Class Notes

These are class notes that I am taking based on the lecture recordings for CS 251, as taught by Professor Dan Boneh in Fall 2022. The photos are either drawings I made or taken from the slides for the class.

Table of Contents

Lecture 1 :	Introduction	page 2
Lecture 2 :	Bitcoin Nuts and Bolts	page 8
Lecture 3 :	Bitcoin Nuts and Bolts, II and Wallets	page 14
Lecture 4 :	Consensus, I	page 20
Lecture 5 :	Consensus, II	page 24
Lecture 6 :	Consensus, III	page 30
Lecture 9 :	Stablecoins	page 34
Lecture 10:	Decentralized Exchanges	page 37
Lecture 11:	Lending Systems	page 40
Lecture 12:	Legal Aspects and Regulation	page 45
Lecture 13:	Privacy on the Blockchain	page 48
Lecture 14:	zk-SNARKs	page 54
Lecture 15:	Constructing a SNARK	page 63
Lecture 16:	Scaling the Blockchain, I	page 70
Lecture 17:	Scaling the Blockchain, II	page 74
Lecture 18:	Recursive SNARKs	page 78
Lecture 19:	MEV and Bridging	page 81
Lecture 20:	The Future of Blockchain	page 85

Lecture 1: Introduction

This is a course about blockchain, so the first important question is: what is a blockchain?

An abstract answer is: a blockchain provides coordination between many parties, when there is no singular trusted party.

If a trusted party exists, that party can just coordinate everything, and we don't need a blockchain. But, for example, in finance, there is often no trusted party. So there are many applications for blockchains in finance, but they have interesting uses even outside the financial world, and we will see this soon.

It is important when a new technology comes along to ask the fundamental question: what is the new idea here? We will answer that question about blockchain now.

- **2009: Bitcoin released**
  - → The new innovation here was a new type of data structure. Specifically, it was a practical *public append-only* data structure.
    - ∗ This means that you can only write to the data structure, and you cannot erase anything that's been written.
    - ∗ How?
    - ∗ Replication: if we replicate the data all over the world, then it becomes difficult to erase something from every copy of the data structure.
    - ∗ How do we keep the replicas consistent?
    - ∗ This is a question we will spend a lot of time in the course answering.
  - → Once the data structure was created, they also created a new type of currency (BTC) based on the data structure.
- **2015: Ethereum released**
  - → The new innovation here was creating a computer based on the above data structure.
    - ∗ We can think of a computer as something that maintains and changes a state, and it turns out this is something we can do on the blockchain.
  - → Also, programs on the blockchain are composable: one program on the chain can call another.
- **2017-2022: growth of decentralized finance, NFTs, etc**

So what is this good for?

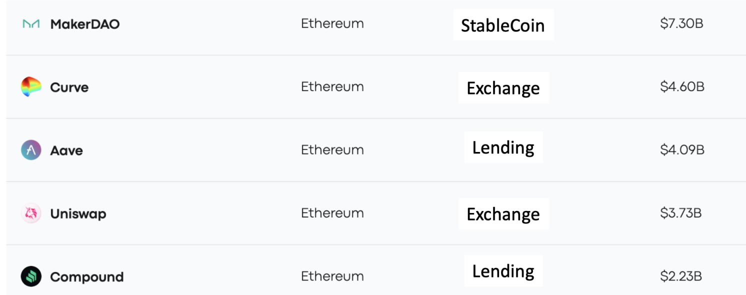
1. We can use this for a **digital currency**, such as bitcoin and ethereum, and this has the benefit of being accessible to anyone with internet access.

   Prof Boneh recommends reading the article Bitcoin Has Saved My Family for an example of why this is useful.

2. Once we run programs on these, we can use them for **decentralized applications** (DAPPs). These enable:
   - decentralized finance (DeFi): financial instruments managed by public programs
   - digital assets (NFTs): art, game assets, domain names
   - decentralized organizations (DAOs): the closest thing today is a "partnership" where people come together and share their assets to accomplish a goal

3. This is also just fun, because writing decentralized programs is a new/different way of thinking about programming.

   (This is remarkable because often these are public, small programs, on the scale of 1000 lines of code, managing billions of dollars of assets.)



MakerDAO	Ethereum	StableCoin	$7.30B
Curve	Ethereum	Exchange	$4.60B
Aave	Ethereum	Lending	$4.09B
Uniswap	Ethereum	Exchange	$3.73B
Compound	Ethereum	Lending	$2.23B

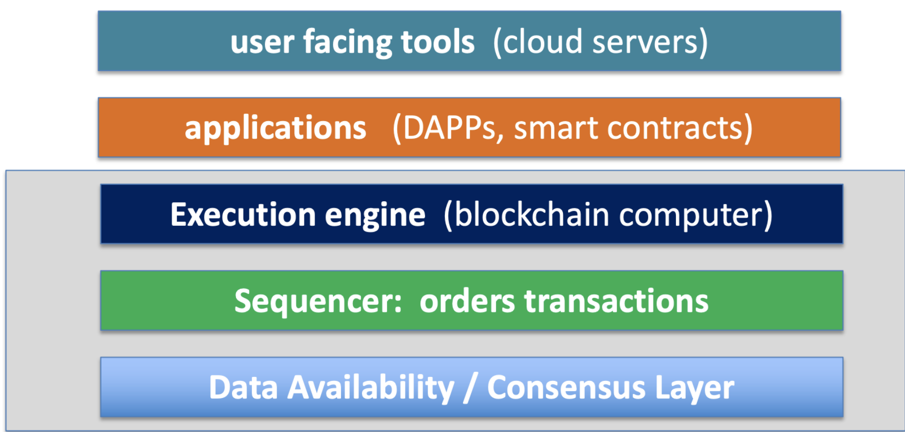
Sep. 2022

Many central banks are now also moving into the space of digital currencies.

Let's go one level deeper, and ask again: what is a blockchain?

A blockchain is made up of 5 different layers:

- The bottom layer is the magical write-only layer, which we call the **data availability** or **consensus** layer.
- The next layer is the **sequencer**, which makes sure the transactions appear in order.
- The next layer is the **execution engine** which we can think of as the blockchain computer, or where the programs run.
- The previous three layers are the fundamental blockchain; on top of that is where **applications** are written.
- The topmost layer is the **user-facing** layer, which is what you would see on e.g. a blockchain trading website.

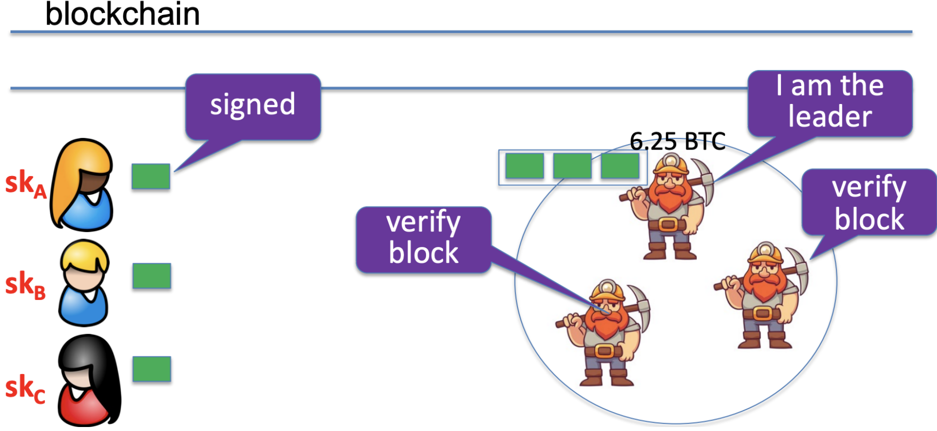


Let's look more carefully at the **consensus layer**.

As a reminder, this is the public append-only data structure, and it has the following properties:

- persistence: once we add data, it can never be removed
- safety: all honest participants see the same data (with some caveats we won't discuss yet)
- liveness: all honest participants can add new transactions (a.k.a. censorship resistance)
- open: (this is more of an optional property) anyone can add to the chain, there is no authentication required

Let's look visually at how blocks are added to the chain.



We have three end-users with their own secret key, and a pool of what are called miners or validators.

The end-users first write their transacations down, and sign it using their secret key. These transactions are then sent to the validators, and a leader is selected at random out of the validators. The leader creates a new block out of these transactions, and posts it to the blockchain. (In the bitcoin system, the leader would get paid for doing this, but the ethereum system is now more complicated, and we will discuss it later.) The other validators or miners then verify that the new block is valid, and (in the bitcoin system) the reward gets voided if this block is invalid.

If two different leaders are selected at the same time, this causes the blockchain to *fork*, and there is a fork resolution method we will discuss next week.

Why is consensus a hard problem?

In the good case, users will send transactions to different miners all over the world, and in the end, the miners will all agree on all the transactions and their order.

One problem with this is: network delays could cause transactions to appear in different orders to different miners.

Another problem is: if a network gets disconnected, then different sets of miners could have different sub-lists, and we have to be able to merge the lists in the correct order once the network gets re-connected.

Finally, what happens if some miners crash or are malicious?

Now let's look at the **blockchain computer**.

Definition 1.1. A **decentralized application** is an application that is run on the blockchain; this means that the code and the current state are written on the chain.

The way this works is that the application will accept transactions from users, change state based on this information, and write the new state on the chain.

On top of the blockchain computer, we have **user-facing servers**, so that users don't directly have to interact with the decentralized applications.

This course is a very unusual intersection of cryptography, distributed systems, and economics.

What we will cover:

1. the starting point: Bitcoin mechanics
2. consensus protocols
3. ethereum and decentralized applications
4. DeFi: decentralized applications in finance
5. private transactions on a public blockchain
6. scaling the blockchain: how to operate much faster
7. interoperability among chains: how to bridge between different blockchains

---

We are done with introduction, and now we begin the technical part of this course.

Today, we will provide the cryptography background necessary for the rest of the course.

Definition 1.2. A **cryptographic hash function** is an efficiently computable function $H : M \to T$ (mapping a message to a tag) where $|M| \gg |T|$ (the size of the tag is much smaller than the size of the message).

We often say that $T = \{0, 1\}^{256}$, so the output of the hash function is a 256-bit string.

Definition 1.3. A **collision** for $H : M \to T$ is a pair $x \neq y \in M$ such that $H(x) = H(y)$.

By the pigeonhole principle, since $|M| \gg |T|$, *many* collisions exist.

Definition 1.4. A function $H : M \to T$ is **collision resistant** if it is "hard" to find even a single collision for H.

A typical example of a collision-resistant hash function is SHA-256, which maps a message of at most 2^{64} bytes to a 32-byte tag, such that no one can find a collision for SHA-256 (if tomorrow you woke up and found a singular collision for SHA-256, that would be New York Times material).

Another popular hash function is SHA-3, which is a modern version of SHA-256.

To learn more about collision-resistant hash functions, take CS 255!

We use hash functions to commit to data on a blockchain:

Alice has a large file m. She posts $h = H(m)$, which is only 32 bytes.

Bob reads h. Later, Alice tells Bob that her message was m'.

Bob verifies that $H(m') = h$, and because our hash function is collision-resistant, he is assured that $m = m'$.

We say that $h = H(m)$ is a **binding commitment** to m; once we post h we can reveal m at any later time, but then we cannot trick anyone by changing m after the time we posted h.

Later on, we will talk about **hiding commitments**. Importantly, this is *not* a hiding commitment, since h may reveal some information about m.

Now, let's say we want to commit to a list (of transactions):

In general, whenever we talk about a list, you should be thinking of a list of transactions.

Alice has $S = (m_1, m_2, \ldots, m_n)$.

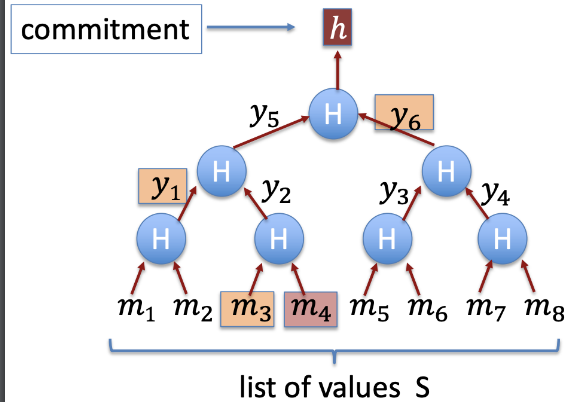
Our GOAL is:

1. Alice posts a short binding commitment to S (some h which is a commitment to S)
2. Bob reads h.
3. If Bob is given m_i and some proof π_i, then Bob can run some algorithm $\text{verify}(h, i, m_i, \pi_i)$ and accept if $S[i] = m_i$ and reject otherwise.

Moreover, an adversary should not be able to trick the verifier: an adversary would try to come up with some m_i' which is *not* actually $S[i]$ and some π_i' that convinces the verifier to accept the false m_i'. It should not be able to succeed.

The instinctive way to do this might be to open up the entire list (so π_i is just the entire list), and then we can just verify that $h = H(S)$ and $S[i] = m_i$. But this is linear in the size of the list; can we do something logarithmic in the size of the list?

The answer is Merkle Trees (Merkle came up with them when he was a PhD student in 1989).



What we do is we build a hash tree, so at each level of the tree, we pass the nodes into the hash function in pairs, in order to build the next level. The h at the top level is the commitment that Alice reveals.

Then, if we want to verify, for example, m_4, we just need to provide m_3, y_1, y_6, and then Bob can hash his way up the tree, computing

$$\begin{aligned} y_2 &= H(m_3, m_4) \\ y_5 &= H(y_1, y_2) \\ h' &= H(y_5, y_6), \end{aligned}$$

and then accepting if $h = h'$ and rejecting otherwise.

Theorem 1.5. For a given n, if H is a collision-resistant hash function, then the adversary cannot find some (S, i, m, π) such that $|S| = n$, $m \neq S[i]$, h is the commitment for S, and verify(h, i, m, π) accepts.

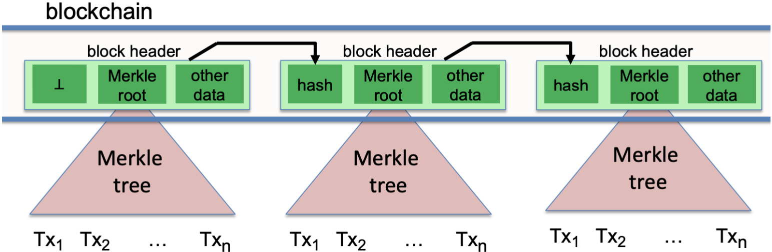
That is, the adversary cannot come up with any sequence of messages S where it can trick the verifier about some message in the sequence.

To prove this, you would prove the contrapositive (if the adversary can trick the verifier, this is not a collision-resistant hash); this was a homework problem in 255.

How is this useful?

Well, to post a block of transactions S on the chain, we only need to write the commitment to S, which keeps the chain small. Then, when it's necessary, we can reveal and prove individual transactions.

So within each block of the block chain, we have **block headers**. These contain: the hash of the previous block header, the Merkle root or commitment to the transactions, and other necessary data.



Another cute application of hash functions is proof of work:

Our GOAL is:

to have a computation problem that takes time $\Omega(D)$ to solve but the solution only takes time $O(1)$ to verify.

(Here, D is called the **difficulty**.)

How do we do this?

Well consider a collision-resistant hash function $H : X \times Y \to \{0, 1\}^{256}$.

Then, our puzzle could be: given an input $x \in X$, output a $y \in Y$ such that $H(x, y)$ (when read as a number) is at most $2^n/D$.

The verifier is given x, y and just needs to compute $H(x, y)$ and accept if it's at most $2^n/D$.

Theorem 1.6. If H is a "random function" then the best algorithm requires D evaluations of H in expectation.

But this is a parallel algorithm: if you have more machines, you can solve the puzzle faster.

This is used in the Bitcoin consensus protocol, using SHA-256 applied twice.

Lecture 2: Bitcoin Nuts and Bolts

In the physical world, Bob provides his physical signature to prove that he is authorizing a transaction from his bank account.

But in the digital world, it would be very easy to copy Bob's signature from one place to another. So Bob's **digital signature** is a function of Bob's secret key, and whatever document he is signing.

Definition 2.1. A **signature scheme** is a triple of algorithms:

- Gen() outputs a key pair (pk, sk), which stands for "public key" and "secret key"
- $\text{Sign}(\text{sk}, \text{msg})$ outputs a signature σ based on the secret key and message
- $\text{Verify}(\text{pk}, \text{msg}, \sigma)$ uses the public key to check whether σ is a valid signature (for Bob) on the given msg - it outputs `accept` or `reject`

Definition 2.2. Informally, a signature scheme is **secure** if an adversary sees Bob's signature on many messages of his choosing, and then cannot forge Bob's signature on any new message.

Families of signature schemes:

1. RSA signatures (old, not used in blockchain):

   long signatures and public keys (≥ 256 bytes) but fast to verify

2. discrete-log signatures (Schnorr and ECDSA):

   these are used for Bitcoin and Ethereum,

   short signatures (48 or 64 bytes) and public keys (32 bytes) which is important because storing data on the blockchain is costly

3. BLS signatures:

   used in Ethereum 2.0, Chia, Dfinity

   also 48 bytes, can do "signature aggregation:" compressing multiple signatures from different people into a single signature

4. post-quantum signatures:

   safe against attacks from quantum computers, but very long (≥ 600 bytes)[1]

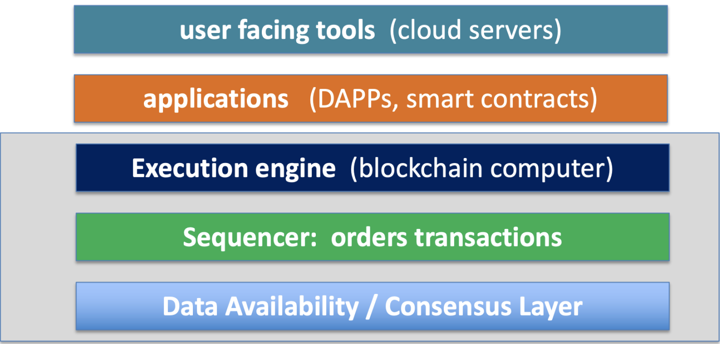
Signatures are important for the blockchain because when Bob sends in his transaction, he signs it to confirm that he is really the one transferring money from his account, and no one else is trying to make a transaction in his name without his permission. When the miners verify the transactions that they receive, they are mainly verifying that the signature matches the transaction received.

[1] There are many cryptographic schemes which are not secure against quantum attacks. However, we have developed ways to do cryptography on normal computers that would still be secure against quantum computers, but these tend to have longer keys/be slower, so we won't implement them until quantum computers are a real threat.

Now we transition to: Bitcoin mechanics.

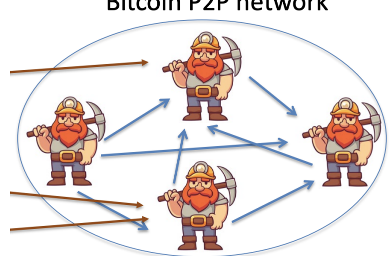
Oct 2008: paper by "Satoshi Nakomoto" (pseudonym) explaining the mechanics of Bitcoin
Jan 2009: Bitcoin network launched

Remember the bitcoin layers from last class?



Today, we will briefly explain the consensus layer, so that we can focus on the execution engine. Then, next week, we will return to explaining the consensus layer and sequencer in more detail.

Overview of the Consensus Layer
First, there is a peer-to-peer network of thousands of miners or validators, where each miner is connected to about 8 other miners in the network.



When a miner receives a (signed) transaction, it broadcasts it to all the other miners in the network, by broadcasting to its neighbors, and then the neighbors broadcast to their neighbors, and so on. Each miner then validates this transaction adds this transaction to their **mempool**, which is their personal list of transactions which haven't been added to the chain yet.

> **Remark 2.3.** Miners have a lot of power! They see all the transactions before adding them to the blockchain, so if it is in their benefit to try to change the transactions or add them in a different order, they will.

Then, around every 10 minutes, each miner will create a candidate block from their mempool. A random miner is selected (we will see how next week) and they broadcast their block to the network. All the miners validate the new block, and then it is added to the chain and those transactions are deleted from the miners' mempools.

Once the block is added to the chain, the selected miner gets paid 6.25 BTC for creating the block. This is known as the **coinbase transaction** and it is the first transaction in any block. This is the only way new bitcoin is created (economically, this means the remaining bitcoin in the world is worth slightly less), and the amount of bitcoin paid gets halved every four years (meaning only 21 million bitcoin will ever be created - we are currently at 19.1 million bitcoin).

Remark 2.4. Miners can also choose the order of transactions in a block - someone who wants to make an earlier trade than you can persuade a miner to stick their transaction before yours in a block. How do we deal with this?

We will talk about this at length later.

Next week, we will talk about how it is (very) difficult to remove a block from the chain or to prevent a block from being posted, which are important and useful properties of the blockchain.

Bitcoin blockchain

The first block in the bitcoin blockchain was called the **genesis block** and contains a copy of a news article from that day, to mark the "start date" of bitcoin.

