Blockchains: An Introduction to Blockchain Technology

Lecture 2, Bitcoin Mechanics:
Transactions, the UTXO Model, Script, and Key Management

Lecture Notes

Abstract

Lecture 1 introduced the two cryptographic primitives every blockchain is built from (hash functions and digital signatures) and the layered picture of a blockchain system. In lecture 2, we open the black box behind the phrase "a signature authorises a payment" and work through, in order: how a transaction travels from a user to the chain; the anatomy of a block header and of a transaction; the *UTXO* (unspent-transaction-output) model of money; **Bitcoin Script**, the small stack language that locks and unlocks every coin; the richer locks it enables (hashlocks, timelocks, multisignature, pay-to-script-hash); two important changes to the transaction *format* itself (SegWit, Taproot); a handful of real applications built purely from Script (co-signed custody, escrow with a judge, a time-locked inheritance vault); and finally **key management**, how a hardware wallet turns one backed-up secret into an unlimited supply of addresses, and how to check a balance without holding the key that can spend it. Consensus (*which* block is appended and why it cannot later be rewritten) is deferred to Lecture 3; here we take it for granted that roughly every ten minutes one valid block is appended.

Contents

1 From primitives to a running system

A *digital signature scheme* is a triple of algorithms (Lecture 1):

$$\text{Gen}() \to (pk, sk), \qquad \text{Sign}(sk, \text{msg}) \to \sigma, \qquad \text{Verify}(pk, \text{msg}, \sigma) \to \{\mathsf{accept}, \mathsf{reject}\}.$$

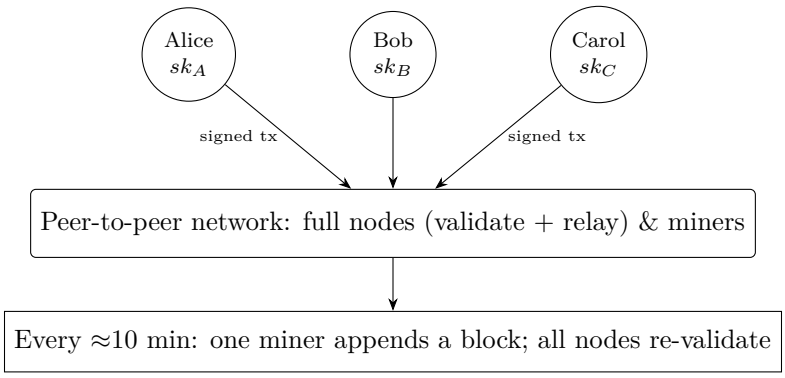
Its security guarantee is *unforgeability*: without sk, nobody can produce a signature that verifies under pk on a message that was not already signed. Bitcoin uses this in one specific way (*authorising the spending of coins*), and the whole of this lecture is an unfolding of that sentence. The question we answer is mechanical and precise:

> *When Alice "sends Bob a bitcoin", what exact bytes does she publish, what does every other computer check before believing her, and what stops anyone (including the miner who packages her payment) from redirecting or altering it?*

Two vocabulary notes before we start. A **full node** is any computer running Bitcoin's validation software and storing enough state to check every rule for itself; there are tens of thousands of them, and most are not miners. A **miner** is a full node that additionally spends energy searching for valid block headers (Lecture 1's proof of work), and is thereby occasionally entitled to append the next block. Everything a miner appends is re-checked, independently, by every full node; a miner who breaks the rules simply produces a block that the rest of the network discards.

2 The Bitcoin ledger: blocks, headers, and the chain

2.1 How a transaction reaches the chain



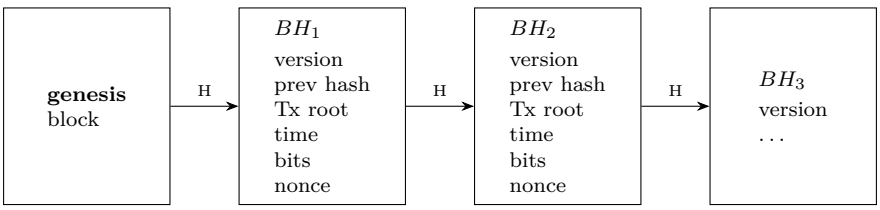
The lifecycle of a payment:

1. **Create and sign.** The user builds a *transaction* (Section 3.2) describing which existing coins to consume and which new coins to create, and signs it with the secret key(s) that control the coins being spent.
2. **Broadcast.** The transaction is handed to a few peers and spreads by gossip (Lecture 1) until essentially every node has a copy.
3. **Validate and pool.** Each full node that receives it checks it against *all* consensus rules. A well-formed, currently-valid transaction that has not yet been confirmed is held in that node's **mempool** (a local set of pending transactions). An invalid one is dropped and not relayed.
4. **Mine.** Roughly every ten minutes, one miner selects a subset of its mempool (typically the fee-densest transactions), orders them, prepends a special *coinbase* transaction (Section 3.3), and (if it has found a header meeting the proof-of-work target) broadcasts the resulting **block**.
5. **Confirm.** Every node re-validates the block and, if it is valid, appends it. The transaction now has *one confirmation*; each subsequent block adds another. Lecture 3 explains why a transaction buried under a few blocks is, in practice, permanent.

Note two consequences that recur later. First, a transaction is public *before* it is confirmed (it sits in thousands of mempools), so miners and bots can see it and react to it. Second, the miner who wins a block chooses the *order* of transactions inside it; ordering is not dictated by arrival time.

2.2 The blockchain is a chain of 80-byte headers

A block has two parts: a large *body* (the list of transactions) and a tiny, fixed-size *header*. The chain, strictly speaking, is the chain of *headers*; each header commits to its block's body via a Merkle root (Lecture 1) and to the previous header via a hash.



