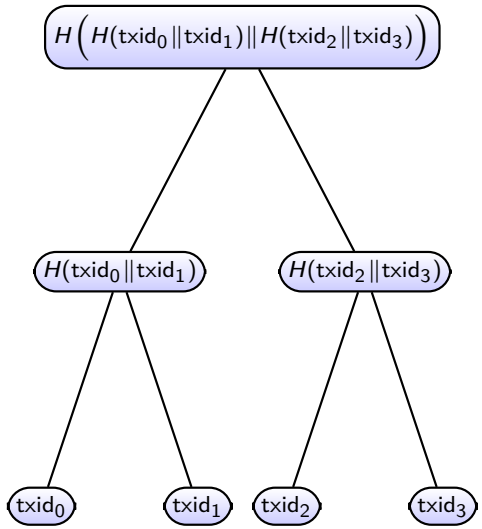
Field Type	Size (bytes)
nValue	8
ScriptPubKeyLen	1/3/5/9
ScriptPubKey	

- `nValue` contains the number of satoshis as UTXO. Should be less than or equal to block reward
- Coinbase UTXO cannot be spent unless it has received 101 confirmations.

$B_t =$

BlockSize
nVersion(B_v)
nhashPrevBlock(B_p)
nMerkleRoot(B_m)
nTime(B_t)
nBits(B_d)
nNonce(B_n)
txnCounter
txn_0
txn_1
txn_2
txn_3

- nMerkleRoot $= H\Big(H(\text{txid}_0 \| \text{txid}_1) \| H(\text{txid}_2 \| \text{txid}_3)\Big)$
  - And it is computed as follows



Signature Generation

- The P2PKH `scriptSig` requires a signature computation on the transaction.
  - The signature is computed on a *part* of the transaction.
  - Give that we now know the exact serialized structure of a transaction, the exact part of it on which the signature is computed is determined as follows:

- txn

nVersion
0x02
hash0
n0
scriptSigLen0
scriptSig0
nSequence0
hash1
n1
scriptSigLen1
scriptSig1
nSequence1
0x01
nValue0
scriptPubkeyLen0
scriptPubkey0
nLockTime

  - Consider, `scriptSig0` : $\langle \text{signature} \rangle \ \langle \text{pk} \rangle$
  - The $\langle \text{signature} \rangle$ is generated on a well defined message
  - The message is derived from the txn
  - Denote the message by: $\text{m}_{\text{txn},k}$ where k signals that it is for the input number k.
- The following standards dictate the selection of $\text{m}_{\text{txn},k}$

nHashType	Value
SIGHASH_ALL	0x00000001
SIGHASH_NONE	0x00000002
SIGHASH_SINGLE	0x00000003
SIGHASH_ANYONECANPAY	0x00000080

- $\text{signature} = \text{sign}_{\text{sk}}(\text{m}_{\text{txn},k}) \| \text{lsb}(\text{nHashType})$

- $\text{m}_{\text{txn},0}$ following `SIGHASH_ALL` type

nVersion
0x02
hash0
n0
prevScriptPubKeyLen0
prevScriptPubKey0
nSequence0
hash1
n1
0x00
nSequence1
0x01
nValue0
scriptPubkeyLen0
scriptPubkey0
nLockTime
nHashType

LockScript : Different Types

1. P2PKH
2. P2PK
3. Multi-Sig
4. P2SH
5. OP_RETURN

---

2. P2PK

- P2PK is a simpler form of P2PKH
- P2PK lock script

```
<Public-Key> OP_CHECKSIG
```

- The corresponding unlock script

```
<sig>
```

Multi-Sig

3. A Multi-Sig lock script enables various forms of joint ownership of an UTXO
   - A m-of-n multi-sig allows an UTXO to be locked to n users such that, later any m out of n users together can spend the same
   - m-of-n multisig lock script

     `M <PK_1> <PK_2> --- <PK_N> N OP_CHECKMULTISIG`
   - The corresponding unlock script

     `OP_0 <sig_i1> <sig_i2> --- <sig_iM>`

     where $1 \leq$ i1 $<$ i2 $< \cdots <$ iM $\leq N$.

- The `OP_0` is there to fix a bug in `OP_CHECKMULTISIG` implementation
- Example: for `2 <PK_1> <PK_2> <PK_3> 3 OP_CHECKMULTISIG`
  - Correct: `OP_0 <sig_1> <sig_3>`
  - Incorrect: `OP_0 <sig_2> <sig_1>`
- Example:

txn	Input		Output	
	txid	scriptSig	UTXO	scriptPubkey
T_1			x ฿	m $\langle \text{PK}_{A_1} \rangle \cdots \langle \text{PK}_{A_n} \rangle$ n OP_CHECKMULTISIGVERIFY
T_2	$H(T_1)$	$\langle \text{Sign}_{\text{sk}_{A_{i_1}}}(T_2) \rangle \cdots \langle \text{Sign}_{\text{sk}_{A_{i_m}}}(T_2) \rangle$		

- The T_2 transaction is validated by the network participants as follows: check if, $\text{Verify}_{\text{pk}_{A_{i_j}}}(\langle \text{Sign}_{\text{sk}_{A_{i_j}}}(T_2,) \rangle, T_2) = \text{valid}$, for all j in $1 \leq j \leq m$.

A Fair Exchange Contract using Multi-Sig

- Suppose, Alice wishes to purchase online from Bob. Assume, Bob accepts payment in bitcoins.
- But, they don't trust each other!!
  - Alice get's the product and keep her payment
  - Bob get's the payment but doesn't send the product
- Solution#1: Alice locks the payment UTXO to the following lock script

```
2 PK_Alice PK_Bob 2 OP_CHECKMULTISIG
```

- Pros
- Alice can't get the product and keep her payment.
- Bob won't get paid unless Alice gets the product
- Cons
- Not useful if there is a dispute - such as product return
- Bob doesn't ship the product, and thus Alice money is lost (not to Bob)
- Solution#2: An improved solution
- Alice and Bob invites an arbitrator - Max
- Alice locks the payment UTXO to the following lock script

```
2 Alice_PK Bob_PK Max_PK 3 OP_CHECKMULTISIG
```

- Max can arbitrate and decides who gets the UTXO if there is a dispute.

Pay to Script Hash

4. A P2SH LScript contains a hash of an another arbitrary-type lock script - called **redeem script**
   - Consider a Receiver who is going to get ฿ from a Sender
   - Receiver creates a redeem script (any type) - LS_{redeem}
   - Hashes it: `h_redeem` $= HASH160(LS_{redeem})$
   - Provides h_{redeem} to a Sender.

   - Sender creates a P2SH lock script

     `OP_HASH160 <h_redeem> OP_EQUAL`

   - Later, when the Receiver wants to consume this UTXO, it provides the following two scripts
     - LS_{redeem}
     - ULS_{redeem}

● ____ **Validation**

- Two scripts are combined and executed in two stages:
  - LS_{redeem} || `OP_HASH160 <h_redeem> OP_EQUAL`
  - ULS_{redeem} || LS_{redeem}

P2SH Addresses

- **P2PKH Address**

Input: Public-key pk
Process:
$H1 \leftarrow \text{SHA256}(\text{pk.uncompressed})$
$H2 \leftarrow \text{RIPEMD160}(H1)$
$H3 \leftarrow A||H_2$ where
$$A = \begin{cases} 0\text{x}00 & \text{mainnet} \\ 0\text{x}6\text{f} & \text{testnet} \end{cases}$$
$H4 \leftarrow \text{SHA256}\,(\text{SHA256}(H3))$
$H5 \leftarrow H3||H4[1:4] = A||H2||H4[1:4]$
where $H4[1:4]$ denotes the first 4 bytes of $H4$.
Bitcoin Address $\leftarrow$ Base58Encoding($H5$)

Address starts with 1 (Main)

Address starts with m/n (Test)

- **P2SH Address**

Input: $\text{LS}_{\text{redeem}}$
Process:
$H1 \leftarrow \text{SHA256}(\text{LS}_{\text{redeem}})$
$H2 \leftarrow \text{RIPEMD160}(H1)$
$H3 \leftarrow A||H_2$ where
$$A = \begin{cases} 0\text{x}05 & \text{mainet} \\ 0\text{x}\text{C}4 & \text{testnet} \end{cases}$$
$H4 \leftarrow \text{SHA256}\,(\text{SHA256}(H3))$
$H5 \leftarrow H3||H4[1:4] = A||H2||H4[1:4]$
where $H4[1:4]$ denotes the first 4 bytes of $H4$.
P2PSH Address $\leftarrow$ Base58Encoding($H5$)

Address starts with 3 (Main)

Address starts with 2 (Test)

ashoka UNIVERSITY

Benifits of P2SH Address

- A Receiver might use a complex script to receive payment
- Instead of a asking a Sender to use this complex script as lock script
  - Obtain a P2SH Address
  - give it the Sender
- Sender extracts Redeem Script hash `h_redeem` $\leftarrow$ P2SH address
- Creates the lock script as

```
OP_HASH160 <h_redeem> OP_EQUAL
```

- Complex scripts are replaced by shorter hash digest in the txn output
- Script can be encoded as address - sender's wallet don't need complex engineering to implement P2SH
- Shifts the burden of constructing the script to the recipient and not the sender
- Shifts the txn fee cost of a long txn from the sender to the receiver
- Receiver has to include the long redeem script to spend it

Pay to Script Hash: The flow

- Suppose, a user D requests user C_1 to transfer ฿, locking it using a non-standard type lockscript !!
  - Step-(1): The user D prepares the following
    - Prepare a lockscript $\text{LS}_{\text{redeem}}$
    - Prepare the corresponding $\text{ULS}_{\text{redeem}}$
    - Prepare $\text{P2SH}(\text{LS}_{\text{redeem}})$
    - Finally send $\text{P2SH}(\text{LS}_{\text{redeem}})$ to C_1
  - Step-(2): C_1 extracts $\text{h}_{\text{redeem}} = \text{HASH160}(\text{LS}_{\text{redeem}})$ from $\text{P2SH}(\text{LS}_{\text{redeem}})$.

    It then propose the following transaction

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{C_1 \to D}$	$H(T_{B \to \{C_1, C_2\}})$	0	$\langle \text{Sign}_{\text{sk}_{C_1}} (m_{T_{C_1 \to D}, 0}) \rangle \langle \text{pk}_{C_1} \rangle$	x_1 ฿	OP_HASH160 $\langle \text{h}_{\text{redeem}} \rangle$ OP_EQUALVERIFY

- After $T_{C_1 \to D}$ is accepted by the network, the user D can spend the x_1 ฿ later by pushing the following transaction to the network

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{D \to E}$	$H(T_{C_1 \to D})$	0	$\text{ULS}_{\text{redeem}}$ $\text{LS}_{\text{redeem}}$	x_2 ฿	OP_DUP OP_HASH160 $\langle \text{HASH160}(\text{pk}_E) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY

5 OP_RETURN

- A data output lock script can be used to store data on the blockchain.
  - Lock script of this type

    `OP_RETURN <data>`

    where `data` is limited to 40 bytes

  - The `data` is mostly hash digest (32-bytes) of arbitrary data
  - General structure of `data`= some prefix $\|$ hash digest
  - prefix identifies the application
  - For example: prefix = 0x444f4350524f4f46 for application `https://proofofexistence.com/`

  - No unlock script correspond to `OP_RETURN`
  - UTXO is set to 0
  - Unlocking script containing `OP_RETURN` is considered invalid
  - One `OP_RETURN` as output per txn

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
T_{C_1}	$H(T_{B\to\{C_1,C_2\}})$	n	$\langle \text{Sign}_{sk_{C_1}}(T_{C_1})\rangle\ \langle pk_{C_1}\rangle$	$x_1 = 0$	OP_RETURN $\langle$data$\rangle$

Table of Contents

1 Lecture#1 (18/01) [Course Overview, Bitcoin Introduction]

2 Lecture#2 (21/01) [Hash Functions]

3 Lecture#3 (25/01) [Random Oracle Model, Iterated Hash Construction]

4 Lecture#4 (28/01) [Hash Puzzle, Signature Scheme]

5 Lecture#5 (01/02) [Group Theory]

6 Lecture#6 (04/02) [Discrete Logarithm Problem, DSA]

7 Lecture#7 (04/02) [Elliptic Curves, ECDSA]

8 Lecture#8 (11/02) [Bitcoin Address Generation]

9 Lecture#9 (15/02) [Bitcoin Transaction and its Validation]

10 Lecture#10 (18/02) [Bitcoin Consensus]

11 Lecture#11 (22/02) [Block Mining, txn Serialization]

12 Lecture#12 (25/02) [txn Serialization, PaytoScriptHash]

13 Lecture#13 (01/03) [More lockScripts, txn-malleability]

14 Lecture#14 (04/03) [SegWit transaction]

15 Lecture#15 (15/03) [Analysis of Bitcoin's Consensus]

16 Lecture#16 (21/03) [Hyperledger Fabric]

17 Lecture#17 (25/04) [Ethereum]

18 Lecture#18 (29/03) [Ethereum Contract Accounts]

19 Lecture#19 (05/04) [Payment Channels, Coin Mixers]

20 Lecture#20 (08/04) [Coin Mixers, MixEth, TumbleBit]

21 Lecture#21 (19/04) [Tokens]

ashoka UNIVERSITY

nLocktime Limitations

- nLocktime has the limitation that while it makes it possible to spend some outputs in the future, it does not make it impossible to spend them until that time.
  - Suppose, a user C_1 gives a payment transaction $T_{C_1 \to D}$ to D by setting nLocktime to 2 weeks.

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{C_1 \to D}$	$H(T_{B \to C_1})$	0	$\langle \text{Sign}_{sk_{C_1}}(T_{C_1 \to D}) \rangle \ \langle pk_{C_1} \rangle$	x_1 ฿	$\begin{cases} \text{OP\_DUP OP\_HASH160 } \langle \text{HASH160}(pk_D) \rangle \\ \text{OP\_EQUALVERIFY OP\_CHECKSIGVERIFY} \end{cases}$
nLockTime = **Current Time + Two Weeks**					

  - However, there are a few issues:

1. D cannot submit the $T_{C_1 \to D}$ to the network to redeem the money until 2 weeks have elapsed. D may submit the transaction after 2 weeks
2. C_1 can create another transaction, **double-spending** the same inputs with nLocktime = 0. Thus C_1 can spend the same UTXO (that $T_{C_1 \to D}$ is going to spend after 2 weeks) before the 2 weeks have elapsed.

  - Thus D has no guarantee that C_1 won't do some thing like the following:

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{C_1 \to E}$	$H(T_{B \to C_1})$	0	$\langle \text{Sign}_{sk_{C_1}}(T_{C_1 \to E}) \rangle \ \langle pk_{C_1} \rangle$	x_1 ฿	$\begin{cases} \text{OP\_DUP OP\_HASH160 } \langle \text{HASH160}(pk_E) \rangle \\ \text{OP\_EQUALVERIFY OP\_CHECKSIGVERIFY} \end{cases}$
nLockTime = 0					

Check Lock Time Verify (CLTV)

- Thus, the only guarantee that nLocktime provides is that D will not be able to redeem it before 2 weeks have elapsed. There is no guarantee that D will get the mondey
- To achieve such a guarantee, the timelock restriction must be placed on the reference UTXO itself and be part of the locking script!!
- This is achieved by the following timelock, called "**Check Lock Time Verify (CLTV)**" _____ **The Syntax**: C_1 prepares the following transaction and immediately sends it to the network for execution.

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{C_1 \to D}$	$H(T_{B \to C_1})$	0	$\langle \text{Sign}_{sk_{C_1}}(T_{C_1 \to D}) \rangle \langle pk_{C_1} \rangle$	x_1 ฿	$\langle tw \rangle$ OP_CHECKLOCKTIMEVERIFY OP_DROP OP_DUP OP_HASH160 $\langle \text{HASH160}(pk_D) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
nLockTime = 0					

- where tw = now + two weeks (write it in two way, using block heights or Unix timestamps)
- If h is the current block height, put tw $= h + 2016$ as 2016 roughly translates to two weeks.
- After $T_{C_1 \to D}$ is accepted by the network, D may prepares and submits the following txn at an appropriate time in future:

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{D \to F}$	$H(T_{C_1 \to D\}})$	0	$\langle \text{Sign}_{sk_{C_1}}(T_{D \to F}) \rangle \langle pk_{C_1} \rangle$	x_1 ฿	OP_DUP OP_HASH160 $\langle \text{HASH160}(pk_F) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
nLockTime = tw_1					

- $T_{D \to F}$ is validated as follows. It is considered invalid if $tw_1 <$ tw.
- OP_CHECKLOCKTIMEVERIFY in LScript for $T_{C_1 \to D}$ checks the above condition.
- If $tw_1 >$ tw, the script execution continues. Otherwise, it halts and $T_{D \to F}$ deemed invalid.

Check Lock Time Verify (CLTV)

- After execution, if OP_CHECKLOCKTIMEVERIFY is satisfied, the $\langle \text{tw} \rangle$ that preceded it remains as the top item on the stack.
- It is dropped, with OP_DROP, for correct execution of rest of the script.

- Therefore, with the above mechanism, C_1 can no longer **double-spend** the money as $T_{C_1 \to D}$ is immediately accepted by the network by spending the UTXO of $T_{B \to C_1}$.
- Also, D cannot get the money before the two weeks - for $T_{D \to F}$ to be valid, D is forced to set nLocktime to two weeks !!

Script with IF/ELSE

- Suppose C_1 has paid D such that a timelock two weeks has been enabled before D can redeem the money.
  - What if after a week both C_1 and D come to an agreement that it is okay for D to get the money immediately.
  - But our method doesn't allow that - D is now forced to wait unnecessarily !!
  - _____ Here is a way out!
  - The Bitcoin Script supports composing conditions under "IF" and "ELSE".
  - C_1 can pay D using the following transaction:

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{C_1 \to D}$	$H(T_{B \to C_1})$	0	$\langle \text{Sign}_{sk_{C_1}}(T_{C_1 \to D}) \rangle \langle pk_{C_1} \rangle$	x_1 ฿	OP_IF 2 $\langle pk_{C_1} \rangle$ $\langle pk_D \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ELSE $\langle tw \rangle$ OP_CHECKLOCKTIMEVERIFY OP_DROP OP_DUP OP_HASH160 $\langle \text{HASH160}(pk_D) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY OP_ENDIF
nLockTime = 0					

Hash-Locked

- Hash-Locked :
  - This type of LScript allows us to lock Bitcoins by specifying a value such that any body, at a later point, can redeem the coins by providing a pre-image of the value under HASH160.
  - In the following example, assume HASH160$(k) = h$.

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
T_{C_1}	$H(T_{B\to C_1})$	0	$\langle \text{Sign}_{sk_{C_1}}(T_{C_1})\rangle\ \langle pk_{C_1}\rangle$	x_1 ฿	OP_HASH160 $\langle h\rangle$ OP_EQUALVERIFY
nLockTime = 0					

  - A user D will be able to redeem the above provided it has a pre-image of h

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{D\to F}$	$H(T_{C_1})$	0	$\langle k\rangle$	x_1 ฿	OP_DUP OP_HASH160 $\langle \text{HASH160}(pk_F)\rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
nLockTime = 0					

ECDSA Singnature Malleability

- **Fact** If (c_1, c_2) is an ECDSA signature, then $(c_1, q - c_2)$ is also a valid ECDSA signature.