After that, each block header has:

- a version number (4 bytes)
- a hash of the previous block header (32 bytes)
- the self-reported time the miner assembled the block (4 bytes)
- bits - will be used for "proof of work" and explained next class (4 bytes)
- nonce - will be used for "proof of work" and explained next class (4 bytes)
- root of the merkle tree of transactions (32 bytes)

For the rest of this lecture, we can think of the blockchain as an append-only sequence of transactions.

Transaction Structure

(This is for generic, non-coinbase, transaction.)

A transaction for Bob contains:

- the inputs (the assets you want to give in the transaction)
- the outputs (who gets the transaction)
- the witnesses (part of the input)
- the locktime (optional, specifies the earliest block number that can include the transaction)

Then, we hash the transaction (excluding the witnesses), and this is the transaction id, a 32-bit identifier.

Each input contains:

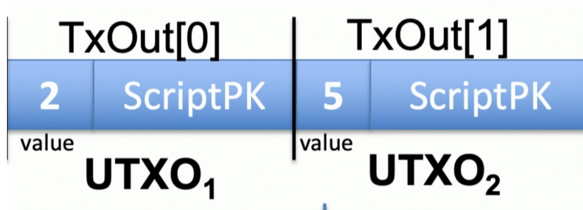
- the transaction id and output index for when Bob got the money
- the ScriptSig, a program used as a signature, to prove that Bob actually has this money and can spend it

Each output contains:

- the value (8 bytes)
- the ScriptPK, a program which tells us who is now allowed to spend this money

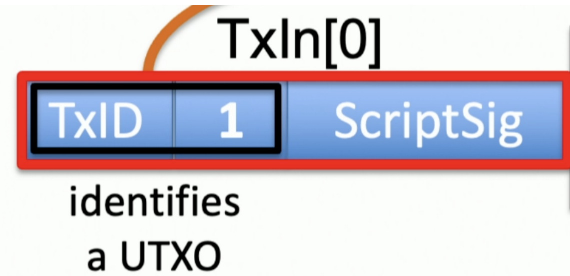
(Note that the value is in units of 10^{-8} bitcoins, so we don't have fractional values but we can trade fractional amounts of bitcoins.)

Let's look an example of this. The outputs of our first transaction might look something like this:



Here, UTXO stands for Unspent Transaction Output; this is the money that can actually be spent at any given time.

Then, an input of our next transaction can look like this:



so that we are spending the bitcoin that we got in TxOut[1] in our previous transaction.

At this point, TxOut[1] becomes a TXO, or a spent transaction, and we cannot use it again. So there are no "account balances" in the same sense in the blockchain; to count all the money you own, you have to go through all your UTXOs and add up the values. Of course, online wallets or bitcoin trading services will do this for you.

This means you cannot partially spend a UTXO; if you want to send someone 2 out of 5 of the bitcoin in your UTXO, you would have to send the remaining 3 to yourself in the same transaction.

How do we validate the input on this transaction?

For each input, miners check that:

- The program ScriptSig — ScriptPK (the two concatenated programs) returns true.
- The TxID and index corresponds to an output in the current UTXO set.
- The sum of all the input values is greater than or equal to the sum of all the output values.

The miners have a set of all UTXOs in their memory, and they remove the spent UTXOs from the set each time they add a transaction to the blockchain.

Bitcoin transaction fees (paid to the miners) are based on network congestion, so they spike when the bitcoin value crashes and people are selling their bitcoin.

UTXOs with very little value are called **dust**; if these are less than the value of a transaction fee then you will only be able to use them if you create a transaction with a lot of dust input, to aggregate all your dust into a clean UTXO output.

The ScriptSig and ScriptPK are in a very strange language called Bitcoin Script:
This is not a Turing Complete language, because there are no loops. Instead, it is what's called a stack machine.

There are a number of possible operations:

- `OP_1` pushes 1 onto the stack, `OP_2` pushes 2 onto the stack, and so on until `OP_16`
- `OP_DUP` will duplicate the top value of the stack
- `OP_IF` `OP_ELSE` and `OP_ENDIF` help with control
- `OP_VERIFY` will abort and fail if the top of the stack is false (0)
- `OP_RETURN` will abort and fail
- `OP_EQVERIFY` will pop the top two elements of the stack and abort and fail if they are not equal
- adding, subtracting, and, SHA256 hash, check a signature on the ticket, check if the value at the top of the stack is after the transaction locktime

Example 2.5. A common script looks like: `<sig> <pk> DUP HASH256 <pkhash> EQVERIFY CHECKSIG` What does this do?

Well, we start out with an empty stack. After the first two instructions, the stack looks like:

`<sig> <pk>`

and then the `DUP` tells us to duplicate the top value, so that our stack looks like

`<sig> <pk> <pk>`.

Then, `HASH256` pops the top value from the stack and hashes it, so we get

`<sig> <pk> <hash>`.

We push the next value to the stack to get

`<sig> <pk> <hash> <pkhash>`

and then `EQVERIFY` pops `<hash>` and `<pkhash>` and checks if they're equal. If they're not equal (so we have a different public key than we wanted), we fail. If they are equal, then we have

`<sig><pk>`

on the stack, and then `CHECKSIG` will pop these two terms, and write 1 on the stack if this is a valid signature (using the transaction as the message) and 0 otherwise. Then, the program terminates successfully.

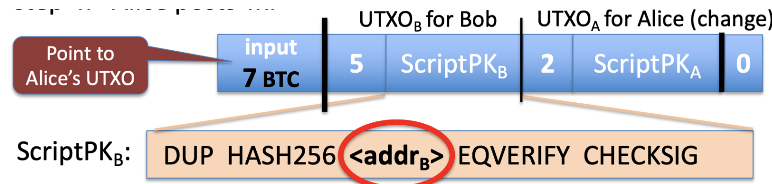
So what is this program for?

Well this is one option for our ScriptSig (the `<sig> <pk>` at the beginning) and ScriptPK (the rest of the script). We can see if this program outputs a 1 then we have a a valid signature, and if not, our signature is invalid.

There are multiple transaction types. The first is **P2PKH**, or pay to public key hash:

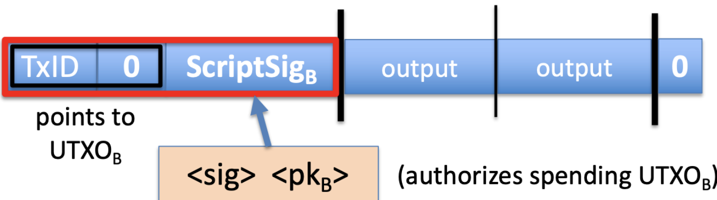
Let's say Alice wants to pay Bob 5 BTC. To do so:

1. Bob generates his signature key pair $(\text{pk}_B, \text{sk}_B)$ using the Gen() function.
2. Bob computes his bitcoin address as $a_B = H(\text{pk}_B)$.
3. Bob sends a_B to Alice.
4. Alice posts a transaction that looks like:



(Note that here the ScriptPK$_B$ is exactly the second half of the script we just analyzed!)

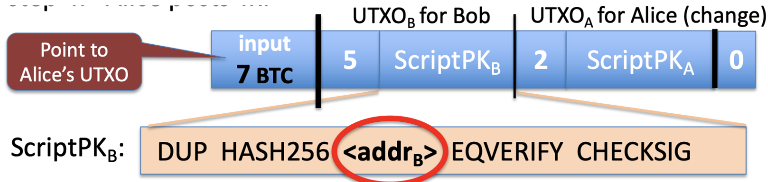
5. When Bob wants to spend this UTXO, he then creates the following ticket (which has the first half of the script we just analyzed):



Lecture 3: Bitcoin Nuts and Bolts, II and Wallets

Remember that transactions on the blockchain cannot be erased. This is a double-edged sword - we wanted the blockchain to have this feature so that we know for sure that all valid transactions are recorded on the blockchain, but it also means that if we make a mistake when sending a transaction, we cannot undo this.

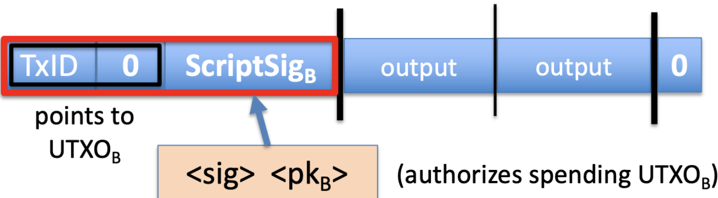
Last time, we discussed the P2PKH transaction, where for Alice to send 5 bitcoin to Bob, she would post a transaction with a `ScriptPK` that looks like this:



What if an adversarial miner tries to steal this money, by replacing `addr_B` with their own address?

Well, when Alice posts the transaction, she posts it along with a signature on the whole transaction (stored in the `ScriptSig` for her input). If a miner tries to modify the transaction, then Alice's signature would no longer be valid, and the modified transaction would not be accepted.

As a reminder, we showed last time that when Bob wants to spend this transaction, he creates a ticket with an input that looks like:



We analyzed this script (Bob's `ScriptSig` concatenated with Alice's `ScriptPK`) very carefully last time, and we noted that it first checks that the hash of `<pk_B>` matches `<addr_B>`, and then checks that `<sig>` is a valid signature for Bob's ticket, using Bob's public key.

There's a problem here - we can't sign the entire transaction if the transaction contains the signature inside of it. So `<sig>` here is actually a signature on the entire transaction except any `ScriptSig`s.

Note 3.1. Since the bitcoins are secured only using your secret key, if you lose your secret key, you lose all your bitcoin, and if someone steals your secret key, they now have all of your bitcoin.

Note that in P2PKH:

- Alice specifies the recipient within her `ScriptPK`.
- The recipient's PK is hashed, so the recipient is not actually revealed until Bob claims his bitcoin. (this blocks against some quantum attacks)
- Since the ticket is signed by Alice, the miner cannot modify `<addr_B>` and steal the funds.

There is a problem with ECDSA. Specifically, the way ECDSA works, if you have a valid $(m, \texttt{sig})$ pair for Bob, an adversary can create a different $\texttt{sig}'$ for Bob on m.

Generally, we wouldn't care about this - Bob already signed this message, why would he care if someone changes his signature to a different valid signature?

But in the ticket, there is a problem. If the ticket ID is the hash of the entire ticket, including the signatures, then an adversarial miner can change the signature, changing the ticket ID. Then, Bob will never see his ticket ID show up on the blockchain, and assume his transaction never went through, when actually his money has been spent already.

To combat this, we keep the signatures in the **witness** section of the transaction, and have Bob's ticket ID be the hash of everything but the witnesses.

Our next transaction type is called **P2SH** or pay to script hash. (We will be discussing the way this worked before SegWit in 2017).

In this transaction:

1. Bob writes a "redeem script" and publishes the hash of the script itself.
2. Alice sends funds, with the recipient address being the hash of this script.
3. Bob can spend UTXO if he can satisfy the script.

The main idea here is that, using a script, we can allow for more complex conditions for when the recipient can actually get the money.

So Alice's `ScriptPK` looks like:

```
HASH160  <H(redeem script)>  EQUAL
```

and Bob's `ScriptSig` looks like:

```
<sig_1> <sig_2> ... <sig_n> <redeem script>
```

Then, the miner, reading the concatenated scripts from left to right, will first check if the provided `<sig_i>`'s satisfy the redeem script, and then check if hash of the redeem script matches the hash that Alice posted.

Example 3.2. We will look at an example of this, called **multisig**.

The goal of multisig is to require signatures from multiple people in order to spend a UTXO. A redeem script for this would look like:

```
<2>  <PK_1>  <PK_2>  <PK_3>  <3>  CHECKMULTISIG
```

where the 3 at the end means we are inputting three public keys and the 2 at the beginning means we are expecting two of the corresponding signatures.

So the `ScriptPK` would have a hash of the redeem script as the address, and a valid `ScriptSig` would have two corresponding signatures and then the redeem script, such as:

```
<0> <sig1> <sig3> <redeem script>
```

Note that in the `ScriptSig`, the redeem script is written in the clear, so the redeem script is public at

the time when we spend the UTXO. Bitcoin added a recent update, which we'll talk about later, where you are now able to spend a UTXO without ever publicly posting your redeem script in plaintext.

Note 3.3. Two notes about `CHECKMULTISIG`:

- The `<0>` at the beginning of the `ScriptSig` is because the actual `CHECKMULTISIG` operation has a bug where it pops an extra element from the top of the stack and expects that element to be 0.
- `CHECKMULTISIG` expects the signatures to be in the same order as the public keys. So it would accept
```
<0> <sig1> <sig3> <2> <PK1> <PK2> <PK3> <PK4> <4> CHECKMULTISIG
```
but not
```
<0> <sig3> <sig1> <2> <PK1> <PK2> <PK3> <PK4> <4> CHECKMULTISIG
```

Let's look at some situations where multisigs would be useful:

Protecting Assets with a Co-Signatory

A custody server could help you protect your assets. The way this works is: Alice's UTXOs are sent to a script that requires a 2-of-2 multisignature, with the public key of Alice and the custory server.

Then, when Alice wants to spend her UTXOs, she sends her public key to the custody server. The custody server uses some process of verifying this is actually Alice (such as calling her on the phone), and if they successfully verify her, they send over a transaction signed by the custody server. Alice then adds her signature onto the transaction, and submits it to the miners.

This prevents someone who got access to Alice's signature from immediately stealing her bitcoins. Note that the custody server cannot steal her bitcoins (because any transaction also requires Alice's signature) but it can refuse to sign and therefore prevent Alice from accessing her UTXOs.

To work around this, usually Alice actually uses a 2-of-3 multisig, where the third public key is a secondary public key for Alice, which she keeps locked away somewhere more secure, and would only use if the custody server tries to block her access.

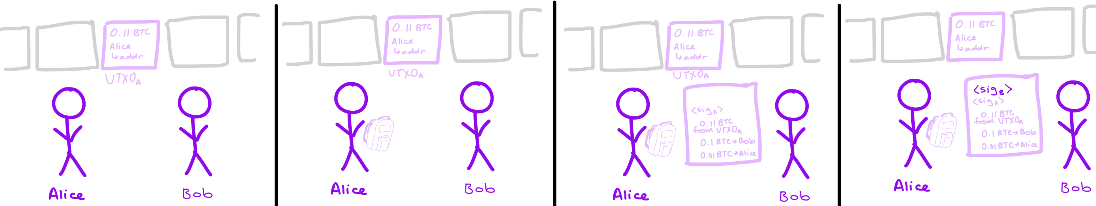
Escrow Service

Let's say Bob wants to sell Alice a backpack for 0.1 BTC (as a note, at the time of writing this, this means the backpack is worth around $2000). We want Alice to pay only after the backpack arrives, but we don't want her to skip out on paying.

We do the following:

1. Alice creates `addr`, which is a 2-of-3 address with PK_A, PK_B, and PK_J, where J, the judge, is a neutral third party.
2. Alice posts a payment of 0.11 BTC to `addr` and tells Bob she wants a backpack for 0.1 BTC (called $UTXO_A$)
3. Once Bob sees the transaction on the blockchain, he mails the backpack to Alice.
4. Once Alice gets the backpack, she signs a transaction that sends 0.1 BTC to Bob and 0.01 BTC back to herself, from $UTXO_A$. *(The 0.01 BTC being sent back to Alice here is to give her motivation to create this transaction.)*
5. Bob adds his signature to the transaction and posts it to the blockchain.

Here is a picture of what this would look like, but note that the picture does not have a very accurate description of the way a transaction is structured:



So this is what happens if everyone behaves well. What happens if someone acts adversarially?

If the backpack never arrives (so Bob is at fault), Alice and the judge sign a transaction to give Alice her money back from $UTXO_A$.

If Alice refuses to pay, after she recieves the backpack, Bob and the judge sign a transaction to get the money from $UTXO_A$.

If Alice and Bob are both at fault, then the judge signs a transaction that gives them both 0.05 BTC (and gives the judge the remaining 0.01 BTC as payment), and either Alice or Bob can sign the transaction and submit it to the blockchain.

But this service relies on the judge always being honest. We assume the judge is someone Alice and Bob mutually agree on, and the judge is running some sort of escrow service, so he has motive to be honest in order to keep getting clients for his escrow service.

Cross Chain Atomic Swap

Let's say Alice has 5 BTC and Bob has 2 LTC (litecoin). They want to swap. We want a sequence of transactions such that either *both* the bitcoin and the litecoin are successfully transferred, or *neither* are transferred, and there is no way of getting "stuck halfway" so that one person ends up with both the currencies.

How do we do this?

This will be a programming project later, so you will see this in detail later!

We now transition to the discussion of wallets.

One user can have many public keys (and corresponding secret keys), since there is at least one per type of currency, and maybe even one per UTXO.

The purpose of a **wallet** is to

- generate new public/secret key pairs, and store the secret keys
- post and verify transactions
- show account balances

How do we manage lots of secret keys?

There are many options:

- the **cloud** holds secret keys like a bank does (but there's a saying in crypto: "not your keys, not your coins")

- keeping them on your **laptop** or **phone**
- using external **hardware**
- printing them on **paper**
- storing them in your **brain** (generally a bad idea, good secret keys are not easily memorized)

Lets talk about hardware wallets in more detail.

Hardware wallets connect to your laptop or phone wallet using Bluetooth or USB, and manage many secret keys.

You have an (up to 48-digit) PIN to unlock the hardware, and then you use the screen and buttons on the hardware to verify and confirm your transactions.

If you lose your hardware wallet, you lose all your funds. How do we prevent this?

One idea is:

Generate a secret 256-bit seed k_0.

Then, HMAC is a well-known pseudorandom generator - this means it is a deterministic algorithm such that the output "looks random."

So we generate secret key sk_i as $\mathrm{HMAC}(k_0, i)$, and then the corresponding ECDSA public key is just g^{sk_i}, for some generator g.

If you don't know k_0, then $\mathrm{pk}_1, \mathrm{pk}_2, \mathrm{pk}_3, \ldots$ look like random unlinkable addresses, and you can't even trace them back to the same person.

But if you lose your wallet, you can input k_0 into your new hardware, and get back all your secret keys.

How do you store k_0?

Many hardware wallet brands come with a list of 2048 words, where each word corresponds to an 11-bit string. When you initialize your hardware, they give you the list of 24 words, and you are asked to store the 24 words and repeat them back to the ledger.

How do you store the 24 words?

If you're boring, you could just write them down on a piece of paper, and keep it in a fireproof safe. Another option is Crypto Steel, where you get a bunch of tokens with the letters of the alphabet, and spell out your words in order on a rod, which would be stored in a fireproof cylinder.

But you would have to be careful with your unused letters. If you throw them away, someone can go through your garbage and figure out, via process of elimination, what your words were.

A more common attack is the "pre-initialized hardware wallet" where people would sell pre-initialized hardware wallets (advertised as being easier to set up) and since the person who initialized your wallet knows your k_0, they can later steal all your secret keys.

Lecture 4: Consensus, I

The Byzantine Generals problem encapsulates the problem of reaching consensus.

Definition 4.1. The **Byzantine generals problem** has the following setup:

- there are n generals, one of which is the *commander*
- some generals are *loyal* and others are *traitors*
- the commander sends an order to all generals (either *attack* or *retreat*)
  - → if the commander is loyal, he sends the same order to all generals
  - → if the commander is a traitor, then he can send different orders to different generals
- the generals all communicate with each other:
  - → loyal generals have to say the same order as the commander
  - → traitors can say whatever they want

Then, all generals decide whether to *attack* or *retreat*.
Our GOAL is to have: all loyal generals make the same decision, and if the commander is loyal, they should do what the commander says.

The solution to the Byzantine general problem is a type of **consensus protocol**.

In general, in consensus protocols, we use slightly different terminology:

- instead of generals, we have **nodes**
- instead of a commander, we have a **leader**
- instead of loyal versus traitors, we have **honest** versus **adversarial** nodes

In addition to this, we also have an external **adversary**, which is organizing all the efforts to prevent consensus.

The adversary can corrupt nodes, after which they become **adversarial** nodes.

The adversary can induce:

- **crash faults**, which means the node cannot send or receive any messages
- **ommission faults**, which means the adversary will choose which messages the node can send or receive
- **Byzantine faults**, which means the adversarial nodes can now send bad messages or ignore good messages at will

We will assume the adversary can induce Byzantine faults, because in general we want to protect against the strongest possible adversary.

However, we will assume that all messages are signed, and that the adversary cannot break our signature scheme, since we are building up from cryptographic primitives (such as signature schemes), which we are assuming are secure.

The reason we have a single adversary controlling all adversarial nodes is because this gives the adversarial nodes more power (working as a collective rather than entities with disjoint goals).

There are two types of possible adversaries:

- a **static adversary** decides which nodes to corrupt before communication starts
- an **adaptive adversary** can choose which nodes to corrupt while the protocol is running (so it can observe the behavior of the nodes and then pick ones it thinks are more powerful to corrupt)

We will focus on static adversaries, because the protocol against adaptive adversaries gets very complicated.

We also assume there is an upper bound f on the number of nodes the adversary can corrupt. Usually, we are in the $f < n$ case (can't corrupt all the nodes), $f < n/2$ case (can't corrupt a majority of the nodes), or the $f < n/3$ case (can't corrupt a supermajority of the nodes). This f is called **resilience**.