Every header is **exactly 80 bytes**, in this serialization order:

Field	Size	Meaning
`version`	4 bytes	block format / soft-fork signalling bits
`prev hash`	32 bytes	double-SHA-256 of the *previous* block header
`Tx root`	32 bytes	Merkle root of all transactions in this block
`time`	4 bytes	miner's claimed assembly time (Unix seconds)
`bits`	4 bytes	current proof-of-work target, compactly encoded
`nonce`	4 bytes	the free field the miner varies to solve the puzzle

Because `prev hash` is inside the header, and the header's own hash is what the next header points to, altering any past block changes its header hash and breaks every link after it, exactly the hash-linking argument from Lecture 1, now with a concrete 80-byte object. A new header is added on average every ≈10 minutes; the protocol keeps that interval roughly constant by retargeting `bits` (Section 2.4).

2.3 What the header fields mean

`time` (a *self-reported* timestamp).
Nothing forces a miner to be honest about the clock, so the protocol constrains it from both sides. A header is rejected if its `time` is (i) *not strictly greater than* the median of the previous **11** blocks' timestamps (the "median time past"), or (ii) *more than 2 hours ahead* of the node's network-adjusted time. Timestamps can therefore wobble by minutes but cannot run away, which is enough for the difficulty calculation to work.

`bits` and `nonce` (the mining puzzle).
`bits` encodes the target T; a header is valid only if $\mathrm{SHA256}(\mathrm{SHA256}(\text{header})) \leq T$ interpreted as a 256-bit integer. The miner searches by changing `nonce` (and, once the 2^{32} nonces are exhausted, other malleable inputs such as the coinbase transaction or `time`). A *lower* target means fewer acceptable outputs and more expected hashing per block. This puzzle does double duty: it makes block creation costly *and* it selects the block's proposer in proportion to hash power (Lecture 3).

`Tx root` (the commitment to the body).
The Merkle root over *every* transaction in the block, including the coinbase transaction at position 0. Because the root is in the 80-byte header, a light client that stores only headers can still be handed a $\lceil \log_2 n \rceil$-hash Merkle proof that a specific payment is in a specific block ("simplified payment verification", SPV).

2.4 A real block, read field by field

Block **916 096**, a snapshot from September 2025:

Miner	AntPool (identified from text left in the coinbase transaction)
Timestamp	2025-09-23, 15:54:50 UTC
Transactions	4 811 (≈1 377.6 BTC moved, ≈$154M at the time)
Difficulty	142 342 602 928 674.94 (retargets every 2016 blocks ≈2 weeks)
Merkle root	`4cef12cf...de1e233d3`
Block reward	3.125 BTC
Total fees	0.01335727 BTC (paid to the miner inside the coinbase transaction)

"Difficulty" is a human-friendly rescaling of the target: difficulty 1 corresponds to the easiest target the protocol ever used, and the number above means a valid header is roughly 1.4×10^{14} times rarer than that. Every 2016 blocks, each node recomputes the target from the timestamps of the last retargeting window so that, at the network's current hash rate, blocks would have taken exactly two weeks; the target moves to push the average back toward ten minutes.

3 Transactions and the UTXO model

3.1 There are no accounts, only outputs

Bitcoin has no notion of an account balance stored anywhere. All value lives in **transaction outputs**. Each output is a pair

$$(\text{value in satoshi},\ \text{ScriptPK}),$$

where 1 BTC $= 10^8$ satoshi and *ScriptPK* ("script public key", also called the *locking script* or `scriptPubKey`) is a little program stating the conditions to spend that output. An output that has not yet been consumed by any later transaction is an UTXO.

Definition 3.1 (UTXO)**.** An *unspent transaction output* (UTXO) is an output of some confirmed transaction that no confirmed transaction has yet used as an input. The set of all UTXOs at a given moment is the *UTXO set*; it is the complete description of "who owns what" and is exactly the state that every full node maintains.

A transaction, then, is a request to *destroy* some UTXOs and *create* new ones in their place.

3.2 Transaction structure

A non-coinbase transaction serialises as follows.

Field	What it holds
`version`	serialization-format version (4 bytes)
for each input:	
`TxID`	double-SHA-256 identifying the transaction being spent from (32 bytes)
`out-index`	which output of that transaction is being spent (4 bytes)
`ScriptSig`	the *unlocking script* (for SegWit inputs this data moves to the *witness*)
`sequence`	per-input flag: RBF signalling (BIP125) and relative timelocks (BIP68)
for each output:	
`value`	amount in satoshi; #BTC = `value`$/10^8$ (8 bytes)
`ScriptPK`	the *locking script*
`nLockTime`	earliest block height ($< 5 \times 10^8$) or Unix time at which this transaction may be included (4 bytes)

Key points:

- An input does *not* carry a value. Its value is whatever the referenced UTXO says; a validator looks it up in the UTXO set.
- An input names its UTXO by the pair (`TxID`, `out-index`) ("the k-th output of the transaction whose id is `TxID`"). This is called an *outpoint*.
- `TxID` = SHA256(SHA256(serialised transaction)). For legacy inputs the serialisation includes the `ScriptSig`s; for SegWit inputs it does not (Section 6.3), which is the whole point of SegWit.
- `nLockTime` is a property of the *whole* transaction and is enforced only if at least one input has `sequence` $\neq$ `0xFFFFFFFF`; a transaction with every input's sequence set to the maximum ignores `nLockTime` entirely. This interaction matters for the timelock scripts in Section 5.2.

Optional, deeper detail (not examinable)

The value 5×10^8 is the threshold that tells `nLockTime` (and `OP_CHECKLOCKTIMEVERIFY`) whether a number means a *block height* or a *Unix timestamp*: values below 500,000,000 are heights (Bitcoin will not reach block 500 million for millennia), values at or above it are seconds since 1970. A script and the transaction spending it must use the *same* interpretation or validation fails.