- **secp256k1 curve parameters**
  - $p : 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$
  - $E : y^2 = x^3 + 7$, defined over $\mathbb{Z}_p$
  - $G = (x, y) \in \mathcal{G}_E$ is the base point, where $x =$

    79be667e f9dcbbac 55a06295 ce870b07
    029bfcdb 2dce28d9 59f2815b 16f81798

    and $y =$

    483ada77 26a3c465 5da4fbfc 0e1108a8
    fd17b448 a6855419 9c47d08f fb10d4b8
  - order$(G) = q$ is a prime, where $q =$

    FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
    BAAEDCE6 AF48A03B BFD25E8C D0364141
- **KeyGen**
  - Pick $a \xleftarrow{\$} \mathbb{Z}_q$. Compute $H = aG$
  - Set $\begin{cases} \text{pk} = (E, p, q, G, H) \\ \text{sk} = (a, \text{pk}) \end{cases}$
- **Sign**(msg.txt, sk)
  1. Compute $h =$ hash(msg.txt)
  2. Pick $r \xleftarrow{\$} \mathbb{Z}_q$. Compute rG.
  3. Let $rG = (u, v)$, where $u, v \in \mathbb{Z}_p$
  4. $c_1 = u \bmod q$ (If $c_1 = 0$, goto (2))
  5. $c_2 = r^{-1}(h + ac_1) \bmod q$ (If $c_2 = 0$, goto (2))
  6. Output signature $\sigma = (c_1, c_2)$
- **Verify**$(msg.txt, \sigma = (c_1, c_2), \text{pk})$
  - Compute $h =$ hash(msg.txt)
  - Compute $e_1 = hc_2^{-1} \bmod q$
  - Compute $e_2 = c_1 c_2^{-1} \bmod q$
  - Compute $J = e_1 G + e_2 H$
  - Let $J = (u, v)$
  - Output **valid** $\iff u \bmod q = c_1$

ECDSA Singnature Malleability

- Claim Suppose, $(c_1, c_2) \leftarrow \text{Sign}(m, \text{sk} = a)$, then

$$\text{Verify}((c_1, q - c_2), m, \text{pk} = H) = \texttt{valid}$$

- Indeed, if $h = \text{SHA256}(\text{SHA256}(m))$, $e_1 = h(q - c_2)^{-1} = h(-c_2)^{-1} \bmod q$, and $e_2 = c_1(q - c_2)^{-1} = c_1(-c_2)^{-1} \bmod q$, then

$$\begin{aligned}
e_1 G + e_2 H &= h(-c_2)^{-1} G + c_1(-c_2)^{-1} H \\
&= -hc_2^{-1} G - c_1 c_2^{-1} H \\
&= -(hc_2^{-1} G + c_1 c_2^{-1} H) \\
&= -(hc_2^{-1} G + c_1 c_2^{-1} aG) \\
&= -(hc_2^{-1} + c_1 c_2^{-1} a) G \\
&= -c_2^{-1}(h + c_1 a) G \\
&= -rG \\
&= -(u, v) \\
&= (u, -v)
\end{aligned}$$

txn Malleability

txn =

nVersion
0x02
hash0
n0
scriptSigLen0
$\langle (c_1, c_2) \rangle$ $\langle$pk$\rangle$
nSequence0
hash1
n1
scriptSigLen1
scriptSig1
nSequence1
0x01
nValue0
scriptPubkeyLen0
scriptPubkey0
nLockTime

txn′ =

nVersion
0x02
hash0
n0
scriptSigLen0
$\langle (c_1, q - c_2) \rangle$ $\langle$pk$\rangle$
nSequence0
hash1
n1
scriptSigLen1
scriptSig1
nSequence1
0x01
nValue0
scriptPubkeyLen0
scriptPubkey0
nLockTime

- Both, txn and txn′, are valid transactions
- But, HASH(txn) $\neq$ HASH(txn′), i.e. txid $\neq$ txid′

- We call txn, malleable.

Table of Contents

1 Lecture#1 (18/01) [Course Overview, Bitcoin Introduction]

2 Lecture#2 (21/01) [Hash Functions]

3 Lecture#3 (25/01) [Random Oracle Model, Iterated Hash Construction]

4 Lecture#4 (28/01) [Hash Puzzle, Signature Scheme]

5 Lecture#5 (01/02) [Group Theory]

6 Lecture#6 (04/02) [Discrete Logarithm Problem, DSA]

7 Lecture#7 (04/02) [Elliptic Curves, ECDSA]

8 Lecture#8 (11/02) [Bitcoin Address Generation]

9 Lecture#9 (15/02) [Bitcoin Transaction and its Validation]

10 Lecture#10 (18/02) [Bitcoin Consensus]

11 Lecture#11 (22/02) [Block Mining, txn Serialization]

12 Lecture#12 (25/02) [txn Serialization, PaytoScriptHash]

13 Lecture#13 (01/03) [More lockScripts, txn-malleability]

14 Lecture#14 (04/03) [SegWit transaction]

15 Lecture#15 (15/03) [Analysis of Bitcoin's Consensus]

16 Lecture#16 (21/03) [Hyperledger Fabric]

17 Lecture#17 (25/04) [Ethereum]

18 Lecture#18 (29/03) [Ethereum Contract Accounts]

19 Lecture#19 (05/04) [Payment Channels, Coin Mixers]

20 Lecture#20 (08/04) [Coin Mixers, MixEth, TumbleBit]

21 Lecture#21 (19/04) [Tokens]

ashoka UNIVERSITY

Segregate Witness Type txn (SegWit)

- SegWit type txn was introduced to solve the problem of txn malleability.

0. We first introduce SegWit type Locking ($\texttt{scriptPubKey}^{\texttt{W}}$) and Unlocking ($\texttt{scriptSig}^{\texttt{W}}$) scripts.

---

1. SegWit-Output A regular txn output that locks an UTXO with a SegWit scriptPubkey is called SegWit output.
2. SegWit-Input A regular txn input that unlocks a SegWit-output is called a SegWit-input.
3. SegWit-txn A regular txn is called SegWit-type if at least one of its inputs is SegWit

---

0. P2WPKH (Pay to Witness Public Key Hash) $\texttt{scriptPubKey}^{\texttt{W}}$ and $\texttt{scriptSig}^{\texttt{W}}$

- $\texttt{scriptPubKey}^{\texttt{W}}$: { `OP_0 <PubKeyHash>`

- $\texttt{scriptSig}^{\texttt{W}} = \begin{cases} \texttt{scriptSig:} & \texttt{(empty)} \\ \texttt{scriptWitness:} & \texttt{<Signature> <PublicKey>} \end{cases}$

- Where do we keep scriptWitness in txn serialisation ??

SegWit txn Serialisation

nVersion
MarkerByte = 0x00
FlagByte = 0x01
Number of Inputs N
Input_0
⋮
Input_{N-1}
Number of Outputs M
Output_0
⋮
Output_{M-1}
Witness_0
⋮
Witness_{N-1}
nLockTime

- SegWit txn Serilisation
  - If the txn has N inputs, there are N witness structures
  - The kth witness structure correspond to the kth input ($0 \leq k \leq N-1$)
- If an input is not a SegWit input, its corresponding witness structure is just a single byte containing $0x00$

Example

- $T_{A \to B}$

Inputs
$H(T_{U \to A})$
0
$\langle \text{Sign}_{\text{sk}_A}(\text{m}_{T_{A \to B}}, 0) \rangle \ \langle \text{pk}_A \rangle$
Outputs
x฿
OP_0 $\langle \text{HASH160}(\text{pk}_B) \rangle$

- $T_{A \to B}$ has a `SegWit` output.
- But, $T_{A \to B}$ is not a `SegWit txn`

- $T_{B \to C}$

Inputs
$H(T_{A \to B})$
0
Empty
$H(T_{V \to B})$
0
$\langle \text{Sign}_{\text{sk}_B}(\text{m}_{T_{B \to C}}, 1) \rangle \ \langle \text{pk}_B \rangle$
Outputs
x฿
OP_DUP OP_HASH160 $\langle \text{HASH160}(\text{pk}_C) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
Witnesses
$\langle \text{Sign}_{\text{sk}_B}(\text{m}_{T_{B \to C}}, 0) \rangle \ \langle \text{pk}_B \rangle$
$0x00$

- The 0th input of $T_{B \to C}$ is a `SegWit` input
- Thus, $T_{B \to C}$ is a `SegWit txn`

SegWit txn Validation

Inputs
$H(T_{A\to B})$
0
Empty

$H(T_{V\to B})$
0
$\langle \text{Sign}_{\text{sk}_B}(^{\text{m}}T_{B\to C},1)\rangle\ \langle \text{pk}_B\rangle$
Outputs
x฿
OP_DUP OP_HASH160 $\langle \text{HASH160}(\text{pk}_C)\rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
Witnesses
$\langle \text{Sign}_{\text{sk}_B}(^{\text{m}}T_{B\to C},0)\rangle\ \langle \text{pk}_B\rangle$
0x00

- SegWit Input Validation
  - $\texttt{scriptPubkey}^{\text{W}}$: OP_0 $\langle \text{HASH160}(\text{pk}_B)\rangle$
  - $\texttt{scriptSig}^{\text{W}}$:

$$\begin{cases} \texttt{scriptSig}: & (\texttt{empty}) \\ \texttt{scriptWitness}: & \langle \text{Sign}_{\text{sk}_B}(^{\text{m}}T_{B\to C},0)\rangle\ \langle \text{pk}_B\rangle \end{cases}$$

- Validation
  - From $\texttt{scriptPubKey}^{\text{W}}$ construct,
  - $\texttt{scriptPubkey} = \begin{cases} \text{OP\_DUP OP\_HASH160 } \langle \text{HASH160}(\text{pk}_B)\rangle \\ \text{OP\_EQUALVERIFY OP\_CHECKSIGVERIFY} \end{cases}$
  - Finally, execute: $\texttt{scriptWitness}\|\texttt{scriptPubkey}$

Transaction Identifiers for `SegWit txn`

nVersion
MarkerByte = 0x00
FlagByte = 0x01
Number of Inputs N
Input_0
⋮
Input_{N-1}
Number of Outputs M
Output_0
⋮
Output_{M-1}
Witness_0
⋮
Witness_{N-1}
nLockTime

- For a `SegWit` txn, define
- $\texttt{txn}^- = \texttt{txn} \setminus \{\texttt{MarkerByte}, \texttt{FlagByte}, \texttt{Witnesses}\}$
- Define, `txid` and `wtxid` as follows

$$\begin{aligned} \text{txid} &= \text{SHA256}(\text{SHA256}(\texttt{txn}^-)) \\ \text{wtxid} &= \text{SHA256}(\text{SHA256}(\texttt{txn})) \end{aligned}$$

- UTXO's are identified by txid - see the next slide for clarity on this!

Example

- $T_{B \to C}$

Inputs
$H(T_{A \to B})$
0
Empty
$H(T_{V \to B})$
0
$\langle \text{Sign}_{\text{sk}_B}(\text{m}_{T_{B \to C},1}) \rangle \ \langle \text{pk}_B \rangle$
Outputs
x฿
OP_DUP OP_HASH160 $\langle \text{HASH160}(\text{pk}_C) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
Witnesses
$\langle \text{Sign}_{\text{sk}_B}(\text{m}_{T_{B \to C},0}) \rangle \ \langle \text{pk}_B \rangle$
$0x00$

- $T_{C \to D}$

Inputs
$\text{txid}_{T_{B \to C}} = H(T^{-}_{B \to C})$
0
$\langle \text{Sign}_{\text{sk}_C}(\text{m}_{T_{C \to D},0}) \rangle \ \langle \text{pk}_C \rangle$
Outputs
y฿
OP_0 $\langle \text{HASH160}(\text{pk}_D) \rangle$

- So, where do we use $\text{wtxid}_{T_{C \to D}}$?

SegWit Coinbase txn

- In a valid block, the coinbase txn
  - is regular if all txn in the block are regular
  - is SegWit if at least one txn is SegWit

SegWit Compatibility

- A sender's wallet must be SegWit-enabled in order to
  - create a txn having SegWit outputs, or
  - create a SegWit txn, i.e., one of its inputs is consuming a SegWit output, create a transaction, or
  - detect the status of SegWit transaction.

- Case-1
- Suppose, a user A (who uses SegWit enabled wallet) wants to pay a receiver B by a SegWit output
- In that case A must know that B also uses a SegWit wallet.
- Additionally B should provide a P2WPKH address (not P2PKH) to A for the payment.

- P2WPKH Address
- BIP-173 describes the format for P2WPKH (and P2WSH) address
- It uses Base32 encoding (instead of Base58 encoding)
- The format is also called bech32 addresses
- The format uses 32 lower-case-only alphanumeric character set - carefully selected to reduce errors from misreading or mistyping.
- The underlying checksum is based on *BCH* error detection algorithm - allowing not only detection but also correction of errors.

SegWit Compatibility

- Mainnet P2WPKH

  `bc1qhp8prqltrlcd8c5t9xh7rvts7t9kv999zsjvr9`
- Mainnet P2WSH

  `bc1qgdjqv0av3q56jvd82tkdjpy7gdp9ut8tlqmgrpmv24sq90ecnvqqjwvw97`
- Testnet P2WPKH

  `tb1qgfdez6jd3theaek7ud9q8stx0cnpjhx4zv48zx`
- Testnet P2WSH

  `tb1qrp33g0q5c5txsp9arysrx4k6zdkfs4nce4xj0gdcccefvpysxf3q0sl5k7`

- If P2WPKH(pk_B) is given, then A extracts HASH160(PK_B) out of it.

- Case-2
- Suppose B's wallet is SegWit-enabled but A's is not, and B wants its payment using SegWit output !
- **Wayout:** B can prepare a SegWit P2WPKH LScript and make a regular P2SH address (prefix is 3) out of it.
- A can be presented with the P2SH address which can be processed by its regular wallet, without really knowing that the redeem script inside it is a SegWit P2WPKH.

SegWit Compatibility

- Case-3
- Consider a miner, N, who hasn't yet upgraded to SegWit-enabled Bitcoin Core.
- Suppose N receives a SegWit txn $T_{B \to C}$ (i.e., on of its input is consuming a SegWit output)
- In validating $T_{B \to C}$, N will ignore the witness field.
- It will take it as a regular txn and look for scriptSig field of the input, which is empty in this case
- But, the empty field will not cause the regular script execution to fail, because the SegWit output's LScript, $\mathsf{OP\_0}\ \langle \mathsf{HASH160}(\mathsf{pk_B}) \rangle$, allows any one to unlock it !!
- Does that mean the bitcoin locked to a SegWit LScript can be unlocked by any bode ? Not really !!

- Suppose M is an another miner who is SegWit enabled and also receives the $T_{B \to C}$.
- Because M is SegWit enabled, it will look to the appropriate witness field for the correct ULScript !!

- In summary, if the majority of the miner's are SegWit enabled, then SegWit outputs are secure!

Incentives for upgrading to SegWit enabled wallets/Bitcoin Core

- Addressing the issue of transaction malleability
- Efficient signature generation (leading to low transaction fee)
- More transactions per block (Good for miners)
- Good for clients who doesn't want to keep the entire Blockchain ledger

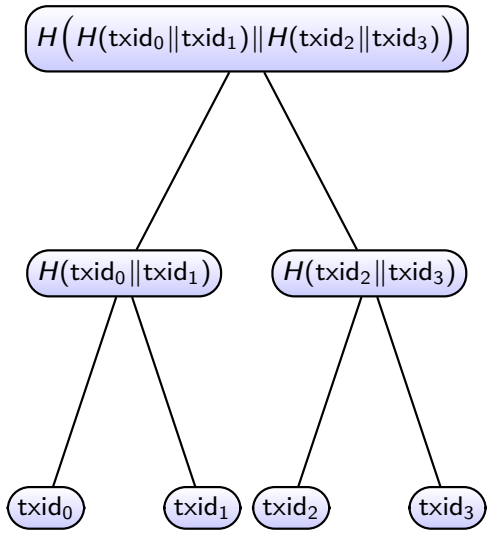
Why is the wtxid calculated and how is it used?

If B =

BlockSize
Version(B_v)
PrevBlocHash(B_p)
MTR(B_m)
TimeStamp(B_t)
Difficulty(B_d)
Nonce(B_n)
txnCounter
txn_0
txn_1
txn_2
txn_3

No SegWit txn in B

- Then $\texttt{MTR} = H\Big(H(\text{txid}_0 \| \text{txid}_1) \| H(\text{txid}_2 \| \text{txid}_3)\Big)$
- And it is computed as follows



- B is valid, if $H(B_v \| B_p \| B_m \| B_t \| B_d \| B_n) \leq B_d$
- Any tampering of txn_i's will result in new B_m^*.
- New B_m^* will make B invalid (High Probability)
- That is $H(B_v \| B_p \| B_m^* \| B_t \| B_d \| B_n) > B_d$

Notation: $H(\cdot) = \texttt{SHA256}(\texttt{SHA256}(\cdot))$

Why is the wtxid calculated and how is it used?

If B =

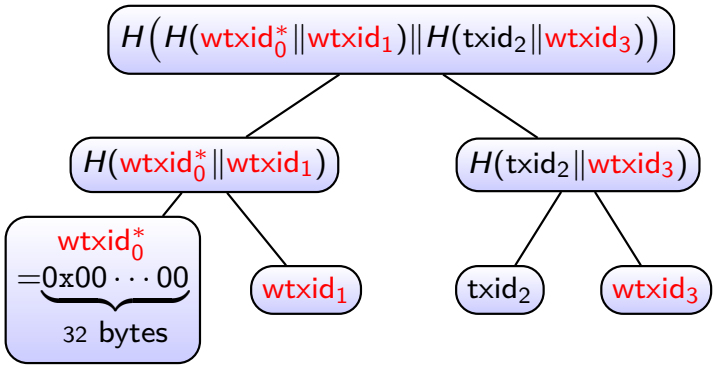
BlockSize
Version(B_v)
PrevBlocHash(B_p)
MTR(B_m)
TimeStamp(B_t)
Difficulty(B_d)
Nonce(B_n)
txnCounter
txn_0^{sw}
txn_1^{sw}
txn_2
txn_3^{sw}

- txn_0^{sw} (coinbase) is computed as follows

$\text{txn}_0^{\text{sw}} =$

nVersion
Number of Inputs = 1
Dummy Input
Number of Outputs = 2
Output 0 (P2PKH output)
nValue = 0
scriptPubkey Length = 0x26
scriptPubkey = OP_RETURN 0$xAA21A9ED$ $H(\text{WMTR}\|\text{WRV})$
Number of stack items = 0$x01$
Length of stack item1 = 0$x20 bytes$
Witness reserved value(WRV)
nLockTime

- $\text{WMTR} = H\Big(H(\text{wtxid}_0^*\|\text{wtxid}_1)\|H(\text{txid}_2\|\text{wtxid}_3)\Big)$
- And it is computed as follows



- $\text{MTR} = H\Big(H(\text{txid}_0\|\text{txid}_1)\|H(\text{txid}_2\|\text{txid}_3)\Big)$
- And, it is computed as usual

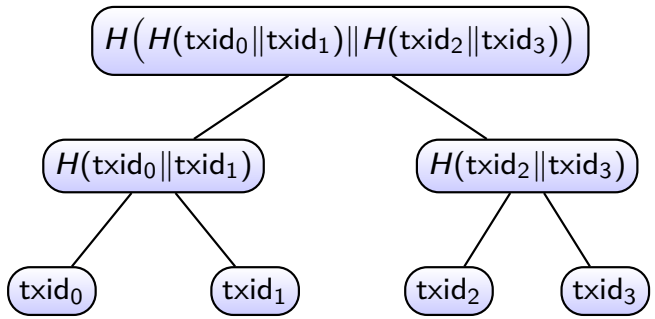


Table of Contents

1 Lecture#1 (18/01) [Course Overview, Bitcoin Introduction]

2 Lecture#2 (21/01) [Hash Functions]

3 Lecture#3 (25/01) [Random Oracle Model, Iterated Hash Construction]

4 Lecture#4 (28/01) [Hash Puzzle, Signature Scheme]

5 Lecture#5 (01/02) [Group Theory]

6 Lecture#6 (04/02) [Discrete Logarithm Problem, DSA]

7 Lecture#7 (04/02) [Elliptic Curves, ECDSA]

8 Lecture#8 (11/02) [Bitcoin Address Generation]

9 Lecture#9 (15/02) [Bitcoin Transaction and its Validation]

10 Lecture#10 (18/02) [Bitcoin Consensus]

11 Lecture#11 (22/02) [Block Mining, txn Serialization]

12 Lecture#12 (25/02) [txn Serialization, PaytoScriptHash]

13 Lecture#13 (01/03) [More lockScripts, txn-malleability]

14 Lecture#14 (04/03) [SegWit transaction]

15 Lecture#15 (15/03) [Analysis of Bitcoin's Consensus]

16 Lecture#16 (21/03) [Hyperledger Fabric]

17 Lecture#17 (25/04) [Ethereum]

18 Lecture#18 (29/03) [Ethereum Contract Accounts]

19 Lecture#19 (05/04) [Payment Channels, Coin Mixers]

20 Lecture#20 (08/04) [Coin Mixers, MixEth, TumbleBit]

21 Lecture#21 (19/04) [Tokens]

ashoka UNIVERSITY

The Flow

- txn Creation
  - Bitcoin transactions are created by wallets and pushed into the network

---

- txn Validation
  - Transactions soon reach all full nodes in the network
  - Every node would validate received transactions
  - Validation requires each node to use `Bitcoin Core` software, and the special execution environment therein
  - The validated transactions are put into each node's `transaction pool`
  - Unconfirmed `txn` The `txns` that are present in the transaction pool of a node, but not in its blockchain.

- txn Ordering
  - The network now get together to reach a consensus on a set of unconfirmed `txns` next.
  - To this node, every node prepares a valid block containing a set of unconfirmed `txns` from its pool.
  - Network is said to reach a consensus if all nodes agree on the same block.
  - Confirmed `txn` A `txn` is tagged as confirmed if it is included into a valid block.

- In our discussion so far we looked at the blockchain, the global public ledger (list) of all transactions, which everyone in the bitcoin network accepts as the authoritative record of ownership.
- The blockchain is not created by a central authority, but is assembled independently by every node in the network.
- Somehow, every node in the network, acting on information transmitted across insecure network connections, can arrive at the same conclusion and assemble a copy of the same public ledger as everyone else.
- We now examine the process (Bitcoin's decentralized consensus - we have already discussed it partially, time to fill in the missing details) by which the bitcoin network achieves global consensus without central authority.

