Blockchains: An Introduction to Blockchain Technology

Lecture 1, Foundations:
Motivation, the Blockchain Stack, and Cryptographic Primitives

Lecture Notes

Abstract

These notes accompany the first lecture. They cover what a blockchain *is*, why blockchains exist and what problem they solve, the layered architecture of a blockchain system, and the two cryptographic primitives that everything in the rest of the course is built from: *cryptographic hash functions* and *digital signatures*. No prior cryptography background is assumed. If you are interested in cryptography, please take the Cryptography course offered by Prof. Bart Mennink in Period 2. Later lectures revisit each primitive in the context of a concrete system (Bitcoin in Lecture 2, consensus in Lecture 3, Ethereum in Lecture 4, and so on); the goal here is to fix vocabulary and intuition.

Contents

---

1 What a blockchain is

Before asking what blockchains are *for*, it is worth pinning down what one actually is. A *blockchain* is a shared record of transactions that is:

Replicated.
Many computers (*nodes*) each keep a full copy of the record, instead of one authoritative copy held by a central operator.

Append-only.
Transactions are added in batches called *blocks*. Existing entries are never edited or deleted. The record only ever grows.

Hash-linked.
Each block contains a cryptographic hash of the previous block, so the blocks form a *chain*. Changing any past block changes its hash and breaks every link after it, and every node would notice. (Hashes are defined in Section 4; for now, treat the hash of a block as a short fingerprint that no one can forge.)

Protocol-governed.
A set of rules (the *consensus* protocol) decides which block may be appended next and keeps the replicas in agreement, with no party in charge.

So the word "blockchain" names three things at once: a *data structure* (a hash-linked list of blocks, each block a set of transactions), a *network* (the nodes that store and check it), and a *protocol* (the rules by which the network extends the structure). The following sections make each of these precise; the rest of *this* section is about why anyone would want such a thing.

2 Why blockchains exist: what is a blockchain for?

2.1 Coordination without a trusted party

The abstract purpose of a blockchain is

> *to let many parties coordinate on a shared state when there is no single party they all trust.*

The phrase "shared state" should be read broadly: an account ledger, the set of who owns which asset, the current configuration of a program, the order in which a set of events happened. "Coordination" means the parties end up *agreeing* on that state, and can each independently check that the agreed state was reached by the rules.

The important qualifier is the last one. If there *is* a party that everyone trusts (a bank, a broker, a government registry, a company's database), then that party can simply keep the authoritative copy of the state, and there is no need for a blockchain. A blockchain is the answer to the harder question: *what if no such party exists, or the parties are unwilling to appoint one?* Financial systems are the running example throughout this course, because there the participants are often mutually distrusting and there frequently is no neutral party acceptable to all of them.

2.2 What was actually new

Blockchains combine older ideas (hash functions, digital signatures, replicated state machines, peer-to-peer networking), but a few genuinely new combinations appeared over the last fifteen years.

2009, Bitcoin.
A *practical, public, append-only data structure* that is kept honest not by an administrator but by *replication* (many parties store a copy and check each other) and *incentives* (the parties doing the work are paid in the system's own asset). On top of this sits a *fixed-supply digital asset* (BTC) that can be transferred between parties without an intermediary.

2015, Ethereum.
The append-only structure is generalised into a *blockchain computer*: a fully programmable environment in which anyone can publish a program that manages digital and financial assets, and the program then runs exactly as written. A key consequence is *composability*: programs already deployed on the chain can call each other, so new applications can be assembled from existing ones.

2017–present, economic applications.
Rapid growth of financial applications built on the Ethereum model (collectively "DeFi"), and, increasingly, *permissioned* blockchains that use the same execution model but restrict who may operate the network.

2.3 What it is used for

A digital currency (stored value). The most basic application is an asset that lives on the chain and can be sent from one party to another over the internet, accessible to anyone with a connection and without asking permission from an intermediary.

Decentralized applications (DAPPs). Once the chain is programmable, the asset is only the beginning. Typical categories:

- **DeFi**, financial instruments implemented as public, verifiable programs: stablecoins, lending markets, exchanges, and so on.
- **DAOs**, decentralized governance: a program holds funds and executes decisions according to on-chain votes (used for investment, donations, shared ownership).
- **Machine-to-machine payments**, automated agents paying small amounts for services, where opening a bank account per agent is impractical.

Underlying all of these is a *new programming model*: writing programs whose execution and state are public, replicated, and not under the author's control after deployment.