3.3 The coinbase transaction and coin issuance

The first transaction in every block is the **coinbase transaction** and it breaks the usual rules on purpose:

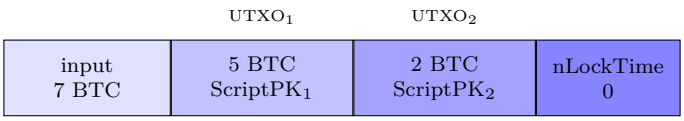
- It has **no real inputs**: its single input's outpoint is the all-zero `TxID` with `out-index` `0xFFFFFFFF`, and its "`ScriptSig`" is an arbitrary 2–100-byte field (the miner stuffs an identifier, and, since BIP34, the block height, in here).
- Its outputs may pay up to block reward + total fees of the block. The *block reward* is fixed by the protocol; the fees are the sum over all other transactions of (inputs − outputs).
- It is the **only** mechanism that creates new BTC. Every satoshi in existence entered through some block's coinbase.
- The reward **halves every 210 000 blocks** (≈4 years): 50 BTC (2009) → 25 (2012) → 12.5 (2016) → 6.25 (2020) → **3.125** (2024) → 1.5625 (≈2028) → $\cdots$. The geometric sum $50 \cdot 210{,}000 \cdot (1 + \frac{1}{2} + \frac{1}{4} + \cdots) = 21{,}000{,}000$, so the supply asymptotes to just under **21 million BTC**. Once the reward rounds to zero, miners are paid by fees alone.

Optional, deeper detail (not examinable)

Coinbase outputs are subject to a **100-block maturity** rule: they cannot be spent until 100 further blocks have been built on top. This protects against a short chain reorganisation orphaning the block and retroactively invalidating payments that were made by spending a reward that no longer exists.

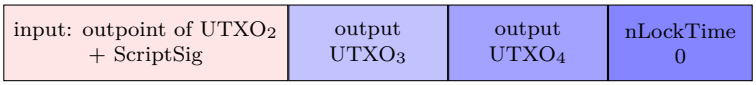
3.4 Funding and spending: a worked pair

A funding transaction. Tx_1 takes one input worth 7 BTC and produces two outputs:



After Tx_1 confirms, the UTXO set gains UTXO_1 (5 BTC) and UTXO_2 (2 BTC), and loses whatever the 7 BTC input pointed at.

A spending transaction. Tx_2 spends *only* UTXO_2:



Once Tx_2 confirms, UTXO_2 is gone from the set forever, replaced by UTXO_3 and UTXO_4. UTXO_1 is untouched and still spendable. Outputs are consumed individually and entirely. There is no such thing as spending "half" of a UTXO; to send a smaller amount you consume the whole UTXO and pay the remainder back to yourself as a second output (a *change* output).

3.5 Validating a transaction: the three checks

For a candidate transaction, every full node checks, **for each input**:

1. **Existence.** The outpoint (`TxID`, `out-index`) names a UTXO in the *current* UTXO set. (Not "a UTXO that once existed". If it has already been spent, it is simply not there.)
2. **Conservation.** $\sum$(input values) $\geq \sum$(output values). The non-negative difference is the *fee*. Creating value is forbidden; spending a little as fee is required in practice.
3. **Authorisation.** Running the input's `ScriptSig` followed by the referenced UTXO's `ScriptPK` (Section 4) terminates with `true` on top of the stack.

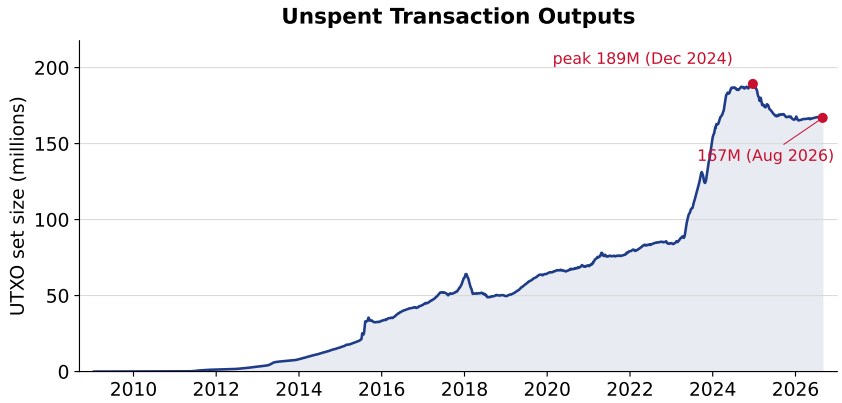
Plus whole-transaction checks: it is well-formed, within size/weight limits, its `nLockTime` is satisfied, no two inputs name the same outpoint, and it does not appear twice. If it passes, and once it is in a confirmed block, the node **removes** its spent UTXOs from the set and **adds** its outputs. The UTXO set is the running result of applying every confirmed transaction in order.

3.6 Fees

$$\text{fee} \;=\; \sum \text{input values} \;-\; \sum \text{output values}.$$

- The fee is not a field; it is implicit in the gap between inputs and outputs. A wallet sets it by making the change output smaller.
- It is chosen by the transaction's creator. Miners, wanting to maximise revenue from a size-limited block, prefer transactions offering more fee *per byte* (more precisely, per weight unit, Section 6.3). A transaction whose fee is too low waits in mempools, sometimes for days, until either the backlog clears or the sender replaces it with a higher-fee version (Replace-By-Fee, signalled by `sequence`).
- It is an **open auction for block space**.
- All fees in a block are collected by the miner through the coinbase transaction. In block 916 096 that was 0.0134 BTC of fees on top of the 3.125 BTC reward.