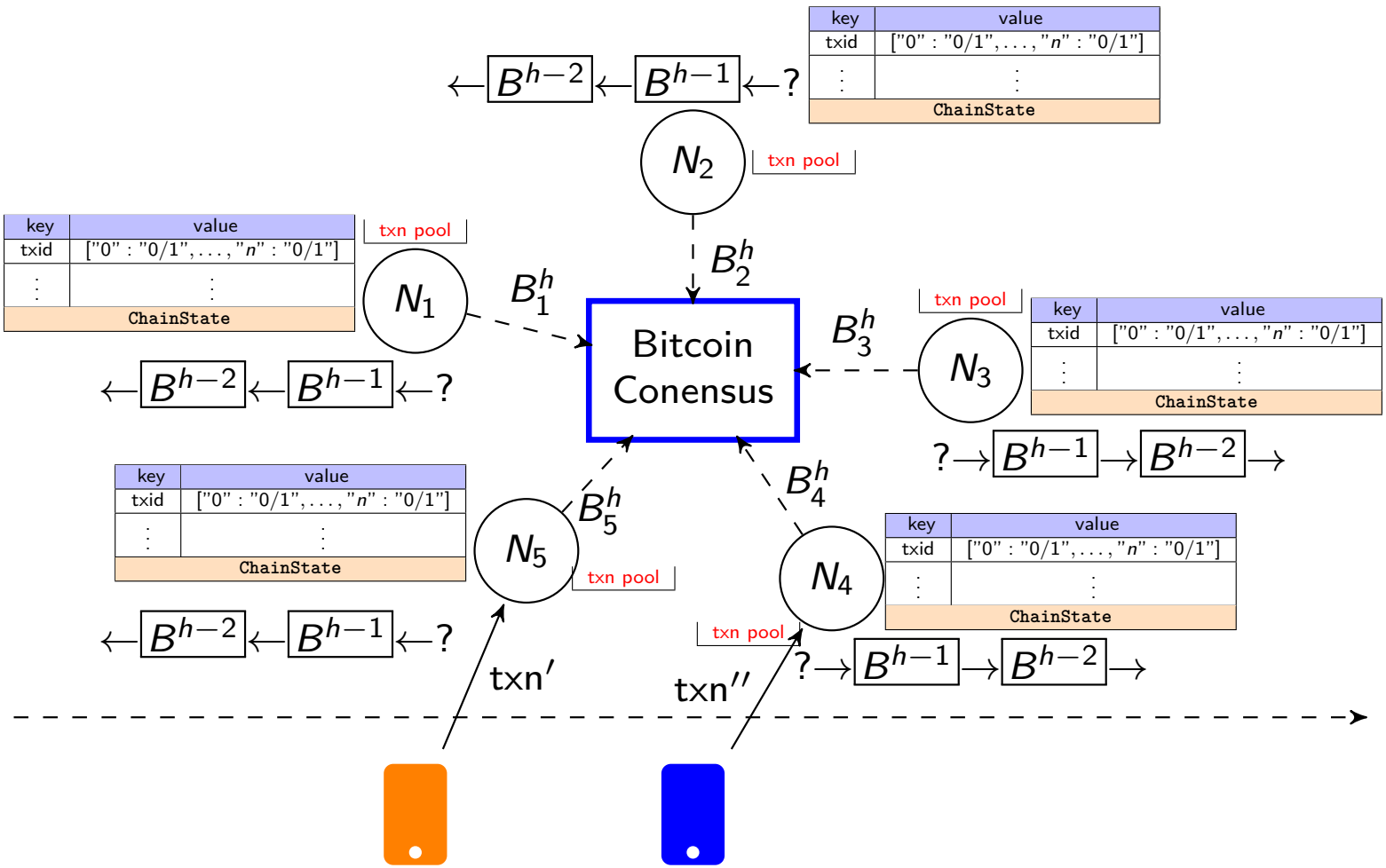
- Bitcoin's decentralized consensus emerges from the interplay of four processes that occur independently on nodes across the network

1. Independent verification of each transaction, by every full node, based on a comprehensive list of criteria
2. Independent aggregation of those transactions into new blocks by mining nodes, coupled with demonstrated computation through a Proof-of-Work algorithm
3. Independent verification of the new blocks by every node and assembly into a chain
4. Independent selection, by every node, of the chain with the most cumulative computation demonstrated through Proof-of-Work

---

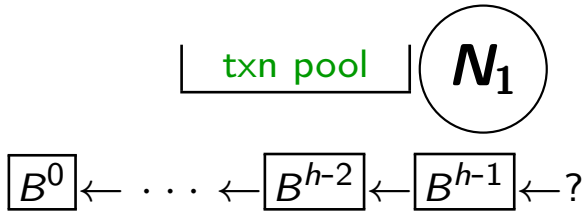
- We now examine these processes and how they interact to create the emergent property of network-wide consensus that allows any bitcoin node to assemble its own copy of the blockchain !!

The Bitcoin Architecture



Key	Value
$h(\text{txn}\ell_1)$	["0" : "UTXO$_{\ell_1 0}$",..., "r_1" : "UTXO$_{\ell_1 r_1}$"]
$h(\text{txn}\ell_2)$	["0" : "UTXO$_{\ell_2 0}$",..., "r_2" : "UTXO$_{\ell_2 r_2}$"]
⋮	⋮
$h(\text{txn}\ell_k)$	["0" : "UTXO$_{\ell_k 0}$",..., "r_k" : "UTXO$_{\ell_k r_k}$"]

Chain State



Block Chain

- N_1 represents a full node in Bitcoin
- N_1's local database - "**Bitcoin Ledger**"
- "**Ledger**" $\equiv$ "**Blockchain**" + "**Chain State**"
- **Chain State** is also called - **World State** of the Ledger

- Chain State is a key-value data base
  - **key**: txid $= h(\text{txn})$
  - **value**: txn **outputs**
  - outputs in red are considered spent
  - outputs in blue are considered un-spent

Transaction Validity

- Suppose, N_1 receives

$$\text{txnq}^* = \begin{cases} \text{Input0} : (h(\text{txn}_{\ell_1}), 0, \text{ULS}_{\text{UTXO}_{\ell_1 0}}) \\ \text{Output0} : \text{UTXO}_{q0} \\ \text{Output1} : \text{UTXO}_{q1} \end{cases}$$

- N_1 retrieves $\text{LS}_{\text{UTXO}_{\ell_1 0}} \leftarrow$ **Chain State**$[h(\text{txn}_{\ell_1})(0)]$
- **Execute:** $\text{ULS}_{\text{UTXO}_{\ell_1 0}} \parallel \text{LS}_{\text{UTXO}_{\ell_1 0}}$
- If no error, add txnq^* to txn pool

- txnq^* is then relayed into the network by N_1

  This ensures that only valid transactions are propagated across the network, while invalid transactions are discarded at the first node that encounters them.

- Simultaneously, N_1 updates its local **Chain State** as follows:

Key	Value
$h(\text{txn}\ell_1)$	["0" : "UTXO$_{\ell_1 0}$",..., "r_1" : "UTXO$_{\ell_1 r_1}$"]
$h(\text{txn}\ell_2)$	["0" : "UTXO$_{\ell_2 0}$",..., "r_2" : "UTXO$_{\ell_2 r_2}$"]
⋮	⋮
$h(\text{txn}\ell_k)$	["0" : "UTXO$_{\ell_k 0}$",..., "r_k" : "UTXO$_{\ell_k r_k}$"]
$h(\text{txnq}^*)$	["0" : "UTXO$_{q0}$", "1" : "UTXO$_{q1}$"]

txn pool N_1

Chain State

② Independent aggregation of txns into new blocks

② Independent aggregation of those transactions into new blocks by mining nodes, coupled with demonstrated computation through a Proof-of-Work algorithm

- After validating transactions, a mining node will add them to the memory pool, or transaction pool, where transactions await until they can be included (mined) into a block.

● Mining a block - aggregating transactions from mempool into a *valid* block !!

- Once a valid block is prepared by the mining node, it transmits the block to all its peers.

---

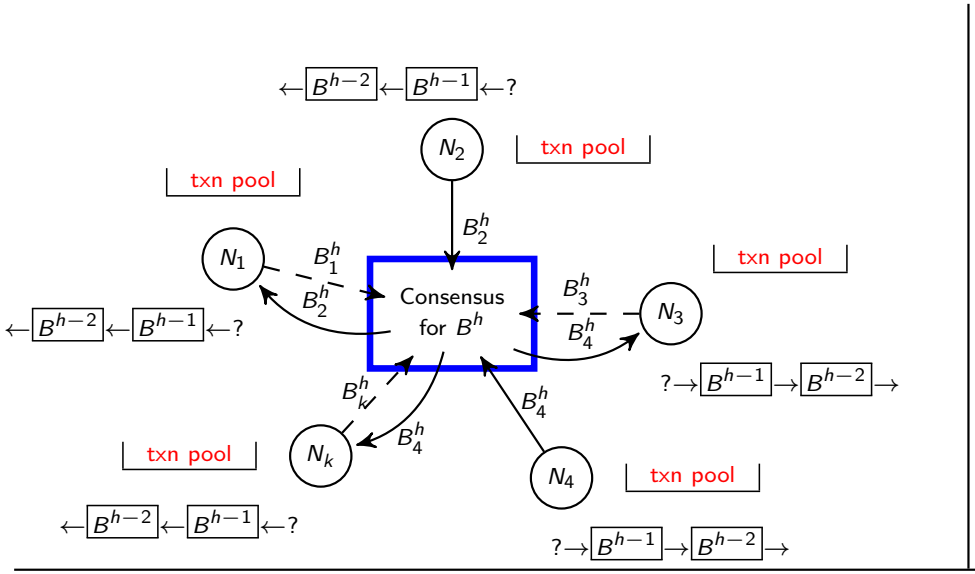
③ Independent verification of the new blocks by every node and assembly into a chain

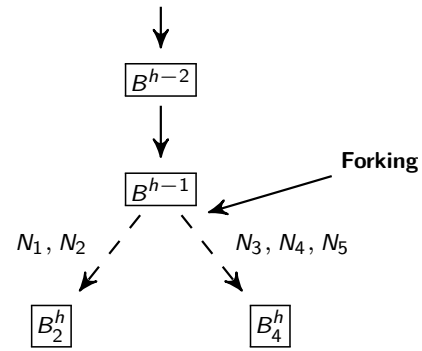
- All mining nodes listen for blocks, propagated on the bitcoin network.
- The arrival of a new block has special significance for a mining node - it acts as an announcement of a winner - receiving a valid new block means someone else won the competition and they lost.
- After receiving the block, they first validate it, and then they abandon their efforts to find a block at the same height and immediately start computing the next block in the chain, using received block as the "parent"

ashoka UNIVERSITY

The problem of multiple chains

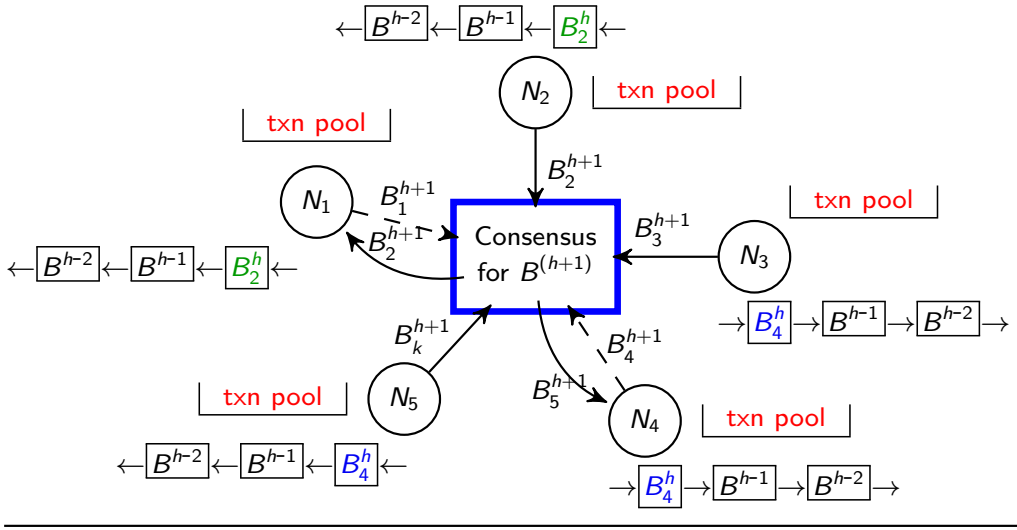
- Suppose two miners solve the Proof-of-Work algorithm within a short period of time from each other.
- As both miners discover a solution for their respective candidate blocks, they immediately broadcast their own "winning" block to their immediate neighbours who begin propagating the block across the network.
- Each node that receives a valid block will incorporate it into its blockchain, extending the blockchain by one block.
- If that node later sees another candidate block extending the same parent, it connects the second candidate on a secondary chain.
- As a result, some nodes will "see" one candidate block first, while other nodes will see the other candidate block and two competing versions of the blockchain will emerge.

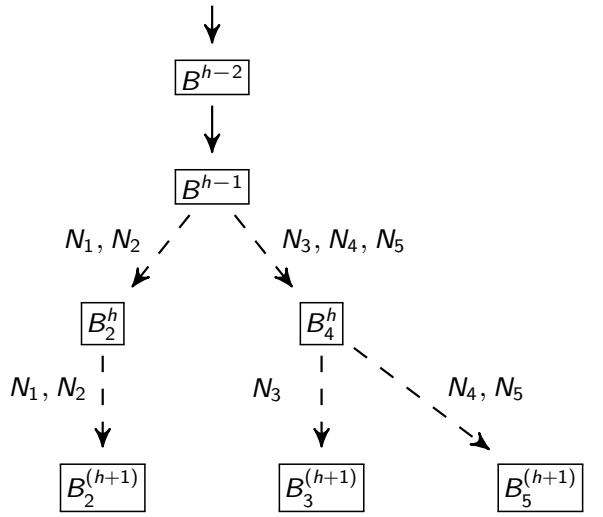




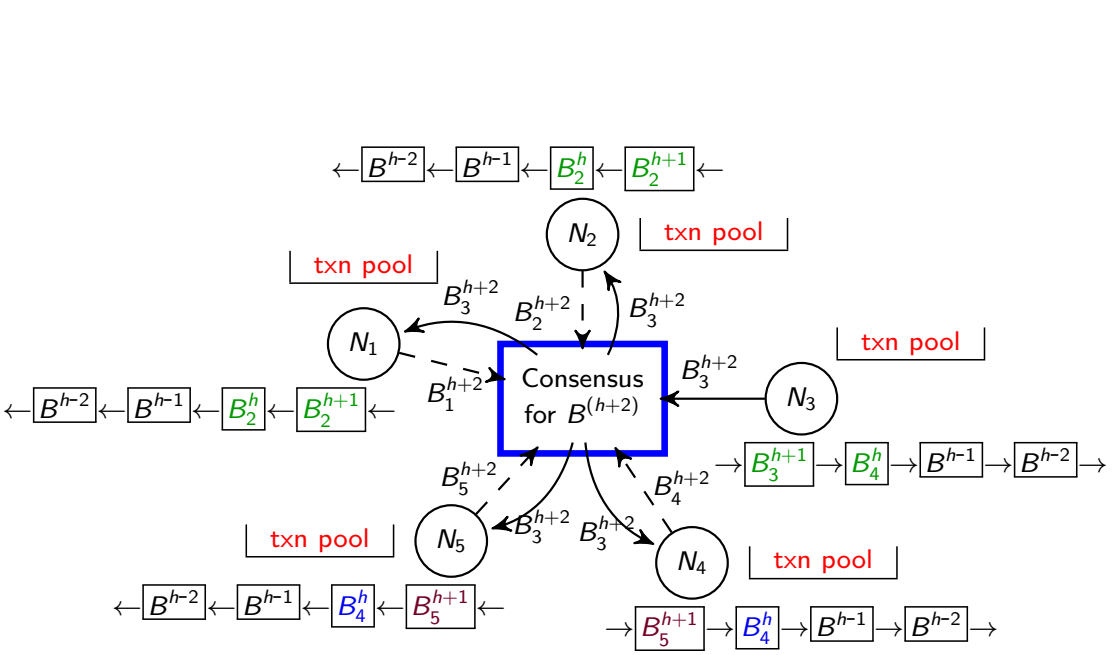
- The miner N_1 is aware of the two different branches (/chains) of the ledger that now exist. Given that the length for these two chains are same, N_1 is allowed to mine $B^{(h+1)}$ extending its own branch! Similarly, the other miners will work to extend their own branches.

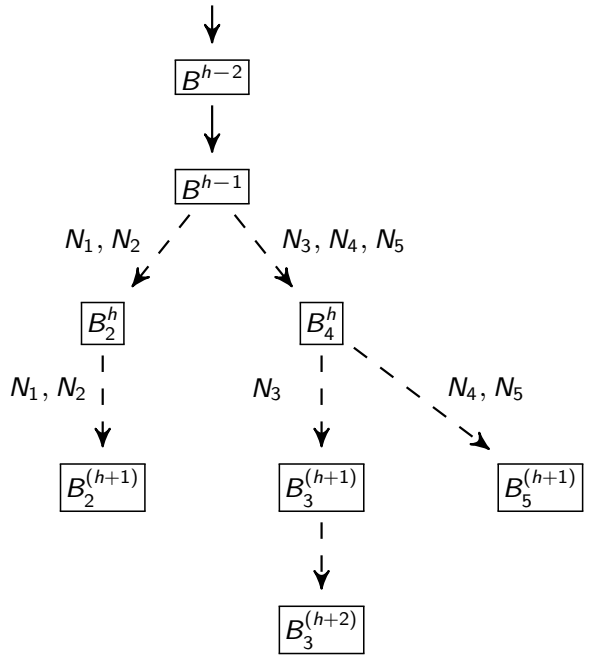
The problem of multiple chains





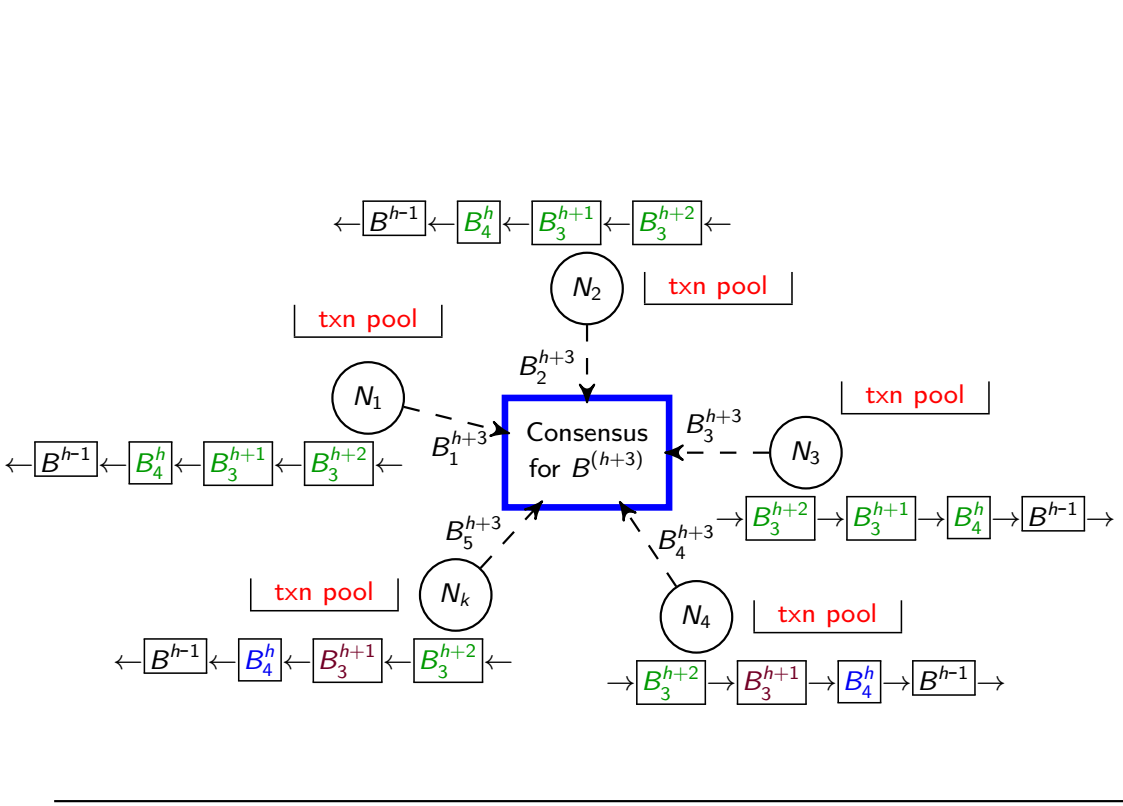
The problem of multiple chains





- In the selection of $B^{(h+2)}$, there is a clear winner, which is $B_3^{(h+2)}$
- ▶ What should N_1, N_2, N_4 and N_5 be doing to mine $B_{(h+3)}$?
- Should they continue to mine $B^{(h+2)}$, and once done, begin mining $B^{(h+3)}$ atop it?
- Should they discard their respective chains and begin mining atop $B_3^{(h+2)}$, i.e., switch to N_3's chain?
- ● Bitcoin consensus mechanism requires a miner to switch to longest available chain !