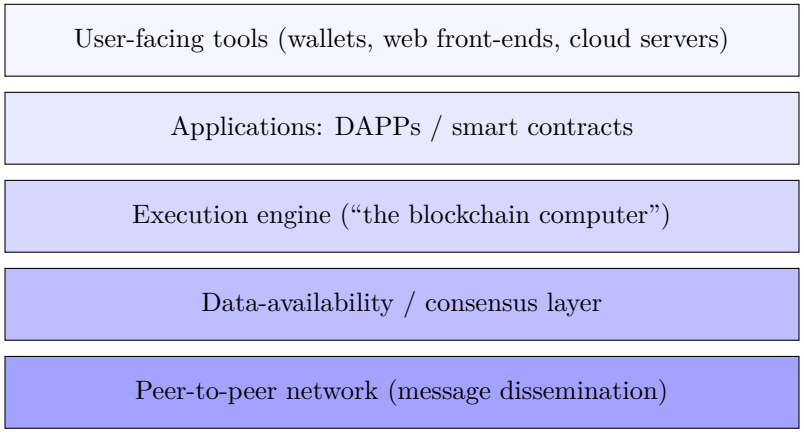
2.4 Motivation in one sentence

> *Blockchains let mutually distrusting parties agree on shared state, without a referee.*

The rest of the lecture is about *how*, technically, that agreement is reached and made useful.

3 The blockchain stack

Section 1 described a blockchain as a data structure, a network, and a protocol. Zooming in on a working system, it is useful to think of it as a stack of layers, each relying on the guarantees of the layer beneath it.



We build the picture from the bottom up.

3.1 The peer-to-peer network layer

Every layer above depends on one service: getting a message from any node to every other node. That service is provided by a *peer-to-peer network*, for our purposes roughly "the Internet."

Structure.
There is no central server. Each node maintains connections to a small, changing set of other nodes, its *peers*. A joining node contacts a few hard-coded *bootstrap* peers, then learns about more peers from them.

Dissemination.
New data spreads by *gossip* (flooding): a node that receives a previously unseen transaction or block forwards it to its peers, who forward it to theirs, and so on. After a few hops almost every node has it. There is no global broadcast primitive, only this hop-by-hop relay.

Guarantees.
Best-effort only. Messages may be delayed, dropped, duplicated, or delivered out of order, and the network may *partition* into groups that cannot reach one another. These are precisely the conditions that make the consensus layer's job hard (Section 3.4).

Trust.
The transport authenticates nothing. A peer can send anything. This is tolerable because content is self-verifying at the layer above: every transaction carries a digital signature and every block a hash, so a node checks what it receives regardless of which peer relayed it.

Permissionlessness.
Anyone can run a node and connect; there is no gatekeeper for the network itself.

Attacks that target this layer directly (eclipse attacks, surrounding a victim with attacker-controlled peers, and network-level denial of service) and the design of gossip protocols that resist them are active research and out of scope for this course.

3.2 The consensus layer, informally

At the base is a *public append-only data structure*, maintained by replication: many nodes each store a copy of the data and run a protocol that keeps their copies consistent. The properties we want from it are:

Persistence.
Once data has been added, it can never be removed.

Safety (consistency).
All honest participants hold the same data.

Liveness.
Honest participants can get new transactions added.

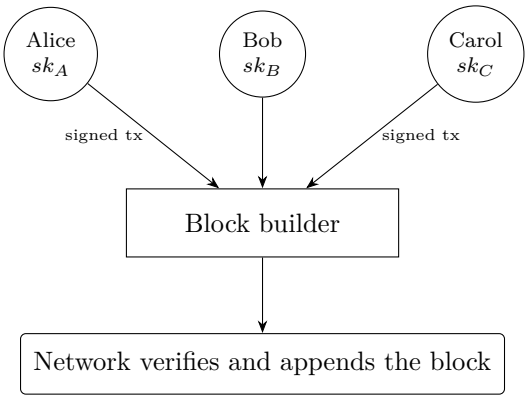
Permissionlessness (sometimes).
Anyone can submit data; there is no registration or authentication step to become a participant.

Two properties are deliberately *absent* from this list, and each is a later lecture in its own. *Scalability*: because every node re-executes every transaction, the base layer's throughput is capped by design; raising it is the subject of Lecture 6 (payment channels and rollups). *Privacy*: every transaction and account balance is visible to every node, so confidentiality has to be added on top (Lecture 7). Two further caveats matter from the start, because the informal statements above are idealisations.

Remark 3.1 (How strong "can never be removed" is depends on the protocol)**.** Under proof of work (Bitcoin) there is no absolute finality: "can never be removed" means *prohibitively expensive to remove*, not mathematically impossible. A sufficiently resourced party can in principle rewrite recent history (a "deep reorganisation"), and the design merely makes doing so cost more than it could ever be worth. Chains with a BFT-style finality gadget (Ethereum's proof of stake) are different: once a block is *finalized*, reverting it is not just costly but requires at least 1/3 of validators to provably violate the protocol and be slashed, so under the protocol's assumptions a finalized block genuinely cannot be undone.