During the communication protocol, nodes can send messages to each other. We assume time moves in discrete rounds, so we say that a node sent a message in some round i.
The adversary can delay the sending of messages, with the constraint that:

- in a **synchronous network**, there is some Δ such that the adversary must deliver all messages within Δ rounds from when they were sent
- in a **asynchronous network**, the adversary can delay messages as long as it wants, as long as it delivers every message after some finite number of rounds

But we have a problem - synchronous networks are not super accurate to real life, and asynchronous networks are very difficult to work with. So instead we will model this as a

- **partially synchronous network**, in which there is a Δ and a GST (global stabilization time). The adversary can delay messages for arbitrarily long before the GST, but after the GST, it must deliver all messages within Δ rounds.

Ok, so now let's look at a specific problem.

Our motivation for the state machine replication problem is the following:

Originally, we have one central bank that keeps a list of transactions. But with the blockchain, we are trying to decentralize this process. So we have a bunch of smaller machines that are trying to keep the list of transactions. But we might be telling only a few machines about our transactions, or some machines are offline, and we want all the machines to agree on the order of the transactions in the end.

We call this list of transactions the **log** or **ledger**.

Definition 4.2. The **State Machine Replication** (SMR) protocol has the following setup:

- there are two types of nodes - **clients** and **replicas**
- the replicas receive transactions and execute some protocol to determine the log
- the clients don't talk to each other, they just learn the log from the replicas

The GOAL is that the clients all learn the same log.

SMR differs from the Byzantine generals problem in two ways:

- now, the clients have to *continuously* output a correct log, rather than just a single attack/retreat
- now, the clients who are trying to learn are different from the replicas that are communicating, rather than just both being generals

The reason we want the clients to agree on the order of transactions is to avoid the double-spend attack:

Definition 4.3. Let's say I want to buy two cars, but I only have money for one car.

I create transaction t_1 which sends my money to Car Company A and transaction t_2 which sends my money to Car Company B.

Say Car Company A works with a client that has transactions in the order $t_1 t_2$, so it ignores t_2 as invalid and sends you a car, while Car Company B works with a client that has transactions in the order $t_2 t_1$, so it ignores t_1 as invalid, and sends you a car.

Now you have two cars!

So what do we want from our SMR protocol?

Definition 4.4. A **secure** SMR protocol has two guarantees:

- **safety**: Let's say we have client i with log i and client j with log j. It's fine if one log has fewer transactions than the other (one client hasn't received all the transactions yet) but the shorter log must agree with the start of the longer log. We say that one log is a *prefix* of the other.
- **liveness**: There must be some time T_{conf} such that, if we submit a transaction to an honest replica, all clients must have it in their log after at most T_{conf} rounds.

We include liveness because it prevents censorship of certain transactions; a simple protocol with safety and not liveness has the clients never write anything to their log.

Blockchain protocols are the same as SMR protocols, except we work with blocks, which have groups of transactions, rather than individual transactions.

So let's put this all together:

We want a *SMR protocol* that is *secure* under *partial synchrony* with the best possible *resilience* (for replicas) against a *Byzantine adversary* that controls the corrupted replicas.

In 1988, DLS showed that this is not possible for a resilience greater than or equal to $n/3$.

So we will try to do this for $f < n/3$.

Note that in this scenario, the clients just assume that all the honest replicas will agree on the transaction log, so they can talk to all replicas and keep the transaction log that the majority of them agree on. Thus, we just need to focus on having all the honest replicas get the same log.

Let's try a first attempt, which we will call **baby streamlet**.

Some logistical things:

- We divide our time into **epochs**, where each epoch is 2Δ rounds.
- The n replicas are fixed before protocol execution starts, and every replica knows the public keys of all other replicas.

Then, in each epoch e:

- We randomly assign a leader node L_e using a hash function.
- The leader looks for the longest chain it has seen so far and *proposes* a new block extending that chain. This block is labeled e and signed by the leader L_e.

- Then the client looks for the longest chain, and *finalizes* the block at the end of that chain. If there are multiple such longest chains, it finalizes the one whose end block has a smaller epoch.
- Once the block is finalized, it is added to the log.

But this is not secure!

An adversarial leader can extend the longest chain with B_2, sign it, and label it with e, and send it to client Alice. Meanwhile, he can also extend the longest chain with B_3, sign it, label it with e, and send it to client Bob. Bob and Alice both see that the longest chain has been extended by L_e, and they finalize it. Now, Bob and Alice have different chains ☹

(Note that Bob and Alice will eventually get B_3 and B_2, respectively, but only after the GST, and by that time it will be too late.)

Let's try to fix this. We will call our second attempt **teen streamlet**.

We have the following adjustment:

- After a leader posts a block, the replicas **vote** on the proposed blocks by signing it.
- A block is considered **notarized** if 2/3 of the replicas have voted on it.
- Now, at epoch e, the leader L_e finds the longest *notarized* chain and proposes a new block extending that.
- After Δ rounds after epoch e has started, the replicas vote on the first valid epoch-e proposal from L_e that extends the longest notarized chain it sees. If there is no such chain, it doesn't vote.
- The client finalizes a block once it is notarized.

Now, since $f < n/3$, we have prevented the attack from Baby Streamlet - for two blocks in the same epoch to get finalized, they had to both be notarized, which means at least $2/3 + 2/3 - 1 = 1/3$ of the replicas voted for both blocks. But less than 1/3 of the replicas are adversarial, and honest replicas won't vote for more than one block. So this attack is now impossible!

Lecture 5: Consensus, II

Last time, we defined safety as "for all honest replicas, one replica's version of the log must be a prefix of the other." Let's discuss what this means a bit further.

Definition 5.1. If we have two streams A and B, then A **concatenated** with B, or $A||B$, is B appended to the end of A.
For example, if $A = \text{tx}_1\text{tx}_2$ and $B = \text{tx}_3\text{tx4}$, then $A||B = \text{tx}_1\text{tx}_2\text{tx}_3\text{tx}_4$ and $B||A = \text{tx}_3\text{tx}_4\text{tx}_1\text{tx}_2$.

Definition 5.2. A stream A is a **prefix** of B (denoted $A \preccurlyeq B$) if there exists some C (which could be the empty stream) such that $A||C = B$.
If $A = \text{tx}_1\text{tx}_2\text{tx}_3\text{tx}_4$, $B = \text{tx}_1\text{tx}_2\text{tx}_3$, and $D = \text{tx}_1\text{tx}_3\text{tx}_4$, then $B \preccurlyeq A$, but D is *not* a prefix of A.

Now, we said that security for SMR is defined by two properties - safety and liveness. We can re-define safety to mean: our protocol has **safety** if for any honest replicas A and B, either $\text{LOG}_A \preccurlyeq \text{LOG}_B$ or $\text{LOG}_B \preccurlyeq \text{LOG}_A$.

Last time, we left off trying to prove safety for teen streamlet. (Note that no streamlet can have liveness until *after* GST since before that the adversary can delay messages arbitrarily.) To simplify things, from this part onwards we will ignore the clients altogether, and assume the replicas are also the ones finalizing the epochs.

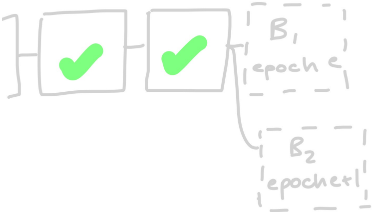
We showed last time that assuming $f < n/3$, so less than a third of the replicas are adversarial, then the adversary cannot carry out the same attack as against Baby Streamlet - we cannot have two blocks finalized at the same height for the same epoch.

But what if these two blocks correspond to different epochs?

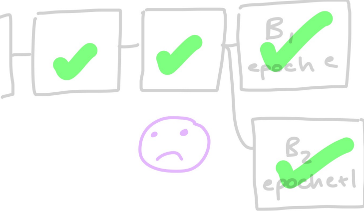
Consider the following scenario. For ease of notation, we will assume there are 100 replicas and therefore we need 67 votes to approve a block:

- In epoch e, an adversarial replica proposes block B_1, but the adversary only shows it to 66 honest replicas before voting time.
- These 66 honest replicas see that it is a valid block and vote on it, but it is not notarized because less than 67 replicas have signed it.

- Since the chain up to B_0 is the longest notarized chain, the next replica (who is honest) proposes a block B_2 extending B_0.



- The honest replicas all see that the chain up to B_0 is the longest notarized chain and that B_2 is valid, so they all vote on B_2, and it gets notarized.
- A dishonest replica now votes on B_1, so B_1 is now also notarized. Now there are two notarized chains with the same height.



This uses the fact that dishonest replicas do not need to follow policy, so they are able to sign a block after the voting period is over for that block.

Ok, let's try to fix this. We will call our final attempt **streamlet** (not adult streamlet, sadly ☹).

This is very similar to teen streamlet:

- In epoch e, the leader L_e looks for the longest notarized chain it has seen so far, and proposes a new block extending that.
- Once it has been Δ rounds since the start of epoch e, the replicas vote on the proposal by signing it. Honest replicas will vote on the first proposal they see from L_e that is marked "epoch e," extends the longest notarized chain they see, and is a valid block. If there is no such proposal at this time, they do not vote.
- Anytime after epoch $e+1$, if the longest notarized chain the client sees ends in "epoch $e-1$" "epoch e" "epoch $e+1$" (so the blocks are from consecutive epochs), the client finalizes the block labeled epoch e and all blocks before that.

So our only modification is that we wait for the next block in our chain to be notarized properly before we finalize the current block.

This version of streamlet is secure! But it still has problems:

- for each block to be added to the chain and finalized, we require $\theta(n^3)$ messages to be sent - this is a very high communication complexity
- it expects that all replicas can maintain synchronized epochs the entire time

There are better protocols out there: for example, HotStuff requires only $\theta(n)$ messages for each block and does not need perfect synchrony.

How do we choose which nodes participate in consensus? There are two options:

- **permissioned**: there are a fixed set of nodes, decided beforehand (this is what we assumed up to this point)
- **permissionless**: anyone meeting certain criteria can join the consensus (this is how blockchain works)

In general, permissionless protocols are vulnerable to the Sybil attack.

Definition 5.3. In the previous section, we assumed the adversary controlled at most f nodes, where f usually meant a minority or a superminority. But in a permissionless protocol, if the criteria are easy to meet, the adversary can just generated arbitrarily many nodes, and take over the entire network. This is known as a **Sybil attack.**

So, we want our consensus protocols to have **Sybil resistance**, where it is very difficult for the adversary to create disproportionately many nodes.

To do this, we base our criteria to join on some bounded resource:

- the **proof of work** protocol (used in Bitcoin) requires you to have some amount of computational power
- the **proof of stake** protocol (now used in Ethereum) requires you to have some number of coins, which we showed earlier was designed to be finite
- the **proof of space/time** protocol (used in Chia and Filecoin) requires you to have some amount of total storage over time

So the adversary cannot carry out a Sybil attack, because, for example, in the proof of work protocol, his total computational power is limited, and if he tries to create many adversarial nodes, each node would have a small fraction of his computational power, so they wouldn't be able to do much.

Let's look at how Bitcoin uses proof of work in its consensus protocol.

We keep a difficulty level D, where D is some number between 1 and 2^{256}.

Remember that each block header on the bitcoin contains a `nonce`. To mine a new block, the miner must find a new nonce such that the hash of the new block header is less than $2^{256}/D$. Remember that the new block header contains the hash of the previous block header, the Merkle root of the transaction tree, and a nonce. So we want to find a nonce such that

$$H\left(H(B_{\text{prev}}), \text{Merkle root}, \texttt{nonce}\right) < \frac{2^{256}}{D}.$$

Since we model the hash function as essentially random, the only way for miners to find this D is for them to try random nonces until they reach one that has this output. As we discussed before, this means that in expectation, it will take D total attempts for the miners to be able to add a new block – though, since this is random, it is possible that certain nonces are found very quickly or very slowly.

We assume that D is known to all the miners, and that it is adjusted externally based on the total computational power of the miners so that each new block is added to the blockchain approximately once every 10 minutes.

So that's how bitcoin uses the proof of work - what does the actual consensus protocol look like?

Bitcoin uses **Nakamoto consensus**:

- In any round, each miner attempts to extend (we say it *mines on top of*) the heaviest chain it sees.

→ By heaviest, we mean vaguely "the chain that has required the most compute power so far." For now, we can assume this just means the longest chain it sees.

- Then, each miner looks at the longest chain it sees, and confirms the block k steps away from the end (along with its prefix).

  → This is similar to what we were doing in streamlet; we confirm a block only once we know that it has been extended into a longer valid chain, to avoid an attack like the one we saw on teen streamlet.

This means the leader is whoever first creates a block with a valid nonce, so the leader selection process requires proof of work (and an adversary making multiple nodes does not have a higher chance of being selected as a leader).
Note that because the Nakamoto consensus does not care who created the blocks, our consensus protocol still works when miners are randomly joining or dropping out of the network. Since our protocol works in this setting, we say it has **dynamic availability**.

In general, consensus in the internet setting is characterized by **open participation**, which means that

- anyone (who meets some criteria) can create nodes to join the protocol, and in particular an adversary can try to create many nodes to over-represent themselves
- honest participants can come and go at will

In this setting, we watn to limit the adversary's participation (or have Sybil resistance), and have liveness in the protocol even when honest nodes join and leave at will (or dynamic ability).

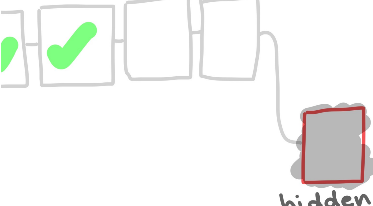
Bitcoin is not secure under partial synchrony; we want to show that it is secure under synchrony against a Byzantine adversary.

What is the best possible resilience of Bitcoin?

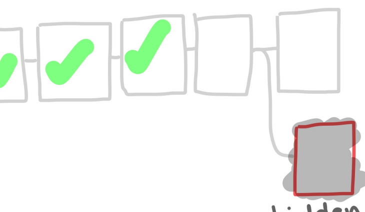
To answer this, let's look at one attack against this consensus protocol.

Remember that we have a k-deep confirmation rule for Bitcoin. We will say in this example that $k = 3$.

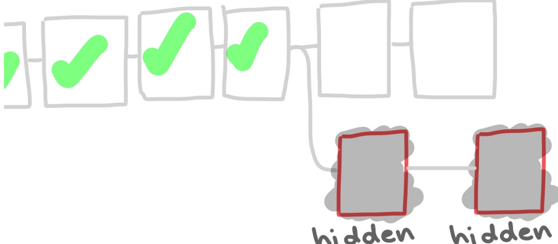
Let's say an adversary is able to create a block on the blockchain. Instead of sharing this block to get approved, they keep it hidden from all of the honest miners.



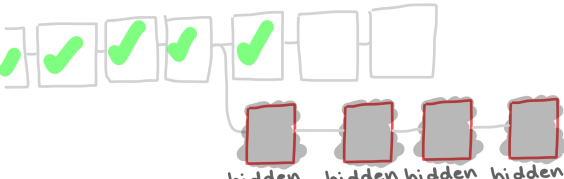
Then, because the miners do not see this hidden block, they keep building on the original chain.



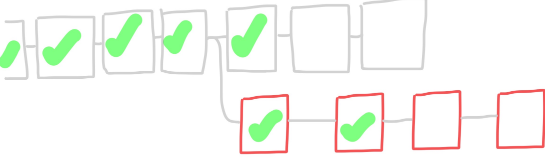
At the same time, the adversaries keep building on the hidden adversarial chain.



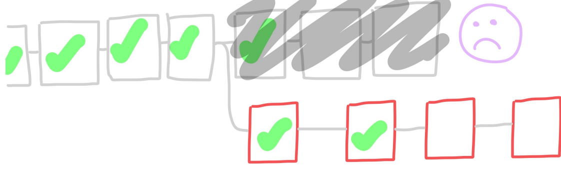
After there are 3 additional blocks on the honest chain, the first additional block to the honest chain is confirmed.



If at any point after this, there are more blocks on the adversarial chain than the honest chain, then the adversaries can reveal their hidden chain. This is now the longest (valid) chain that the miners see, so they confirm the 3rd-from-last transaction on the adversarial chain.



Now the adversarial chain is the longest valid chain all miners see, and of the additional transactions on the honest chain have effectively been un-done.



This is known as **Nakamoto's Private Attack**.

When is this possible? This requires some probabilistic analysis, but the answer is that generally, for large enough k, it is very unlikely that the adversaries are able to create a longer chain than the honest nodes unless they have more than half of the computing power.

So assuming our resilience $f < n/2$, we are secure against this attack.

Lecture 6: Consensus, III

Last lecture, we discussed how bitcoin uses proof-of-work in its mining process, to prevent the adversary from adding arbitrarily many nodes to the network. The proof-of-work process required coming up with a nonce such that the hash of the new block header (including this nonce) is at most $2^{256}/D$, where D is the difficulty. In expectation, it would take around D random guesses to come up with a working nonce, so we want D to be larger if the total computing power of the network is large and smaller if the total computing power is small.

The way bitcoin works is: at the start of each new **period** of 2016 blocks, we come up with a new difficulty D_i (and corresponding target $T_i = 2^{256}/D_i$). We want this new target to be an adjustment of the previous target, so that if we mined the previous 2016 blocks too quickly, we want to decrease the target (or increase the difficulty), and vice versa.

How do we get the time it took to mine the previous 2016 blocks? We look at the timestamps included in each block header.

For this adjustment, we have the equation

$$T_{i+1} = \frac{T_i t_i}{2016 \times 10 \text{ min}},$$

where T_i is the target for the previous period and t_i is the total time it took to mine the 2016 blocks in the previous period. We can see that this says, if our mining power is the same this period as it was last period, then we've adjusted our difficulty so it should now take us 10 minutes per block. But we have some rules to prevent extreme adjustment: we must have

$$\frac{1}{4}T_i \leq T_{i+1} \leq 4T_i.$$

Ok, so now that we know difficulty adjustment, we return to our question from last lecture:
Can we show that bitcoin is secure under synchrony against a Byzantine adversary? What would be the best possible resilience?

We showed last time that we need $f < n/2$, because otherwise we are vulnerable to a Nakamoto private attack. As a side note, we say that transactions lost in a Nakomoto private attack got **reorged** (short for "reorganized") because they used to be part of the longest chain but they no longer are.

But there is a concept called **forking** that will let an adversary successfully carry out a Nakomoto private attack even when $f < n/2$!

Here is my explanation of this from an Ed post:

> For a specific example, we can pretend $\beta = 40\%$. Then, after 1000 minutes, we expect that 100 blocks have been created, and since the adversarial miners have 40% of the mining power, we expect that they have found about 40 of these blocks. But every time they find a block, they add it to this adversary chain to try to carry out a Nakomoto private attack. This is what we mean by "the adversary chain grows at a rate proportional to β" because we expect that once 100 blocks have been created, 40 of them are actually in this hidden adversary chain.
>
> In theory, this would be ok, because it would mean the other 60 of them are in the honest chain, so the honest chain is longer than the adversarial chain, and the adversaries cannot carry out their private attack. But if there are network delays, then maybe some of the honest miners

don't immediately realize that B_1 has been added to the honest chain, so they keep searching for a block to be added to B_0, rather than B_1. If they only get the notification that B_1 has been added to the chain once they have found another block to be added to B_0, then this new block has gone to waste. This is the forking problem: due to network latency, some miners don't realize that the chain has already been extended, so they waste their time searching for useless blocks.

Let's say we lose half of the honest blocks due to forking (as in, half of the blocks that the honest miners found go to waste because they added on to an old part of the chain). Then, there are only 30 blocks in the honest chain, while we said before that there are 40 blocks in the adversarial chain, so the adversarial chain is longer than the honest chain, and the adversaries can carry out the Nakomoto private attack. This is bad!

So we want to reduce the proportion of the honest mining power lost to network delays; we want to reduce the proportion of mining time spent looking for useless blocks. We can't change the actual amount of time lost to network delays, because we don't really control the delays. But we can reduce the proportion of time they waste by increasing the amount of time it takes to mine a block, or increasing our difficulty D.

Theorem 6.1. If $\beta < 1/2$, then there exists some small enough mining rate λ, where λ depends on β and the network delay Δ, such that Bitcoin satisfies security (meaning safety and liveness) except with error probability $e^{-\Omega(k)}$ under a synchronous network.

Note that we can always force the mining rate to be less than λ (in expectation) by increasing the difficulty D.

This is we require one block every 10 minutes rather than, e.g. one block every second.

Is this the best we can do?
No! Bitcoin only works under synchrony, while streamlet works under partial synchrony. Also, the reason we say we *confirm* blocks, rather than *finalizing* them, is because there is some error probability when we confirm our blocks.

Also, because we are using proof-of-work, which requires a lot of computing power, Bitcoin uses a lot of energy - as of 2021, it uses more energy than the Netherlands. ☹

This is why Ethereum has now moved from proof-of-work to proof-of-stake, which uses coins rather than computing power to prevent the adversary from arbitrarily creating new nodes. Proof-of-stake ethereum adds to the open participation and block reward features of blockchain to include **finality and accountable safety**, and a method called **slashing** which creates a punishment for dishonest participants.

In a proof-of-stake consensus protocol, the nodes lock up (or **stake**) some of their coins to be eligible to participate in consensus; the more coins they lock up, the more their vote counts and the higher their chances are of being elected as leader.

If a node is caught doing adversarial things, then their stake can be burned as punishment; this is called **slashing**.