Switching to Longest Chain !!



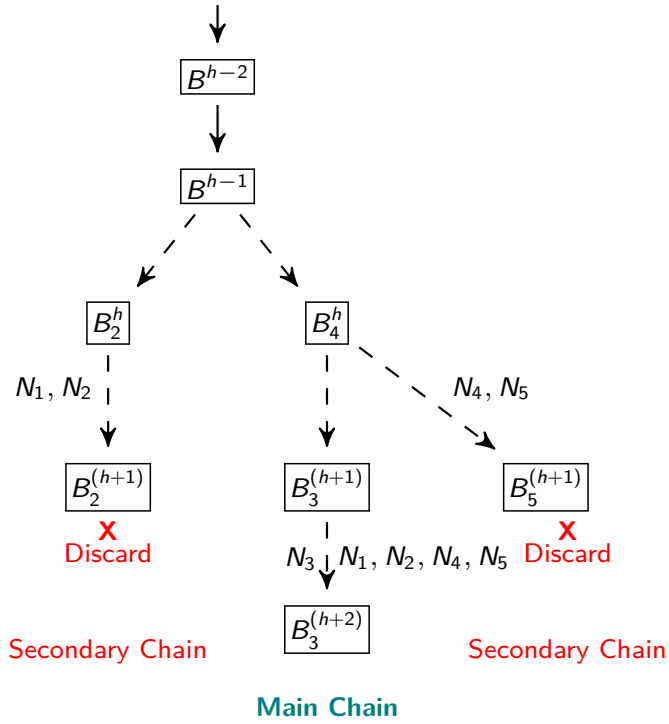


Table of Contents

1 Lecture#1 (18/01) [Course Overview, Bitcoin Introduction]

2 Lecture#2 (21/01) [Hash Functions]

3 Lecture#3 (25/01) [Random Oracle Model, Iterated Hash Construction]

4 Lecture#4 (28/01) [Hash Puzzle, Signature Scheme]

5 Lecture#5 (01/02) [Group Theory]

6 Lecture#6 (04/02) [Discrete Logarithm Problem, DSA]

7 Lecture#7 (04/02) [Elliptic Curves, ECDSA]

8 Lecture#8 (11/02) [Bitcoin Address Generation]

9 Lecture#9 (15/02) [Bitcoin Transaction and its Validation]

10 Lecture#10 (18/02) [Bitcoin Consensus]

11 Lecture#11 (22/02) [Block Mining, txn Serialization]

12 Lecture#12 (25/02) [txn Serialization, PaytoScriptHash]

13 Lecture#13 (01/03) [More lockScripts, txn-malleability]

14 Lecture#14 (04/03) [SegWit transaction]

15 Lecture#15 (15/03) [Analysis of Bitcoin's Consensus]

16 Lecture#16 (21/03) [Hyperledger Fabric]

17 Lecture#17 (25/04) [Ethereum]

18 Lecture#18 (29/03) [Ethereum Contract Accounts]

19 Lecture#19 (05/04) [Payment Channels, Coin Mixers]

20 Lecture#20 (08/04) [Coin Mixers, MixEth, TumbleBit]

21 Lecture#21 (19/04) [Tokens]

ashoka UNIVERSITY

A Milestone !!

- At this point in the course, we have accomplished the goal of understanding how Bitcoin works at a technical level !!
- We now understand the underlying blockchain-based consensus mechanism that plays a breakthrough role in making Bitcoin work !!
- **Next Goal:** To explore how newer distributed applications can be built using blockchain !!
- To achieve this, we first bring abstraction to the building-blocks of Bitcoin so that a framework for constructing blockchain based distributed applications could emerge !!

- The first component of any application is formed by its users
- As users interact with the application, the history of its interaction needs to be recorded in a **data base**
- **Updating the data base** Every interaction with the application can be represented as users making some transactions; each transaction proposing to create or update a database entry (lets refer to it as an **asset**); and finally the application validating the transactions as per the prescribed application rules before executing the proposed task (create/update a data base entry)
- **Distributed data base** The other component is the set of operational nodes managing the overall application, i.e., together receiving transactions; validating, ordering and executing in a way such that it leads to a unique view of the data base

- For **Bitcoin**, we have
- Assests $\equiv$ Bitcoin data base entries

Key	Value
txid	Asset = [UTXO amount, LockScript]

- Asset Creation
- An asset creation is done using a txn, and subsequently, the newly created asset, is placed in the database using the key = txid
- Asset Update
- A transaction must be made to update an Asset.
- The prescribed "asset-update-rule" is used to validate the transaction first and then the update is executed in the database.
- Finally, the mining nodes of the Bitcoin's network do the validation, ordering and execution of these txns as per the Bitcoin's blockchain-based distributed consensus mechanism !!

A Milestone !!

- For **Bitcoin**, we have
- Assests $\equiv$ Bitcoin data base entries

Key	Value
txid	Asset = [UTXO amount, LockScript]

- Asset Creation
- An asset creation is done using a txn, and subsequently, the newly created asset, is placed in the database using the key = txid
- Asset Update
- A transaction must be made to update an Asset.
- The prescribed "asset-update-rule" is used to validate the transaction first and then the update is executed in the database.
- Finally, the mining nodes of the Bitcoin's network do the validation, ordering and execution of these txns as per the Bitcoin's blockchain-based distributed consensus mechanism !!

- **Security** Incorrect asset updates will cause users to lose Bitcoins !!
- Bitcoin's blockchain-based distributed consensus mechanism gives an assurance of not allowing incorrect asset updates despite the fact that it is run by mutually distrusted mining nodes !!

- **Anonymous Feedback System with Accountability - an application**
- **User Interaction**
- The application allows users to post feedbacks anonymously
- It also allows users to flag a already posted anonymous-feedback
- **Accountability:** If an anonymous feedback receives enough number of flags, the application must make the feedback non-anonymous by revealing the identity of the user who posted it
- **Transaction:** Users make transactions to post feedbacks, make transactions to flag feedbacks
- Posting a new feedback $\equiv$ Asset creation !!

Key	Value
txid	Asset = [Feedback, Who Posted , Who Flagged = {}]

- Flagging a feedback $\equiv$ Asset update !!
- **Anonymous feedback turning non-anonymous** $\equiv$ Asset query !!
- Asset update and query is correctly done as per the prescribed application rulebook - which outlines when is precisely the identity of the user who posted the feedback can be revealed; other users can flag the feedback only once; or users are not allowed to flag their own feedbacks, etc.

ashoka UNIVERSITY

A Milestone !!

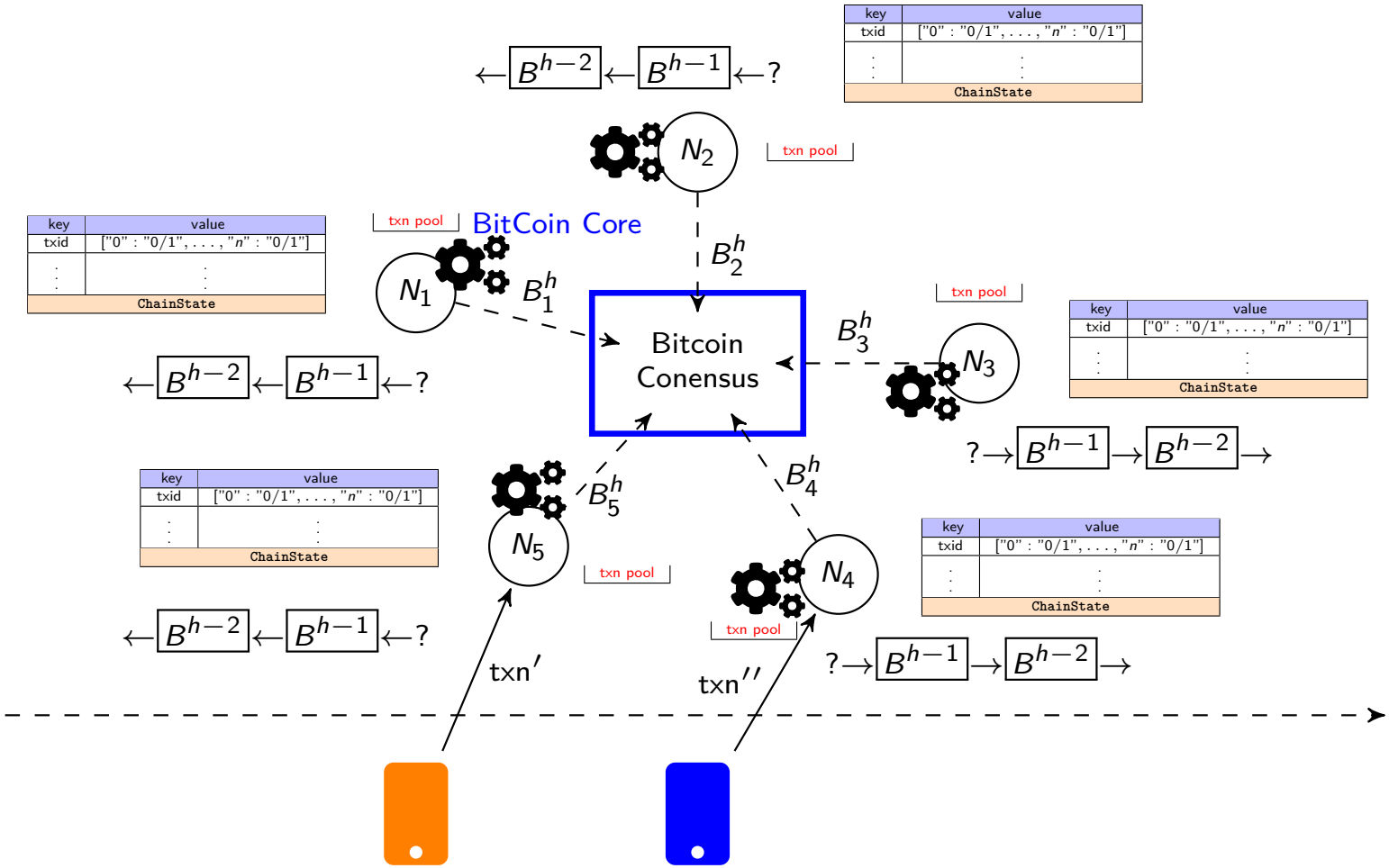
- **Anonymous Feedback System with Accountability - an application**
  - **User Interaction**
  - The application allows users to post feedbacks anonymously
  - It also allows users to flag a already posted anonymous-feedback
  - **Accountability:** If an anonymous feedback receives enough number of flags, the application must make the feedback non-anonymous by revealing the identity of the user who posted it
  - **Transaction:** Users make transactions to post feedbacks, make transactions to flag feedbacks
  - Posting a new feedback $\equiv$ Asset creation !!

Key	Value
txid	Asset = [Feedback, Who Posted , Who Flagged = {}]

  - Flagging a feedback $\equiv$ Asset update !!
  - **Anonymous feedback turning non-anonymous** $\equiv$ Asset query !!
  - Asset update and query is correctly done as per the prescribed application rulebook - which outlines when is precisely the identity of the user who posted the feedback can be revealed; other users can flag the feedback only once; or users are not allowed to flag their own feedbacks, etc.

  - **The Goal** To run and manage the application not by a single server, but by a set of mutually distrusted servers
  - Can we use Bitcoin's blockchain based consensus mechanism for the same ??
  - **Looks like we definitely can !!**

1. **Application** It can be described as users **creating** and **updating** a few types of **assets** and they do so, by proposing different types of **transactions**.
2. **Distributed Database** The database for all assets (as **key-value** pairs), and all the transactions, are implemented in distributed fashion among multiple servers (peers).
3. **Distributed Consensus** The peers take part in a distributed algorithm, at a regular interval, to reach a consensus on a set of newly proposed transactions for their,
   - validation, (as per the prescribed **rules** of the application)
   - ordering, and
   - execution (leading to creation and update of assets)



A Few Observations !!

- In Bitcoin's Blockchain-based peer-to-peer network, any one is allowed to join as a peer in the network
- At any point of the time, no clarity on the total number of peers present in the network, or every peer's identity, etc.
- This, in the presence of arbitrary number of malicious peers, makes the security of managing assets creation and updating them, an extremely difficulty process

---

- The problem of consensus gets much easier if we have a restriction in place on peer enrolment ?
- With the knowledge of peer identities and the total number of peers present in the system, we may avoid, altogether, the expensive Bitcoin's distributed consensus and replace it by a well known distributed consensus algorithm !!

- We insist that the consensus at every step still requires an agreement on a block of transactions, though the block preparation no longer require the expensive proof-of-work; we also insist to continue maintaining a blockchain (chain of blocks)!!

- We call Bitcoin's blockchain-based peer-to-peer network as **Permission-less**
- By enforcing a restriction on the peer enrolment, we call the resulting blockchain as **Permissioned Blockchain**.
- A permissioned blockchain can be of the following types:
- Enterprise Blockchain Framework: Limiting the entry of peers by allowing them from a single organisation
- Consortium Blockchain Framework: Limiting the entry of peers by only allowing them from a pre-approved set of organisations (not trusted by each other)

- Consider deploying the application "Anonymous Feedback System with Accountability" on a permissioned-blockchain !!
  - The application allows users to post feedbacks, anonymously
  - It also allows users to flag a already posted anonymous-feedback
  - If an anonymous feedback receives enough number of flags, the application must make the feedback non-anonymous by revealing the identity of the user who posted it

---

- Asset Type

Key	Value
txid	Asset = [Feedback, Who Posted , Who Flagged = {}]

where txid is the transaction identifier of the txn that created this asset in the first place.

- Rulebook A Rulebook outlines how a transaction must be validated before it is allowed to create an asset or modify an existing one.
- Network Configuration We might insist that, the distributed network must have peers allowed only from two pre-approved organisations.

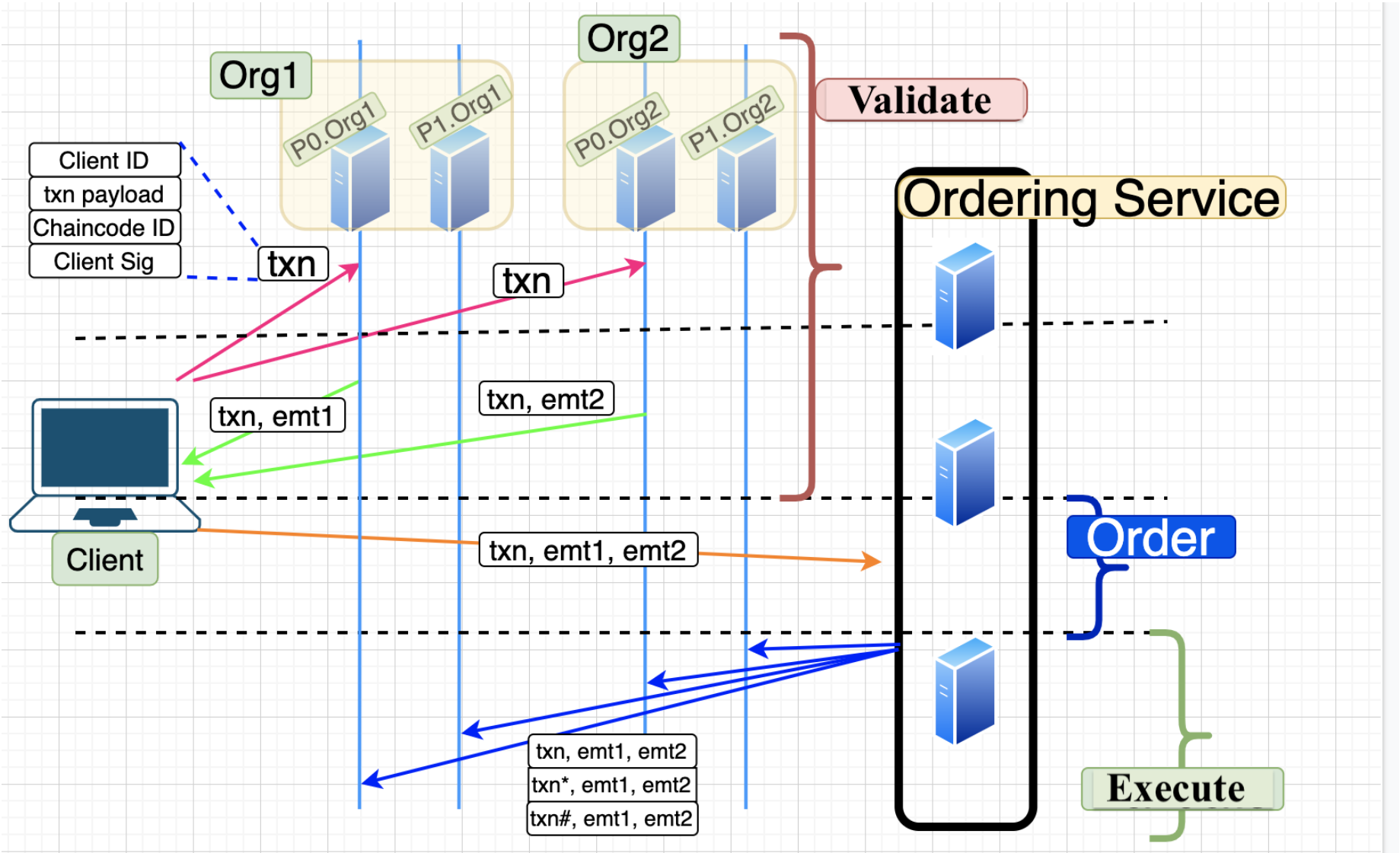
- Does there exists a framework that lets you deploy the above application on a permission-blockchain the following way (modular way !!):

- Step-(1) Put in place, a peer-to-peer to network, where each peers is from one of the two pre-approved organisation
- Step-(2) Import the **Application Rulebook** into each of these peers, where the rule book prescribed the asset types, and the rules that must be followed to update existing assets !!
- Step-(3) Configure all peers with the same distributed consensus algorithm that they can use to validate, order and execute newly proposed transactions!