Remark 3.2 (Safety holds for settled data)**.** All honest nodes agree on blocks that are *finalized* or buried deep enough in the chain. Very recent blocks can differ between nodes for a short time (a temporary "fork"); this is resolved within a few blocks. Lecture 3 makes this precise.

3.3 How data gets added



The lifecycle of a change to the shared state:

1. A user creates a *transaction* describing the change and *signs* it with their secret key (Section 6), so that anyone can check the change was authorised by the right party.
2. A *block builder* collects many transactions and packages them into a candidate *block*, in some order.
3. The network *verifies* the block (every signature valid, every transaction allowed by the current state, the block itself well-formed) and only then *appends* it. Invalid blocks are rejected by honest nodes.

3.4 Why agreement is hard

If the network were perfectly reliable and every participant honest, keeping the replicas consistent would be trivial. The difficulty comes from four failure modes that a real protocol must tolerate simultaneously.

Network delay.
Messages can arrive late or out of order. Two honest nodes can therefore observe the *same* set of transactions in *different* orders. A protocol cannot assume everyone sees events in one global sequence.

Network partition.
The network can split into groups that temporarily cannot reach each other at all. Each side then knows only its own transactions, and must decide whether to keep making progress without hearing from the others.

Crash faults.
A node can simply stop (power loss, a bug, being switched off). The remaining nodes must continue without knowing whether, or when, it will return.

Byzantine (malicious) faults.
A faulty node can behave *arbitrarily*, including in ways designed to break the protocol. The characteristic move is *equivocation*: telling different things to different peers (e.g. telling one half of the network "the next block is B" and the other half "the next block is B'"). Handling equivocation is the heart of the Byzantine Generals Problem, treated in Lecture 3.

A consensus protocol is a set of rules that produces persistence, safety, and liveness *despite* all four of these happening at once, as long as the faulty fraction of participants stays below some threshold. Bitcoin's answer (Nakamoto consensus, built on proof of work) and the Proof-of-Stake answer used by Ethereum are the subject of Lecture 3.

3.5 The execution engine

The consensus layer decides *which transactions happened and in what order*. It does not, by itself, say what those transactions *mean*. That is the job of the *execution engine* sitting directly above it.

- Its input is the ordered, agreed-upon stream of transactions coming out of consensus.
- It applies a *deterministic state-transition function*

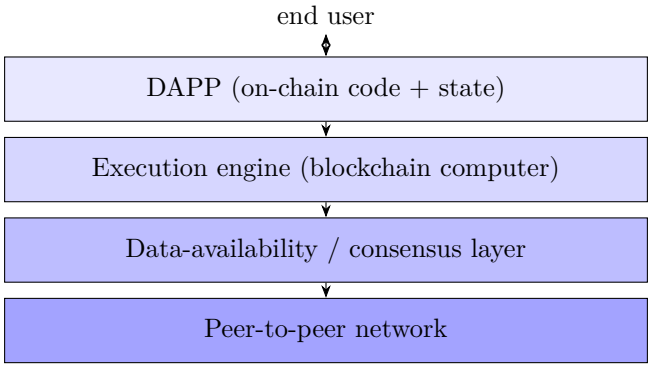
$$\delta : \ (\text{state},\ \text{transaction}) \ \longmapsto \ \text{state}'.$$

- Determinism is essential: because every honest node runs the *same* δ on the *same* ordered input, every honest node computes the *same* new state. Anyone can re-run the computation and check the result.
- The resulting state (or a short commitment to it, see Section 5.2) is written back into the consensus layer, so that the agreed state is itself part of the append-only record.

This shared, re-executable computer is what allows *code*, and not just a currency balance, to live on-chain.

3.6 DAPPs and the whole stack

A *decentralized application* (DAPP) is a program whose code and state live on-chain and which advances by consuming user transactions. The most developed class of DAPPs is decentralized finance (exchanges, lending markets, stablecoins), which is the subject of Lecture 5.



The user interacts with the DAPP (usually through a wallet and a web front-end); the DAPP's logic runs on the execution engine; the resulting state transitions are ordered and recorded by the consensus layer; and every message between nodes travels over the peer-to-peer network at the base (Section 3.1).

4 Cryptographic primitive 1: hash functions

Every blockchain system rests on *two* primitives: *hash functions*, covered in this section, and *digital signatures*, covered in Section 6.

4.1 What a hash function is

Definition 4.1 (Hash function)**.** A *hash function* is an efficiently computable function

$$\mathrm{H} : M \longrightarrow T, \qquad \text{with } |M| \gg |T|,$$

that maps an input of (almost) arbitrary size to a fixed-size output. The output set is $T = \{0,1\}^n$ for some fixed n; the value $\mathrm{H}(m)$ is called the *hash* (or *digest*) of m, and n is a parameter of the particular hash function.