This allows us to create accountability for the miners (or a way of punishing dishonest miners) which we did not have in Bitcoin.

Definition 6.2. If a protocol has an **accountable safety resilience** of $n/3$, it is secure whenever there are less than $n/3$ adversarial nodes AND if there is a safety violation, all observers of the protocol can provably identify up to $n/3$ of the protocol violators. Also, no honest node can be falsely accused.

So this is *stronger* than safety because even when there are more than $n/3$ adversaries, we can catch (some of) them if they do anything bad!

Definition 6.3. We say that a protocol provides **finality** if it preserves safety during periods of asynchrony.

So our ideal internet consensus would have:

- Sybil resistance (via proof-of-work or proof-of stake)
- dynamic availability (so that even if there is low participation, transactions can still be processed)
- finality and accountable safety

Is there an SMR protocol that provides both dynamic availability and finality? No ☹

There is a theorem that shows that this is impossible - it is in the optional slides that were not covered in lecture.

But we can deal with this by splitting our chain into two parts:

- we say that some prefix of our chain is the **finalized chain** which requires synchrony and sufficient participation to be live, but is always safe
- if we don't have synchrony or if the participation drops too low, we just add to the end of our chain; this unfinalized part is called the **available chain**. this is safe and live under synchrony and dynamic participation, but it is not finalized

I am skipping writing notes for lectures 7 and 8 because I am out of time ☹ However, Aditya Saligrama has notes for these lectures here and here.

Lecture 9: Stablecoins

A quick review of Solidity:

- everything is in the form of a **contract** (similar to a class in an object-oriented language)
- contracts have state variables which they can modify and functions that can be called externally
- they can **inherit** code from other contracts
- there are multiple types of contracts: `contracts`, `interfaces`, `abstracts`, and `libraries`
- there are some global objects, such as `block` and `tx`

Definition 9.1. A **fungible token** is interchangeable; for example, a dollar is fungible because my dollar bill is functionally identical to yours, and it doesn't make a difference if we swap them.

Example 9.2. One of the most important type of contract in Ethereum is the **ERC20 contract**. Each ERC20 contract corresponds to a new type of currency that can be exchanged.

The ERC20 contract just has an array called `balances` (in its state space) and six functions:

```
function transfer(address _to, uint256 _value) external returns (bool);
function transferFrom(address _from, address _to, uint256 _value) external returns (bool);
function approve(address _spender, uint256 _value) external returns (bool);
function totalSupply() external view returns (uint256);
function balanceOf(address _owner) external view returns (uint256);
function allowance(address _owner, address _spender) external view returns (uint256);
```

If we have funds in an ERC20 contract, then this would be stored under our name in `balances`, and since this array is public (like all of Ethereum), we can look up other people's balances using the `balanceOf` function. We spend our balance using the `transfer` function; this automatically sends the balance from `msg.sender`.

The code for the ERC20 transfer function looks like this:

```
contract ERC20 is IERC20 {
  mapping (address => uint256) internal _balances;

  function transfer(address _to, uint256 _value) external returns (bool) {
    require(_balances[msg.sender] >= _value, "ERC20_INSUFFICIENT_FUNDS");
    _balances[msg.sender] -= _value;
    _balances[_to] += _value;
    emit Transfer(msg.sender, _to, _value);    // write log message
    return true;
}}
```

Note that we avoid underflow errors by checking that `_balances[msg.sender] >= value` before transferring the money, and the compiler automatically checks for overflow errors before running the code.

To call functions in other contracts, we have an address for that token (stored as an address). Then, we cast that address to the desired type, say `ERC20Token`, and then we can call messages specific to that contract.

The way that contracts work in practice is:

- On the chain, we have a bunch of different contracts: say, `ERC20 Coin A`, `ERC20 Coin B`, `ERC721 NFTD`.
- We have a bunch of services that interface with the blockchain, such as Compound.
- The end-user Alice talks to Compound and asks to borrow, for example, 100 Coin B tokens. Compound then calls the `transfer` function within the Coin B contract and asks to transfer 100 tokens from the Compound account to Alice's account, effectively sending 100 tokens to Alice.

Definition 9.3. A **StableCoin** is a coin designed to trade at a fixed price; for example, one coin could always be worth 1 USD.

Definition 9.4. A **collateralized** StableCoin is backed up by some other asset, while an **un(der)collateralized** StableCoin is not.

A **centralized** StableCoin is managed by a central party, while an **algorithmic** StableCoin is just run by public code on the chain.

Example 9.5. Types of stablecoin are:

- custodial stablecoins are centralized and collateralized (such as USDC)
- digital currency for a central bank are centralized and un(der)collateralized
- synthetics are algorithmic and collateralized (such as DAI)

For a custodial stablecoin, such as USDC:

- a centeral custodian (such as a bank) will deploy a ERC20 contract on the chain, with a `balances` array
- Bob gives the custodian 100 USD
- in response, the custodian calls the `mint` function on its contract to mint 100 USDC and store it in the `balances` array under Bob's name
- Alice gives the custodian 7 EUR
- in response, the custodian calls the `mint` function on its contract to mint 6.8 USDC and store it in the `balances` array under Alice's name
- then, when Bob wants to transfer USDC to Alice, he does not need to involve the custodian at all, because he can just call the `transfer` function on the contract, and it will transfer money from his balance to Alice's
- if Bob wants to withdraw money, he will ask the custodian to withdraw 60 USD for him, and the custodian will call the `burn` function on the contract to burn 60 USDC from Bob's balance. then, it will send 60 USD to Bob.

This gives the custodian very strong powers: it can censor customers or arbitrarily burn money from people's balances.

Can we have a stable asset that is not run by custodians?

These are known as **synthetics**.

The collateral also cannot be something custodial like the USD; it will have to be a cryptocurrency. But most cryptocurrencies, like ETH, are not stable.

Example 9.6. One solution to this is the **MakerDAO** system.

In this system, there are two tokens. DAI is the stable coin (and, last year, was always worth between 0.99 and 1.01 USD) and MKR is used for governance.

Alice has 1 ETH and wants to pay Bob in DAI. How does she do this?

First, she creates a `vault` on the MakerDAO contract. This consists of a wallet that she controls and a vault for locked funds.

Then, she deposits her 1 ETH into the vault, and at the time she deposits it, her 1 ETH is worth 3000 USD. Then, she withdraws 2000 DAI from the vault into her wallet, using her locked ETH as collateral. This is known as **minting**.

Now, she can pay Bob using the DAI in her wallet, and at any time get her 1 ETH back by repaying her debt in DAI.

Alice has to pay interest on her borrowed DAI; this is known as the **stability fee**. So her debt started out at 2000 DAI, and after some time, it goes up to 2001 DAI, and so on.

Then, if the value of DAI dips below a dollar, the stability fee increases. This encourages people to repay their loans, which causes less DAI to exist in the world, so the value of DAI goes up. If the value of DAI goes above a dollar, the stability fee decrease, which encourages people to take out more loans, causing more DAI to exist in the world, so the value of DAI goes down.

At any point in time, your collateral must be at least 130% of your debt. If your debt increases so this is no longer true, portions of your collateral will be sold on the open market and used to repay your debt until this percentage is achieved.

The rest of this lecture is a guest lecture by Griffin Dunaif, about NFTs.

Lecture 10: Decentralized Exchanges

This is a guest lecture by Dan Robinson, at Paradigm.

Definition 10.1. A **decentralized exchange** is a type of decentralized application, built with smart contracts on the blockchain. It allows users to directly trade ERC20 tokens or NFTs.

Advantages of a decentralized exchange:

- non-custodial: there is no third-party custodian with power over your exchanges (unlike something like Coinbase)
- censorship-resistant: anyone who can use the blockchain can make a decentralized exchange
- permissionless
- convenient: don't have to deposit into an external system (unlike the MakerDAO example)
- programmable
- atomic

Negative aspects include: transaction fees, because we are trading directly on the chain.

Types of Decentralized Exchange:

1. **on-chain orderbook**

   Market-makers place orders on the chain, and users fill them on the chain. But these are gas-inefficient, because every step requires additional gas.

2. **off-chain orderbook** (such as OpenSea)

   Market-makers sign their orders off the chain and users sign the orders and then submit it to the chain. This improved some of the gas inefficiency, because now orders only cost gas when they were filled, but it is less programmable because smart contracts cannot see orders unless you manually show it to them.

   (We might be concerned about censorship issues with sharing orders off chain. It is true that there will be censorship issues if a specific bulletin board refuses to post your order, but your order can still be filled if you post it somewhere else, and a bulletin board service cannot prevent anyone from signing a given order.)

3. **Dutch auction**

   In a brief, intuitive sense, Dutch auctions are the opposite of English auctions, where users place their order on the chain at an extreme price, and then the price adjusts over time (lowering if it is an offer and increasing if it's a bid) until someone fills it.

   The main problem with this is it's very slow, because we have to wait for the price to adjust to market value.

4. **automated market maker** (most decentralized exchanges use this)

   Market makers deposit their assets into a pool, and users trade with the pool at an algorithmically determined price. This is gas-efficient, makes it easier for regular people to be market-makers, and is very programmable (it is easy to make a smart contract that can automatically check the state of the pool).

How do automated market makers work?

Let's consider something just comparing two assets, because it's easier for us to visualize (automated market makers actually work with many different assets at once). As an example, we have ETH, which is risky, trading against USD, which is stable.

The automated market maker has reserves, where it has x ETH and y USD. It offers to buy or sell ETH at some price p (in USD). At any point, p must be a function of the reserves x and y, along with possibly some outside information. The reason for this is: if your reserve x is going down, then it's a sign that a lot of people believe it's better right now to have 1 ETH than p USD, so you should be charging more for your ETH.

How to make your own AMM (automated market maker):
Let's say we want an AMM that has a 50/50 portfolio of ETH and USD - this means that if we have x ETH (which we think is worth p USD each) and y USD, we want $x \cdot p = y$; we want the value of our dollars and our ETH to be the same.

Rewriting this equation, we get that we want our marginal price to be $p = y/x$ when someone makes a small trade. Adding a bit of calculus, we represent "small trade" as $-\frac{dy}{dx}$ because we expect that y decreases a small amount and x increases a small amount when we make this trade, so we want

$$-\frac{dy}{dx} = y/x,$$

which has the solution $y = k/x$, for a constant k (this means that if we start off with some initial x and y, we want to set a price that keeps $x \cdot y$ constant).

We call this the **constant product market maker**.

You can see some visualizations of this here.

Another way of expressing the constant product market maker is:

We have a given **liquidity** $\ell = \sqrt{xy}$. We recall that $p = y/x$, so then we can get that

$$x = \ell/\sqrt{p}, \qquad y = \ell\sqrt{p}$$

with a bit of algebra.

So most trades we make against a market maker are in the form of changing p but keeping ℓ constant, or changing ℓ but keeping p constant.

Another market maker is the **constant sum market maker**, which will always trade at a constant price.

This means that $p \cdot x + y$ will always stay constant.

And there are many other rules we can use to create automated market makers!

Areas to improve in AMMs:

- decreasing gas costs
- slippage: someone could trade right before you, drive your cost up, and then sell back their assets at the slightly higher price
- impermanent loss: because you are selling as the price goes up, you make less money than if you had sold it all at once once the price reached its peak

could instead use a rebalancing strategy, where you just trade to fit your equation at the end of the day, rather than constantly - this does better in practice

Lecture 11: Lending Systems

So far we have talked about:

- how consensus protocols work
- Bitcoin (using UTXOs and the Bitcoin scripting language)
- Ethereum (the EVM and Solidity)

Now, we are talking about *decentralized finance*. We talked about exchanges and stablecoins, and today we are talking about lending.

For the rest of the course, we will discuss:

- privacy on the blockchain
- scaling the blockchain
- interoperability across blockchains

But this is up to you! If there is something you want to learn about blockchains that isn't on this list, send the teaching team an email!

> **Reminder 11.1.** Our goal for this class is to approach this topic from a computer science perspective. This is not a finance class, so the material covered here should not be taken as financial advice.

What do banks do for the economy?

They bring together **borrowers** and **lenders** to make it easier for people to borrow or lend money. The lenders (or liquidity providers) get interest on the money they deposit, and the borrowers pay interest on the money they borrow.

The difference between the borrow interest and the deposit interest is called **spread**, and it is how the bank makes money.

The bank also aborbs risk for the lenders, in the sense that if Bob the borrower defaults on his loan (so he doesn't pay it back), the bank takes the loss, and still gives Alice the lender her money back.

In the crypto world, centralized finance, or **CeFi lending**, works exactly like a bank.

But the difference between the real world and the crypto world is that in the real world, the bank can assume Bob probably won't default on his loan, and can take legal measures if he does. But in the crypto world, Bob is completely anonymous, so it is very easy for him to run away with the money he borrowed.

How do the CeFi systems discourage this?

The CeFi systems require Bob to lock up some **collateral** for his loan.

Let's say Bob wants to borrow 1 ETH, and at the time, 1 ETH is worth 100 UNI (as in Uniswap tokens). He locks up 500 UNI as collateral and gets 1 ETH back; at this time, his loan is **over-collateralized**, and we say that Bob's **debt position** is:

$$+500 \text{ UNI}$$
$$-1 \text{ ETH.}$$

From here, three things can happen:

1. Bob repays his debt:
   At this point, the bank gives him back his 500 UNI, minus any interest he had to pay on his loan, and his debt position is cleared out.
2. Bob defaults on his loan:
   At this point, the bank keeps 100 UNI, which is the equivalent of the 1 ETH he borrowed, plus some penalty fee for him defualting, plus the interest he collected on the loan so far, and gives Bob the rest of his UNI back. Again, his debt position is cleared.
3. the value of the loan increases relative to the collateral:

   Let's say that all of a sudden, 1 ETH is worth 400 UNI. Then, the bank is no longer happy with the ratio of debt to collateral, so they **liquidate** the collateral by selling it on the open market. They give Bob back his remaining 100 UNI, minus a penalty fee and the interest, and his debt position is cleared.

Let's review some of the vocabulary we used so far:

Definition 11.2. The **collateral** is assets that Bob gives to the bank as a form of a security deposit against his loan.

Definition 11.3. **Over-collateralization** means Bob has to provide collateral that's worth more than the loan he's taking out.

Definition 11.4. **Under-collateralization** means Bob's collateral is worth less than the loan he took out.

Definition 11.5. The **collateral factor** is some number between 0 and 1, which is the maximum-value loan that can be borrowed using 1 unit of collateral. This is decided by the bank, and is based on the type of the collateral.

Definition 11.6. When the value of the collateral drops (or the value of the loan increases) to the point where the ratio

$$\frac{\text{Bob's loan value}}{\text{Bob's collateral value}}$$

is above the *collateral factor*, the bank no longer feels comfortable with the loan it gave out, so it sells some of Bob's collateral on the open market, and clears that portion of Bob's *debt position*, until the ratio drops back below the *collateral factor*. This process of selling a portion of the collateral is known as **liquidation**.

For low-volatility assets, it's unlikely that liquidation is necessary, so the CeFi would set a high collateral factor to encourage people to borrow from that CeFi.

For high-volatility assets, the bank wants to give itself time to liquidate the collateral before the value of the collateral drops below the value of the loan (and the bank starts losing money), so it sets a low collateral factor.

Example 11.7. In Compound, the lending process is based on computing the **health** of each person's

debt position, where the health is calculated as

$$\text{BorrowCapacity} = \sum_i \big(\text{value}(\text{collateral}_i) \times \text{CollateralFactor}_i\big)$$
$$\text{health} = \frac{\text{BorrowCapacity}}{\text{value}(\text{TotalDebt})}.$$

If the health ever drops below 1, liquidation is triggered until the health goes back over 1.

But the problem with CeFi lending is that it relies a lot on *trust*:

- we are trusting the CeFi organization to not run away with our money, or to not get hacked or make a bad calculation
- the CeFi gets a significant portion of the interest, rather than the lender herself; this spread (borrow interest minus deposit interest) is decided entirely by the CeFi

But the point of blockchain is to not rely on trust in financial institutions!

So let's write some code that can act as a substitute for this centralized lending system.

We call this **DeFi lending**, which is short for decentralized finance.

The first idea we might come up with for a DeFi system is called a **lending order book**.

Here, lenders post on the order book (on-chain) to say that they have X amount of a given asset, and they will lend it with a given interest rate. Then, borrowers come along, and post on the order book that they want to borrow Y amount of a given asset, for a given interest rate. The order book algorithm then matches borrowers and lenders with compatible amounts and interest rates.

But this doesn't work well:

- It's computationally expensive, because every time someone posts an offer or wants to change their current offer, they have to post more information to the blockchain.
- It's risk-heavy, because loans are never spread out over multiple borrowers - one borrower has all the money you loaned out, so if that person defaults, you lose all your money.
- It's difficult to withdraw money, because you have to wait for your corresponding borrower to return your money.

A better approach to this is called **liquidity pools**:

Here, lenders send in their assets, and it gets added to a pool of all assets of that type. When borrowers want to borrow money, they don't have to wait to get matched with a specific lender, because they just borrow from the pool as a whole. To do so, they have to over-collateralize they loan by supplying a different asset to the pool as well.

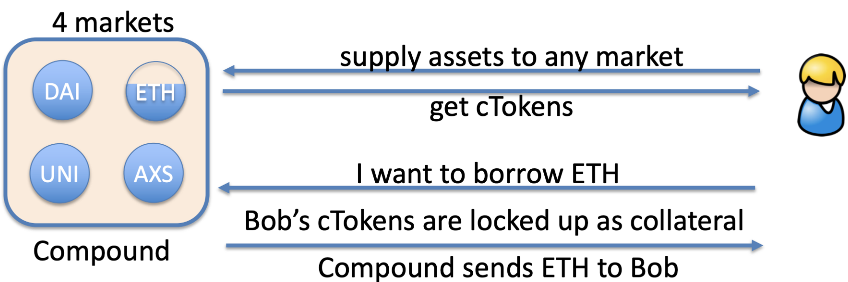
How do we record who gave how many assets?

Well, when someone sends in, e.g. 1000 DAI, then they will get back Y cDAI, where Y is determined by the current exchange rate, which changes every block. This exchange rate increases over time, which means if you wait to redeem your cDAI you will get more DAI in return; this is how interest is collected.

This also allows you to use your collateral or the money you loaned out, because you can trade your cDAI in the open market; in fact, you can even trade cDAI back for regular DAI.

This is a cool feature of the liquidity pools, because it makes it very easy to give your "locked up" liquidity to someone else, and they can redeem it, along with the accrued interest, later on by simply trading the cDAI back to the liquidity pool.

Borrowing from the DeFi works similarly:



Bob pays interest on his loan, because when he trades his ETH back for c tokens, the exchange rate will have increased, so he gets back fewer c tokens than he originally deposited.

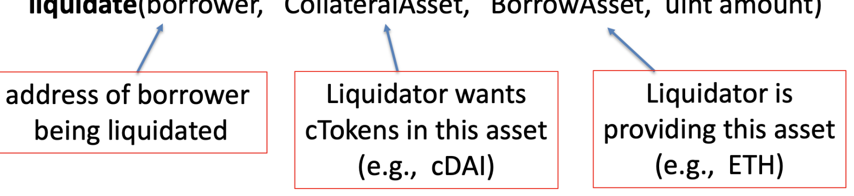
Specifically, the exchange rate (how many ETH you get for 1 cETH) is calculated as

$$\text{exchangeRate} = \frac{\text{underlying balance} + \text{total borrow balance} - \text{reserve}}{\text{cETH supply}},$$

where the underlying balance is the total number of ETH supplied to the pool and total borrow balance is the total number of ETH borrowed from the pool.

It is possible to show that there are no easy ways to make "free money" against this DeFi, but this requires analysis of this algorithm to show.

If a borrower's health goes below 1, then anyone can liquidate the borrower's collateral, by calling the function:

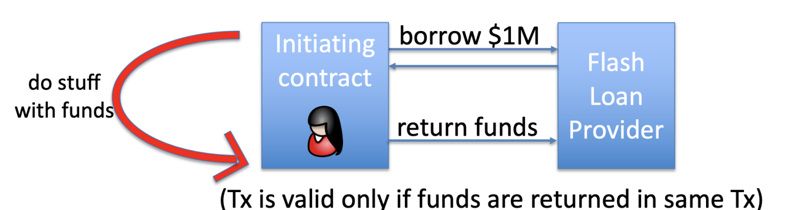


This function transfers liquidator's ETH into ETH market, and gives the liquidator cDAI from user's collateral

The important part of DeFi is that these are entirely algorithms that are written beforehand; there is no singular organization who can randomly change how the lending protocol works or run away with your money, and the code is available for anyone to read before they make their trades.

Flash Loans are a (technologically) cool thing that is only possible in crypto.

Definition 11.8. A **flash loan** is a loan that is taken and repaid in a single transaction. This means there is no collateral required, because there is zero risk for the lender.



These are useful for:

- risk-free arbitrage
- collateral swap
- price oracle manipulation in attacking a DeFi

Example 11.9. Let's look at how to use flash loans for **risk-free arbitrage**. Arbitrage is when you can trade against two different people for two different exchange rates, so you can borrow an asset for cheaper than you sell it for, and make free money from this difference.