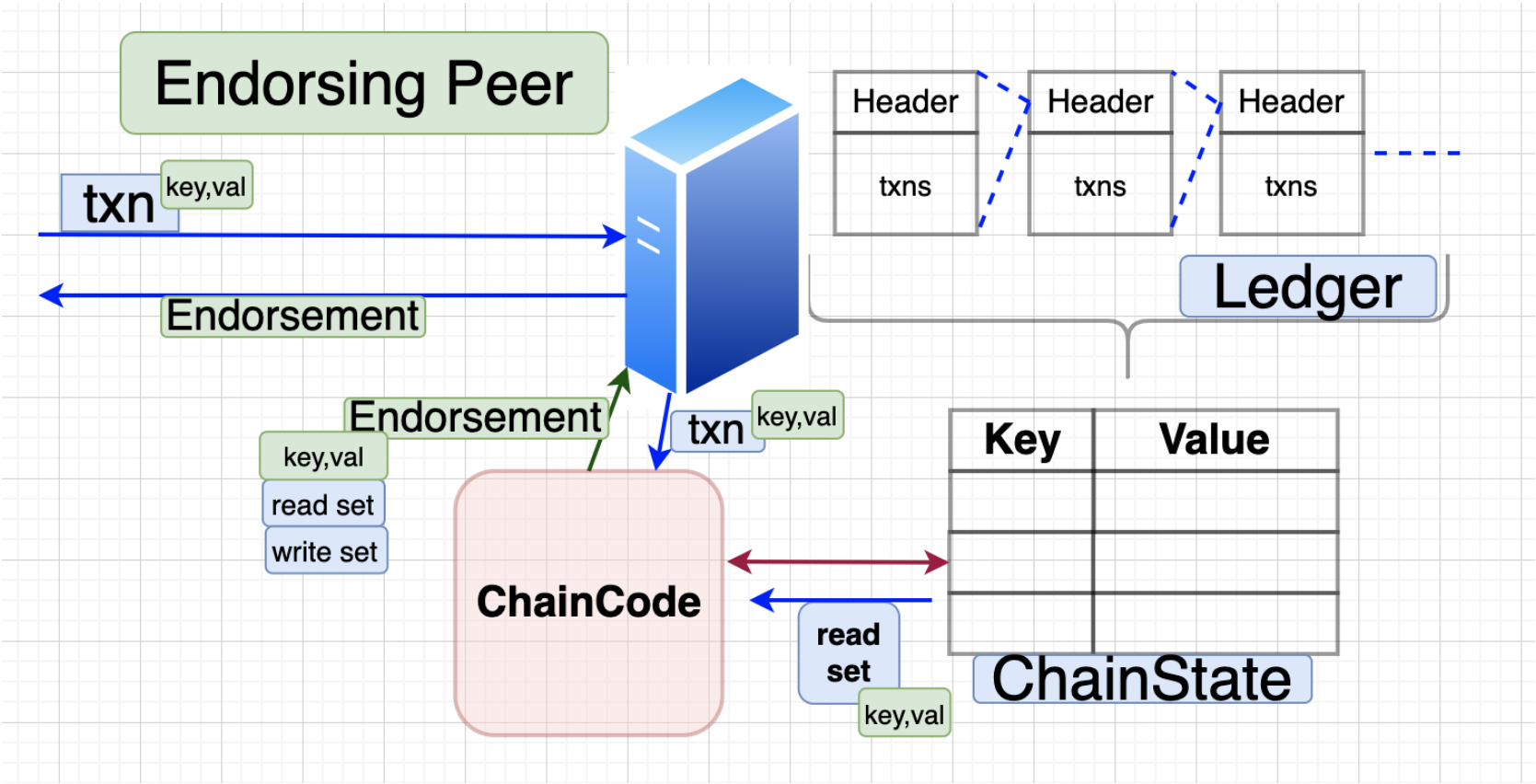
Hyperledger Fabric

- **Hyperledger Fabric** `https://www.hyperledger.org/use/fabric` is an open-source framework for deploying and operating Permissioned Blockchains!
- The Permissioned Blockchains, in turn, then run dedicated distributed applications.
- Fabric is one of the Hyperledger `https://www.hyperledger.org/` projects hosted by the Linux Foundation

Hyperledger Fabric: Architecture and Flow of the Deployed Blockchain-based Distributed System



Peer Architecture: Endorsing Peer



Peer Architecture: Committing Peer

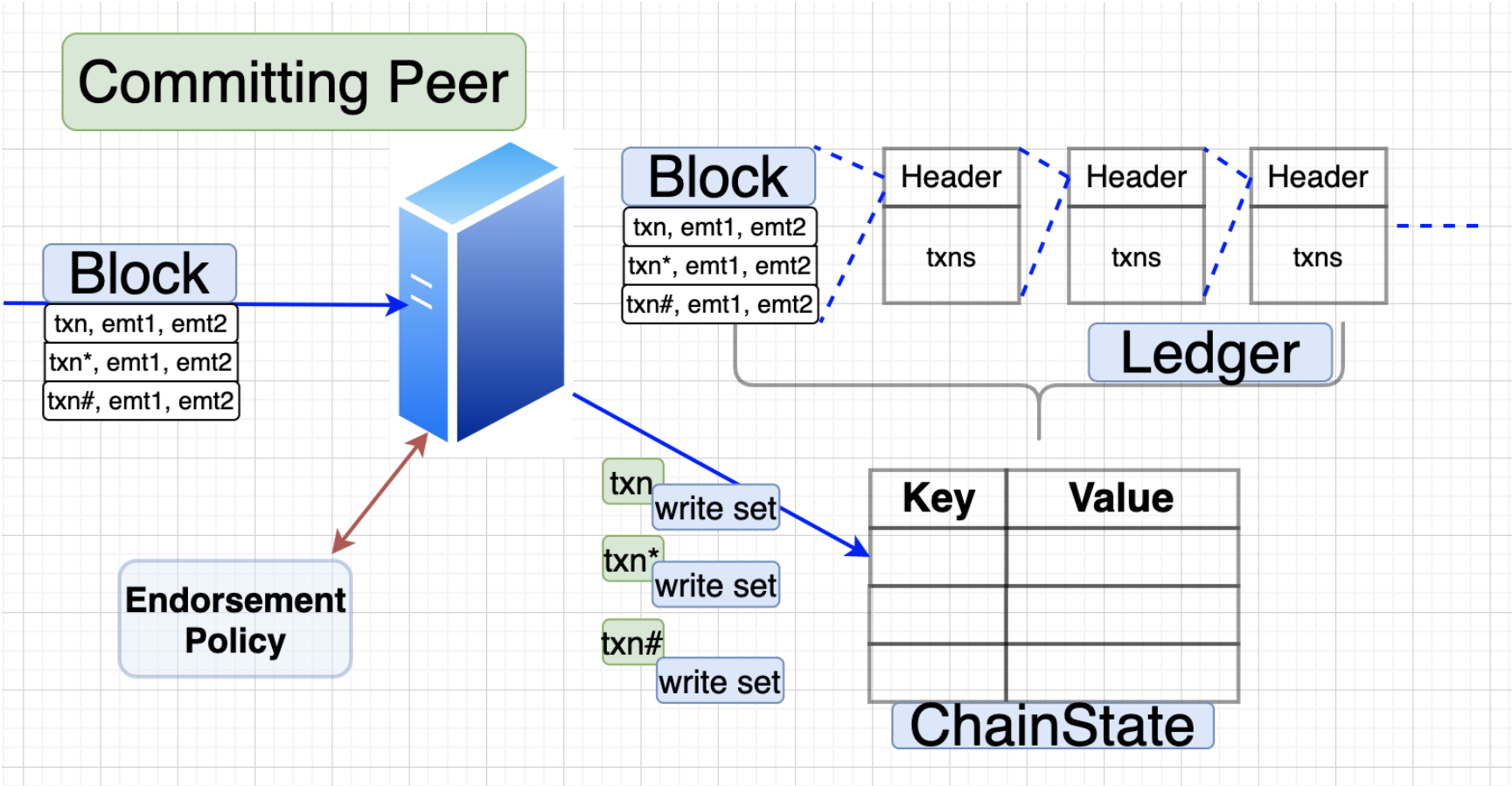


Table of Contents

1 Lecture#1 (18/01) [Course Overview, Bitcoin Introduction]

2 Lecture#2 (21/01) [Hash Functions]

3 Lecture#3 (25/01) [Random Oracle Model, Iterated Hash Construction]

4 Lecture#4 (28/01) [Hash Puzzle, Signature Scheme]

5 Lecture#5 (01/02) [Group Theory]

6 Lecture#6 (04/02) [Discrete Logarithm Problem, DSA]

7 Lecture#7 (04/02) [Elliptic Curves, ECDSA]

8 Lecture#8 (11/02) [Bitcoin Address Generation]

9 Lecture#9 (15/02) [Bitcoin Transaction and its Validation]

10 Lecture#10 (18/02) [Bitcoin Consensus]

11 Lecture#11 (22/02) [Block Mining, txn Serialization]

12 Lecture#12 (25/02) [txn Serialization, PaytoScriptHash]

13 Lecture#13 (01/03) [More lockScripts, txn-malleability]

14 Lecture#14 (04/03) [SegWit transaction]

15 Lecture#15 (15/03) [Analysis of Bitcoin's Consensus]

16 Lecture#16 (21/03) [Hyperledger Fabric]

17 Lecture#17 (25/04) [Ethereum]

18 Lecture#18 (29/03) [Ethereum Contract Accounts]

19 Lecture#19 (05/04) [Payment Channels, Coin Mixers]

20 Lecture#20 (08/04) [Coin Mixers, MixEth, TumbleBit]

21 Lecture#21 (19/04) [Tokens]

ashoka UNIVERSITY

Summary: Bitcoin, Hyperledger Fabric

- Bitcoin
  - Distributed Application (DAPP): peer-to-peer currency
  - Underlying Platform
    - Peer-to-peer distributed network
      - Every peer must be running the same Bitcoin-core software
    - Network is permission-less
    - Distributed consensus: Nakamoto Consensus (Proof-of-Work)
  - Bitcoin wallets to connect users to Bitcoin network

- Hyperledger Fabric
  - Distributed Application (DAPP): Arbitrary, not fixed
  - Underlying Platform: Based on your input application, it lets you bootstrap a, customised, underlying platform.
    - Peer-to-peer distributed network
    - Network is permissioned
    - Distributed consensus: it lets you choose from a set of available distributed consensus protocols.
- Configure each peer node, with a smart contract that governs the application. These smart contracts are written using general purpose programming languages
  - Fabric wallets to connect users to the network

ashoka UNIVERSITY

Running Arbitrary DAPP On Permission-less Network

- Ethereum
  - Distributed Application (DAPP): Arbitrary, not fixed
  - Underlying Platform: Fixed
    - Peer-to-peer distributed network
      - Every peer must be running a software, built as per the Ethereum Yellow-paper specification
    - Network is permission-less
    - Distributed consensus: Similar to Bitcoin's Proof-of-Work (Plans to upgrade to Proof-of-Stake). This implies the availability of an incentive mechanism to reward peers.
      - Ethereum, also implements a peer-to-peer currency - called "Ether"
- ~~Configure each peer node, with a smart contract that governs the application~~ This is clearly not to be done this way! The peer-to-peer network is permission-less, and therefore dynamic.
  - Making a transaction to create a key-value pair, where the value contains the smart contract governing a specific application !!
  - Once approved, the smart contract finds its place into the Blockchain's world-state and gets an immutable status
  - Subsequent transactions related to the application can be executed (by peers) as per the smart contract
  - Ethereum wallets to connect users to Ethereun network

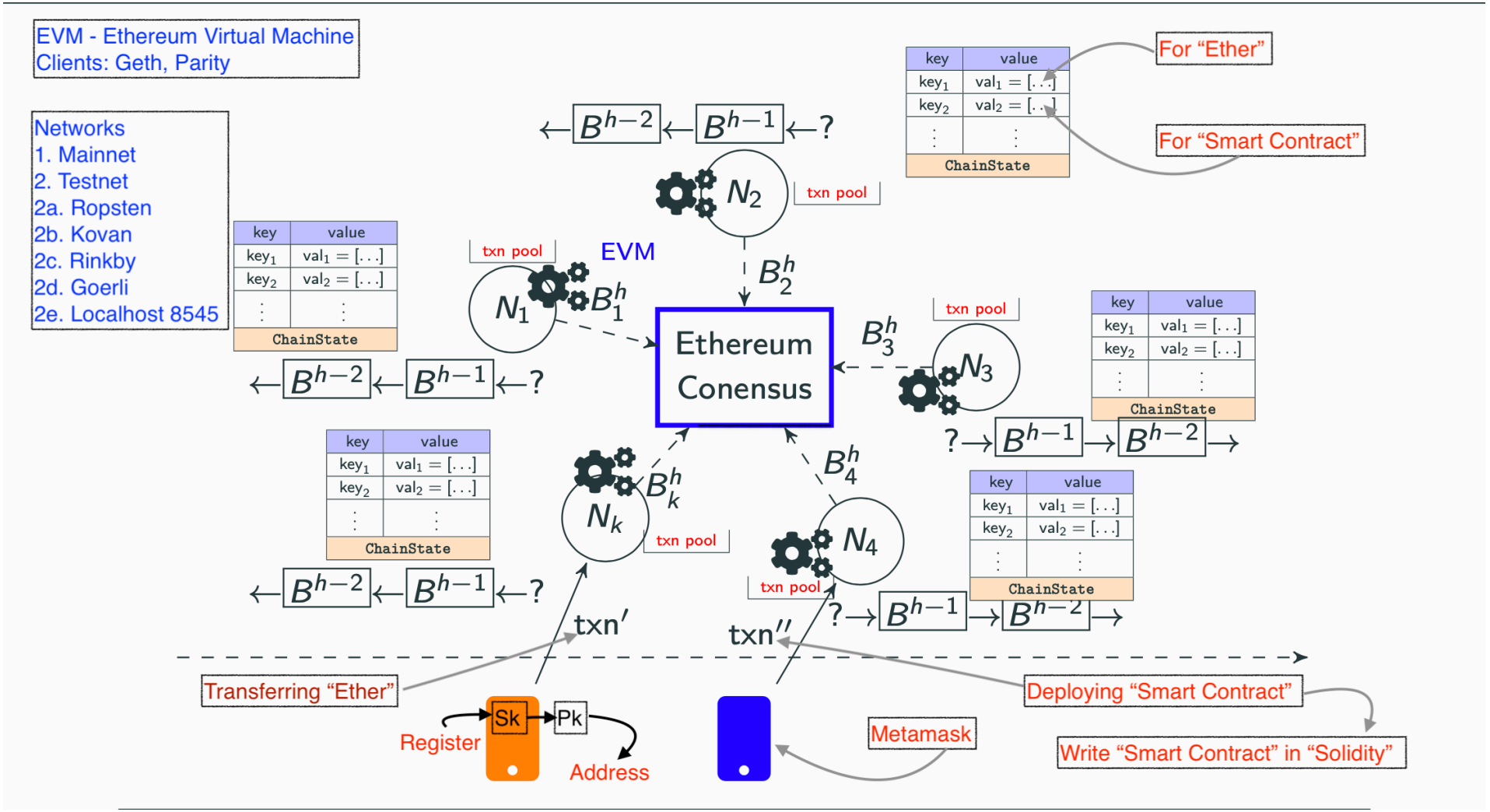
Ethereum

- **Key-Value** pair types in the Ethereum Blockchain's world-state!

- **Two Types**

1. **Key-Value** pair to manage a user's **Ether** balance
2. **Key-Value** pair to manage a smart contract, implementing a distributed application

The Ethereum Architecture



Introduction

- Pointers
  - https://ethereum.org/
  - https://github.com/ethereum
  - https://ethereum.org/learn/
  - https://github.com/ethereum/wiki/wiki/White-Paper
  - https://ethereum.github.io/yellowpaper/paper.pdf
  - https://ethereum.org/developers
  - https://github.com/ConsenSys/ethereum-developer-tools-list
- Wallets
  - https://metamask.io/
- Ethereum Explorers
  - https://blockchair.com/ethereum
  - https://etherscan.io/
  - https://etherchain.org/

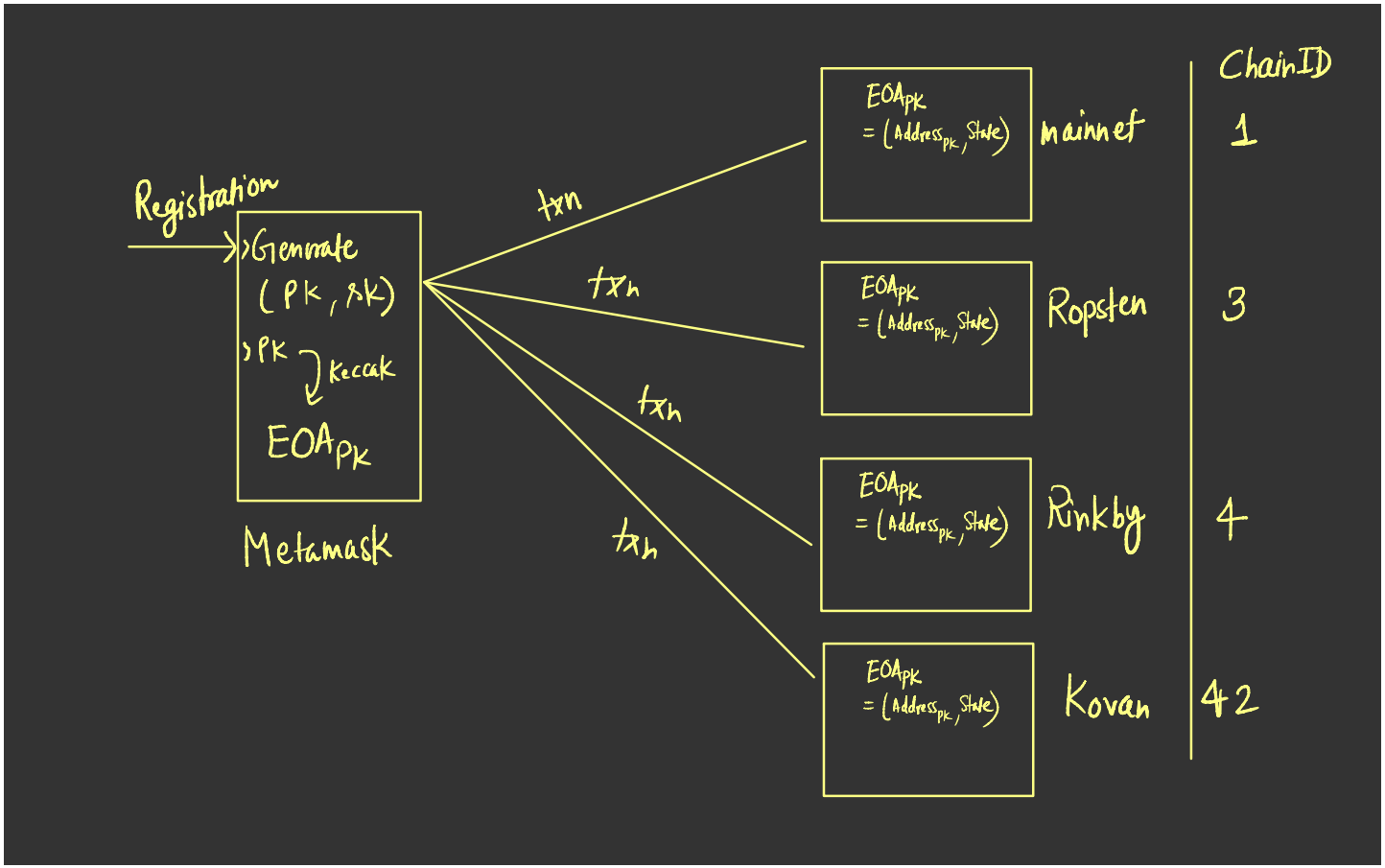
Bitcoin Vs Ethereum

	Bitcoin	**Ethereum**
Proposed by	Satoshi Nakamoto (Pseudonym)	Vitalik Buterin (Original Creator). Later Gavin Wood became Co-founder [Link]
Currency Unit	1 BTC = 10^8 satoshi	1 ETH = 10^{18} Wei
First Block was mined on	January 3, 2009 [Link]	July 30, 2015 [Link]
Smart Contract low level Language	Script	EVM bytecode (Solidity - high level language)
Block Interval	10 mins	15 seconds [Link]
Consensus	Proof of Work (Nakamoto)	Proof of Work (Ethash), Proof of Stake (Future, Eth 2.0)
Currency supply	fixed	variable

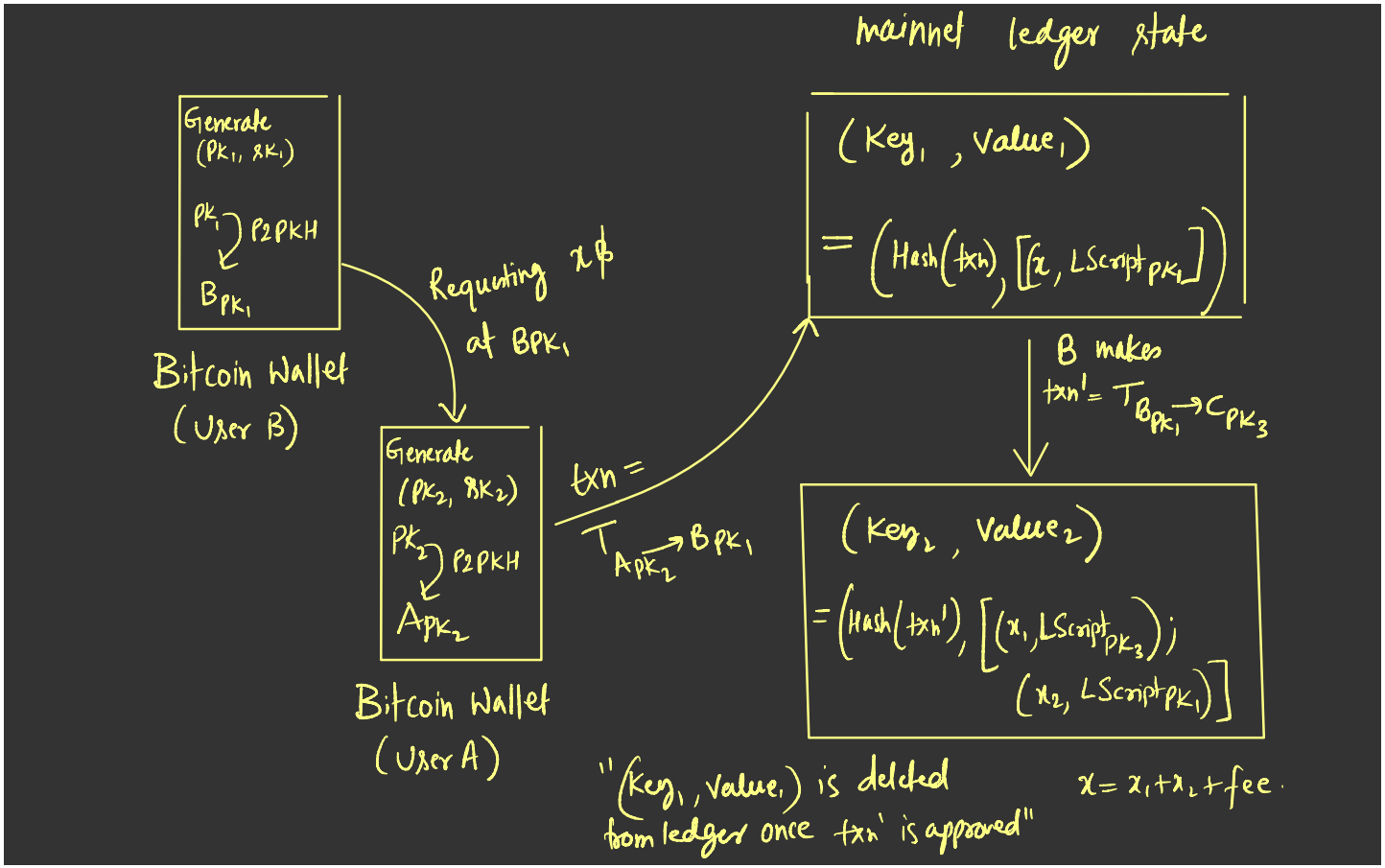
Externally Owned Account

- Registration with a wallet
1. Keys, Address generation
   - The wallet generates an ECDSA key pair (pk, sk)
     - secp256k1 curve is used (the same as Bitcoin)
   - $Address_{pk}$ = right most 160 bits (least significant bits) of Keccak256(pk), where Keccak is a hash function, winner of SHA3 competition. Keccak256 outputs are 256 bits.
2. Creation of an Externally Owned Account (EOA)
   - A transaction is made (by the wallet) to create a key-value pair in the world-state database
   - $(key, val) \equiv (Address_{pk}, State)$
   - The same key-value pair is created in the mainnet and all testnets (Metamask makes the underlying transactions in these networks)
   - The key-value pair of this type is called an Externally Owned Account, controlled by pk (EOA_{pk})
   - The address of the account EOA_{pk} is $Address_{pk}$
   - The State of the account has following components
     - `nonce`: represents the number of transactions sent from this address
     - `balance`: represents the number of Wei owned by this account

Externally Owned Account



EOA Vs UTXO


mainnet ledger state
(Key1, Value1)
= (Hash(txn), [(x, LScript PK1)])
Generate (PK1, SK1)
PK1 → P2PKH → BPK1
Bitcoin Wallet (User B)
Requesting x ฿ at BPK1
Generate (PK2, SK2)
PK2 → P2PKH → APK2
Bitcoin Wallet (User A)
txn = T APK2 → B PK1
B makes txn' = T BPK1 → CPK3
(Key2, Value2)
= (Hash(txn'), [(x1, LScript PK3), (x2, LScript PK1)])
"(Key1, Value1) is deleted from ledger once txn' is approved"
x = x1 + x2 + fee.