Informally, think of $\mathrm{H}(m)$ as a short, fixed-size *fingerprint* of m: cheap to compute in the forward direction, and (for a *cryptographic* hash function) behaving as though the output had been assigned at random, so that it can be neither inverted nor made to collide. The next subsection makes "cryptographic" precise.

4.2 What we need from a hash function

For the uses in this course a cryptographic hash function must satisfy three properties, stated here informally. The boxed formal versions are optional, useful if you have seen game-based security definitions before, not required for the exam.

1. Collision resistance. It is infeasible to find *any* two inputs $x \neq y$ with $\mathrm{H}(x) = \mathrm{H}(y)$. Because $|M| \gg |T|$, collisions certainly exist; the requirement is that no efficient algorithm can *find* one. This is the property that lets a hash stand in for its data: if you and I each compute $\mathrm{H}(m)$ and get the same 32-byte value, we are (assuming collision resistance) almost certainly holding the same m.

Optional, formal definition (not examinable)

A *collision* for H is a pair $x \neq y$ with $\mathrm{H}(x) = \mathrm{H}(y)$. The family $\{\mathrm{H}_\lambda\}$ is *collision resistant* if for every probabilistic polynomial-time algorithm $\mathcal{A}$,

$$\Pr\big[(x, y) \leftarrow \mathcal{A}(1^\lambda) \ : \ x \neq y \ \wedge \ \mathrm{H}_\lambda(x) = \mathrm{H}_\lambda(y)\big] \ \leq \ \mathrm{negl}(\lambda),$$

with λ the security parameter and negl a negligible function. For a fixed function such as SHA-256 one instead quotes the best known attack (here $\approx 2^{128}$ evaluations, from the birthday bound).

2. Preimage resistance (one-wayness). Given only a hash value h, it is infeasible to find any x with $\mathrm{H}(x) = h$. The function can be evaluated forwards but not inverted.

Optional, formal definition (not examinable)

H is *preimage resistant* if for every probabilistic polynomial-time $\mathcal{A}$, with x drawn uniformly from the input space and $h = \mathrm{H}(x)$,

$$\Pr\big[x' \leftarrow \mathcal{A}(1^\lambda, h) \ : \ \mathrm{H}(x') = h\big] \ \leq \ \mathrm{negl}(\lambda).$$

3. Output looks random. The output carries no structure an adversary can exploit: flipping one input bit flips about half the output bits, unpredictably, and no shortcut steers the output toward a chosen region faster than trial and error.

Optional, formal definition (not examinable)

Formally this is modelled by the *random-oracle* idealisation: H is replaced by a black box that, on each never-before-seen input, returns a value drawn uniformly and independently from $\{0,1\}^n$. Proofs "in the random oracle model" assume every party (including the adversary) only accesses H through this box.

Example 4.1 (SHA-256)**.** SHA-256 is conjectured to meet all three, as

$$\{\, x : \mathrm{len}(x) < 2^{64} \ \textit{bits} \,\} \ \longrightarrow \ \{0,1\}^{256}.$$

The length bound is on the message length in *bits*, and is large enough to be irrelevant in practice.

4.3 Common families

Family	Output size n	Where it appears
SHA-2 (SHA-256, ...)	224/256/384/512 bits	Bitcoin uses SHA-256
RIPEMD-160	160 bits	Bitcoin address hashing (`HASH160`)
SHA-3 / Keccak	224/256/384/512 bits	Ethereum uses Keccak-256

Two details that trip people up later in the course:

- **Bitcoin's `HASH160` is a composition, not RIPEMD-160 alone.** It is $\texttt{HASH160}(x) = \mathrm{RIPEMD160}(\mathrm{SHA256}(x))$, chaining two different hash families to turn a public key into a short, 160-bit address.
- **Ethereum's "`SHA3`" is the original Keccak-256 submission, not the later NIST SHA3-256 standard.** The two differ in their padding rule; both produce 256-bit outputs. When Ethereum code or documentation says `sha3`, it means Keccak-256.

5 Built from hashing: commitments, Merkle trees, and proof of work

None of the three constructions in this section is a new primitive. A *commitment* is one call to a hash function; a *Merkle tree* is many hash calls arranged in a tree; *proof of work* is a search for an input that makes one hash come out small. All three inherit their security from the single assumption that the underlying hash function is well-behaved, each leaning on a different subset of the three properties of Section 4.2:

Construction	Publish / do	Hash properties it needs
Commitment	$H(m)$ now, open m later	one-wayness (hiding) + collision resistance (binding)
Merkle tree	one 32-byte Merkle root for a list	collision resistance (binding) only
Proof of work	an input whose hash $\leq$ target	one-wayness + output looks random

5.1 A hash as a commitment

Suppose Alice holds some data m and publishes only

$$h = \mathrm{H}(m) \qquad \text{(32 bytes for SHA-256)}.$$

Think of h as a *sealed envelope*: Alice has committed to m without revealing it. Later she can "open" the commitment by publishing m, and anyone can check $\mathrm{H}(m) = h$. Two properties are of interest.