Let's say Uniswap is trading at an exchange rate of 1 USDC = 1.002 DAI, and Curve is trading at an exchange rate of 1 USDC = 1.001 DAI.
Then, we can take out a flash loan from Aave (a flash loan provider). In a singular transaction, we:

- take out a loan of 1M USDC
- trade against Uniswap to get 1.002M DAI
- trade against Curve to get 1.001M USDC
- return our 1M USDC and keep our free 1000 USDC

Note that in this process, Curve and Uniswap's algorithms adjust their exchange rates based on this trade, and their exchange rates become more equalized. So we cannot do this for infinite money, just until the point where the two exchange rates become equalized.

Lecture 12: Legal Aspects and Regulation

This is a guest lecture by Jake Chervinsky, from the Blockchain association. These notes are a very brief summary; I highly recommend that you watch the lecture recording if you have not already.

Definition 12.1. A **regulation** is a rule that prohibits certain conduct or that requires your conduct to meet certain obligations, and call for specific punishments for violators.

We have regulations in order to: control market power, facilitate competition, protect consumers, promote stability, achieve governmental interests, ...

Basics of Regulations

Who makes regulations?

In the US, regulations are made at both the state and federal level. The legislature makes the laws, the executive office enforces the laws, and the judiciary interprets the laws.

How should you think about regulations?

(I am not your lawyer, and I am not here to give you advice - please don't take anything here as legal advice.) It is good to learn the basics of regulations so that you know when you may be running into legal issues, but hire a lawyer before you do anything potentially risky.

Regulators and Frameworks

The top 5 regulators for crypto:

- SEC (securities and exchange commission)

  Mission: to protect investors, facilitate capital formation, and maintain fair, orderly, and efficient markets

  The federal securities laws arose in the 1930s after the Great Depression, since this was thought to be caused by financial abuse and information assymetry.

  The SEC places regulations on "securities," but these regulations require central parties that are registered with the SEC, making it is nearly impossible to legally trade securities in a decentralized manner on the blockchain. Thus, people in the crypto industry put a lot of effort into making sure their digital assets are not considered securities.

  A specific example of this is the **Howey test**, where a digital asset is considered an **investment contract** (a type of security) if the SEC can prove it has four given properties. The fourth property says that people who own the asset expect to profit "based on the efforts of others," and reading into past case law, this means that the issuer of the asset is putting in some sort of work to make the asset more valuable over time. Many people in the crypto industry try to argue that their asset is not an investment contract because there is no *centralized* issuer who is trying to make the asset profitable.

  In 2018 and 2019, it was decided that Bitcoin and Ether are "sufficiently decentralized," so they do not count as investment contracts. However, in more recent years, the chair of the SEC has declared that almost all other digital assets are securities, and they have been regulating by *enforcement* rather

than outlining more specific rules about how the Howey test applies to digital assets.

- CFTC (the commodity futures trading commission)

  Mission: to promote the integrity, resilience, and vibrancy of US derivatives markets (such as futures, options, and stocks)

  There is currently a sort of "turf war" about whether digital assets should be considered commodities or securities; the industry generally hopes that they will be considered commodities, because the head of the CFTC is more crypto-friendly than the head of the SEC.

- FinCEN (the financial crimes enforcement network)

  They enforce the anti money-laundering laws, and laws against financing terrorism.

  Specifically, they have a Bank Secrecy Act which requires that banks (and centralized financial institutions) collect and report the personal information of all their customers.

  However, if you are in a decentralized financial system on the blockchain, there is no central party to record your information and report it to FinCEN, so there are currently legal battles around if and how such transactions should be surveiled.

- OFAC (office of foreign assets control)

  Sanctions are a foreign policy tool used to economically penalize certain foreign countries. OFAC makes a list of special parties (such as foreign government officials and people from entire countries) that people from the US are not allowed to trade with.

  Sanctions laws have *strict liability*, which means even if we traded with someone on this list unknowingly, we are still subject to punishment from the law.

- IRS (internal revenue service)

  There are many unanswered questions about how aspects of crypto, such as mining and DeFi transactions, should be taxed.

  Last year, the Infrastructure Investment and Jobs Act (IIJA) appended the Tax Code with five provisions specific to crypto, which are all problematic; they generally require you to report all your transactions, to the point of requiring you to know your counterparty in large trades - this is the subject of a major lawsuit at the moment.

<u>Current Issues and Priorities</u>

- Tornado Cash:
  This was a tool, deployed at the start of Ethereum, that allowed people to pool all of their assets and then get their money back in a new wallet, unlinked to their depositing account, so that they could get more privacy over their financial transactions.

  This was used by a lot of money launderers, and the OFAC ended up sanctioning the Tornado Cash protocol. Previously, sanctions were just used against people, not types of software, and there are two lawsuits in the court right now challenging the idea that OFAC can sanction a piece of software.

- Ooki is a DAO where people can get governance tokens and then be allowed to vote on changes to protocol; regulators want to treat these governance token holders as a central party responsible for the trades on Ooki.

Lecture 13: Privacy on the Blockchain

This week, we start a new unit on how to do privacy on the blockchain. This will go into the world of zero-knowledge proofs and SNARKs; we will spend this week talking about applications of these two, and then next week we will go into the math of how these work.

(This lecture starts with a short review of Uniswap and automated market makers, which I am not covering in the notes, but I recommend you watch the lecture if you would like to better understand these ideas!)

Why do we need privacy on the blockchain?

- manufacturers don't want everyone to know how much they pay for individual parts
- companies that pay employees in crypto want their employees and salaries to be private information
- end-users want their rent, donations, and purchases to be private
- business want their "business logic" in smart contracts to be private

(Dan Boneh is offering a new course on DAOs in the spring, jointly with a political science professor.)

There are two types of privacy:

- **pseudonymity**: everyone has a long-term consistent pseudonym

  This allows people to build up a reputation, but it also means that all of your transactions are linked, so they might collectively reveal a lot of information about who you are. For example, if I only use my pseudonym (the hash of my public key) during working hours in California, it becomes clear over time that I probably live on the West Coast. If I regularly use this to buy coffee at Coupa, then I probably live on or near campus. If I go on a trip and buy something in an East Coast store, then someone can figure out that I am on vacation.
- **full anonymity**: user's transactions are unlinkable - you cannot tell that two distinct transactions came from the same user

Of course, there are two sides to privacy - we would like to have privacy in our day-to-day operations, but we also know that full privacy helps people conduct illegal or very unethical operations, without others knowing.

Thus, there is an important question of: privacy from who?

- In Bitcoin and Ethereum, there is **no privacy**: everyone can see all transactions.

  What would a world with no privacy look like?

  People would pretty quickly figure out how to make their own transactions private; for example, if you don't want everyone to know you went to the doctor, you might convince a stranger to pay the doctor for you, in exchange for some sort of currency or goods outside of the blockchain.
- In standard bank systems, there is **privacy from the public** but there is a trusted operator that can see all transactions.
- Under **semi-full privacy**, the trusted operator cannot see transactions, but local law enforcement can.
- A service called Tornado used to provide **full privacy**, so that no one could see any transactions. We will talk about this more next lecture.

There is no correct answer to the question: which of these worlds should we live in? Everyone has different opinions, and it is up to us as a society to decide which to build.

The challenge is: how do we support positive applications of private payments, while preventing negative or criminal ones?

While this may seem like a policy question, we can actually use computer science techniques, such as zero-knowledge proofs, to help us create privacy while ensuring legal compliance! We will discuss this more in the next few lectures.

Do we have privacy in Bitcoin or Ethereum?

We know that all transactions are public, and everyone can see the "from" addresses and the "to" addresses. However, it is also true that any user can generate many different public keys, and does not need to announce which public keys are hers. Would this grant her anonymity in her transactions?

Well, let's say we have the following transaction between Alice and Bob:

inputs: A1: 4, A2: 5 outputs: B: 6, A3: 3

If Bob is a merchant, and Alice is buying a book from him, then Alice learns that the address B is linked to this merchant, and Bob learns that $A1, A2$, and $A3$ are Alice's addresses.

If Bob is a bank, and Alice is doing an exhange with him, then Bob knows that $A1, A2$, and $A3$ are linked to Alice, and because of the Know Your Customer laws, is also required to report this information to the government.

But is that it? If the bank just knows $A1, A2, A3$, can Alice just create new addresses for other transactions and move on with her life?

Well, there are a few de-anonymization strategies that allow us to link different addresses together. We have the following two heuristics:

- If multiple addresses are listed as transaction inputs, they probably belong to the same user.
- Remember that, in most transactions from Alice to Bob, the combined value of her inputs ends up being more than what she actually wants to send Bob, and she just sends this extra money back to herself.

  We say that this extra address is a *change address*, and it belongs to the same user that owns all input addresses.

  How do we know which of the outputs is the change address?

  Well, if one of the outputs receives less BTC then it is probably the change address, because if the actual output was worth less than the individual inputs, we wouldn't need multiple inputs.

One paper used these heuristics to see how many users they could de-anonymize.
They found 3.3 million clusters of addresses (where each cluster corresponds to a single user) by applying these two heuristics.
Then, they identified 1070 addresses as the publicly-available addresses of merchants such as Coinbase, which allowed them to figure out which clusters belonged to such merchants.
They also realized that 15% of all clusters interacted with one of these merchants at some point in time, which means anyone with authority could find out who they are due to KYC laws, and then know exactly

how much Bitcoin each person owns.

It is kind of surprising that even such simple heuristics can de-anonymize so much of the blockchain! With even more data available and more complicated heuristics, there are groups that are able to de-anonymize almost every Bitcoin transaction.

Companies such as Chainalysis or Elliptic that do such de-anonymization processes commercially; they sell, for example, a verification service that checks whether people you are trading with are on a blocklist, so you don't make any illegal trades.

So, can we build private coins on a public blockchain?

As we mentioned before, if you want to privately pay a doctor, you wouldn't pay them yourself, you would get a random stranger to pay them for you.

We can formalize this process in what's called a **mix**.

What is a mixer?

A mixer is a service on the internet (it is important that it is on the internet and not on the blockchain) that randomizes people's addresses for them. Let's look at how this works.

Attempt 1: Simple Mixer

Alice, Bob, and Carol each have 1 BTC that they want to send privately. They each post transactions on the blockchain that send their 1 BTC to the mixer's address M. Then, they each generate new (randomized) addresses X, Y, Z and send this to the mixer privately. For full security, this should be done in some encrypted way.

Then, the mixer sends 1 ETH each to the new addresses X, Y, Z, in a random order. Then, everyone knows that, for example, X belongs to Alice, Bob, or Carol, but no one besides the mixer knows who out of the three people it belongs to.

We say that this creates an **anonymity set of size 3**.

But the problem with this is that it requires a lot of trust in the mixer - there is no obligation for them to return their money to you.

We will begin by fixing the problem of this being a very small anonymity set. If Alice wants more anonymity, she could continue mixing with more people. After the first round, she has an anonymity set of size 3. If she mixes with 3 more people in the second round, and all of them have already mixed once, then she ends up with an anonymity set of size 9, and she can keep going this way to get exponential anonymity.

The other benefit of repeated mixing is that, if we use different mixers each time, then even if one of the mixers reveals their information, it is not enough to figure out Alice's address in the end.

Following the theme of this course, the natural next question is: can we mix without having to trust a third-party mixer?

The answer is yes! There are on-chain mixing services such as CoinJoin on Bitcoin and Tornado Cash on Ethereum.

Let's look at how CoinJoin works:

Example 13.1 (CoinJoin). Let's say Alice, Bob, and Carol want to mix together. Alice has the UXTO $A1$, which has 5 BTC, Bob has the UTXO $B1$, which has 3 BTC, and Carol has the UTXO $C1$, which has 2 BTC.

In some sort of public forum, such as PasteBin, Alice writes down:

- her input address $A1$
- that she has 5 BTC (this is verifiable because anyone can check the value at $A1$)
- her change address $A3$

Bob and Carol write down their corresponding information.

Then, they will anonymously post mix addresses $B2, A2, C2$. Everyone can see the mix addresses, but no one knows which address corresponds to whom.

Then, all three of them prepare and sign a transaction with the following information:

- **inputs**: $A1 : 5$ BTC, $B1 : 3$ BTC, $C1 : 2$ BTC
- **outputs**: $B2 : 2$ BTC, $A2 : 2$ BTC, $C2 : 2$ BTC, $B3 : 1$ BTC, $A3 : 3$ BTC

and then one of them can combine all of their signatures and post this transaction to the blockchain.

In order for the outputs to be anonymous, they all need the same output value, so each person gets the same amount of bitcoin (and then the *publicly known* change addresses for Alice and Bob get back the remaining money from their inputs).

In practice, each CoinJoin transaction mixes about 40 inputs. But there is a drawback: because all users have to sign the transaction, any of them can disrupt the process just by refusing to sign. This also means no one has to trust the mix – if you aren't going to get your money back, you would just refuse to sign the transaction, and then the process would not go through.

Let's take this a step further:

Can we have fully private transactions on a public blockchain?

We would naively think: no, how could anyone verify transactions if they are private?

But crypto magic actually allows us to do private transactions on a publicly verifiable blockchain! We will use two tools, called **commitments** and **zero-knowledge proofs**.

We will explain how these tools work soon, but for now, let's see how we would use them to allow private transactions.

A (hiding) commitment is a short amount of data that reveals nothing about the actual transactions, but, if the user later has to reveal their transaction, they cannot pretend it was something else - there is no other transaction they can reasonably come up with that would match the commitment.

Then, on the public blockchain, we would just have a commitment to the previous state, a commitment to the new transaction, a zero-knowledge proof, and then a commitment to the new state after the transaction.

How do you verify that this transaction is valid?

This is the magic of zero-knowledge proofs! The zero-knowledge proof provides a (publicly verifiable) proof

that the transaction is consistent with the previous state and that the new state is actually what the blockchain would look like after the transaction, without revealing information about what this transaction or state actually is.

We can begin talking about how this works technically.

We say that a **commitment scheme** consists of two functions:

- `commit(msg, r)` which takes in the message and some secret random bits, and outputs a short "commitment string" `com`
- `verify(msg, com, r)` which takes in the message and the random bits from before, as well as the commitment string. It outputs `accept` if `com` is the true output of `commit(msg, r)` and `reject` otherwise

We say that commitments have to be **binding** and **hiding**.

Definition 13.2. We say that a commitment is **binding** if no efficient adversary can come up with some $(m_1, r_1), (m_2, r_2)$ such that $m_1 \neq m_2$ but

$$\texttt{verify}(m_1, \texttt{com}, r_1) \text{ and } \texttt{verify}(m_2, \texttt{com}, r_2)$$

both accept.

That is, no adversary can come up with a new message that maps to their old commitment string - once they have committed to something, they cannot change the message.

Definition 13.3. We say that a commitment is **hiding** if `com` reveals nothing about the committed data. Formally, we say that if `commit(m, r) = com`, and r is selected uniformly at random from some set R of random strings, then `com` is statistically independent of m.

How do we do this?

Example 13.4. One example is **hash-based commitment**. We take H to be a collision-resistant hash function that maps (m, r) pairs to some output c, where the set of all possible random strings is much larger than the set of all possible outputs (practically, this means we need to input a very large random string, but we get a very short commitment out of it).

Then, we just have out `commit` function output $H(m, r)$ and we have our `verify` function accept if $c = H(m, r)$ and reject otherwise.

This is binding because H is collision-resistant.

This is hiding if we add a mild assumption to our hash function H; it turns out this is true for most hash functions we use, as long as our random string is large enough (compared to the length of our output).

Now, we will change topics, to talk about how to build succinct zero-knowledge proofs.

The intuition is:
A **SNARK** (succinct non-interactive argument of knowledge) is a succinct proof that a certain statement is true.

Example 13.5. If our statement is

"I know a message m such that SHA-256$(m) = 0$"

then a trivial proof would be the message m.

But a SNARK needs to be *short* and *fast to verify.* If the message m is very large, then the trivial proof is neither.

Even though the message could be 1 GB long, we will show (next lecture) a proof that is only 400 bytes and takes milliseconds to verify!

A **zk-SNARK** is a SNARK with the additional property that the proof "reveals nothing" about the message m.

Remember that we want our proofs to be short and fast to verify because we are positing them on chain, so long or expensive proofs will take a lot of gas to verify.

Lecture 14: zk-SNARKs

Zero-knowledge SNARKs have many applications on the blockchain:

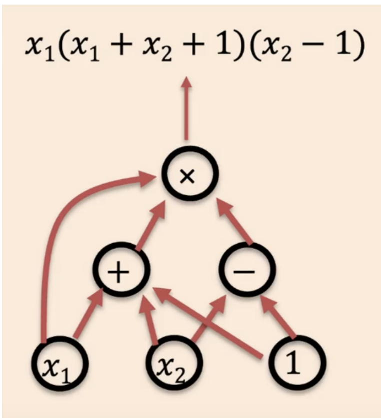
- they allow for private transactions on a public blockchain
- they allow for proofs that a private transaction is in compliance with banking laws
- and more!

We begin by reviewing arithmetic circuits.

Definition 14.1. Let us say that we have a finite field $\mathbb{F} = \{0, 1, \ldots, p-1\}$ (if you don't know what a finite field is, it's not super important to this lecture - you can just think of it as the set of all integers mod p) for some prime $p > 2$.

Then an **arithmetic circuit** is a special type of function $C : \mathbb{F}^n \to \mathbb{F}$ (so it takes n inputs and has one output).

We think of it as a directed acyclic graph whose inputs are $1, x_1, \ldots, x_n$ and whose internal nodes are the operations $+, -, \times$. So an arithmetic circuit defines some polynomial on n variables and gives us a way of computing the polynomial:



We say that $|C|$ is the number of **gates**, or internal nodes, in C.

It turns out anything computable in polynomial time can be captured by an arithmetic circuit.

Example 14.2. The circuit

$$C_{\text{hash}}(h, m) = h - \text{SHA256}(m)$$

outputs 0 if h is the hash of m and outputs something nonzero otherwise. This turns out to have around 20K gates.

Similarly, we have a circuit

$$C_{\text{sig}}(pk, m, \sigma)$$

which outputs 0 if σ is the valid signature for m with public key pk, and it outputs something nonzero if this signature is invalid.

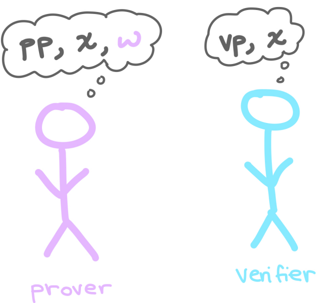
Now, we move on to **non-interactive arguments of knowledge** (NARKs).

In particular, we will understand how a preprocessing NARK works.

Let's say we have a public arithmetic circuit C, which takes in an input pair (x, w) and outputs a single number in $\mathbb{F}$. Here, we say that x is the publicly known **statement**, and it is an element of $\mathbb{F}^n$, and w is a secret **witness**, and it is an element of $\mathbb{F}^m$.

Then, the NARK first does some preproccessing, or **setup**. So it runs some function $S(C)$, to get some information about C, and publicly outputs pp (the prover's parameters) and vp (the verifier's parameters).

Then, we have the following setup:



and the prover wants to convince the verifier that they know some w such that $C(x, w) = 0$.

Definition 14.3. Formally, a **preprocessing NARK** consists of three functions S, P, V, where:

- $S(C) = vp, pp$ is some preprocessing function that outputs information about the circuit.
- $P(pp, x, w) = \pi$ is a function that takes in pp and some (x, w) that makes our circuit output 0, and outputs a proof π that we know such a w.
- $V(vp, x, \pi)$ takes in vp, the same x, and the proof π, and outputs `accept` or `reject` based on whether π provides convincing evidence that the prover knows a valid w.

We say that a good preprocessing NARK needs to be complete and knowledge sound.

Definition 14.4. A **complete** preprocessing NARK is one where, for any (x, w) such that $C(x, w) = 0$,

$$V(vp, x, P(pp, x, w))$$

accepts with probability 1.

Definition 14.5. An adaptively **knowlege sound** preprocessing NARK is, intuitively, one where, if V accepts the proof π for a given x, then P must know a valid witness w for that x.

More practically, this means a (more powerful) extractor E could use this information to extract w from P.

Optionally, a preprocessing NARK can also be **zero-knowledge**, which means informally that (C, pp, vp, x, π) give no additional information about what the witness w could be.

Now, we are ready to define a SNARK, or a *succinct* NARK.

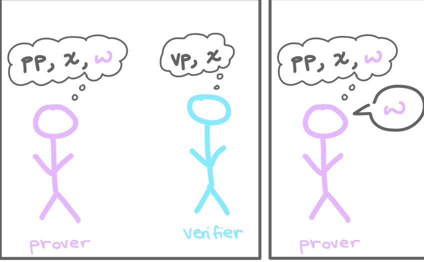
Definition 14.6. Formally, a **succinct preprocessing NARK** consists of the same three functions S, P, V, but with the additional requirements that:

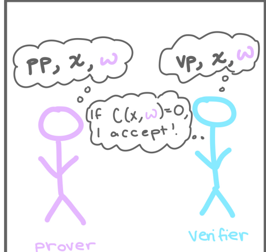
- $\pi = P(pp, x, w)$ is short (so $|\pi| = O_\lambda(\log|C|)$)
- V runs quickly; we say it must run in $O_\lambda(|x|, \log|C|)$ time.