Transaction for Transferring Ether

- $EOA_1 \xrightarrow{txn} EOA_2$
  - This is to transfer some ether from EOA_1 to EOA_2
  - EOA_1 initiates the transaction
  - The transaction structure is

Field	Value
[nonce]	T_n
[gasPrice]	T_p
[gasLimit]	T_g
[to]	T_t
[value]	T_v
[data]	T_d
[v]	T_w
[(r,s)]	(T_r, T_s)

- `Nonce`
  - $T_n = EOA_1.State.nonce$

- `gasPrice`
  - The validation of T requires series of computational steps
  - Computational steps are further broken into atomic computational opcodes
  - Each atomic computation requires some units of gas to be paid
  - gasPrice $= T_p =$ # of Wei to be paid per unit of gas

- `gasLimit`
  - $T_g =$ maximum amount of gas to be consumed in validating T
  - This is paid up-front
  - $T_g \times T_p$ Wei are deducted from sender EOA
  - Any unused gas is refunded back to sender EOA
  - Consumed gas goes to miner's EOA

Table of Contents

1. Lecture#1 (18/01) [Course Overview, Bitcoin Introduction]
2. Lecture#2 (21/01) [Hash Functions]
3. Lecture#3 (25/01) [Random Oracle Model, Iterated Hash Construction]
4. Lecture#4 (28/01) [Hash Puzzle, Signature Scheme]
5. Lecture#5 (01/02) [Group Theory]
6. Lecture#6 (04/02) [Discrete Logarithm Problem, DSA]
7. Lecture#7 (04/02) [Elliptic Curves, ECDSA]
8. Lecture#8 (11/02) [Bitcoin Address Generation]
9. Lecture#9 (15/02) [Bitcoin Transaction and its Validation]
10. Lecture#10 (18/02) [Bitcoin Consensus]
11. Lecture#11 (22/02) [Block Mining, txn Serialization]
12. Lecture#12 (25/02) [txn Serialization, PaytoScriptHash]
13. Lecture#13 (01/03) [More lockScripts, txn-malleability]
14. Lecture#14 (04/03) [SegWit transaction]
15. Lecture#15 (15/03) [Analysis of Bitcoin's Consensus]
16. Lecture#16 (21/03) [Hyperledger Fabric]
17. Lecture#17 (25/04) [Ethereum]
18. Lecture#18 (29/03) [Ethereum Contract Accounts]
19. Lecture#19 (05/04) [Payment Channels, Coin Mixers]
20. Lecture#20 (08/04) [Coin Mixers, MixEth, TumbleBit]
21. Lecture#21 (19/04) [Tokens]

ashoka UNIVERSITY

Transaction

- ▶ $\text{EOA}_1 \xrightarrow{\text{txn}} \text{EOA}_2$
- — This is to transfer some ether from EOA_1 to EOA_2
- — EOA_1 initiates the transaction
- — The transaction structure is

Field	Value
[nonce]	T_n
[gasPrice]	T_p
[gasLimit]	T_g
[to]	T_t
[value]	T_v
[data]	T_d
[v]	T_w
[(r,s)]	(T_r, T_s)

- - to T_t contains 160-bit address EOA_2
- - vlaue
- ▶ $T_v = \#$ of Wei's to be transferred to EOA_2
- - data $T_d = \phi$

v, r, s

- - T_w, T_r, T_s together enables signature verification, where
- - $T_w = \begin{cases} 27/28 \\ 2 \times \text{ChainID} + 35 \\ 2 \times \text{ChainID} + 36 \end{cases}$
- - $(T_r, T_s) = (c_1, c_2) = \text{Sign}(\text{msg}_{\text{txn}}, \text{sk})$

Message for Signature Generation

- ▶ Define $L_S(T)$ as follows:
- ▶ $L_S(T) =$
  $\begin{cases} T_n \| T_p \| T_g \| T_t \| T_v \| T_d, & \text{if } T_w \in \{27, 28\} \\ T_n \| T_p \| T_g \| T_t \| T_v \| T_d \| \text{ChaidID}, & \text{otherwise} \end{cases}$
- ▶ Set $\text{msg}_{\text{txn}} = \text{Keccak256}(L_S(T))$

- ChainID
  Ethereum mainnet - 1
  Ropsten - 3
  Rinkby - 4
  Kovan - 42

ECDSA

- **secp256k1 curve parameters**
  - $p : 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$
  - $E : y^2 = x^3 + 7$, defined over $\mathbb{Z}_p$
  - $G = (x, y) \in \mathcal{G}_E$ is the base point, where $x =$

    79be667e f9dcbbac 55a06295 ce870b07
    029bfcdb 2dce28d9 59f2815b 16f81798

    and $y =$

    483ada77 26a3c465 5da4fbfc 0e1108a8
    fd17b448 a6855419 9c47d08f fb10d4b8
  - order$(G) = q$ is a prime, where $q =$

    FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
    BAAEDCE6 AF48A03B BFD25E8C D0364141
- **KeyGen**
  - Pick $a \xleftarrow{\$} \mathbb{Z}_q$. Compute $H = aG$
  - Set $\begin{cases} \text{pk} = (E, p, q, G, H) \\ \text{sk} = (a, \text{pk}) \end{cases}$
- **Sign**(msg.txt, sk)
1. Compute $h = \text{hash}(\text{msg.txt})$
2. Pick $r \xleftarrow{\$} \mathbb{Z}_q$. Compute rG.
3. Let $rG = (u, v)$, where $u, v \in \mathbb{Z}_p$
4. $c_1 = u \bmod q$ (If $c_1 = 0$, goto (2))
5. $c_2 = r^{-1}(h + ac_1) \bmod q$ (If $c_2 = 0$, goto (2))
6. Output signature $\sigma = (c_1, c_2)$

- **Verify**$(msg.txt, \sigma = (c_1, c_2), \text{pk})$
  - Compute $h = \text{hash}(\text{msg.txt})$
  - Compute $e_1 = hc_2^{-1} \bmod q$
  - Compute $e_2 = c_1 c_2^{-1} \bmod q$
  - Compute $J = e_1 G + e_2 H$
  - Let $J = (u, v)$
  - Output **valid** $\iff u \bmod q = c_1$

Going beyond simple Ether transfers

- An example case in Bitcoin: How do we lock Bitcoins to a joint ownership ?
- In the following, a user C is making a transaction $T_{C \to A,B}$ to lock x ฿to the join ownership of A and B

txn	Input		Output	
	txid	scriptSig	UTXO	scriptPubkey
$T_{C \to A,B}$			x ฿	2 $\langle PK_A \rangle \langle PK_B \rangle$ 2 OP_CHECKMULTISIGVERIFY

- Approval of $T_{C \to A,B}$ leads to the creation of the following (**key**,**value**) pair entry in the Chainstate

key	**value**
⋮	⋮
Hash($T_{C \to A,B}$)	[x, 2 $\langle PK_A \rangle \langle PK_B \rangle$ 2 OP_CHECKMULTISIGVERIFY]

- We can think of of the above (**key**,**value**) entry as a **Joint Account** in Bitcoin, with a balance of x ฿. The account is governed by the following smart contract, written using Bitcoin's Script Language,

$$2 \ \langle PK_A \rangle \langle PK_B \rangle \ 2 \ \ OP\_CHECKMULTISIGVERIFY$$

- Spending of the balance from this account can be done in the following way

txn	Input		Output	
	txid	scriptSig	UTXO	scriptPubkey
$T_{A,B \to D}$	Hash($T_{C \to A,B}$)	$\langle Sign_{sk_A}(T_{A,B \to D}) \rangle \ \langle Sign_{sk_B}(T_{A,B \to D}) \rangle$	x ฿	OP_DUP OP_HASH160 $\langle$HASH160(pk_D)$\rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY

- To validate $T_{A,B \to D}$, the smart contract of the join account is run on scriptSig of $T_{A,B \to D}$ as an input.

Going beyond simple Ether transfers

- Can we hope to do the same in Ethereum ??
- Creating a simple "externally owned account" will not do.
- We should have a mechanism to open a different type of account to achieve this!

- Let us call this new type as **Contract Accounts** !!
- Clearly, we should be providing some storage for this type of account, to store a smart contract which will control the account!
- The storage can also be allowed to store and manage some data!
- A few clarifications?
- What do the (**key**,**value**) pairs, representing contract accounts, look like ?
- Who can open a contract account?

- We can now address the issue of join ownership of some funds in Ethereum as follows
- In our discussion, we saw that a user C wants to put some fund under the joint ownership of A and B
- Let us assume, EOA_A, EOA_B and EOA_C are externally owned accounts controlled by pk_A, pk_B and pk_C respectively
- A transaction can be made from EOA_C to create a contract account with a balance of x **Ether** and a smart contract that enforce a joint ownership (between pk_A and pk_B) on the balance!

Contract Accounts in Ethereum

- EOA_C makes the transaction the following transaction

Field	Value
[nonce]	$T_n = \text{EOC}_C$.state.nonce
[gasPrice]	T_p
[gasLimit]	T_g
[to]	$T_t = 0x00\cdots00$
[value]	$T_v =$ x Ether
[data]	$T_d =$ Smart Contract
[v]	T_w
[(r,s)]	(T_r, T_s)

- Approval of this transaction leads to the creation of the following (**key**,**value**) pair in the Chainstate

key	**value**
⋮	⋮
Address	nonce = 0 balance = x Ether storageRoot = Root hash of a MPT (storage) codeHash = Hash of EVM code (smart contract)

where Address of the new account is defined as being the rightmost 160 bits of the Keccak256 hash of the RLP encoding of the structure containing the sender address EOA_C and EOA_C.state.nonce.

Table of Contents

1 Lecture#1 (18/01) [Course Overview, Bitcoin Introduction]

2 Lecture#2 (21/01) [Hash Functions]

3 Lecture#3 (25/01) [Random Oracle Model, Iterated Hash Construction]

4 Lecture#4 (28/01) [Hash Puzzle, Signature Scheme]

5 Lecture#5 (01/02) [Group Theory]

6 Lecture#6 (04/02) [Discrete Logarithm Problem, DSA]

7 Lecture#7 (04/02) [Elliptic Curves, ECDSA]

8 Lecture#8 (11/02) [Bitcoin Address Generation]

9 Lecture#9 (15/02) [Bitcoin Transaction and its Validation]

10 Lecture#10 (18/02) [Bitcoin Consensus]

11 Lecture#11 (22/02) [Block Mining, txn Serialization]

12 Lecture#12 (25/02) [txn Serialization, PaytoScriptHash]

13 Lecture#13 (01/03) [More lockScripts, txn-malleability]

14 Lecture#14 (04/03) [SegWit transaction]

15 Lecture#15 (15/03) [Analysis of Bitcoin's Consensus]

16 Lecture#16 (21/03) [Hyperledger Fabric]

17 Lecture#17 (25/04) [Ethereum]

18 Lecture#18 (29/03) [Ethereum Contract Accounts]

19 Lecture#19 (05/04) [Payment Channels, Coin Mixers]

20 Lecture#20 (08/04) [Coin Mixers, MixEth, TumbleBit]

21 Lecture#21 (19/04) [Tokens]

ashoka UNIVERSITY

txn Scalability on Blockchain

- Although blockchain applications have achieved record growth, their ability to scale and increase transaction capacity is fundamentally at odds with their approach to security through wide replication.
- As each transaction that is processed via the network has to be stored on the blockchain, there is a fundamental limit on how many transactions can be processed per second.
- For instance, in Bitcoin with its 1MB block size, and a block creation rate of approx. 10 minutes, the network is currently (quoting from 2016) limited to process up to 7 transactions per second.
- In contrast, centralised payment systems, e.g., the VISA network processes during peak times more than 50,000 transactions per second.
- A natural solution that offers to increase the block size, or the block creation rate will run into several problems.
- The scalability problem is further amplified by "micro-transactions" (low value transactions), which is one of the most "sought-after applications" when blockchain-based currencies go mainstream.
- In some applications, micro-transactions require instant approval.

- **Payment channels** are one of the prominent tool to improve scalability of blockchain-based currencies including Bitcoin.
- They allow two users to rapidly exchange money between each other without sending transactions to the blockchain.
- This is achieved by keeping the massive bulk of transactions off-chain, and using the blockchain only when parties involved in the payment channel disagree, or when they want to close the channel.
- Because off-chain transactions can always be fairly settled by the users via the blockchain, there is no incentive for them to disagree, and hence honest behaviour is enforced.
- Payment channels reduce transaction fees, allow for instantaneous payments and limit the load put on the blockchain.

ashoka UNIVERSITY

Eltoo: A Payment Channel Protocol for Bitcoin

- Eltoo (`https://blockstream.com/eltoo.pdf`) is a state of the art payment channel protocol.
- It forms a one directional payment channel between a sender A and a receiver B.

- Eltoo consists of a three phases
  - setup phase,
  - negotiation phase and
  - settlement phase.
- The setup phase involves moving some funds into an address on Blockchain controlled by all participants. The setup phase is executed on Blockchain.
- The negotiation phase is the core of the protocols and consists of repeated adjustments on the distribution of funds to participants. All negotiations are executed off Blockchain.
- Finally, the settlement phase simply enforces the agreed upon distribution on the blockchain. The settlement phase is executed on Blockchain.

- Payment channels only create a relationship between two parties. Requiring everyone to create channels with everyone else does not solve the scalability problem.
- The scalability can be achieved using a large network of payment channels, called *payment network*, which enables users to route transactions via intermediary hubs. The Lightning Network is one example `https://lightning.network/lightning-network-paper.pdf`
- A further generalisation of payment channels are *state channels*, which radically enrich the functionality of payment channels. Concretely, the users of a state channel can, besides payments, execute entire complex smart contracts described in form of self-enforcing programs in an off-chain way. Raiden `https://raiden.network/` is probably the most prominent project whose goal is to implement state channels over Ethereum.

Eltoo: Online Version (Simple)

- **Setup Phase**

- Both endpoints A and B generate and exchange two public/private key-pairs:
- A settlement key pair (pk_{A_s} and pk_{B_s}) used to spend the funds to a settlement transaction and thus terminate the protocol.
- An update key-pair (pk_{A_u} and pk_{B_u}) used to replace a previous update, continuing the protocol.

txn	Input			Output
	txid	ScriptSig	UTXO	ScriptPubkey
T_{u0}			x	OP_IF 10 OP_CSV 2 $\langle pk_{A_s} \rangle$ $\langle pk_{B_s} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ELSE 2 $\langle pk_{A_u} \rangle$ $\langle pk_{B_u} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ENDIF
T_{s0}	$H(T_{u0})$	$\langle Sign(T_{s0}, sk_{A_s}) \rangle$ $\langle Sign(T_{s0}, sk_{B_s}) \rangle$ OP_TRUE	x	OP_DUP OP_HASH160 $\langle HASH160(pk_A) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY

- The OP_CSV in the IF branch creates a timeout during which only the ELSE branch is valid, giving precedence to update transactions, and only allowing settlement transactions after the timeout expires.

Eltoo: Online Version (Simple)

- **Negotiation Phase**

txn	Input		Output	
	txid	ScriptSig	UTXO	ScriptPubkey
T_{u0}			x	OP_IF 10 OP_CSV 2 $\langle pk_{A_s} \rangle$ $\langle pk_{B_s} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ELSE 2 $\langle pk_{A_u} \rangle$ $\langle pk_{B_u} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ENDIF
T_{s0}	$H(T_{u0})$	$\langle \text{Sign}(T_{s0}, sk_{A_s}) \rangle$ $\langle \text{Sign}(T_{s0}, sk_{B_s}) \rangle$ OP_TRUE	x	OP_DUP OP_HASH160 $\langle \text{HASH160}(pk_A) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
T_{u1}	$H(T_{u0})$	$\langle \text{Sign}(T_{u1}, sk_{A_u}) \rangle$ $\langle \text{Sign}(T_{u1}, sk_{B_u}) \rangle$ OP_FALSE	x	OP_IF 10 OP_CSV 2 $\langle pk_{A_s} \rangle$ $\langle pk_{B_s} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ELSE 2 $\langle pk_{A_u} \rangle$ $\langle pk_{B_u} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ENDIF
T_{s1}	$H(T_{u1})$	$\langle \text{Sign}(T_{s1}, sk_{A_s}) \rangle$ $\langle \text{Sign}(T_{s1}, sk_{B_s}) \rangle$ OP_TRUE	x-1 1	OP_DUP OP_HASH160 $\langle \text{HASH160}(pk_A) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY OP_DUP OP_HASH160 $\langle \text{HASH160}(pk_B) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY

Eltoo: Online Version (Simple)

- **Negotiation Phase**

txn	Input		Output	
	txid	ScriptSig	UTXO	ScriptPubkey
T_{u0}			x	OP_IF 10 OP_CSV 2 $\langle \text{pk}_{A_s} \rangle$ $\langle \text{pk}_{B_s} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ELSE 2 $\langle \text{pk}_{A_u} \rangle$ $\langle \text{pk}_{B_u} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ENDIF
T_{s0}	$H(T_{u0})$	$\langle \text{Sign}(T_{s0}, \text{sk}_{A_s}) \rangle$ $\langle \text{Sign}(T_{s0}, \text{sk}_{B_s}) \rangle$ OP_TRUE	x	OP_DUP OP_HASH160 $\langle \text{HASH160}(\text{pk}_\text{A}) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
T_{u1}	$H(T_{u0})$	$\langle \text{Sign}(T_{u1}, \text{sk}_{A_u}) \rangle$ $\langle \text{Sign}(T_{u1}, \text{sk}_{B_u}) \rangle$ OP_FALSE	x	OP_IF 10 OP_CSV 2 $\langle \text{pk}_{A_s} \rangle$ $\langle \text{pk}_{B_s} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ELSE 2 $\langle \text{pk}_{A_u} \rangle$ $\langle \text{pk}_{B_u} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ENDIF
T_{s1}	$H(T_{u1})$	$\langle \text{Sign}(T_{s1}, \text{sk}_{A_s}) \rangle$ $\langle \text{Sign}(T_{s1}, \text{sk}_{B_s}) \rangle$ OP_TRUE	x-1 1	OP_DUP OP_HASH160 $\langle \text{HASH160}(\text{pk}_\text{A}) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY OP_DUP OP_HASH160 $\langle \text{HASH160}(\text{pk}_\text{B}) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
T_{u2}	$H(T_{u1})$	$\langle \text{Sign}(T_{u2}, \text{sk}_{A_u}) \rangle$ $\langle \text{Sign}(T_{u2}, \text{sk}_{B_u}) \rangle$ OP_FALSE	x	OP_IF 10 OP_CSV 2 $\langle \text{pk}_{A_s} \rangle$ $\langle \text{pk}_{B_s} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ELSE 2 $\langle \text{pk}_{A_u} \rangle$ $\langle \text{pk}_{B_u} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ENDIF
T_{s2}	$H(T_{u2})$	$\langle \text{Sign}(T_{s2}, \text{sk}_{A_s}) \rangle$ $\langle \text{Sign}(T_{s2}, \text{sk}_{B_s}) \rangle$ OP_TRUE	x-2 2	OP_DUP OP_HASH160 $\langle \text{HASH160}(\text{pk}_\text{A}) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY OP_DUP OP_HASH160 $\langle \text{HASH160}(\text{pk}_\text{B}) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY

Eltoo Offline: Setup Phase

txn	Input		Output	
	txid	ScriptSig	UTXO	ScriptPubkey
T_{u0}			x	OP_IF 10 OP_CSV 2 $\langle \text{pk}_{A_{s0}} \rangle$ $\langle \text{pk}_{B_{s0}} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ELSE $\langle S_0 + 1 \rangle$ OP_CHECKLOCKTIMEVERIFY 2 $\langle \text{pk}_{A_u} \rangle$ $\langle \text{pk}_{B_u} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ENDIF
T_{s0}	$H(T_{u0})$	$\langle \text{Sign}(T_{u0}, \text{sk}_{A_{s0}}) \rangle$ $\langle \text{Sign}(T_{u0}, \text{sk}_{B_{s0}}) \rangle$ OP_TRUE	x	OP_DUP OP_HASH160 $\langle \text{HASH160}(\text{pk}_\text{A}) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY

1. A creates funding transaction T_{u0} and share it (unsigned T_{u0}) with B
2. Before A broadcasts T_{u0}, A requires B to create and share T_{s0} (signed)
3. Finally A broadcasts T_{u0} to the Bitcoin network

Eltoo Offline: Negotiation Phase

txn	Input		Output	
	txid	ScriptSig	UTXO	ScriptPubkey
T_{u0}			x	OP_IF 10 OP_CSV 2 $\langle pk_{A_{s0}} \rangle$ $\langle pk_{B_{s0}} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ELSE $\langle S_0 + 1 \rangle$ OP_CHECKLOCKTIMEVERIFY 2 $\langle pk_{A_u} \rangle$ $\langle pk_{B_u} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ENDIF
T_{s0}	$H(T_{u0})$	$\langle Sign(T_{u0}, sk_{A_{s0}}) \rangle$ $\langle Sign(T_{u0}, sk_{B_{s0}}) \rangle$ OP_TRUE	x	OP_DUP OP_HASH160 $\langle HASH160(pk_A) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
1. A creates funding transaction T_{u0} and share it (unsigned T_{u0}) with B 2. Before A broadcasts T_{u0}, A requires B to create and share T_{s0} (signed) 3. Finally A broadcasts T_{u0} to the Bitcoin network				
Negotiation Phase				
T_{u1}	NA	$\langle Sign(T'_{u1}, sk_{A_u}) \rangle$ $\langle Sign(T'_{u1}, sk_{B_u}) \rangle$ OP_FALSE Note 1. $T'_{u1} = [T_{u1}.Input \backslash \{txnref\} \| T_{u1}.output]$ 2. nLocktime $= S_1 > S_0 + 1$	x	OP_IF 10 OP_CSV 2 $\langle pk_{A_{s1}} \rangle$ $\langle pk_{B_{s1}} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ELSE $\langle S_1 + 1 \rangle$ OP_CHECKLOCKTIMEVERIFY 2 $\langle pk_{A_u} \rangle$ $\langle pk_{B_u} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ENDIF
T_{s1}	NA	$\langle Sign(T'_{s1}, sk_{A_{s1}}) \rangle$ $\langle Sign(T'_{s1}, sk_{B_{s1}}) \rangle$ OP_TRUE Note $T'_{s1} = [T_{s1}.Input \backslash \{txnref\} \| T_{s1}.output]$	x-1 1	OP_DUP OP_HASH160 $\langle HASH160(pk_A) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY OP_DUP OP_HASH160 $\langle HASH160(pk_B) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
1. A and B collaborate to create transaction T_{u1}. Before signing T_{u1}, they first negotiate T_{s1} 2. Both T_{u1} and T_{s1} are not broadcasted, but instead stored for possible use during the settlement phase				

Eltoo Offline: Negotiation Phase

txn	Input		Output	
	txid	ScriptSig	UTXO	ScriptPubkey
Negotiation Phase				
T_{u1}	NA	$\langle \text{Sign}(T'_{u1}, \text{sk}_{A_u}) \rangle \ \langle \text{Sign}(T'_{u1}, \text{sk}_{B_u}) \rangle$ OP_FALSE Note 1. $T'_{u1} = [T_{u1}.\text{Input} \backslash \{\text{txnref}\} \| \ T_{u1}.\text{output}]$ 2. nLocktime $= S_1 > S_0 + 1$	x	OP_IF 10 OP_CSV 2 $\langle \text{pk}_{A_{s1}} \rangle \ \langle \text{pk}_{B_{s1}} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ELSE $\langle S_1 + 1 \rangle$ OP_CHECKLOCKTIMEVERIFY 2 $\langle \text{pk}_{A_u} \rangle \ \langle \text{pk}_{B_u} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ENDIF
T_{s1}	NA	$\langle \text{Sign}(T'_{s1}, \text{sk}_{A_{s1}}) \rangle \ \langle \text{Sign}(T'_{s1}, \text{sk}_{B_{s1}}) \rangle$ OP_TRUE Note $T'_{s1} = [T_{s1}.\text{Input} \backslash \{\text{txnref}\} \| \ T_{s1}.\text{output}]$	x-1 1	OP_DUP OP_HASH160 $\langle \text{HASH160}(\text{pk}_\text{A}) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY OP_DUP OP_HASH160 $\langle \text{HASH160}(\text{pk}_\text{B}) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
1. A and B collaborate to create transaction T_{u1}. Before signing T_{u1}, they first negotiate T_{s1} 2. Both T_{u1} and T_{s1} are not broadcasted, but instead stored for possible use during the settlement phase				
T_{u2}	NA	$\langle \text{Sign}(T'_{u2}, \text{sk}_{A_u}) \rangle \ \langle \text{Sign}(T'_{u2}, \text{sk}_{B_u}) \rangle$ OP_FALSE Note 1. $T'_{u2} = [T_{u2}.\text{Input} \backslash \{\text{txnref}\} \| \ T_{u2}.\text{output}]$ 2. nLocktime $= S_2 > S_1 + 1$	x	OP_IF 10 OP_CSV 2 $\langle \text{pk}_{A_{s2}} \rangle \ \langle \text{pk}_{B_{s2}} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ELSE $\langle S_2 + 1 \rangle$ OP_CHECKLOCKTIMEVERIFY 2 $\langle \text{pk}_{A_u} \rangle \ \langle \text{pk}_{B_u} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ENDIF
T_{s2}	NA	$\langle \text{Sign}(T'_{s2}, \text{sk}_{A_{s2}}) \rangle \ \langle \text{Sign}(T'_{s2}, \text{sk}_{B_{s2}}) \rangle$ OP_TRUE Note $T'_{s2} = [T_{s2}.\text{Input} \backslash \{\text{txnref}\} \| \ T_{s2}.\text{output}]$	x-2 2	OP_DUP OP_HASH160 $\langle \text{HASH160}(\text{pk}_\text{A}) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY OP_DUP OP_HASH160 $\langle \text{HASH160}(\text{pk}_\text{B}) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
1. A and B collaborate to create transaction T_{u2}. Before signing T_{u1}, they first negotiate T_{s2} 2. Both T_{u2} and T_{s2} are not broadcasted, but instead stored for possible use during the settlement phase				

Eltoo Offline: Settlement Phase

txn	Input		Output	
	txid	ScriptSig	UTXO	ScriptPubkey
Negotiation Phase				
T_{u1}	NA	$\langle \text{Sign}(T'_{u1}, \text{sk}_{A_u}) \rangle$ $\langle \text{Sign}(T'_{u1}, \text{sk}_{B_u}) \rangle$ OP_FALSE Note 1. $T'_{u1} = [T_{u1}.\text{Input} \backslash \{\text{txnref}\} \| T_{u1}.\text{output}]$ 2. nLocktime $= S_1 > S_0 + 1$	x	OP_IF 10 OP_CSV 2 $\langle \text{pk}_{A_{s1}} \rangle$ $\langle \text{pk}_{B_{s1}} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ELSE $\langle S_1 + 1 \rangle$ OP_CHECKLOCKTIMEVERIFY 2 $\langle \text{pk}_{A_u} \rangle$ $\langle \text{pk}_{B_u} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ENDIF
T_{s1}	NA	$\langle \text{Sign}(T'_{s1}, \text{sk}_{A_{s1}}) \rangle$ $\langle \text{Sign}(T'_{s1}, \text{sk}_{B_{s1}}) \rangle$ OP_TRUE Note $T'_{s1} = [T_{s1}.\text{Input} \backslash \{\text{txnref}\} \| T_{s1}.\text{output}]$	x-1 1	OP_DUP OP_HASH160 $\langle \text{HASH160}(\text{pk}_\text{A}) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY OP_DUP OP_HASH160 $\langle \text{HASH160}(\text{pk}_\text{B}) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
1. A and B collaborate to create transaction T_{u1}. Before signing T_{u1}, they first negotiate T_{s1} 2. Both T_{u1} and T_{s1} are not broadcasted, but instead stored for possible use during the settlement phase				
T_{u2}	NA	$\langle \text{Sign}(T'_{u2}, \text{sk}_{A_u}) \rangle$ $\langle \text{Sign}(T'_{u2}, \text{sk}_{B_u}) \rangle$ OP_FALSE Note 1. $T'_{u2} = [T_{u2}.\text{Input} \backslash \{\text{txnref}\} \| T_{u2}.\text{output}]$ 2. nLocktime $= S_2 > S_1 + 1$	x	OP_IF 10 OP_CSV 2 $\langle \text{pk}_{A_{s2}} \rangle$ $\langle \text{pk}_{B_{s2}} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ELSE $\langle S_2 + 1 \rangle$ OP_CHECKLOCKTIMEVERIFY 2 $\langle \text{pk}_{A_u} \rangle$ $\langle \text{pk}_{B_u} \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ENDIF
T_{s2}	NA	$\langle \text{Sign}(T'_{s2}, \text{sk}_{A_{s2}}) \rangle$ $\langle \text{Sign}(T'_{s2}, \text{sk}_{B_{s2}}) \rangle$ OP_TRUE Note $T'_{s2} = [T_{s2}.\text{Input} \backslash \{\text{txnref}\} \| T_{s2}.\text{output}]$	x-2 2	OP_DUP OP_HASH160 $\langle \text{HASH160}(\text{pk}_\text{A}) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY OP_DUP OP_HASH160 $\langle \text{HASH160}(\text{pk}_\text{B}) \rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
1. A and B collaborate to create transaction T_{u2}. Before signing T_{u1}, they first negotiate T_{s2} 2. Both T_{u2} and T_{s2} are not broadcasted, but instead stored for possible use during the settlement phase				
Settlement Phase				
1. A/B broadcasts T_{u2} to the Bitcoin network, referring to Hash(T_{u0}) 2. A/B broadcasts T_{s2} to the Bitcoin network, referring to Hash(T_{u2})				

Table of Contents

1 Lecture#1 (18/01) [Course Overview, Bitcoin Introduction]

2 Lecture#2 (21/01) [Hash Functions]

3 Lecture#3 (25/01) [Random Oracle Model, Iterated Hash Construction]

4 Lecture#4 (28/01) [Hash Puzzle, Signature Scheme]

5 Lecture#5 (01/02) [Group Theory]

6 Lecture#6 (04/02) [Discrete Logarithm Problem, DSA]

7 Lecture#7 (04/02) [Elliptic Curves, ECDSA]

8 Lecture#8 (11/02) [Bitcoin Address Generation]

9 Lecture#9 (15/02) [Bitcoin Transaction and its Validation]

10 Lecture#10 (18/02) [Bitcoin Consensus]

11 Lecture#11 (22/02) [Block Mining, txn Serialization]

12 Lecture#12 (25/02) [txn Serialization, PaytoScriptHash]

13 Lecture#13 (01/03) [More lockScripts, txn-malleability]

14 Lecture#14 (04/03) [SegWit transaction]

15 Lecture#15 (15/03) [Analysis of Bitcoin's Consensus]

16 Lecture#16 (21/03) [Hyperledger Fabric]

17 Lecture#17 (25/04) [Ethereum]

18 Lecture#18 (29/03) [Ethereum Contract Accounts]

19 Lecture#19 (05/04) [Payment Channels, Coin Mixers]

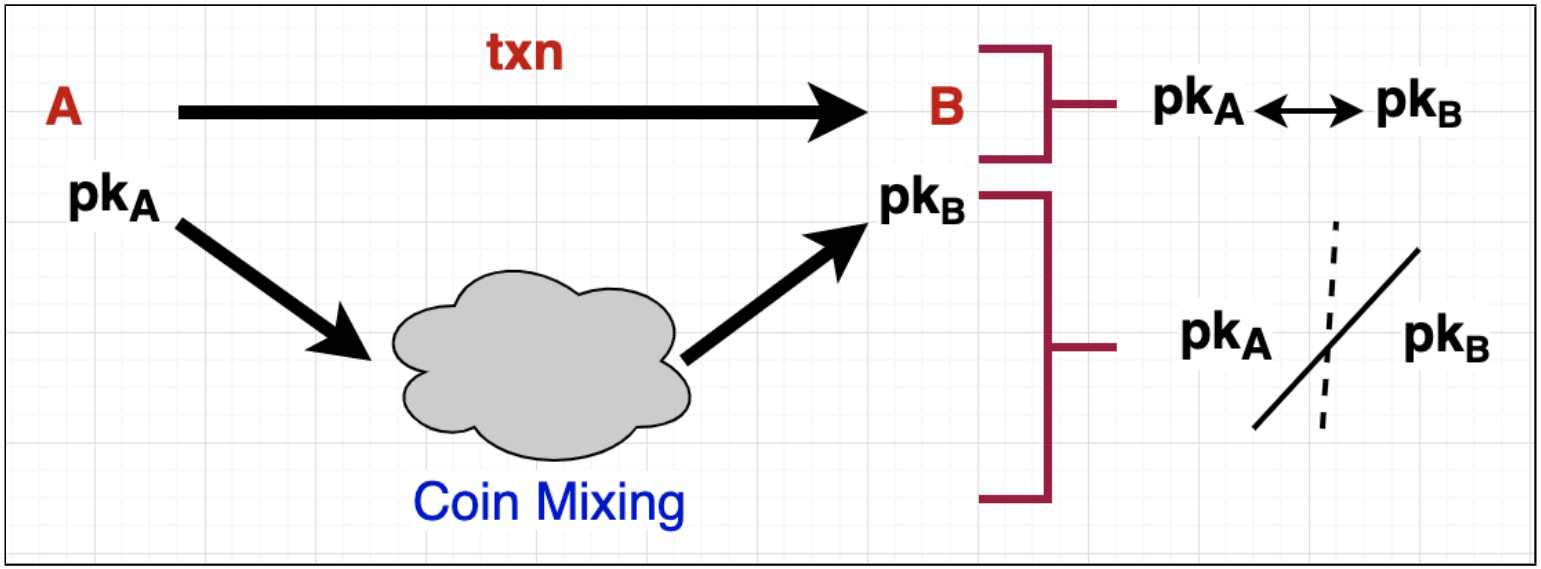
20 Lecture#20 (08/04) [Coin Mixers, MixEth, TumbleBit]

21 Lecture#21 (19/04) [Tokens]

ashoka UNIVERSITY

Anonymity in Bitcoin

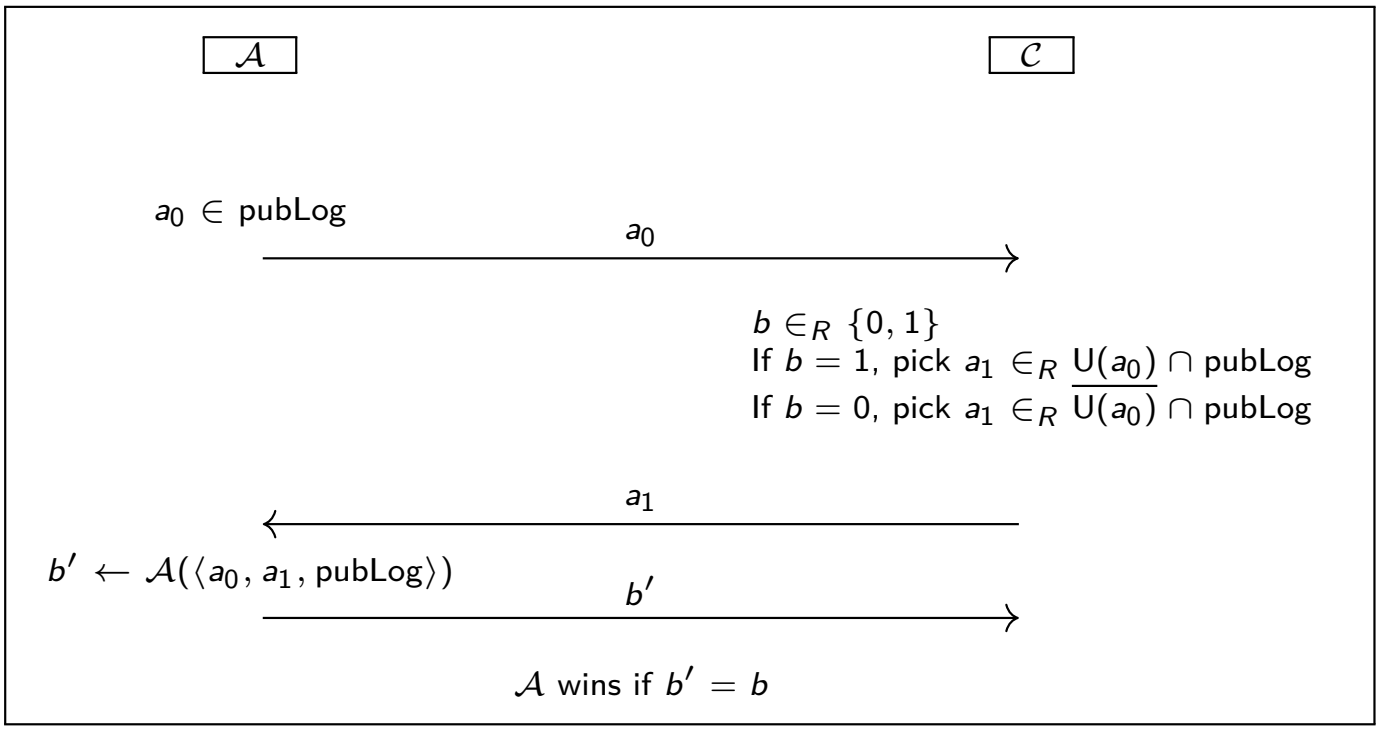
- The temporary nature of users' pseudonymous identities in Bitcoin played a crucial role in its early success.
- However, a decade of intense scrutiny by privacy researchers has provided powerful heuristics using which attackers can effectively link disparate Bitcoin transactions to a common user and, in many cases, to that user's real-world identity.
- This has led to the rise of mixing services which promise to take a user's coins and randomly exchange them for other users' coins to obfuscate their ownership.
- In the common implementation a mixing address receives coins from multiple clients and forwards them randomly to a fresh address for each client.



Coin Mixing

- **Coin Mixer** A coin mixer service has three types of users:
- a tumbler T,
- a list of payers $A_1, \ldots, A_k$, and
- a list of payees $B_1, \ldots, B_k$.
- The mixer operates in epochs.

- **Security: k-anonymity** The coin mixing service is said to satisfy k-anonymity within an epoch, if the following holds: While the blockchain reveals which payers and payees participated in an epoch, no one - not even the tumbler T, can tell which payer paid which payee during that specific epoch.
- More formally, it can be defined using the Address Unlinkability Game: **AddUnL**



- We define $\mathcal{A}$'s advantage in winning **AddUnL** game as follows:

$$\mathrm{Adv}^{\mathcal{A}}_{\mathrm{AddUnL}} = |\mathbb{P}[b' = b] - \frac{1}{2}|$$

- The currency satisfies address unlinkability if for all ppt adversaries $\mathcal{A}$ and for all (a_0, a_1), $\mathrm{Adv}^{\mathcal{A}}_{\mathrm{AddUnL}} \leq \epsilon$, where ϵ is negligible.