Binding.
Alice cannot change her mind. If someone later produces an m' with $\mathrm{H}(m') = h$, they can be sure $m' = m$, because otherwise m and m' would be a collision for H. So a plain hash commitment is *computationally binding*, and this follows from collision resistance alone.

Hiding.
The commitment h should reveal nothing about m until Alice opens it. This is *not* automatic. If m is drawn from a small or guessable set, an attacker can simply enumerate candidates m' and test $\mathrm{H}(m') = h$. A plain hash commitment hides m only if m has enough entropy to make that search infeasible.

Which hash property. Binding rests on *collision resistance* (Section 4.2, property 1). Hiding rests on *preimage resistance* (property 2) *together with* enough entropy in m. One-wayness alone does not hide a value the adversary can guess and re-hash.

Remark 5.1 (Standard fix for hiding)**.** To commit to a low-entropy value and still hide it, commit to

$$h = \mathrm{H}(r \,\|\, m)$$

for a fresh uniformly random *nonce* r. Now guess-and-check fails without knowledge of r, and Alice opens by revealing (r, m). This construction reappears in Lecture 7.

Remark 5.2 (Both properties show up in blockchains)**.** Binding is what lets a hash stand in for data in the append-only record. Hiding is used, for example, in Bitcoin's pay-to-public-key-hash: a spender's public key stays hidden behind its hash until the moment the coins are spent (Lecture 2).

5.1.1 Reducing a whole list to one digest

Often we want a short digest not of one value but of a whole list $S = (m_1, m_2, \ldots, m_n)$ (for instance, all the transactions in a block) so that we can later prove what sits at a given position *without* revealing or re-transmitting the whole list. This is what a Merkle tree does: its output is called the *Merkle root* and each position proof is called a *Merkle proof*.

Interface.

- $h = \text{MerkleRoot}(S)$, one short (32-byte) value standing for the whole list.
- $\pi = \text{GenProof}(S, i)$, given the list S and a position i, returns a short *Merkle proof* π (about $\log_2 n$ hashes) that position i holds $S[i]$.
- $\text{VerifyProof}(h, i, m, \pi)$, given a claimed element m, a position i, and a *Merkle proof* π, returns accept or reject.

The party that holds the whole list (for a block, the node that assembled it) runs GenProof; anyone who knows only the root h runs VerifyProof. The mechanics of both are spelled out in Section 5.2.

Security (binding for lists). No efficient adversary can produce a tuple (S, i, m, π) with

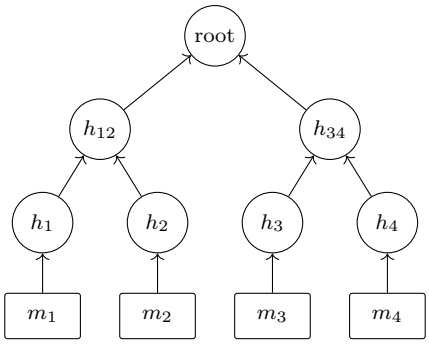
$$m \neq S[i], \qquad h = \text{MerkleRoot}(S), \qquad \text{VerifyProof}(h, i, m, \pi) = \mathsf{accept}.$$

That is, once h is fixed as the Merkle root of S, you cannot construct a convincing Merkle proof for a wrong element at any position.

Remark 5.3. For the transaction-list application only *binding* is needed: the transactions are public anyway, so hiding is not a goal here. This is the typical situation for Merkle trees.

5.2 Merkle trees

Definition 5.1 (Merkle tree)**.** A *Merkle tree* (Merkle, 1979) over a list of n values is a binary tree in which each leaf stores the hash of one list element and each internal node stores the hash of the concatenation of its two children. The hash at the top is the *Merkle root*, a single 32-byte value that stands for the entire list. Any one element can be proved to sit at its position with a *Merkle proof* of only $\lceil \log_2 n \rceil$ hashes.



Building the tree (worked example, $n = 4$).

$$\begin{aligned} \text{Step 1 (hash each leaf):} \quad & h_i = \text{H}(m_i) \quad \text{for } i = 1, \ldots, 4; \\ \text{Step 2 (hash pairs):} \quad & h_{12} = \text{H}(h_1 \parallel h_2), \qquad h_{34} = \text{H}(h_3 \parallel h_4); \\ \text{Step 3 (hash the top pair):} \quad & \text{root} = \text{H}(h_{12} \parallel h_{34}). \end{aligned}$$

Then $\text{MerkleRoot}(S) = \text{root}$.