This is the reason we have the preprocessing function S; we see that our verifier must run in $\log|C|$ time, so it does not have time to even fully read the circuit. Thus, in essence, S takes its time to read the circuit and then outputs vp as a short summary of the relevant information about C.

Thus, a SNARK is a NARK that is *complete*, *knowledge-sound*, and *succinct*, while a zk-SNARK is additionally *zero-knowledge*.

The first thing we may try to build a SNARK is the following protocol:





But this is (generally) not a SNARK!

We can see that:

- If w is long, then π is no longer $O(\log|C|)$.
- If $C(x, w)$ is not easy to compute, then V does not run in $O(|x|, \log|C|)$ time.
- If w is secret, then the prover would not want to publicly post w.

We begin by looking at preprocessing setups:

There are three types of preprocessing setups:

1. A **trusted setup per circut** is a function $S(C, r) = vp, pp$ which takes in some random bits r which must be kept secret from the prover.

   Often, once this setup is done, people destroy the machine that has done the setup, so there is no possible way to recover the random bits.

2. A **trusted but universal setup** is a set of two functions S_{init} and S_{index}.

   S_{init} is a one-time function that takes in the (secret) random bits r and outputs some general parameters gp, which are public.

   Then, $S_{\text{index}}(C, gp)$ generates vp and pp based on the given circuit and the general parameters; this is a public, deterministic algorithm.

3. The best possible algorithm is a **transparent setup**, which is a singular function $S(C)$ which does not use any secret data.

Proof systems have improved over the years, and there are tradeoffs between the types of setup functions and the length of the proof:

	size of proof π	verifier time	setup	post-quantum?
Groth'16	≈ 200 Bytes $O_\lambda(1)$	≈ 1.5 ms $O_\lambda(1)$	trusted per circuit	no
Plonk / Marlin	≈ 400 Bytes $O_\lambda(1)$	≈ 3 ms $O_\lambda(1)$	universal trusted setup	no
Bulletproofs	≈ 1.5 KB $O_\lambda(\log\lvert C\rvert)$	≈ 3 sec $O_\lambda(\lvert C\rvert)$	transparent	no
STARK	≈ 100 KB $O_\lambda(\log^2\lvert C\rvert)$	≈ 10 ms $O_\lambda(\log\lvert C\rvert)$	transparent	yes

Ok, before we actually look at examples of these proof systems, let's formally define knowledge soundness and zero-knowledge:

Definition 14.7. Remember that we informally defined knowledge soundness as "if the verifier accepts π, then P must actually know w." What does it mean for P to know w?

Formally, we say that (S, P, V) is **knowledge sound** if for every pair of polynomial time adversaries $\mathcal{A}_0, \mathcal{A}_1$, such that:

> Once S_{init} outputs some gp, $\mathcal{A}_0$ takes in gp and outputs C, x, st (where st is an extra state that it wants to record and give to $\mathcal{A}_1$).
>
> Then, $S_{\text{index}}(gp, C)$ outputs pp, vp, and $\mathcal{A}_1(pp, x, st)$ outputs some proof π, where $V(vp, x, \pi)$ accepts with probability at least $1/10^6$

then there is an efficient **extractor** E such that

> E uses $\mathcal{A}_0$ and $\mathcal{A}_1$ as a black box (so it can make calls to both with arbitrary inputs) and, if gp, C, x were the parameters generated before, then $E(gp, C, x)$ outputs w, where
>
> $$\Pr[C(x, w) = 0] > 1/10^6 - \epsilon,$$
>
> where ϵ is negligible.

Informally, this means that if the adversary can pick a circuit C and a string x, and generate a proof π that convinces the verifier with non-negligible probability, then our extractor can use this information to find a corresponding w with non-negligible probability.

So that's what knowledge soundness is, but what's a zero-knowledge proof?
Let's look at an example, to first get some intuition.

Example 14.8. A zero-knowledge proof of *Where's Waldo* means you have to convince the verifier you know where Waldo is without revealing to the verifier anything about Waldo's location. So you enter a room, the verifier checks that you have brought nothing with you, and you're given scissors and the Where's Waldo page. You cut out Waldo, burn the rest of the page, and hand the verifier the picture of Waldo.

Definition 14.9. We say that a proof is **zero-knowledge** if, intuitively, the verifier learns nothing about w from π; this means it could have generated the proof π by itself.

We say that (S, P, V) is zero-knowledge if there exists an algorithm

$$\text{Sim}(C, x) \implies (pp, vp, \pi)$$

such that the outputs of our simulator "look like" the real (pp, vp, π).

Importantly, the simulator generates π without knowledge of w. The reason the simulator can do this is because it has more freedom than the prover; the prover had to pick a π that fit given parameters (pp, vp) while the simulator can pick a (pp, vp) that make it easy to find such a π.

Formally, we say that (S, P, V) is zero-knowledge if there is an efficient simulator Sim such that for all $x \in \mathbb{F}^n$ such that there exists a w such that $C(x, w) = 0$, the distribution

$$(pp, vp, \pi) \text{ where } pp, vp = S(C) \text{ and } \pi = P(C, x, w)$$

is indistinguishable from the distribution

$$(pp, vp, \pi) \text{ where } pp, vp, \pi = \text{Sim}(C, x).$$

This is all we have time to say about zero-knowledge protocols. This may seem very abstract, and it would probably take a lot more examples to get an intuition for this, but we don't have time for that in this course – take CS 255 if you are interested in learning more about this!

For now, we will switch gears to talking about a particular application of zero-knowledge proofs.

We will spend the rest of lecture talking about **Tornado Cash**, which was a zero-knowledge mixer launched on Ethereum in 2020.

Let's say Alice wants to privately buy an NFT on Ethereum.

She will send 100 DAI to the `Tornado.cash` contract, and many other people will also send 100 DAI to this mixer.

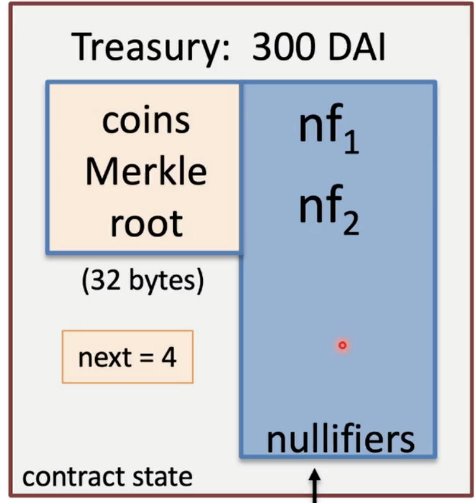
Later, this contract will allow Alice to withdraw her 100 DAI into a fresh address, which has no association to Alice. She can then use this fresh address to buy her NFT privately on the NFT marketplace.

(Recall that we need everyone to send in the same amount of DAI, because otherwise the deposit/withdrawal amounts can be used to link someone's fresh address to their old address.)

Let's dive into how the contract works.

We will say that we are building a 100-DAI pool, which means everyone deposits money in units of 100 DAI. We say that 1 coin is worth 100 DAI.

Then, the contract state has the following information:



where the coins Merkle root is the root of a Merkle tree (stored off-chain) containing a list of all the coins in the pool so far. This list of all the coins is also publicly available off-chain. The `next` variable is the index of the next coin to be added to the pool (so when there are three coins in the pool so far, `next` = 4), and the list of nullifiers we will explain later, but for now keep in mind that there is one nullifier for each coin that has been withdrawn (or spent) from the pool.

So in the current picture, there are three coins in the pool, the contract has 300 DAI, and there are two nullifiers in the pool.

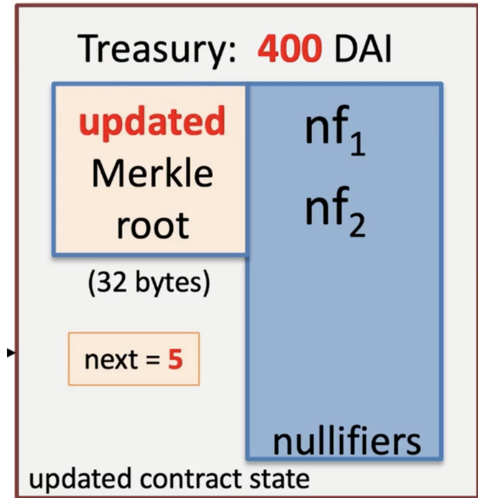
We also have two collision-resistant hash functions H_1, H_2, which map from some set R to 32-byte strings.

If Alice wants to deposit 100 DAI into the pool:

1. She downloads the list of all the coins in the pool, and builds a Merkle proof that the fourth entry of the list is currently 0.
2. She chooses a random $k, r \in R$, and sets C_4 to equal $H_1(k, r)$.
3. She sends 100 DAI to the contract, along with this Merkle proof and C_4.
4. She stores (k, r) secretly, in what is called a *note*.

From this point, the contract will

1. Verify that her Merkle proof is correct.
2. Use C_4 and the current Merkle proof to update the list of all coins and corresponding Merkle root.
3. Update its state:



Note that at this point, an observer knows exactly who deposited which coin.

Now, let's say that Bob wants to withdraw his coin, and have 100 DAI sent to some address A. He has his note (k', r'), which corresponds to coin C_3. He:

- Creates a *nullifier* $nf = H_2(k')$.
- Creates a proof that he owns some coin in the tree, which corresponds to nf, without revealing which coin. To do so:

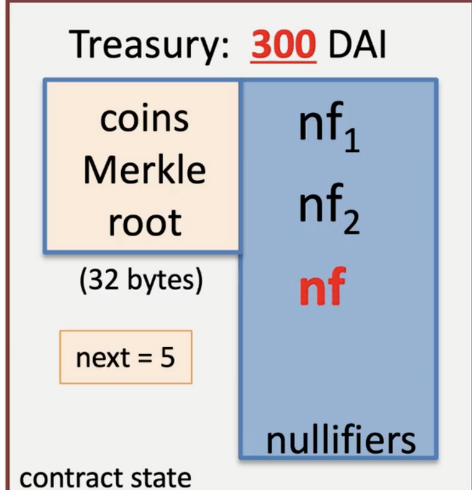
  Bob builds zk-SNARK proof π for
  public statement x = (**root**, **nf**, **A**)
  secret witness w = (k', r', C_3, MerkleProof(C_3))
  where Circuit(x,w)=0 iff:
  (i) C_3 = (leaf #3 of **root**), i.e. MerkleProof(C_3) is valid,
  (ii) $C_3 = H_1(k', r')$, and
  (iii) **nf** = $H_2(k')$.

  Here, the address A is in the statement so the miner cannot just change A to his own address, and steal Bob's coin.

  This assumes the SNARK has a property called **non-malleability**, which means that we cannot use $((\text{root}, nf, A), \pi)$ to build a "similar" proof $((\text{root}, nf, A'), \pi')$ which is still valid, without knowing the witness.

- Bob sends nf, π, A to the contract; he has to do this over Tor so that his IP address is not revealed to the world. Again, *this reveals nothing about Bob's identity or which coin he is spending.*

From here, the contract verifies that π is a valid proof and that nf is not already in the list of nullifiers. Then, it sends 100 DAI to A, and updates the state:



But there is a problem: how does Bob pay for gas on the withdrawal transaction?

If the money for gas comes from Bob's account, then everyone knows Bob made the transaction.

Instead, Bob uses a **relay**: he anonymously sends (nf, π, A) to a relay, which then forwards this information, along with gas for the transaction, to TornadoCash. TornadoCash then sends 100 DAI to A minus the cost for gas, and sends this gas money back to the relay.

(Note that the relay and TornadoCash both take a transaction fee from this process.)

TornadoCash was very heavily used in October 2021; however it can no longer be used.

They also provided a compliance tool, which allows you to prove that you are the original owner of your anonymized coin, in case you want to de-anonymize your transactions for the sake of interacting with banks or merchants.

In March 2022, hackers stole around $600 million from a project called Ronin Bridge. They sent $80 million to TornadoCash to anonymize this money, so it would be impossible to track. It was suspected a month later that this hack came from the Lazarus group, which is a group associated with North Korea - this means that TornadoCash violated US sanctions laws. As a result, in August the US Treasury sanctioned TornadoCash itself, and there has been lots of collateral damage (for people whose funds are now locked in TornadoCash) and two lawsuits since.

In September 2022, the US Treasury clarified that it is not the open-source code itself that is sanctioned, but rather TornadoCash the contract; in particular, it is still legal for this class to be teaching about how TornadoCash works.

A natural question is: how could Tornado have been constructed to comply with US banking regulations?

This is still an open problem, but there are some things they could have done to make the situation slightly better.

1. **deposit filtering**:

   Companies like Chainalysis have services that let you check whether a given address is on a list of sanctioned addresses; TornadoCash could have prevented deposits from any addresses that are known

to be from sanctioned sources.

However, this does give Chainalysis a lot of power - if they decide they don't like someone, they can just add their address to the sanctioned list arbitrarily.

Moreover, this check is usually slow - it takes a couple days for Chainalysis to figure out whether a new address is coming from a sanctioned user, and by that time, the user would have already deposited the money into TornadoCash.

So this does not solve the problem entirely, but it still would have been helpful to do.

2. **withdrawal filtering**:

   At the time of withdrawal, we could require a zero-knowledge proof that the source of the funds is not currently on a sanctioned list; because it takes a couple of days to withdraw the funds, this may give organizations like Chainalysis time to realize that a given address is actually on a sanctioned list.

   To do so, we would modify the coins so that $C_4 = H(k, r), \texttt{msg.sender}$. Then, during withdrawal, part of Bob's zero-knowledge proof would include a proof that `msg.sender` (within C_4) is not currently on the sanctions list.

   This also means that if bad actors eventually get on a sanctions list, their funds become trapped inside Tornado, so they are more encouraged to not take that risk and not use Tornado in the first place.

3. **viewing keys**:

   This is the most privacy-invasive option, while the other two options preserve full anonymity of good actors.

   In this option, we require the nullifier to include an encryption of the deposit's `msg.sender`, using some government public key, so that governments can decrypt the nullifiers and track down the anonymized transactions.

   Unfortunately, there are lots of problems with this design...

Lecture 15: Constructing a SNARK

We start with a very theoretical, PCP-based SNARK. For background, we need to turn to the PCP theorem in complexity theory:

Theorem 15.1. For any arithmetic circuit C, there is a proof system with the property that:

For a very long proof π, the verifier can just read 3λ bits of π (selected at random) and check if they match a valid proof. If the proof is valid, it will always accept, and if the proof is invalid, it will reject with probability $1 - 2^{-\lambda}$.

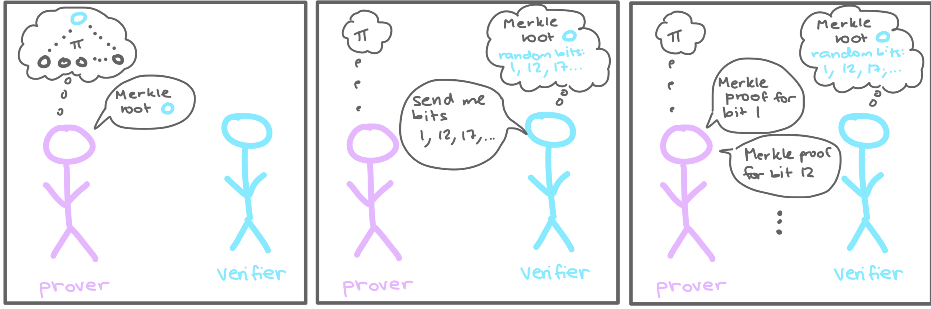
So for such a proof system, the verifier only needs to read 128 bits of the proof, and it will be correct with very high probability.

On its own, this is not a SNARK, because the proof itself is very large.

One idea is that we have the verifier communicate which bits it wants to read, and have the prover only send those bits. But this doesn't work exactly, because if the prover knows exactly which bits the verifier is checking, it could forge those specific bits, and trick the verifier.

We fix this by having the prover first send a Merkle commitment to the full proof!

So we could have a short interactive proof that looks like:



This is not a SNARK, because it is interactive!

Fortunately, this is what's known as a public coin interactive protocol, where the only thing the verifier is doing is generating some random bits and sharing them publicly.

There is a process called the **Fiat-Shamir transform**, which allows us to take public coin protocols, and make them non-interactive.

The important is that the random bits appear to the prover *after* it generates the first message, rather than before. We can force this by having the prover generate its random bits as the hash of the first message and x (the thing it's trying to prove it has a witness for).

So the Fiat-Shamir transform takes an interactive proof that looks like:

and turns it into a non-interactive proof that looks like:

Unfortunately, the PCP SNARK is impractical, because the prover takes very long to generate a proof.

We want the prover to be able to generate a proof in time linear in the size of the circuit - improving prover efficiency will be the focus of the rest of the lecture.

For the rest of the lecture, we will describe things in terms of public-coin interactive proofs, but note that you can always use Fiat-Shamir to turn this into a non-interactive proof.

There are two steps in creating a good general SNARK:

1. a polynomial commitment scheme
2. a polynomial interactive oracle proof (PIOP)

We will explain these two steps now.

What is a polynomial commitment scheme?

Let's say we have a finite field $\mathbb{F}_p = \{1, \ldots, p-1\}$. Again, if you don't know what a finite field is, that's fine - it's not super important for this lecture. But you should maybe look into taking Math 120 at some point ☺

Then, $\mathbb{F}_p^{\leq d}[X]$ is the set of all polynomials in $\mathbb{F}_p[X]$ of degree at most d.

Using this, we can proceed with our definition:

Definition 15.2. A polynomial commitment scheme is a set of functions:

- `setup`(d) generates the parameters pp for working with polynomials of degree at most d

- $\texttt{commit}(pp, f, r)$ outputs com_f, or a commitment to a polynomial f of degree at most d
- $\texttt{eval}$ is a way to prove that, for a pair (x, y), $f(x) = y$. In more detail, $\texttt{eval}$ is actually a SNARK of its own:

Formally: *eval* = (S, P, V) is a SNARK for:

statement $st = (pp, \mathbf{com}_f, x, y)$ with witness $= w = (f, r)$

where $C(st, w) = 0$ iff

$[\, f(x) = y$ *and* $f \in \mathbb{F}_p^{(\leq d)}[X]$ *and* $commit(pp, f, r) = \mathbf{com}_f \,]$

Stanford

We will not be constructing any polynomial commitment schemes today (if you want to see how to construct them, take CS 355!) but here are a few properties of the most commonly used one, called KZG:

- has a trusted setup (so it uses some secret randomness in the setup)
- com_f has a constant size (in fact, it's just a single element of $\mathbb{F}_p$)
- the proof in $\texttt{eval}$ has a constant size (also just a single element of $\mathbb{F}_p$)
- verification in $\texttt{eval}$ takes constant time ($O_\lambda(d)$)

So now we have our polynomial commitment scheme, so we will turn to constructing our polynomial interactive oracle proof.

For any arithmetic circuit $C(x, w)$, where $x \in \mathbb{F}_p^n$, a **polynomial interactive oracle proof** is a proof system that works as follows:

- $S(C)$ outputs public parameters pp and vp, where vp is a set of commitments to some polynomials $f_0, f_{-1}, \ldots, f_{-s}$
- Then, the verifier and the prover have a t-round interaction, where in each round the prover sends a commitment to some polynomial f_i and the verifier sends a random bit r_i.
- The verifier runs some algorithm to check whether it believes the proof. This algorithm takes as input x and the random bits $r_1, \ldots, r_t$, and it can also query the polynomials $f_{-s}, \ldots, f_t$ at various points.

In 2019, a new PIOP called Plonk was constructed - this is widely used in industry today. We will now look at how Plonk works.

Taking Plonk in combination with any polynomial commitment scheme gives us a SNARK, and if we make sure our commitment scheme is randomized, this also gives us a zk-SNARK.

To understand how Plonk works, we will first learn a few tricks which we will use as building blocks for the system.

If you peel away all the layers, everything in the system is built on the following (amazingly simple) key fact:

Remark 15.3. For a given nonzero $f \in \mathbb{F}_p^{(\leq d)}[X]$, for a random $r \in \mathbb{F}_p$,

$$\Pr[f(r) = 0] \leq \frac{d}{p},$$

since there are d roots and p elements.

This is useful because when p is very large (say $p \approx 2^{256}$) but $d \leq 2^{40}$, then d/p is negligible, so the probability that our random number happens to be a root of the polynomial is negligible.

So then we have a simple way of checking if a polynomial is zero:

> Once the prover has committed to a polynomial f, we can just choose a random r. Then, if $f(r) = 0$, the entire polynomial $f = 0$ with high probability, and if $f(r) \neq 0$, f is guaranteed to not be the zero polynomial.

We have a lemma that follows from this key fact:

Lemma 15.4. Assume again that $p \gg d$. Let $f, g \in \mathbb{F}_p^{(\leq d)}[X]$.

For a random $r \in \mathbb{F}_p$, if $f(r) = g(r)$, then $f = g$ with high probability.

This is because, if $f(r) = g(r)$, then this means r is a root of the polynomial $f - g$, so our key fact tells us this means $f - g = 0$ with high probability, so $f = g$ with high probability.

We have a few more useful proof gadgets:

First, let $\omega \in \mathbb{F}_p$ be a kth root of unity; this means it is an element such that $\omega^k = 1 \pmod{p}$.