MixEth

- MixEth: A coin mixing service for Ethereum
- Three distinct entities in MixEth:
  - A MixEth coin mixing smart contract
  - A set of senders $\{\text{pk}_{A_1}, \ldots, \text{pk}_{A_n}\}$
  - A set of receivers $\{\text{pk}_{B_1}, \ldots, \text{pk}_{B_n}\}$
  - Assumption: each user A_i wants to transfer the same amount of Ether, say 1 Ether, to each B_i

1. **Deposit**

- Each A_i transfers 1 Ether to the MixEth contract.
- Each A_i submits pk_{B_i} to the contract

---

2. **Shuffle**

$$
\begin{aligned}
& (\text{pk}_{B_1}, \ldots, \text{pk}_{B_n}) \\
\xrightarrow{\text{Shuffle}} \quad & (c \cdot \text{pk}_{B_{\pi(1)}}, \ldots, c \cdot \text{pk}_{B_{\pi(n)}}) \\
= \quad & (c \cdot \text{sk}_{B_{\pi(1)}} G, \ldots, c \cdot \text{sk}_{B_{\pi(n)}} G)
\end{aligned}
$$

where π is a random permutation on $\{1, \ldots, n\}$ and c is a secret shuffling constant. π and c are kept secret

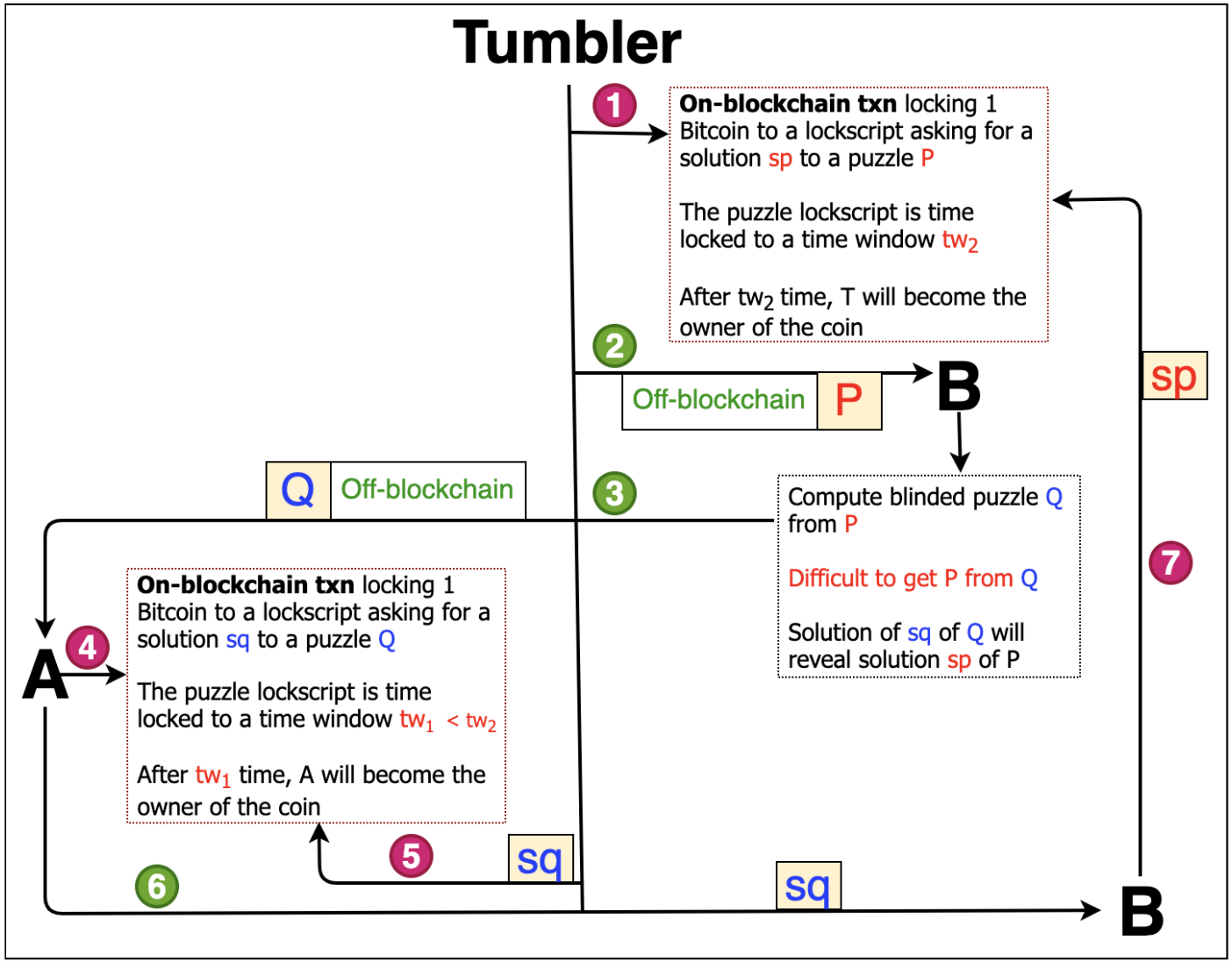
- The following is made available: $G' = c\,G$
- We must have a way to check if the shuffling was done correctly!!

3. **Withdraw**: This is done in two steps - by a receiver, say B_1, to withdraw 1 Ether.

- Step1. B_1 submits a signature $\sigma = \text{Sign}_{\text{sk}_{B_1}}(\text{msg})$ (with group generator as G') and $\text{msg} = \text{sk}_{B_1} G'$.
- Step2. Contract checks if $\text{sk}_{B_1} G' \in \text{Shuffle}(\{\text{pk}_{B_1}, \ldots, \text{pk}_{B_n}\})$?
- It also checks if $\text{Verify}_{\text{sk}_{B_1} G'}(\sigma, \text{msg}) = 1$ (with group generator as G').
- The contract finally transfers and locks 1 Ether to the public key $\text{sk}_{B_1} G'$

- The public key $\text{sk}_{B_1} G'$ does not reveal link to pk_{B_1}
- B_1 can spend this money provided Ethereum allows signature being generated using arbitrary group generators

Coin Mixing: TumbleBit



TumbleBit

txn	Input		Output	
	txid	ScriptSig	UTXO	ScriptPubkey
T Escrows Fund				
T_1			1	OP_IF 2 $\langle pk_T \rangle$ $\langle pk_B \rangle$ 2 OP_CEHCKMULTISIGVERIFY OP_ELSE $\langle$LockTime$\rangle$ OP_CHECKLOCKTIMEVERIFY OP_DROP OP_DUP OP_HASH160 $\langle$HASH160(pk_T)$\rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY OP_ENDIF
T_2	$H(T_1)$	$\langle \sigma^T_{T_2} = [(c, z)] \rangle$ $\langle \sigma^B_{T_2} = [\,] \rangle$	1	OP_DUP OP_HASH160 $\langle$HASH160(pk_B)$\rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY

1. T broadcasts T_1 (On Blockchain)
2. T then computes the following steps locally:
- Compute $\sigma^{**} = \text{Sign}(T_2, sk_T)$
- Pick a random key K
- Compute $c = \text{Enc}_K(\sigma^{**})$
- Compute $z = K^e \pmod N$
3. T finally share the T_2 with B (Off Blockchain)

---

4. To get 1 Bitcoin, B must compute the ScriptSig for the input of T_2 as follows
- $\langle \sigma^T_{T_2} = \sigma^{**} = \text{Sign}(T_2, sk_T) \rangle$ $\langle \sigma^B_{T_2} = \text{Sign}(T_2, sk_B) \rangle$, where B must obtain σ^{**} from (c, z)
5. Before, B **must verify that the tuple** (c, z) **is correctly formed!!**

B Shares the Blinded Puzzle with A

1. B has (c, z)
2. It picks a random r and computes $\bar{z} = r^e z \pmod N$
3. Finally, B shares $\bar{z}$ with A (Off Blockchain)

txn	Input		Output	
	txid	ScriptSig	UTXO	ScriptPubkey
A Pays to T				
1. A submits $\bar{z}$ to T (Off Blockchain)				
1. On receiving $\bar{z}$ from A, T computes the following steps - Compute $\Sigma = \bar{z}^d \pmod N$ - Pick a random key k - Compute $C = \text{Enc}_k(\Sigma)$ - Compute $H = \text{Hash}(k)$ 2. Finally T submits the tuple (C, H) to A (Off Blockchain)				
1. On receiving the tuple (C, H), A **must verify that** (C, H) **is correctly formed!!** 2. Finally, A creates and broadcasts the following transaction (On Blockchain)				
$T3$			1	OP_IF OP_HASH160 $\langle H \rangle$ OP_EQUALVERIFY OP_ELSE $\langle$LockTime$\rangle$ OP_CHECKLOCKTIMEVERIFY OP_DROP OP_DUP OP_HASH160 $\langle$HASH160(pk_A)$\rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY OP_ENDIF
1. T broad the following transaction T_4 to receive 1 Bitcoin				
$T4$	H(T3)	k	1	OP_DUP OP_HASH160 $\langle$HASH160(pk_A)$\rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY
A Shares Blinded Puzzle Solution with B				
1. k is now available for A to see. A then computes $\Sigma = \text{Dec}_k(C)$				
1. A then shares the Σ with B (Off Blockchain) - Note that $\Sigma = \bar{z}^d = (r^e z)^d = (r^e K^e)^d = rK \pmod N$ 2. B computes $K = r^{-1}\Sigma \pmod N$, and consequently $\sigma^{**} = \text{Sign}(T_2, \text{sk}_T) \leftarrow \text{Dec}_K(c)$ 3. B then finally broadcasts T_2 to acquire 1 ฿ (On Blockchain)				
B Consumes the Escrow Fund				
T_2	$H(T_1)$	$\langle \sigma^T_{T_2} = \text{Sign}(T_2, \text{sk}_T) \rangle \langle \sigma^B_{T_2} = \text{Sign}(T_2, \text{sk}_B) \rangle$	1	OP_DUP OP_HASH160 $\langle$HASH160(pk_B)$\rangle$ OP_EQUALVERIFY OP_CHECKSIGVERIFY

Table of Contents

1. Lecture#1 (18/01) [Course Overview, Bitcoin Introduction]
2. Lecture#2 (21/01) [Hash Functions]
3. Lecture#3 (25/01) [Random Oracle Model, Iterated Hash Construction]
4. Lecture#4 (28/01) [Hash Puzzle, Signature Scheme]
5. Lecture#5 (01/02) [Group Theory]
6. Lecture#6 (04/02) [Discrete Logarithm Problem, DSA]
7. Lecture#7 (04/02) [Elliptic Curves, ECDSA]
8. Lecture#8 (11/02) [Bitcoin Address Generation]
9. Lecture#9 (15/02) [Bitcoin Transaction and its Validation]
10. Lecture#10 (18/02) [Bitcoin Consensus]
11. Lecture#11 (22/02) [Block Mining, txn Serialization]
12. Lecture#12 (25/02) [txn Serialization, PaytoScriptHash]
13. Lecture#13 (01/03) [More lockScripts, txn-malleability]
14. Lecture#14 (04/03) [SegWit transaction]
15. Lecture#15 (15/03) [Analysis of Bitcoin's Consensus]
16. Lecture#16 (21/03) [Hyperledger Fabric]
17. Lecture#17 (25/04) [Ethereum]
18. Lecture#18 (29/03) [Ethereum Contract Accounts]
19. Lecture#19 (05/04) [Payment Channels, Coin Mixers]
20. Lecture#20 (08/04) [Coin Mixers, MixEth, TumbleBit]
21. Lecture#21 (19/04) [Tokens]

ashoka UNIVERSITY

Solidity Basics

`bool`	Boolean value, `true` or `false`
Integer	
`intk`	Holds a k-bit signed integer s.t. $8 \leq k \leq 256$ and k is a multiple of 8
`uintk`	Holds a k-bit unsigned integer s.t. $8 \leq k \leq 256$ and k is a multiple of 8
Fixed Point	
`fixedmxn`	holds a decimal value - integer part has m bits where $8 \leq m \leq 256$, decimal part has n bits with $n \leq 18$
`ufixedmxn`	
Address	
`address`	Holds a 160-bit Ethereum address.
Byte Array	
`bytesk`	Fixed size array of bytes, $1 \leq k \leq 32$
`bytes/string`	Variable sized array of bytes
Enum	User-defined type for enumerating discrete values `enum NAME` {LABEL1, LABEL2, . . .}
Array	An array of any type, either fixed or dynamic
`uint32[][3]`	3 dynamic arrays of unsigned integers
Struct	User-defined data containers for grouping variables `struct` NAME {Type1 Variable1; Type2 Variable2; . . .}
Mapping	Hash lookup tables for (key, value) pairs
`mapping(KeyType, ValueType) NAME`	

Solidity Basics

msg Object	= { EOA originated txn call / Contract originated msg call } that launched this contract execution
msg.sender	Represents the address that initiated this call
msg.value	The value of ether sent with this call
msg.sig	The first four bytes of the data payload
txn Object	
txn.gasprice	the gas price in the calling transaction
block Object	contains information about the current block
block.coinbase	the address of the recipient of the current block's fee and block reward
block.difficulty	the difficulty of the current block
block.number	the current block number
block.gaslimit	the maximum amount of gas that can be spent across all txns included in the current block
address Object	Any address, either passed as an input or cast from a contract object, has a number of attributed and methodes
address.balance	the balance of the address in wei
address(this).balance	the current contract balance
address.transfer(amount)	transfers the amount to this adress
Functions	
function Name([parameters]) {public/private/internal/external} [pure/constant/view/payble] [modifier] [returns (return types)]	
public	default; can be called by other contracts or EOA txns, or from within the contract
external	like public, except they cannot be called from within the contract unless explicitly prefixed with the keyword this
internal	they are only accessible from within the contract. They can be called by derived contracts (those that inherit this one)
private	internal function, except they cannot be called by derived contracts
constant/view	A function marked as a view promises not to modify any state
pure	A pure function neither reads nor writes any variables in storage. It can only operate on arguments and return data, without reference to any stored data
payable	A payable function can accept incoming payments

Solidity Basics

Contract Constructor and Selfdestruct

There is a special function that is only used once. When a contract is created, it also runs the **constructor function** if one exists, to initialise the state of the contract. The constructor is run in the same txn as the contract creation. The constructor function is optional !!

```
contract NAME {
    constructor () {
        // this is the constructor
    }
}
```

A contract's life cycle starts with a creation txn from an EOA. If there is a constructor, it is executed as part of contract creation, to initialise the state of the contract as it is being created, and is then discarded. The other end of the contract's life cycle is contract destructions. Contracts are destroyed by a special EVM opcode called **selfdestruct**. It takes one argument - the address to receive any ether balance remaining in the contract account: `selfdestruct(address recipient);`

```
contract Faucet {
    address owner;
    constructor () {
        owner = msg.sender;
    }

    function destroy() public {
        require(msg.sender == owner);
        selfdestruct(owner);
    }
}
```

Function Modifier

```
contract Faucet {
    address owner;
    constructor () {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
    }

    function destroy() public onlyOwner {
        require(msg.sender == owner);
        selfdestruct(owner);
    }
}
```

ashoka UNIVERSITY

FabChat on Ethereum

```
contract fabChat {

    mapping(address => string) users;
    address[] usersArray;

    struct msgContent {
        string username;
        string message;
        bool anonym;
        uint256 flag;
        address[] flaggedUsers;
    }

    msgContent[] public msgs;

    function flagMessage(uint256 msgIndex) public{

        bool alreadyFlagged = true;
        for(uint i = 0; i < msgs[msgIndex].flaggedUsers.length; i++){
            if (msgs[msgIndex].flaggedUsers[i] == msg.sender){
                alreadyFlagged = false;
            }
        }
        require(alreadyFlagged, "You have already Flagged");

        msgs[msgIndex].flaggedUsers.push(msg.sender);
        msgs[msgIndex].flag++;

        if(msgs[msgIndex].flag > (usersArray.length / 2)){
            if(msgs[msgIndex].anonym == true){
                msgs[msgIndex].username = users[msgs[msgIndex].flaggedUsers[0]];
                msgs[msgIndex].flag = 1;
            }
            if(msgs[msgIndex].anonym == false){
                delete msgs[msgIndex];
            }
        }
    }
```

ashoka UNIVERSITY

FabChat on Ethereum

```
    function postMessageAnonymous(string memory m, string memory un) public payable {
        require(msg.value >= 1 ether, "You need to pay more than 1 ether");
        // require(usernamesToAddress[un] == true,"Please register before posting a message.");
        require(keccak256(abi.encodePacked((users[msg.sender]))) == keccak256(abi.encodePacked((un))),
        "Please register before posting a message, or use the correct username associated to your account.");
        address[] memory fu;
        msgs.push(msgContent({username: "anonymous", message: m, anonym: true, flag: 0, flaggedUsers: fu}));
        msgs[msgs.length - 1].flaggedUsers.push(msg.sender);
    }


    function postMessageNonAnonymous(string memory m, string memory un) public {
        require(keccak256(abi.encodePacked((users[msg.sender]))) == keccak256(abi.encodePacked((un))),
        "Please register before posting a message, or use the correct username associated to your account");
        // require(usernamesToAddress[un] == true,"Please register before posting a message.");
        address[] memory fu;
        msgs.push(msgContent({username: users[msg.sender], message: m, anonym: false, flag: 0, flaggedUsers: fu}));
        msgs[msgs.length - 1].flaggedUsers.push(msg.sender);
    }
    function register(string memory userName) public {
        bool alreadyRegistered = true;
        for(uint i = 0; i < usersArray.length; i++){
            if (usersArray[i] == msg.sender){
                alreadyRegistered = false;
            }
        }
        require(alreadyRegistered, "You have already registered. You cannot register twice");
        users[msg.sender] = userName;
        usersArray.push(msg.sender);
        // usernamesToAddress[userName] = true;
    }

    function getMessages() public view returns(msgContent[] memory){
        return msgs;
    }
}
```

Tokens

- A token contract is simply an Ethereum smart contract. A token contract is not much more a mapping of addresses to balances, plus some methods to add and subtract from those balances.
- It is these balances that represent the tokens themselves. Someone "has tokens" when their balance in the token contract is non-zero. That's it! These balances could be considered money, experience points in a game, deeds of ownership, or voting rights.
- "Sending tokens" actually means "calling a method on a smart contract that someone wrote and deployed".

Typical Methods a Token Contract should Provide

`totalSupply`	returns the total units of this token that currently exist.
`balanceOf`	given an address, returns the token balance of that address
`transfer`	given an address and amount, transfer that amount of tokens to that address, from the balance of the address that executed the transfer
`transferFrom`	given a sender, recipient, and amount, transfer tokens from one account to another
`approve`	given a recipient address and amount, authorise that address to execute several transfers up to that amount, from the account that issued the approval
`allowance`	given an owner address and a spender address, returns the remaining amount that the spender is approved to withdraw from the owner
`name`	returns the human-readable name of the token
`symbol`	returns a human-readable symbol for the token
`decimal`	returns the number of decimals used to divide token amounts. For example, if decimals is 3, then the token amount is divided by 1000 to gets its user representation

Token Contract Data Structures

An internal table of token balances, by owner. This allows the token contract to keep track of who owns the tokens:
`mapping (address => uint256) balances`

workflow: `transfer` If Alice wants to send 5 tokens to Bob, her wallet sends a transaction to the token contract's address, calling the transfer function with Bob's address and 5 as the argument. The token contract adjusts Alice's balance (-5) and Bob's balance ($+5$) and issues a Transfer event !!

A Simple Token Contract

```
pragma solidity ^0.6.0;


contract ashokaCoin {
    string public constant name = "ashokaCoin";
    string public constant symbol = "AST";
    uint8 public constant decimal = 0;

    // event Transfer(address indexed from, address indexed to, uint tokens);

    mapping(address => uint256) balances;

    uint256 _totalSupply = 5000;

    constructor() public {
        balances[msg.sender] = _totalSupply;

    }

    function totalSupply() public view returns(uint256) {
        return _totalSupply;
    }

    function balanceOf(address tokenOwner) public view returns(uint256) {
        return balances[tokenOwner];
    }

    function transfer(address recipient, uint256 tokenAmount) public returns(bool) {
        require(balances[msg.sender]>tokenAmount);
        balances[msg.sender] -= tokenAmount;
        balances[recipient] += tokenAmount;
        // emit Transfer(msg.sender, recipient, tokenAmount);
        return true;
    }
}
```

The second data structure is a data mapping of allowances. The contract keeps track of the allowances with a two-dimensional mapping, with the primary key being the address of the token owner, mapping to a spender address and an allowance amount

```
mapping (address => mapping (address => uint256)) public allowed
```

workflow: `approve` & `transferFrom` (used for **ICO**)