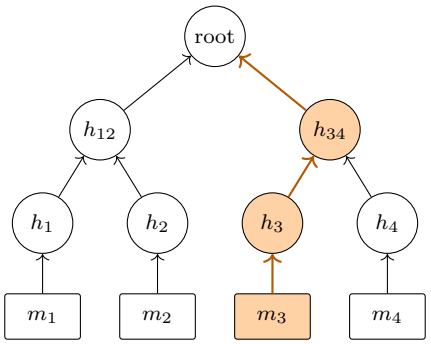
Proving membership (worked example). To prove that $S[3] = m_3$, the prover sends the Merkle proof

$$\pi = (h_4,\ h_{12}) \qquad (\text{length } \log_2 4 = 2).$$

A verifier who knows only the Merkle root h recomputes, bottom-up, the path from the claimed leaf to the root, using each proof element as the missing sibling:

$$\begin{aligned} h'_3 &\leftarrow \mathrm{H}(m_3), \\ h'_{34} &\leftarrow \mathrm{H}(h'_3 \,\|\, h_4), \\ h' &\leftarrow \mathrm{H}(h_{12} \,\|\, h'_{34}), \end{aligned}$$

and accepts iff $h' = h$. Only the hashes on the sibling path are needed; the other leaves never appear.



Shaded: recomputed by the verifier. The Merkle proof $\pi = (h_4, h_{12})$ supplies the two siblings.

5.2.1 Why it is secure

Which hash property. *Collision resistance only* (Section 4.2, property 1). Merkle trees do not need one-wayness or hiding. The committed data (transactions) is public anyway.

Theorem 5.1 (Merkle tree binding)**.** *Fix n. If* H *is collision resistant, then no efficient adversary can find (S, i, m, π) with $|S| = n$, $m \neq S[i]$, $h = \mathrm{MerkleRoot}(S)$, and $\mathrm{VerifyProof}(h, i, m, \pi) =$ accept.*

Proof idea. Suppose such a tuple exists. The honest computation of $\mathrm{MerkleRoot}(S)$ fixes the real hash values along the path from leaf i to the root. Verification of (m, π) with $m \neq S[i]$ recomputes that same path and also arrives at the root h. Walk down from the root comparing the two computations: they agree at the root but disagree at leaf i. Somewhere along the path there must be a node where the two computations feed *different* child-pairs into H but get the *same* output. That is an explicit collision for H. □

5.2.2 Case study: CVE-2012-2459 (Merkle malleability)

The theorem above is about an *idealised* Merkle tree. Bitcoin's concrete construction does not quite match it, and the gap was a real, exploitable bug.

The construction flaw. When a tree level has an *odd* number of hashes, Bitcoin *duplicates the last one* to form a pair before hashing upward.

Consequence: two lists, one root. Take a block with three transactions (t_1, t_2, t_3), and write $h_i = \mathrm{H}(t_i)$. Bitcoin pads the bottom level to (h_1, h_2, h_3, h_3), so

$$\text{root} \;=\; \mathrm{H}\big(\,\mathrm{H}(h_1 \,\|\, h_2) \;\|\; \mathrm{H}(h_3 \,\|\, h_3)\,\big).$$

The four-transaction list (t_1, t_2, t_3, t_3) produces the *same* bottom level and hence the *same* root. Same root $\Rightarrow$ same block header $\Rightarrow$ same block hash, but the second block is *invalid*, because a transaction appears twice. So for Bitcoin's tree the map *list* $\mapsto$ *root* is not injective, and MerkleRoot is not a sound commitment to the list.

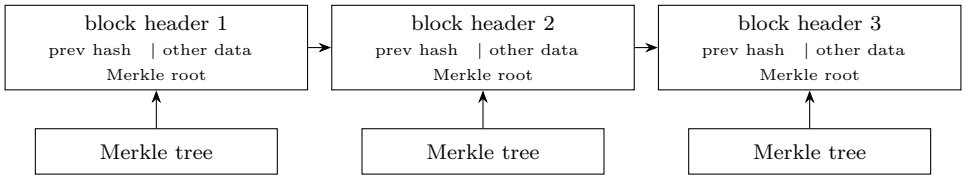
The attack (denial of service / chain split).