3.7 The UTXO set in practice



Every full node (not just miners) keeps the entire UTXO set in a local database so it can perform check 1 above in constant time. As of August 2026 it holds roughly **167 million** entries (down from a ≈189M peak in late 2024, as consolidation and fee pressure pruned dust), occupying on the order of 10–15 GB on disk. Its size, not the length of the chain, is the main scaling pressure on running a node: the full history is append-only and can be served from slow storage, but the UTXO set must be fast to query.

Remark 3.1 (UTXO model vs. account model)**.** Ethereum (Lecture 4) instead keeps an explicit mapping address $\mapsto$ balance and mutates it. Trade-offs: the UTXO model makes transactions

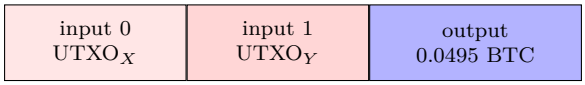
self-contained and trivially parallelisable to validate (independent inputs), gives a natural notion of "coin history", and leaks a little less by default; the account model is more compact, more intuitive for stateful smart contracts, but needs an explicit *nonce* per account to stop replay of a signed transfer. Bitcoin's (`TxID`, `index`) outpoint plays the anti-replay role that Ethereum's nonce does.

3.8 Worked example: consolidating two UTXOs

Alice controls two small UTXOs and wants one tidy one:

$$\text{UTXO}_X:\ 0.030\ \text{BTC},\ \text{ScriptPK}_X \qquad \text{UTXO}_Y:\ 0.020\ \text{BTC},\ \text{ScriptPK}_Y$$

(both locked to her key). She builds Tx_3 with **two inputs** and **one output** of 0.0495 BTC.



Validation:

- Check 3 runs *once per input*, independently: the script pair runs to `true` for input X, *and* it runs to `true` for input Y; and both outpoints $(\texttt{TxID}_X, 0)$, $(\texttt{TxID}_Y, 0)$ are in the UTXO set.
- Check 2 runs *once* on the totals: $0.030 + 0.020 = 0.050 \geq 0.0495$, so the fee is 0.0005 BTC, charged on the aggregate difference, not per input.

Each of Alice's two signatures signs (a digest of) Tx_3, so neither can be lifted onto a different transaction.

4 Bitcoin Script: locking and unlocking coins

4.1 The execution model

Every UTXO carries a **ScriptPK** (locking script), fixed when the UTXO was created. To spend it, the input supplies a **ScriptSig** (unlocking script). A validator combines them and runs the result on a single stack; the input is authorised **iff execution finishes without error and leaves a single `true` on top of the stack**.

> ***Optional, deeper detail (not examinable)***
>
> "Combines" needs care. Before 2010, Bitcoin literally concatenated ScriptSig ‖ ScriptPK and executed the concatenation. Since a 2010 change, the two are run *in sequence with the stack carried over*: first the `ScriptSig` runs on an empty stack, then the `ScriptPK` runs on the resulting stack. Standardness rules also require the `ScriptSig` to be *push-only* (it may push data but not call operational opcodes). We will keep writing "ScriptSig | ScriptPK" as shorthand for this run-in-sequence semantics.

Script is a **stack machine**. A program is a sequence of items; each item is either *data* (pushed onto the stack) or an *opcode* (pops some arguments, pushes some results). Crucially, Script is **not Turing complete**: there are **no loops** and no backward jumps. Every script halts, in time linear in its own length. This is a deliberate design choice (Section 4.3): a validator must be able to bound the cost of checking a transaction before running it.

4.2 A catalogue of opcodes

The essentials for this lecture (numeric "value" is the opcode's byte):

Opcode	Byte	Effect
`OP_0` / `OP_FALSE`	0	push an empty (zero) value
`OP_1` … `OP_16` (`OP_TRUE`=`OP_1`)	81–96	push the integer 1…16
`OP_DUP`	118	duplicate the top stack item
`OP_DROP`	117	discard the top stack item
`OP_EQUAL`	135	pop 2; push `true`/`false`
`OP_EQUALVERIFY`	136	pop 2; fail unless equal
`OP_VERIFY`	105	fail unless top is `true`; pop it
`OP_RETURN`	106	fail immediately (used to mark data-only outputs)
`OP_IF` / `OP_ELSE` / `OP_ENDIF`	99/103/104	pop 1 at `OP_IF`; run the matching branch
`OP_SHA256`	168	pop 1; push its SHA-256
`OP_HASH160`	169	pop 1; push RIPEMD160(SHA256(·))
`OP_HASH256`	170	pop 1; push SHA256(SHA256(·))
`OP_CHECKSIG`	172	pop pk, pop σ; push whether σ signs this tx under pk
`OP_CHECKMULTISIG`	174	t-of-n signature check (Section 5.3)
`OP_CHECKLOCKTIMEVERIFY` (CLTV)	177	fail unless tx `nLockTime` $\geq$ top; does *not* pop
`OP_CHECKSEQUENCEVERIFY` (CSV)	178	analogous, for the input's `sequence` (relative timelock)

Remark 4.1 (What is signed exactly?)**.** `OP_CHECKSIG` does not sign an arbitrary message you choose. It reconstructs a *signature hash* ("sighash") from the spending transaction (which inputs and outputs are covered depends on a *sighash flag* byte appended to the signature) and checks σ against that. The default flag, `SIGHASH_ALL`, commits to the whole transaction, which is why a valid signature cannot be moved to a different transaction or have an output swapped underneath it.