Then, we will consider the set $H = \{1, \omega, \omega^2, \ldots, \omega^{k-1}\} \subseteq \mathbb{F}_p$. We say this is a **subgroup** of our field because it is closed under multiplication; if I multiply any two elements in H, I get back another element of H.

Then, let f be a polynomial in $\mathbb{F}_p^{(\leq d)}[X]$ and let b, c be elements of $\mathbb{F}_p$, where we set d so that it is $\geq k$.

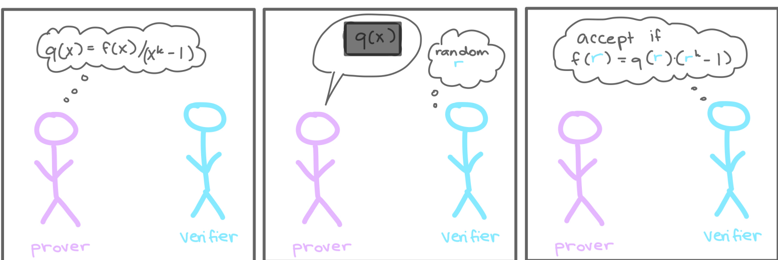
Then, there are efficient PIOPs to do the following:

1. *zero test*: prove that f is identically zero on H (not necessarily on all of $\mathbb{F}_p$)
2. *sum check*: prove that $\sum_{a \in H} f(a) = b$ (where the verifier knows a commitment to f and b)
3. *prod check*: prove that $\prod_{a \in H} f(a) = c$ (where the verifier knows a commitment to f and c)

We will just show how to do the zero test, but I encourage you to pause and try to figure out the other two!

We know that f is identically zero on H if all of the elements of H are roots of f. But H is exactly the set of roots of $X^k - 1$, so f is identically zero on H if (and only if) f is a multiple of $X^k - 1$.

So we can have a PIOP scheme that looks like the following:



where the trick is that the prover can only commit to polynomials, so it can only commit to the true quotient if $f(X)$ is actually a multiple of $X^k - 1$, and then the verifier just needs to check that this is actually a quotient.

Theorem 15.5. This protocol is complete and sound, assuming d/p is negligible. The verifier takes $O(\log k)$ time, plus however long it takes to call `eval verify` twice (to get the values of $q(r)$ and $f(r)$).

Our final useful tool is the *permutation check*:

We say that $W : H \to H$ is a **permutation of** H if for all $1 \le i \le k$, $W(\omega^i) = \omega^j$, for some j.

Then, let $f, g : H \to H$ be polynomials in $\mathbb{F}_p^{(\le d)}[X]$.

Our goal is:

Given commitments to f, g, W, we want to show that for all $y \in H$,

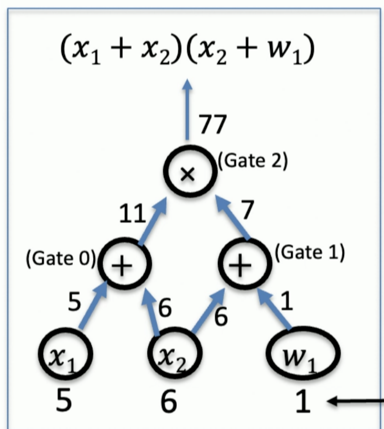
$$f(y) = g(W(y)).$$

So we want to show that f and g give the same outputs on H, but the outputs for f are permuted by W.

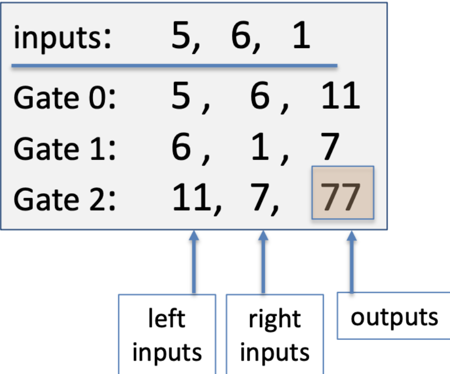
There is a fast PIOP protocol to do this; we don't have time to go over it here, but maybe it will be covered in a section soon!

Now, we have all the tools we need, so we will turn to constructing PLONK.

Let's say we have a random arithmetic circuit $C(x, w)$ and we want to show that we know some w, given x, that makes the output equal to 77. This means we have a set of computations that looks like this:



Our first step is to turn this into a **computation trace**, which is a list of the inputs and outputs at each gate:



What we need to prove is that every computation in the table was done correctly (so that the output equals the sum of the inputs when the gate is a $+$, for example) and that the wiring is done correctly (so when the

output to Gate 0 is 11, the left input to Gate 2 is 11, for example).

Let $|C|$ be the number of gates in the circuit, and let $|I|$ be the number of inputs. Then, we will set

$$d = 3|C| + |I|,$$

so in our example $d = 12$, and

$$H = \left\{1, \omega, \dots, \omega^{d-1}\right\}.$$

We as the prover want to create a polynomial $T \in \mathbb{F}_p^{(\leq d)}[X]$ that corresponds to the correctness of our entire computation trace.

Specifically, we create our polynomial such that:

- $T(\omega^{-j}) =$ input j for all $1 \leq j \leq |I|$
- for all $0 \leq \ell \leq |C| - 1$:
  - $\rightarrow$ $T(\omega^{3\ell}) =$ left input to gate ℓ
  - $\rightarrow$ $T(\omega^{3\ell+1}) =$ right input to gate ℓ
  - $\rightarrow$ $T(\omega^{3\ell+2}) =$ output to gate ℓ

So in our example, the prover comes up with a degree-11 polynomial such that:

inputs:	$T(\omega^{-1}) = 5$,	$T(\omega^{-2}) = 6$,	$T(\omega^{-3}) = 1$,
gate 0:	$T(\omega^{0}) = 5$,	$T(\omega^{1}) = 6$,	$T(\omega^{2}) = 11$,
gate 1:	$T(\omega^{3}) = 6$,	$T(\omega^{4}) = 1$,	$T(\omega^{5}) = 7$,
gate 2:	$T(\omega^{6}) = 11$,	$T(\omega^{7}) = 7$,	$T(\omega^{8}) = 77$

It takes time $d \log_2 d$ for the prover to compute this polynomial based on this information.

So the prover sends a commitment to T to the verifier.

It now needs to prove that T is a correct computation trace; specifically, it needs to show that:

- T encodes the correct inputs
- every gate is evaluated correctly
- the wiring is implemented correctly
- the output of the last gate is 0 (to prove $C(x, w) = 0$)

Proving the last point is easy; we know that $T(\omega^{3|C|-1})$ is the output of the last gate, so we can just open the commitment at this point and check that it is equal to 0.

To prove the first point:

The prover and the verifier both interpolate a polynomial ν that just encodes the x-inputs to the circuit, so that $\nu(\omega^{-j})$ corresponds to the jth x-input. This takes time proportional to the size of the x-input, so the verifier has time to construct this polynomial.

Then, the verifier can just use a zero test to check that $T(h) = \nu(h)$ for the entire subset of H corresponding to the x-inputs.

To prove the second point:

We construct **selector polynomials** of the form $S(\omega^{3\ell}) = 1$ if gate ℓ is an addition gate and $S(\omega^{3\ell}) = 0$ if it is a multiplication gate.

Then, we can see that for the gates to be evaluated correctly, we have that

Observe that, $\forall\, y \in H_{gates} := \{1, \omega^3, \omega^6, \omega^9, \ldots, \omega^{3(|C|-1)}\}$:

$$S(y)\cdot[T(y) + T(\omega y)] + (1 - S(y))\cdot T(y)\cdot T(\omega y) = T(\omega^2 y)$$

left input, right input, left input, right input, output

because this polynomial just says the output of the gate corresponds to the sum of the inputs if it is an addition gate and corresponds to the product of the inputs otherwise.

From here, we can again use a form of a zero check to test that these polynomials are truly equal on all such y.

Computing this selector polynomial takes longer than the amount of time the verifier has. However, since this is not dependent on the input to the commitment, we can compute a commitment to the selector polynomial and give it to the verifier during the setup preprocessing.

To prove the third point, we do a similar process to the second point, but with a wiring polynomial corresponding to the wiring of the circuit.

So in total, our four proofs can be expressed as four zero tests:

gates:	(1)	$S(y)\cdot[T(y) + T(\omega y)] + (1 - S(y))\cdot T(y)\cdot T(\omega y) - T(\omega^2 y) = 0$	$\forall\, y \in H_{gates}$
inputs:	(2)	$T(y) - v(y) = 0$	$\forall\, y \in H_{inp}$
wires:	(3)	$T(y) - T(W(y)) = 0$	$\forall\, y \in H$
output:	(4)	$T(\omega^{3|C|-1}) = 0$ (output of last gate = 0)	

This is amazing because it gives us a very short (constant time) proof of the arithmetic circuit!

There are many extensions of the PLONK scheme for varied applications.

Lecture 16: Scaling the Blockchain, I

We begin by finishing our discussion of SNARKs:

In the real world, we don't write out arithmetic circuits - we write code. There are many platforms out there for writing specifically zero-knowledge code. These are compiled into a SNARK-friendly format, which may be an arithmetic circuit but could also just be EVM code or other assembly languages such as RISC-V. Then, a SNARK machine uses this to generate a witness and then produce a proof.

(The process of generating a proof from a language such as RISC-V is more complex than from a circuit; Dan Boneh might teach a class on practical zero-knowledge proofs soon.)

Now we will switch topics to talking about scaling the blockchain.

Remember that the process of submitting blocks for Bitcoin and Ethereum is very slow; Bitcoin ends up doing about 3 transactions per second, and Ethereum ends up doing about 15. In comparison, Visa does around 2000 transactions per second (and can handle up to 24000) and PayPal handles around 200 transactions per second.

So the natural question is: how do we increase transaction speed on the blockchain?

There are many ideas for this:

- using a faster consensus protocol
- parallelizing: splitting the chain into independent **shards**
- rollup: moving the work somewhere else (we will discuss this next lecture)
- payment channels, which reduce the need to touch the chain

Today we will talk about **payment channels**. This is a more simple approach that increases the transaction speed in very specific cases.

Let's say Alice is transacting with Bob, who runs a coffee shop. Every day, she buys a coffee from Bob and pays him 0.01 BTC on the blockchain. This is very inefficient - for each coffee she is buying from him, she is adding a new transaction to the blockchain. Moreover, she has to pay a transaction fee for each such transaction.

Instead, she could do the equivalent of opening a bar tab. That is, Alice deposits 1 BTC to Bob. Then, every morning when she gets her coffee, Bob records how much of that 1 BTC she has remaining, and at the end of the month, he refunds her the rest of her 1 BTC. Now, even though Alice could be buying hundreds of coffees, we only need two transactions to be posted to the blockchain, so this is much more efficient!

How do we actually implement this, while making sure neither of them can cheat the other?

We will first work in the context of Bitcoin.

Let's say Alice has UTXO A, which has 1 BTC.

Then, the first time she buys a coffee, she sends Bob a signed transaction, giving him 0.01 BTC from UTXO A, and giving herself the rest. But Bob doesn't post that transaction to the chain - he just waits.

The next time she buys a coffee, she updates her tab, so she sends Bob a signed transaction giving him 0.02 BTC from UTXO A and giving herself the rest.

She continues doing this every day, and then at the end of the month, when Bob wants to get paid, he posts the latest transaction to the blockchain. (Note that he can't use multiple transactions, because UTXO A can only be spent once. So he's better off being honest and using the latest transaction, which gives him the most money.)

But Alice can cheat Bob in this scenario! If she spends her UTXO before Bob posts his transaction, he is out of money. How do we prevent this?

Instead of having UXTO A on the blockchain, she starts with a UTXO AB, which requires signatures from both Alice and Bob to spend.

But then if Bob never publishes a transaction, then he doesn't get paid, but Alice also loses all her refunded money! So before Alice posts her transaction AB, Bob gives her a presigned "refund transaction" which is timelocked for, say, 7 days from now. Thus, if Bob does not post any of the transactions Alice sends him, she can eventually get her deposit refunded.

This refund transaction determines the lifetime of the channel, and the money in the initial deposit determines the maximum amount that can be spent.

This is very trivial to do in Ethereum; we can just write a contract that does all of this for us, without having to handle any UTXOs.

But this is just a unidirectional channel: Alice can only send money to Bob, and he cannot send any money back. What if we wanted to open a bidirectional payment channel?

Your first intuition might be to just make two unidirectional channels. But the problem with that is the money sent from Bob to Alice and the money sent from Alice to Bob will not cancel out; if both of them send each other 1 BTC then both channels will be fully spent, rather than the balance resetting back to 0.

This is very easy to do on Ethereum:

Alice and Bob create a contract, and they both send 0.5 ETH to the contract. The contract will also maintain a state, called a nonce, which starts at 0.

Then, off-chain, Bob will send money to Alice by updating the state, so he sends her a signed transaction saying that at nonce 1, Alice has 0.6 ETH, and Bob has 0.4 ETH. Alice then signs this transaction, but they keep it off-chain.

Alice and Bob can continue sending off-chain transactions to each other, but the on-chain state of the contract will not change.

Then, at any point in time, Alice can close the contract by sending the latest signed transaction to the contract, and calling the close function. The contract will verify that this is a valid, signed state, and then enter a "challenge period" of, say, 3 days.

The reason the contract ends the challenge period is that it has no way of verifying that Alice actually sent the latest state, and not an earlier one where she had more of the money. After the three days, if Bob does nothing, then the contract decides that the transaction was valid, returns the money to Alice and Bob, and closes the channel. However, if Alice had submitted an earlier state, then Bob can challenge that by posting a later state to the chain. The contract will check that this is a valid, signed state, and that its nonce recorded a later time than Alice's state, and then it will refund both of them the money as recorded on Bob's transaction.

The main problem with this is: Bob has to be constantly watching the blockchain. If he stops watching the chain for more than 3 days, then Alice can use that as a way to post an earlier transaction and run away with the money. To get around this, Bob will hire a trusted watchtower service, which is an off-chain service (hosted on something like AWS) which just constantly monitors the chain for someone trying to close Bob's payment channels, and responds with the latest state.

Again, the important feature of a payment channel is that Alice and Bob send each other one on-chain transaction to open the channel, then send as many off-chain transactions as they want, and then one on-chain transaction to close the channel. So they can have as many transactions between each other as they want, and record it with only two on-chain transactions.

We can generalize this concept to something we call **state channels**:

For any two-player game, we can do the same thing, by starting the game on-chain, and then sending signed transactions back and forth updating the state, until one player posts the final state of the game to the blockchain, and the contract checks its validity and updates its internal state. If one of the players withdraws, or refuses to sign a move, then we can wait 3 days and then use the refund transaction to end the game.

However, this cannot be done for larger-scale lending protocols, because we don't know in advance who the players are.

Building bidirectional channels is much more difficult on Bitcoin (this is one of the most active areas of development on Bitcoin currently):

We will create a UTXO that can be spent in one of two ways:

- **relative time-lock**: the person who created the UTXO can spend it t blocks after the UTXO was posted to the chain
- **hash lock**: the UTXO contains a hash image X, and someone can redeem it using some **hash preimage** x, where $X = \text{SHA256}(x)$

Using this, we can create a bidirectional payment channel.

First, Alice will generate a random x and send $X = H(x)$ to Bob, and Bob will generate a random y and send $Y = H(y)$ to Alice.

Then, they create the following transactions:

TX1: input: UTXO AB	TX2: input UTXO AB
Out1: pay 7 → A	pay 3 -> B
Out2: either 3 → B, 7 day timelock	either 7 → A, 7 day timelock
or 3 → A now, given y s.t. H(y)=Y	or 7 → B now, given x s.t. H(x)=X
Alice sig	Bob sig

so that either of them has the option of getting (most of) their money back after 7 days. They sign the transactions and send them to each other, so both of them have the options of posting these transactions and refunding their money after 7 days.

Once they both have these refund UTXOs, they post a 2-of-2 multisig UTXO called AB to the chain. Alice contributes 7 BTC to this UTXO, and Bob contributes 3 BTC; this is proportional to the refund transactions they made.

Now, let's say Alice wants to send 1 BTC to Bob. She will pick a new random number x' and send $X' = H(x')$ to Bob. Then, they both create similar refund transactions, with the new amounts and the new hash values:

TX3 input: UTXO AB
Out1: pay 6 → A
Out2: either 4 → B, 7 day timelock
or 4 → A now, given y s.t. H(y)=Y
Alice sig

TX4 input: UTXO AB
pay 4 → B
either 6 → A, 7 day timelock
or 6 → B now, given x' s.t. H(x')=X'
Bob sig

and, moreover, Alice sends her original value x to Bob.

Now, Bob agrees that he was paid 1 BTC, so he gives the coffee to Alice.

What are the ways they could redeem the UTXOs now?

Well, either Alice could post TX3 or Bob could post TX4, and they would get back the money they currently have (Alice would get back 6 BTC and Bob would get back 4 BTC), after 7 days.

But Alice has motive to post the old transaction TX2, when she had more money than she currently does. However, if she posts TX2, then there is a 7 day timelock before TX2 is actually processed. In that time, since Bob now has Alice's signature on TX2, he could use that signature to post the same transaction, but with x, which he now knows. This gives all of Alice's money to Bob, as a penalty for trying to post an old transaction.

Again, this has the same problem of Bob needing to constantly watch for a stale transaction, and we could account for this by hiring a watchtower service.

The next natural question is: let's say that you go to a different coffee shop for each day of the week. You don't want to open seven different payment channels with seven different coffee shops, and you know that you already have a payment channel with your bank, and your bank has payment channels with each of the coffee shops.

We can use this to create **multi-hop payments** with you and the coffee shop, across the two channels.

Let's say Alice wants to pay Carol, though an *untrusted* intermediary Bob.

To do so, Carol generates a random number r, and sends $R = H(r)$ to Bob, who sends it to Alice. Then, Alice creates a transaction to send 1.01 BTC to Bob, which is hash-locked with R and time-locked for Alice to refund. Bob then makes a transaction which sends 1 BTC to Carol, again hash-locked with R and time-locked for him to refund.

Once Carol gets the transaction, she can redeem it using r. At this point, r is revealed, and Bob can use r to redeem his UTXO from Alice, and keep the extra 0.01 BTC as a transaction fee.

If Bob does not forward the transaction, then he will never learn r, so then Alice has time to refund her transaction. If Carol never redeems the transaction, then Bob can redeem his transaction, and then Alice can redeem hers.

We can extend this logic into larger multi-hop transactions, which are known as lightning networks - these are widely used on Bitcoin today.

Lecture 17: Scaling the Blockchain, II

As a reminder, Ethereum can process about 7 transactions per second, while Visa can handle 24000 Tx/s. We want to scale up the blockchain, so that it can handle more transactions per second, in order to keep up with demand and hopefully lower transaction fees.

Today's idea for scaling the blockchain is **rollups**.

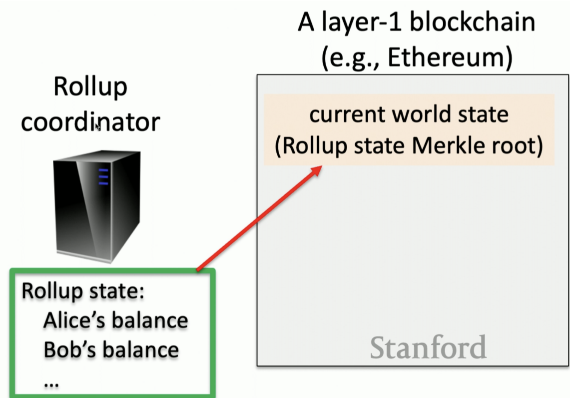
Recall that a basic (what we will call layer-1 or L1) blockchain works as follows:

- At any point in time, it maintains the current world state, which has information like the balances for every account and the current state for every contract.
- Whenever a user posts a transaction, the world state is updated to reflect this change, and this new world state is appended to the blockchain.

The idea behind a rollup is: what if we have another party do most of this work?

As usual for this course, we will start with an idea based on a centralized system, and then turn it into a decentralized process.

Let's say we have a **rollup coordinator** outside the blockchain. They internally manage a rollup state which has balances for every user, and maintain a contract on the blockchain which contains just the Merkle root of the Rollup state.



Then, each end-user can send their transactions to the rollup coordinator instead of to the blockchain. The rollup coordinator can process hundreds of transactions, update its internal state, and then post one update to the blockchain with the list of transactions it processed and its new Merkle root.

We have increased the transaction rate significantly, because now these 100s of transactions require only one update to the L1 blockchain, while before each individual transaction required an update to the L1 blockchain.

Note that Bob's account on the rollup is separate from his account on the blockchain. We have a system that looks something like this:

Rollup contract on layer-1 holds assets of all Rollup accounts (and Merkle state root)

Rollup state (L2)

Alice: 4 ETH, 1 DAI | Bob: 3 ETH, 2 DAI | ...

(coordinator stores state)

Alice: state | Bob: state | Uniswap: state | Rollup contract: 7 ETH, 3 DAI, root | ...

Layer-1 blockchain (L1)

Then, we can see that transferring funds within the rollup (e.g. from Alice's rollup account to Bob's rollup account) is cheap, because we can batch hundreds of those transactions into a singular update to the L1 blockchain.