1. The victim broadcasts a transaction to the mempool and waits for it to be confirmed. A miner picks it up and publishes a valid block B that includes it.
2. The attacker sees B propagating and, before it has reached every node, crafts a mutated copy B': it duplicates a trailing transaction as above, so B' has the *same* block hash as B but an invalid transaction list. The attacker races B' to as many peers as it can.
3. A node that receives B' first checks it, finds the duplicated transaction, and rejects it. As an optimisation to avoid re-validating the same data, it records that *block hash* as permanently invalid.
4. The honest block B then arrives with the *identical* hash. The node discards it against the cached-invalid set without re-examining it.
5. That node can no longer accept the real block at this height, so the victim's transaction never confirms for it. It stalls and forks away from every peer that saw B first.

The fix (Bitcoin Core, May 2012). Detect this specific mutation (a Merkle tree containing the tell-tale duplicated pair) and reject the block *without* marking its hash permanently invalid, so the genuine block can still be accepted later.

5.2.3 Application: keeping the chain small

To record a block containing a large transaction list S, the chain does not need to store S itself. It stores only MerkleRoot(S), the 32-byte Merkle root, inside the *block header*. Each header also contains the hash of the previous header, chaining the headers together.



A light client that stores only the chain of headers can still be given a Merkle proof that a specific transaction is included in a specific block. It just recomputes the path to the root in that block's header. This is how a resource-constrained device verifies "this transaction is on the blockchain" without downloading every block in full.

5.3 Proof of work: hashing as a puzzle

Goal. Make it *computationally expensive* to propose the next block. If proposing a block is cheap, anyone can flood the network with candidate blocks or cheaply produce an alternative history; making it costly removes both problems and, as Lecture 3 shows, is what lets the network agree on *which* history is the real one.

The puzzle. Bitcoin's block header is 80 bytes and contains a 4-byte field called the *nonce*. A valid block is one whose header hashes at or below a threshold T:

$$\text{SHA256}^2(\text{block header}) \;\leq\; T,$$

where $\text{SHA256}^2(x) = \text{SHA256}(\text{SHA256}(x))$ and the target T is a large number the protocol adjusts over time.

- A *lower* target means fewer acceptable outputs, hence more expected hashing before one is found, a harder puzzle, more work per block.
- Once all 2^{32} nonce values are exhausted without success, the miner changes another header input (the timestamp, or the extra-nonce inside the first transaction) and searches the nonce space again.

Which hash properties. Proof of work relies primarily on *two* of the properties from Section 4.2:

- **One-wayness / preimage resistance (property 2).** Given a target T, it is computationally infeasible to find an input x (the block header, nonce included) with $\text{H}(x) \leq T$ other than by brute-force exhaustive search. H cannot be inverted to compute the required nonce.
- **Output looks random (property 3).** The outputs are uniformly distributed and show a strict avalanche effect: there is no analytical shortcut, structural weakness, or input–output correlation that lets a miner predict whether a given nonce will yield a valid hash without actually performing the hash.

How much work. Because the 256-bit output is (heuristically) uniform, a single hash attempt lands at or below T with probability $T/2^{256}$. Each attempt is independent, so the expected number of attempts to find a valid block is

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

and the protocol tunes this by changing T regularly.

Why double SHA-256. Bitcoin's proof-of-work puzzle (and its transaction IDs, and its Merkle-tree hashing) apply SHA-256 *twice*, as in the displayed formula above, rather than once. Satoshi never documented the reason. The usual explanation is *length-extension*: given only $\text{H}(x)$ and the length of x, an attacker can compute $\text{H}(x \,\|\, y)$ for a chosen suffix y *without knowing* x, for hash functions (like SHA-256) built on the Merkle–Damgård construction. Hashing twice breaks this. The attack surface is more relevant to variable-length, attacker-influenced inputs (transaction IDs, Merkle nodes) than to the fixed 80-byte header, but the same construction is used throughout for uniformity.

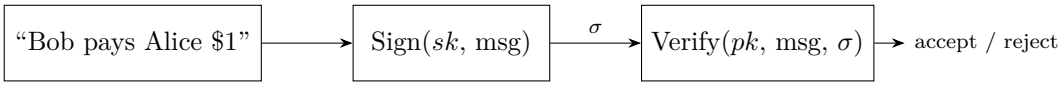
Remark 5.4. Proof of work does double duty: besides making block creation costly, it is also Bitcoin's mechanism for choosing *who* gets to propose the next block. A miner is selected in proportion to the hashing power they contribute. Lecture 3 develops this into a full consensus protocol.

6 Cryptographic primitive 2: digital signatures

The second and last primitive. Where a hash function lets anyone check that data is *unchanged*, a signature lets anyone check that data was *authorised by the holder of a particular key.*

6.1 What a digital signature is

A handwritten signature binds a person to a document: "Bob agrees to pay Alice $1", signed. The signature *depends on the message*, so that a signature valid for one message is useless on any other message.