4.3 Why "no loops" is a feature

A miner must validate every transaction in a candidate block, and a full node must validate every transaction ever, so the cost of running a `ScriptSig` | `ScriptPK` pair has to be predictable from its length alone. Loops would break that: a 100-byte script with a loop could run for billions of steps. Section 4.6 shows a concrete near-miss (`OP_CAT`) where an innocuous-looking opcode let a short script blow up a validator's memory. Ethereum takes the opposite design and *does* allow unbounded computation, then charges per step in "gas" and aborts when the sender's gas runs out (Lecture 4); Bitcoin instead removes the possibility structurally.

4.4 P2PK and P2PKH: the standard lock

The earliest outputs (the first ≈200 000 coinbases, and Satoshi's coins) used **pay-to-public-key** (P2PK): the `ScriptPK` was just $\langle pk \rangle$ `OP_CHECKSIG`, exposing the raw public key on-chain from the moment the coin was created. The modern standard is **pay-to-public-key-hash** (P2PKH), which commits only to a *hash* of the key.

Bob prepares to receive coins:

- He runs $(pk_B, sk_B) \leftarrow \mathrm{Gen}()$ and computes the 20-byte

$$pkh_B \;=\; \mathrm{HASH160}(pk_B) \;=\; \mathrm{RIPEMD160}\big(\mathrm{SHA256}(pk_B)\big).$$

- He publishes an *address* $\mathrm{addr}_B = \mathrm{Base58Check}(\text{version byte} \,\|\, pkh_B)$, a base-58 string with a 4-byte checksum, so a mistyped address is almost always rejected rather than misdirected. The UTXO itself stores only the 20-byte pkh_B, not the address text and not pk_B.

The payer creates a UTXO with locking script

```
OP_DUP OP_HASH160 <pkh_B> OP_EQUALVERIFY OP_CHECKSIG
```

To spend it later, Bob provides the unlocking script $\langle sig \rangle \langle pk_B \rangle$. Here sig is his signature under sk_B on the transaction's sighash (Section 4.2).

Running the combined script. The combined program is

```
<sig> <pk>   OP_DUP OP_HASH160 <pkh> OP_EQUALVERIFY OP_CHECKSIG
```

and it executes as:

Stack (top at right)	Step just performed
(empty)	start
sig, pk	push the two ScriptSig items
sig, pk, pk	`OP_DUP`
sig, pk, HASH160(pk)	`OP_HASH160`
sig, pk, HASH160(pk), pkh	push $\langle pkh \rangle$ from the ScriptPK
sig, pk	`OP_EQUALVERIFY` (fails here if HASH160(pk) $\neq pkh$)
`true`	`OP_CHECKSIG` (verifies sig on the tx under pk)

Termination with `true` $\Rightarrow$ the input is authorised.

4.5 Why P2PKH is built this way

- **The public key is hidden until first spend.** The payer commits to HASH160(pk_B); pk_B appears on-chain only when Bob spends. Benefits: a small hedge against a future break of the signature scheme (a quantum adversary who can derive sk from pk still sees only a hash while the coin sits unspent); and shorter addresses. Caveats: the hedge *evaporates on address reuse*, after the first spend pk_B is public, so re-receiving to the same address exposes it; P2TR (Taproot) outputs publish a key directly; and the ancient P2PK coins were never protected at all.
- **A miner cannot redirect the payment.** Suppose a miner tries to replace pkh_B in the output with pkh_{miner}. That edits the transaction that *created* UTXO$_B$, which invalidates the payer's signature on that transaction (sighash covers the outputs). The malformed transaction is rejected by every other node.

4.6 `OP_CAT` and the cost of validation

`OP_CAT` concatenates the top two stack items: $\langle abc \rangle \langle def \rangle \Rightarrow \langle abcdef \rangle$. Harmless in isolation, but consider

```
OP_DUP OP_CAT OP_DUP OP_CAT ... (repeat 50 times)
```

Each `OP_DUP CAT` doubles the size of the top item, so after 50 rounds it is 2^{50} times the starting size, a script a few hundred bytes long that forces a validator to allocate petabytes. Satoshi's 2010 response was blunt: **disable `OP_CAT`** (along with several other opcodes). A lasting side effect is that Script can no longer, for instance, verify a Merkle branch by hashing concatenations. Modern Bitcoin also caps every stack item at **520 bytes** and the total script at 10 000 bytes, which would limit the blow-up today. Yet `OP_CAT` has still not been re-enabled (proposals to do so, in a length-checked form, are a live debate).

5 Richer locks: hashlocks, timelocks, and multisig

Everything so far changed only *what the locking script says*. Script is expressive enough to encode conditions well beyond "one signature".

5.1 Hashlocks

```
OP_SHA256 <h> OP_EQUALVERIFY <pk> OP_CHECKSIG
```

To spend, the `ScriptSig` must supply $\langle\sigma\rangle\langle x\rangle$ with

- $\text{SHA256}(x) = h$, the spender knows a *preimage* of the committed hash h; **and**
- σ is a valid signature under pk.

The signature clause stops a third party who *sees* x on the network from racing to claim the coin for themselves. Hashlocks are the core of **atomic swaps** and payment channels: the same secret x can unlock a coin on one chain and, once revealed there, be reused to unlock a matching coin on another, so either both transfers happen or neither does. We return to this in Lecture 6.

5.2 Timelocks

An **absolute** timelock makes a coin unspendable until a chosen block height or wall-clock time.

```
<t> OP_CHECKLOCKTIMEVERIFY OP_DROP <pk> OP_CHECKSIG
```

Mechanics:

- `OP_CHECKLOCKTIMEVERIFY` fails unless the spending transaction's `nLockTime` is $\geq t$ *and* that transaction's spending input has a non-final `sequence` (recall Section 3.2: otherwise `nLockTime` is ignored, which would make the check meaningless) and t and `nLockTime` agree on the height-vs-time interpretation.
- CLTV does *not* pop t (it only inspects it), so the explicit `OP_DROP` clears it from the stack.
- What remains, $\langle pk\rangle$ `OP_CHECKSIG`, is an ordinary single-signature check.