However, transferring funds to the rollup from outside is still expensive, because we need to write a transaction on the L1 blockchain to transfer money from our L1 account to the rollup.

Withdrawing our funds from the rollup can be batched with the other rollup transactions, but it requires extra gas, because now there is an extra output on the transaction that the rollup coordinator sends to the L1.

Let's say Alice wants to send her rollup funds to Uniswap. How does she do this?

Well, she could just withdraw her rollup funds, and then send them to Uniswap, but this takes transactions on the blockchain, which is expensive. Instead, what we do is run another instance of Uniswap on the rollup coordinator itself, and then we can transfer our funds directly on the L2.

So the rollup runs exactly like Ethereum does (we can have any balances and contracts within the state, and then we update the state by sending transactions) except that there is no consensus protocol! The reason we don't need the consensus protocol is that we always have a commitment to the current state of the L2, stored within the rollup contract on the L1. So we are expecting the L1 to do the consensus work for the L2 state.

Ok, so that's how the rollup coordinator works. But this is not how rollups work in reality - as stated, the rollup coordinator has complete control over everyone's transactions and balances, and could also just shut down and have everyone's data be unrecoverable. How do we solve these problems, so that we don't require trust in the rollup coordinator?

Problem 1: Can the rollup coordinator lie about transactions?

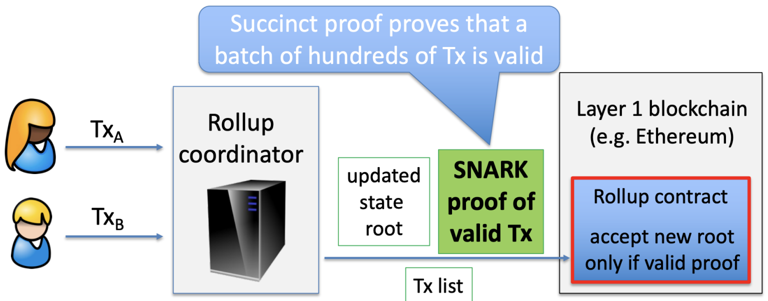
No - recall that Merkle root of the current state of the L2 is stored in the L1. When the rollup coordinator submits an update to the L1, it sends over the new current state as well as a transaction list, and the L1 can then verify that each transaction is signed and properly funded, and check that these transactions correspond to a valid update to the world state.

However, if we do this directly, we lose all of the benefit of the rollup, because now the L1 is doing the work of verifying each transaction individually.

To fix this, we have a few options:

- **validity proofs** (a rollup that uses validity proofs is called a **zk-rollup**, even though we are not using zero knowledge anywhere)

  In this case, the rollup coordinator sends over a SNARK proof that their updated world state is valid:



Specifically, this is a proof that all the user signatures on the transactions are valid, all the Merkle proofs (for the balances that have been changed) are valid, and the new state is the result of applying the transaction list to the old state.

This is better than having the L1 manually verify these things, because by definition SNARK proofs are short and fast to verify, which is important when we have to run them on the slow and expensive L1.

- **fraud proofs** (a rollup that uses fraud proofs is called a **optimistic rollup**)

  In this case, the coordinator submits an update to the L1 without proof, along with some amount of stake to the contract. Then, for 7 days, anyone can submit a fraud proof against the update. If a correct fraud proof is submitted, the update is cancelled and the coordinator's stake is slashed. If an incorrect fraud proof is submitted, the complainer has to pay a penalty.

  Again, the challenge is how to quickly prove fraud to the rollup contract in the L1, without making the contract run through all the transactions.

  The solution to this is just to binary search: the verifier asks both parties for their version of the L2 state Merkle root after various points in the middle of the transaction list, until it finds the first place where the two parties disagree. Then, it just has to check whether that specific transaction is invalid - if it is, then the update is fraudulent and the coodinator gets slashed, and if it is not, then this was a false proof and the complainer has to pay a fee.

  However, there are some difficulties with this idea:

  → The transactions only settle after 7 days, so Alice can only withdraw her transactions from the rollup after 7 days. (This can be solved for fungible tokens by having a 3rd party forward Alice the funds in advance, and then take the payment from the rollup after 7 days.)

  → If a succesful fraud proof is submitted 4 days after the update, all subsequent transactions need to be cancelled and resubmitted.

This means that the rollup coordinator cannot cheat, so all state updates are valid. Thus, if we have a verifier like this coded into our rollup contract, then anyone can act as the rollup coordinator, and the rollup contract will accept their update as long as it comes with a valid proof.

Problem 2: What if the coordinator disappears?

The solution would be to just set up a new coordinator, but in order to do that, we need the current state of the rollup (not just the Merkle root).

Well, we can always reconstruct the state of the rollup by looking at the L1 chain: this can be done just by starting from scratch and simulating every transaction in the transaction list, from every update the rollup

coordinator sent previously.

Thus, anyone can become a coordinator just by running though the previous transactions for the rollup.

So what part of the transaction list do we need to send to the L1?

- For a zk-rollup, we just need the transaction list, because the SNARK proof provides a justification that every transaction was signed and valid.
- For an optimistic rollup, we also need to include the transaction signatures, because these are necessary for checking the validity of the transactions sent.

Lecture 18: Recursive SNARKs

We will talk about recursive SNARKs soon, but we start by finishing our discussion of rollups from last lecture.

A question that was asked:
Let's say that Alice has money in two different L2s, Arbitrum One and Optimism. She posts a transaction to send Bob 2 ETH on Arbitrum One. What's stopping Bob from posting the same transaction to Optimism, to get extra money from Alice?

The answer is that every account on Ethereum has a nonce which gets incremented after every transaction, to prevent replay attacks. Alice has a different nonce on Optimism than she does on Arbitrum One, so the transaction would fail on Optimism because of the invalid nonce.

Last lecture, we discussed posting the transaction list, so that if the coordinator disappears, someone else can recreate the current rollup state by going through the past list. However, this is very costly, because storing the entire transaction list as calldata takes a lot of gas. Is there a cheaper way to handle this?

One proposal (EIP-4444) wants to modify Ethereum so that we can remove the signatures from the calldata after 7 days in order to store less data. However, at the moment, nothing can be erased from calldata in Ethereum.

Another way to make this cheaper is to not store the transaction list on Ethereum at all. Instead, we form what is called the DAC, or Data Availability Committee.

Definition 18.1. The **Data Availability Committee** is a small group of nodes which are trusted to store the transaction list and keep the data available. Then, the L1 accepts an update only if all the DAC members sign the update, to show that they have verified and approve of the new state.

Note that the availability of this data is now dependent on the honesty of the DAC members. However, the actual validity isn't - whenever we are actually using a transaction list from the DAC members, we can just process the list ourselves and check whether the final state matches the state in the L1.

Validium is an example of an L2 using a DAC and validity proofs, and it has the privacy benefit that only DAC members see the transaction data.

	Tx Volume/day	average fee/tx	(on Nov. 15, 2022)
• Ethereum:	1013K Tx	**2.71 USD/Tx**	
• Arbitrum:	345K Tx	0.08 USD/Tx	(optimistic Rollup)
• Optimism:	303K Tx	0.13 USD/Tx	(optimistic Rollup)
• StarkNet:	14K Tx	0.22 USD/Tx	(zkRollup)

Problem 3: Can a coordinator decide to censor a transaction?

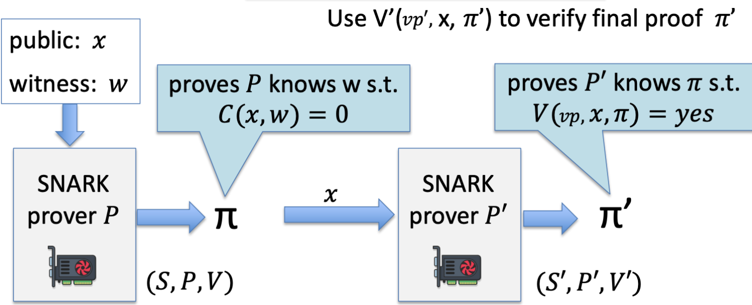
Note that there are usually many coordinators, and if all the coordinators refuse to accept a transaction, users can also become coordinators themselves and submit their own transactions.

However, becoming a coordinator is unwieldy and takes a lot of compute power. One alternative is: End-users could also have the option of sending their transaction to the rollup contract on the L1. The rollup contract will then refuse to accept any state updates until it has received one that has this transaction included. So it combats censorship by freezing the rollup.

We now switch topics to talking about recursive SNARKs.

What is SNARK recursion?

Well, rather than proving that a statement is true, we prove the statement "I know a proof that this is true."



This is a very cool thing, because in essence you are proving that someone else knows a w for the statement, without ever knowing w yourself!

It's also very useful for a variety of applications:

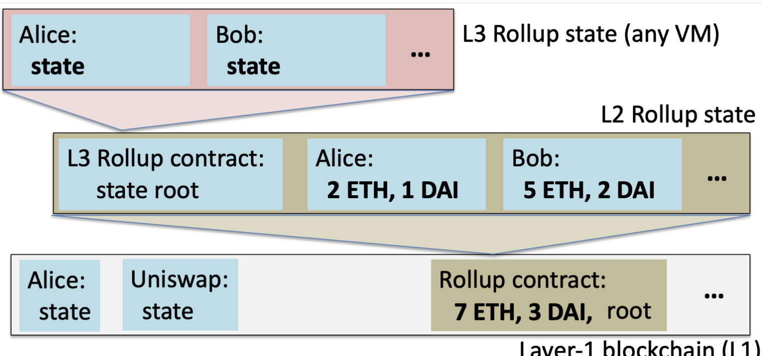
One such application is proof compression.

We can use a very fast prover in the first layer to generate our proof π, and then in the second layer use a slower prover to generate a much smaller proof π'.

This is helpful because we can use the fast prover to get a proof π on a very complicated circuit C, and then our slow prover is just generating a proof for our first verifier V, which is usually significantly less complicated.

Another application is Layer 3 Blockchain.

Within our L2 rollup state, we can store a contract for another rollup, so that we have multiple levels of rollups.



A singular L2 coordinator can support many L3s. Each one can be its own custom blockchain, with its own rules and architecture. The L3 chains can also communicate with each other through the L2 (in the same way that L2s can talk to each other through the L1, but this is much more expensive because we are touching the L1).

Why does this use recursive SNARKs?

Well each L3 coordinator sends a transaction list and the corresponding SNARK proof to its contract on the L2. Then, the L2 collects a batch of these proofs, and builds a proof π that it has a batch of valid proofs, which is what the L1 actually verifies.

Thus, this increases the scaling factor exponentially.

A third example is L2 with private transactions.

This one is a bit more complicated than the previous examples.

The idea is that the way we described rollups so far, the coordinator knows how much money is in everyone's balances. Could we provide a private rollup, where this isn't the case?

Yes. The way we do this is: for each user, the L2 rollup coordinator just stores a hiding and binding commitment to that user's state (such as a hash of her state with some random bits).

Alice is the only person that knows her state and the random bits. How would she send Bob 2 ETH privately?

- She creates and signs a transaction saying this is what she wants to do, and privately sends it to Bob.
- (In her own mind) Alice will compute her updated state. She will also generate new random bits, and generate some h' as the hash of her new state and the new random bits.
- She will build a proof π that h' is a valid update to her state, and send (h', π) to the L2 coordinator.

What we have done so far is have Alice update her state to subtract 2 ETH from her balance. Now, Bob needs to update his state to add this 2 ETH to his balance. He:

- receives the transaction from Alice, and computes his updated state.
- generates new random bits, and generates h'' as the hash of his new state and the new random bits.
- builds a proof π_B that h'' is a valid update to his state (this uses the transaction as a witness).
- sends (h'', π_B) to the L2 coordinator.

Then, using the proofs it got from the users, the Rollup Coordinator updates all the state commitments it has, and generates a proof π'' that it recieved a bunch of valid proofs and updated the states according to those valid proofs, and sends π'' and the new Merkle root to the L1.

(It's not too difficult for the proof of proofs to keep track of whether the transactions involved were consistent, but we are skipping that detail for the sake of this lecture.)

There is a danger here: if Alice loses her random bits, she loses all her funds on the L2, because she can no longer provide a valid proof of her state.

Lecture 19: MEV and Bridging

Before we talk about MEV and bridging, we will give some final thoughts on zero knowledge proofs.

A great story about the applications of zero knowledge proofs, even outside of blockchain:

One large problem today is misinformation. For example, there is lots of news going around about the Russia-Ukraine conflict that includes photos which are actually from a different location or taken at a different time. How do we verify that photos being shared are actually from the event in the news article?

Camera manufacturers very recently came out with the C2PA (content provenance and authenticity) standard for verifying photos. Here's how it works:

1. Sony embeds a secret key sk within each camera, such that it cannot be extracted from the camera.
2. Normally, when a photo is taken, there is data such as the timestamp and location attached to the JPEG. A C2PA camera signs the image, along with the timestamp and location. Now this cannot be externally modified without the signature being invalid.
3. Now, if a reporter uses a C2PA camera and uploads their picture to the New York Times, you as a reader can double-click on the photo, and your browser will check the signature and tell you that it is verified to be taken at that time and location.

Unfortunately, the state of the world is not this simple ☹
Newspapers often modify pictures before publishing them. There is a list of 6 things the Associated Press will allow them to do to photos (so that they cannot be photoshopped); these include putting them in grayscale, cropping, and resizing the photo.

But now that the photo is modified, the signature no longer matches, so your laptop can no longer verify the photo. The C2PA solution to this is to have the photo editor add a signature that verifies that only these six operations were done to the original image. But this is very problematic, because signing keys on the software are very easy to steal.

Instead, we will use zero-knowledge proofs.

Instead of a signature, the processed (as in cropped and resized) photo will come with a zero-knowledge proof π that:

I know a triple (***Orig, Ops, Sig***) such that

1. ***Sig*** is a valid C2PA signature on ***Orig***
2. photo is the result of applying ***Ops*** to ***Orig***
3. metadata(photo) = metadata(***Orig***)

Then, again, the laptop will verify π and show metadata to the user.

Note: this is not deployed yet, but hopefully it will be at this time next year!

An open research question is: can we do a similar thing for videos? Right now, we don't know how to process and generate efficient proofs for that volume of data.

We now return to the world of blockchain. Today, we will cover maximal extractable value (MEV) and bridging, but both of those really deserve much more than just half a lecture. There are also very important topics such as project governance (who gets to decide changes to Compound or Ethereum and how), insurance against bugs or hacks, and interesting/relevant cryptography techniques, which we will note get to cover in this class.

We begin by talking about maximal extractable value:

This is a bit of a discouraging topic, but it is technically very interesting.

Ethereum gives rise to a new type of job, called **searchers**. The entire job of a searcher is to look for opportunities for arbitrage: for example, if the USD/DAI exchange rate is 1.001 on Uniswap and 1.002 on Sushiswap, then a searcher can post a transaction to equalize the markets and get free money (see Example 11.9) for a reminder of how this works. Searchers also profit from liquidation opportunities (remember from Lecture 11 that if a borrower's health gets too low, anyone can liquidate their collateral and profit).

However, there is a big problem in this scenario. Specifically, let's say searcher Alice finds this arbitrage opportunity. They create a transaction trading against these two exhanges, and then post it to the mempool so that a validator can take it and add it to a block. But then:

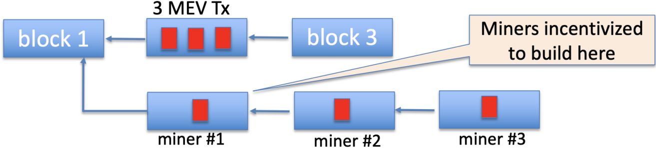
- a validator can see this opportunity, make their own transaction taking advantage of the discrepancy, and stick it before Alice's transaction
- even if all validators are honest, user Bob can see this in the mempool and create a similar transaction, but with Bob as the beneficiary and with slightly higher `maxPriorityFee`, so that the validators post Bob's transaction before Alice's

There are now bots that just scan the mempool, look for transactions that would profit them if they just changed the recipient to themselves, and post those copied transactions with higher `maxPriorityFee`s.

This leads to a problem known as a price gas auction:

Definition 19.1. A **price gas auction** (or PGA) is when two or more searchers compete to post a transaction first. This means both of them repeatedly post the same transaction, but with slightly higher priority fees, until a validator chooses one. This occurs over the span of a few seconds, so it floods the mempool and causes high gas fees.

This also allows for attacks on consensus:
If there is a block with 3 MEV (or arbitrage) transactions, then a miner could cause re-organization by taking one of the MEV transactions for themselves, and then incentivizing the other miners to build upon this new branch and take the remaining MEV transactions:



(This is because the MEV transactions cause extra revenue for miners, much more than normal block rewards.)

With the new Ethereum, there is a solution for this, so block re-orgs used to happen around every half an hour, but they now only happen once a day or so.

Finally, this can also cause centralization:

A searcher could defend against this attack by privately sending their transaction to a trusted validator, instead of posting it to the mempool. This is known as a **private mempool**, which is not how Ethereum is supposed to operate, and in the long run, this will lead to a few trusted validators handling almost all of the transactions.

Last week, validators overall got 4.5k ETH from MEV rewards, so this is a significant amount of money.

So how do we fix this?

We use **proposer-builder separation** (PBS). This means that there are a handful of trusted people that build the blocks, and then there can still be a large group of people that can propose and validate blocks on the network.

This allows for a specific market that is designed for searchers to compete for their place on a block, without interfering with the gas prices or demands of regular users.

The current implementation of PBS is called **MEV-boost**. We will now explain how this works:

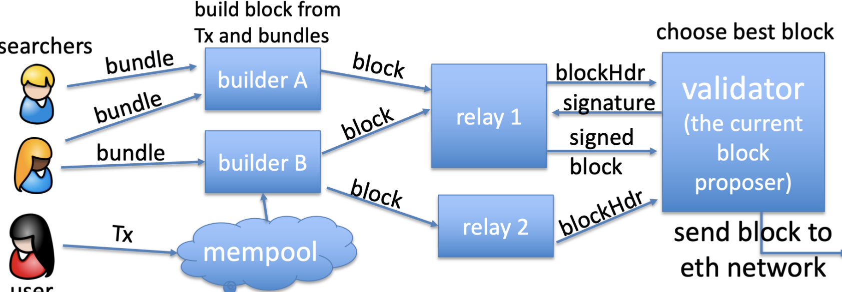
Regular users have transactions they want posted, and searchers have **bundles** (or sequences of transactions) that they want posted in order.

The users post their transactions to the mempool as before, but the searchers have the option of privately sending their transactions to the builders, and making the business transactions they want with the builders. (There are about seven companies that offer building services right now, and their services are very public, so it is beneficial to them to be honest to convince servers to keep using their service.)

Each builder then creates the block that they think will profit the validators the most, and send these blocks to the relays.

The relays again choose the blocks with the most profit to the validators, and try to send them to the validators. However, they cant send the block as-is, becuase then the validators can just steal the transactions. Instead, they send the block header to the validators. The current block proposer (remember this is one of the validators) will choose the block that pays them the most, and sign the block and send it back to the relay.

The relay then sends the block contents to the validator, and then the validator posts it to the network. The validator cannot steal the transactions at this point, because then the relay can post the previously signed block, and then there will be two blocks posted under this validator's name, and he will get slashed.



If the relay doesn't return the block information, the assumption is that the relay crashed and will never send the block information, so the validator will just put together transactions from the mempool

and form its own signed block.

Note that in this system, the relays and the builders are trusted.

However, there are still many problems with this system. For example, in the past 30 days there were only 5 builders that built 80% of all blocks, so there is clear centralization in the builder market, which is bad because it allows for censorship by block builders.

We will now briefly talk about bridges:

Remember that there are many different L1 blockchains, such as Bitcoin, Ethereum, Celo, Avalanche, BSC, Solano, and so on. This creates a **siloing problem**: if someone wants to interact with a marketplace on one chain, but all of their assets are on a different chain, how do they interact?

This is analogous to the pre-internet world - we want to be able to interact with anyone, no matter who your network provider is.

As a first example, let's think about how to trade Bitcoin on Ethereum.

We want to create an ERC20 token on Ethereum that's tied to actual BTC on the blockchain.

The solution for this is **wrapped coins**, so that Asset X on one chain will appear as `wrapped-X` on another chain. For example, on Ethereum we have wBTC, tBTC, and many others.

The wrapped bitcoin wBTC has a **lock-and-mint bridge**.

How does this work?

Well, a custodian has an address on Bitcoin and an ERC20 contract on Ethereum.

If Alice wants to send her BTC to Ethereum, she sends one BTC to the custodian's address on Bitcoin (**locks** her bitcoin), and then the custodian sees that and uses its contract on Ethereum to mint 1 wBTC and send it to Alice.

When Alice wants to move her wBTC back to Bitcoin, she asks the contract to burn her 1 wBTC, and the custodian sees this and sends her back 1 BTC on Bitcoin.

We can do this without a trusted custodian, but we don't have time to finish explaining how to do that in this lecture.

Lecture 20: The Future of Blockchain

This is a guest lecture by Ali Yaha of a16z, about the future of blockchain. It's in a conversation/question-and-answer style, so I'm not going to take notes for it, but it's super interesting, so I recommend you watch the recording!