Definition 6.1 (Signature scheme)**.** A *digital signature scheme* is a triple of algorithms:

- $\text{Gen}() \to (pk, sk)$, generates a public *verification* key pk and a secret *signing* key sk;
- $\text{Sign}(sk, \text{msg}) \to \sigma$, produces a signature on a message;
- $\text{Verify}(pk, \text{msg}, \sigma) \to \{\mathsf{accept}, \mathsf{reject}\}$, checks a signature against a message and public key.

6.2 What we need from a signature scheme

Stated informally (the boxed formal version is optional, as in Section 4.2):

1. Correctness. An honestly produced signature always verifies: $\text{Verify}(pk, \text{msg}, \text{Sign}(sk, \text{msg})) = \mathsf{accept}$ for every key pair from Gen and every message.

2. Unforgeability (EUF-CMA). An adversary who may request valid signatures on *any* messages of their choosing still cannot produce a valid signature on a *new* message, one never queried.

3. Strong unforgeability. Blockchains usually want more: the adversary cannot even produce a *different* valid signature $\sigma' \neq \sigma$ on a message that was *already* signed. Plain ECDSA does *not* have this: given a valid (r, s), the pair $(r, -s \bmod n)$ is also valid. In Bitcoin this "signature malleability" let a third party alter a transaction's ID without invalidating it, and was one of the motivations for SegWit (Lecture 2).

Optional, formal definition (not examinable)

EUF-CMA game. The challenger runs $(pk, sk) \leftarrow \text{Gen}()$ and gives pk to $\mathcal{A}$. $\mathcal{A}$ may repeatedly submit messages m_i and receive $\sigma_i \leftarrow \text{Sign}(sk, m_i)$; let Q be the set of queried messages. $\mathcal{A}$ finally outputs (m^*, σ^*) and *wins* if $\text{Verify}(pk, m^*, \sigma^*) = \mathsf{accept}$ and $m^* \notin Q$. The scheme is *existentially unforgeable under chosen-message attack* if $\Pr[\mathcal{A} \text{ wins}] \leq \text{negl}(\lambda)$ for every probabilistic polynomial-time $\mathcal{A}$. For *strong* unforgeability, replace the winning condition by $(m^*, \sigma^*) \notin \{(m_i, \sigma_i)\}$, a fresh signature on an old message now also counts as a forgery.

6.3 Families of signature schemes

Scheme	Notable property	Used in
RSA	signature / key $\geq$ 256 bytes; fast verification	rare on-chain (RSA accumulators, VDFs)
ECDSA (secp256k1)	public key 33/65 B; signature 65–72 B†	Bitcoin (legacy), Ethereum
Schnorr	public key 32 B, signature 64 B; linear structure	Bitcoin (Taproot)
Ed25519	public key 32 B, signature 64 B; fast, misuse-resistant	Solana, Cardano, Stellar, Near, Aptos, Sui
BLS	public key 48 B, signature 96 B; signatures *aggregate*	Ethereum consensus, Chia, ICP
ML-DSA	signature $\geq$ 2,420 B; post-quantum	PQ-migration candidate

Secret keys are 32 bytes across ECDSA, Schnorr, and Ed25519 (all on 256-bit-scale curves). The 33 B $\rightarrow$ 32 B shrink in *public* key size from ECDSA to Schnorr is part of Taproot's efficiency argument. † Bitcoin serialises ECDSA signatures in DER ($\approx$70–72 B); Ethereum uses a fixed (r, s, v) encoding (65 B), where v allows recovering the public key from the signature.

6.4 Where signatures are used

- **Authorising transactions**, "I approve spending these funds." Every transaction carries one or more signatures.
- **Governance**, votes in a DAO are signed messages.
- **Consensus**, in Proof-of-Stake systems, validators sign their votes on blocks.

7 Summary

The whole lecture in three points:

1. A blockchain is a *replicated, append-only, hash-linked record of transactions governed by a consensus protocol with no party in charge*. It exists to let *mutually distrusting parties agree on shared state without a trusted referee*; if a trusted party is available, use it instead.
2. A blockchain system is a *stack*: a peer-to-peer network that disseminates messages best-effort; a data-availability / consensus layer that orders transactions despite delay, partition, crashes, and Byzantine behaviour; an execution engine that applies a deterministic state-transition function to that ordered stream; and DAPPs running on top.
3. *Two fundamental cryptographic primitives*: *hash functions* (collision resistance $\Rightarrow$ a hash can stand in for its data) and *digital signatures* (bind an authorisation to a specific message, unforgeably). *Commitments*, *Merkle trees* (commit to a list with 32 bytes, prove any element with $\log_2 n$ hashes), and *proof of work* (turn hashing into a tunable cost for proposing blocks) are applications of hashing.

Looking ahead. Lecture 2 applies both primitives to a real system: Bitcoin's transaction structure and UTXO model, Bitcoin Script, and wallet / key management.