Net effect: the UTXO cannot be spent before t, and behaves like plain pay-to-pubkey afterwards. This is the building block for vaults, escrow with a refund deadline, payment channels, and the inheritance script of Section 7.3.

Optional, deeper detail (not examinable)

There are *four* timelock mechanisms, easily confused:

	Absolute	Relative
transaction-level	`nLockTime` field	`sequence` field (BIP68)
script-level (opcode)	`OP_CHECKLOCKTIMEVERIFY` (BIP65)	`OP_CHECKSEQUENCEVERIFY` (BIP112)

"Relative" means "N blocks *after the UTXO being spent was itself confirmed*", which is what payment channels need. The opcode versions let the *output's own script* demand the delay, rather than trusting the spender to set the transaction field.

5.3 Multisig and `OP_CHECKMULTISIG`

Goal: spending requires any t of n named keys to sign. The script for 2-of-3:

```
<2> <PK1> <PK2> <PK3> <3> OP_CHECKMULTISIG
```

and a `ScriptSig` that spends it (the leading `OP_0` is explained below; under P2SH the redeem script is appended too, Section 5.4):

```
<0> <sig1> <sig3>
```

`OP_CHECKMULTISIG` executes in one shot:

1. pop the count n ($= 3$); pop n pubkeys PK_3, PK_2, PK_1;
2. pop the count t ($= 2$); pop t signatures sig_3, sig_1;
3. pop **one extra** item and ignore it, a long-standing implementation bug reads one argument too many; the convention is to supply a dummy `OP_0`;
4. walk the pubkey list and the signature list *both left-to-right*: each signature must verify against the *next as-yet-unused* pubkey. Pubkeys may be skipped; signatures may not. Push `true` iff all t signatures were matched.

Because the scan is one-directional, **the signatures must be given in the same relative order as their pubkeys** (here sig_1 before sig_3). Stack trace:

Stack (top at right)	Step
(empty)	start
0, sig_1, sig_3	push ScriptSig items
0, sig_1, sig_3, 2, PK_1, PK_2, PK_3, 3	push ScriptPK items
`true`	`OP_CHECKMULTISIG` (as above)

Optional, deeper detail (not examinable)

`OP_CHECKMULTISIG` does not exist in Tapscript (the Taproot script version); there, t-of-n is built from `OP_CHECKSIGADD`, which accumulates a count of valid signatures and avoids both the dummy-`OP_0` wart and the batch cost.

Abstractly the address of a multisig arrangement should be a hash of the *policy*, something like H($PK_1, \ldots, PK_3$, "2-of-3"). But a plain output script has nowhere to *put* a policy the payer was never told; that is what P2SH solves.

5.4 P2SH: pay to script hash

Pay-to-script-hash (BIP16, 2012) lets the *payee* specify the spending conditions as a *redeem script*, and lets the payer commit to just its hash.

- The payee builds a redeem script R (e.g. the 2-of-3 multisig above), computes HASH160(R), and publishes it as an address in base-58.
- The payer sends funds to a UTXO with locking script

  ```
  OP_HASH160 <HASH160(R)> OP_EQUAL
  ```

- To spend, the redeemer supplies

  ```
  <sig1> ... <sign> <R>
  ```

  i.e. the satisfying inputs *and* the redeem script itself, revealed now for the first time.

A validator checks **two** things:

1. HASH160(R') = the hash in the `ScriptPK`, where R' is the last item of the `ScriptSig`, the revealed script really is the one the address paid to;
2. running R' with the supplied signatures on the stack returns `true`, the conditions are actually met.

So the **payee** chooses (and, until spend time, hides) the policy; the payer only ever handles a short hash and a familiar-looking address. The redeem script is still bounded by the 520-byte push limit, so a P2SH bare-multisig tops out around **15** compressed pubkeys. "Arbitrarily complex" has a hard ceiling. SegWit's P2WSH (Section 6.5) is the same idea with the witness discount and a 256-bit (SHA-256) script hash.

6 How the transaction format evolved

Sections 4–5 varied the *lock*. The next two changes are to the transaction's **serialisation format**, how the bytes are laid out and, critically, which bytes the `TxID` is computed over.

6.1 Signature malleability

ECDSA malleability. For every valid ECDSA signature there is a second, differently-encoded byte string that is *also* a valid signature on the exact same message from the exact same key. Anyone relaying the transaction can swap the original signature for this alternate encoding, with no knowledge of the private key and no change to what the transaction does.

Why this bit:

- The signature is part of the legacy transaction serialisation, so changing it changes `TxID` = SHA256(SHA256(tx)).
- A transaction's author therefore *cannot know its final TxID* until it confirms. A third party (or the miner) may have altered it in flight.
- Software that assumed "I signed it, so I know its TxID" could be confused into thinking its own transaction had failed.

Note that malleability never lets an attacker *steal* funds or redirect an output. The sighash still binds those. It only lets them rename the transaction.

6.2 Case study: Mt. Gox

Mt. Gox, based in Tokyo, was the largest Bitcoin exchange in the world, handling over 70% of all BTC trades by 2013. In February 2014 it halted all withdrawals and, days later, filed for bankruptcy with about **850 000 BTC** missing. Its public explanation was transaction malleability: attackers, it claimed, mutated the TxID of withdrawal transactions before confirmation; Gox's software watched for the *original* TxID, never saw it confirm, and re-sent the withdrawal, paying twice.

The mechanism is real, but the *attribution* did not survive scrutiny. Decker and Wattenhofer (ESORICS 2014) measured every malleability event on the chain and found that at most ≈302 000 BTC had ever been involved in any malleability attack *network-wide*, and at most ≈386 BTC could plausibly have been taken from Gox this way, against an 850 000 BTC hole, with coins that had, by later analysis, been leaking from Gox since 2011. The lesson for these notes: keep malleability as a genuine protocol *defect* worth fixing; drop the "it caused the Gox collapse" story.

6.3 Segregated Witness (SegWit, 2017)

Fix: move each input's signature data out of the `ScriptSig` and into a separate **witness** structure that is *not* part of the bytes the TxID is hashed over:

$$\texttt{TxID} \;=\; \mathrm{H}(\text{transaction } \textit{without} \text{ witnesses}).$$

Consequences:

- For **SegWit inputs**, the TxID no longer depends on any signature, so their malleability is gone. (Legacy-style inputs still exist and are still malleable; SegWit fixes it only for coins spent the new way.)
- A second identifier, the `wtxid`, *does* commit to the witness; a block carries a Merkle root over wtxids too, in the coinbase transaction.
- Witness bytes are billed at a **discount**. Bitcoin replaced the 1 MB block-*size* cap with a 4 000 000 *weight-unit* cap, where a non-witness byte costs 4 weight units and a witness byte costs 1. Moving signatures into the witness thus makes signature-heavy transactions cheaper and raised effective throughput without a hard-fork size increase.

6.4 Taproot (2021)

Taproot (BIPs 340–342, activated November 2021) bundles two ideas:

- **Schnorr signatures** (BIP340) replace ECDSA on the same secp256k1 curve. Schnorr is provably secure under cleaner assumptions, non-malleable, 64 bytes fixed, and (importantly) *linear*: a sum of public keys has a matching sum of signatures, so n parties can jointly produce *one* signature that looks exactly like a single-signer's (key aggregation, e.g. MuSig2).
- **MAST** (Merklized Alternative Script Trees): a Taproot output commits to a Merkle root of *several* alternative spending scripts. To spend, you reveal only the one branch you are using plus its Merkle path. The other branches never appear on-chain.

The upshot: a complex policy (multisig, timelocked fallbacks, ...) that *is* exercised cooperatively leaves the same 1-input, 1-signature footprint as the simplest possible payment (better privacy and lower fees).

Optional, deeper detail (not examinable)

Schnorr vs. BLS aggregation. Lecture 1 introduced BLS as "the aggregatable signature scheme"; Schnorr's key aggregation above is a different mechanism aimed at a different problem, and it is worth keeping the two apart.

- **Schnorr (MuSig2, used here):** a fixed, known group of co-signers *interactively* combines their keys into one joint key *before* anything is signed, then interactively produces one joint signature on one message. On-chain this looks exactly like an ordinary single-signer payment. Nobody outside the group can even tell it was a multisig.
- **BLS (used in Ethereum's consensus layer):** many signers each sign *independently*, with no coordination, possibly on *different* messages; afterwards, anyone can combine their separate signatures into one compact aggregate, non-interactively, after the fact. This is the shape Ethereum needs: thousands of validators attesting every slot, whose signatures a block proposer must bundle without asking every validator to sit through a signing round.

In short: Schnorr aggregation trades coordination cost for a signature that is indistinguishable from a single signer's; BLS aggregation gives up that indistinguishability but removes the need for coordination at signing time.

6.5 A field guide to output types

Type	Idea
P2PK	raw public key on-chain; earliest coins only
P2PKH	hash of a public key
P2SH	hash of a payee-chosen redeem script (2012)
P2WPKH	native SegWit single-sig; same idea as P2PKH, witness-discounted
P2WSH	native SegWit script hash; same idea as P2SH, 256-bit hash, discounted
P2TR	Taproot: Schnorr key-path or MAST script-path (2021)
`OP_RETURN`	a provably *unspendable* output carrying arbitrary data.

By *value*, legacy still dominates (April 2025: P2PKH + P2SH + P2PK ≈63%, P2TR <1%). By *output count* the ordering flips, P2TR is the single most common UTXO type (≈34%), because most Taproot outputs are tiny "dust" from inscription activity.

7 Script in the wild: applications

None of the following needs anything beyond Sections 4–5.

7.1 Co-signed custody

Alice keeps her funds in a UTXO locked to addr = 2-of-2(PK_A, PK_S), where PK_S belongs to a co-signing service. To spend, she posts a transaction carrying $\langle sig_A \rangle \langle sig_S \rangle$.

- Theft of SK_A *alone* does not move her coins. The service's signature is also required, and the service applies its own checks (rate limits, 2FA, ...) before adding it.
- But 2-of-2 is fragile: lose *either* key and the funds are frozen forever. Real co-signing custody therefore uses **2-of-3** (customer key, service key, and an offline recovery key) so any one loss is survivable.

7.2 Escrow with a judge

Alice wants to buy a backpack for 0.1 BTC from Bob, paying only once it arrives, but without the option to simply refuse to pay. They use addr = 2-of-3(PK_A, PK_B, PK_J) with a mutually trusted judge J who *never holds the money*, only ever co-signs.

Happy path.

1. Alice posts 0.11 BTC to addr, creating UTXO_A. Bob sees it on-chain and ships.
2. The backpack arrives. Alice signs a transaction $\text{UTXO}_A \to (PK_B{:}\,0.1,\ PK_A{:}\,0.0095)$ (0.0005 BTC fee) with $\langle sig_A \rangle$ and sends it to Bob.
3. Bob adds $\langle sig_B \rangle$ and broadcasts. 2-of-3 is satisfied by $\{A, B\}$; the judge was not needed.

Disputes. The judge's signature substitutes for the missing party:

1. **Backpack never arrives** (Bob at fault): Alice and J sign $\text{UTXO}_A \to PK_A$; Alice is refunded.
2. **Alice never signs** (Alice at fault): Bob and J sign $\text{UTXO}_A \to PK_B$; Bob is paid.
3. **Both partly at fault**: J signs a split, e.g. sending UTXO_A to 0.05 BTC for PK_A, 0.05 for PK_B, and 0.0095 for PK_J; either party co-signs to execute it.

Because it is 2-of-3, the judge *alone* can never move the money; the worst a corrupt judge can do is take one side in a genuine dispute.

7.3 A time-locked inheritance vault

Combine an `OP_IF`/`OP_ELSE` branch with CLTV to build a "dead man's switch":

```
OP_IF <t> OP_CHECKLOCKTIMEVERIFY OP_DROP <PK_heir> OP_CHECKSIG
      OP_ELSE <PK_owner> OP_CHECKSIG OP_ENDIF
```

- The **owner** spends at any time by taking the `OP_ELSE` branch: `ScriptSig` = $\langle sig_{\text{owner}} \rangle \langle 0 \rangle$ (the `OP_0` makes `OP_IF` take the else branch).
- The **heir** can spend only after time t, via the `OP_IF` branch: `ScriptSig` = $\langle sig_{\text{heir}} \rangle \langle 1 \rangle$, and the transaction must set `nLockTime` $\geq t$ (which CLTV enforces).
- The branch selector is pushed *on top of* the signature, because `OP_IF` reads the top of the stack first.

As long as the owner is alive they periodically move the coins to a fresh vault, resetting the clock. If they stop, the heir's branch eventually matures. No lawyer, no custodian. The policy is enforced by the script.

8 Key management and hardware wallets

Locks and unlocks assume the user *has* the right secret keys available and kept safe. A real user accumulates many key pairs (ideally a fresh one per received payment, for privacy) across possibly several chains. Managing them by hand is hopeless; this section is how it is actually done.

8.1 What a wallet does

A **wallet**:

- generates key pairs and stores the secret keys;
- builds, signs, and broadcasts transactions, and verifies incoming ones;
- scans the chain for UTXOs it controls and displays a balance.

It is not a place where coins "are". Coins are UTXOs on the chain. The wallet just holds the keys that can spend them, which is why "*not your keys, not your coins*" and, equally, "lose the keys, lose the coins".

8.2 Storing one secret key

Custodial / cloud.
An exchange (Coinbase, ...) holds the key for you; you have a login, not a key. Convenient, and exactly as trust-requiring as a bank.

Software wallet.
A key stored by an app on a laptop or phone (MetaMask, ...). The key is exposed to whatever else runs on that general-purpose, internet-connected device.

Hardware wallet.
A dedicated device (Trezor, Ledger, Keystone, ...) whose secret key *never leaves the device*: the host sends it an unsigned transaction, the device shows the details on its own screen, the user approves on-device, and only the signature comes back.

Paper / brain.
Print or memorise the key. Both are fragile (loss, damage, forgetting) and used mainly as a curiosity or deep cold backup.

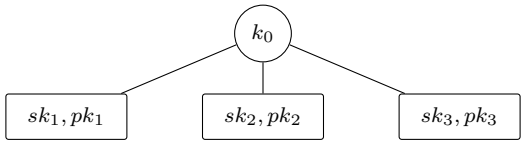
The rest of this section assumes the **hardware-wallet** model, today's standard for meaningful holdings.

8.3 Many keys from one seed: the naive scheme

Storing hundreds of independent 32-byte keys, each needing its own backup, does not scale. Instead, derive them all from a single **seed** $k_0 \in \{0,1\}^{256}$ generated once on the device:

$$sk_i \leftarrow \mathrm{HMAC}(k_0, i), \qquad pk_i \leftarrow g^{sk_i} \qquad (i = 1, 2, \ldots).$$

Here g is the fixed, public group generator (for Bitcoin, the secp256k1 base point, identical for every user), and g^{sk_i} denotes elliptic-curve scalar multiplication (written multiplicatively). HMAC is a keyed hash (Lecture 1); modelled as a pseudorandom function, it makes each sk_i computationally independent of the others.



Properties:

- **One backup for everything.** k_0 lives on the device and in an offline backup (Section 8.4). Lose the device ⇒ buy a new one, restore k_0, recompute every key.
- **Addresses look unrelated.** Without k_0, $pk_1, pk_2, \ldots$ are *unlinkable*. There is no visible structure tying them together until two of them are spent as inputs to *one* transaction. That publicly asserts common ownership, the **common-input-ownership heuristic**, the workhorse of chain-analysis (Lecture 7).

8.4 The seed phrase (BIP39)

The offline backup of k_0 is a human-transcribable **mnemonic**, almost always **24 words**:

- Each word is one of **2048** in a fixed, language-specific list (BIP39), so a word encodes $\log_2 2048 = 11$ bits.
- $24 \times 11 = 264$ bits = **256 bits of entropy** + an **8-bit checksum** (the first 8 bits of the SHA-256 of the entropy). The checksum lets a wallet reject a mis-copied phrase.

Optional, deeper detail (not examinable)

The device's actual seed is *not* those 264 bits. BIP39 feeds the mnemonic through PBKDF2 (HMAC-SHA512, 2048 iterations), with salt `"mnemonic"` ‖ passphrase, to produce a **512-bit** seed.

Optional, deeper detail (not examinable)

Practical warning. Never accept a *pre-initialised* hardware wallet (one that arrives with a seed phrase already printed, or that shows you one on first boot without generating it in front of you). Funds sent to it can be swept by whoever chose that seed. A genuine device generates k_0 on-device, from its own entropy, at setup time.

8.5 The viewing-key problem

In the naive scheme, *checking your balance requires* k_0: you need it to regenerate $pk_1, pk_2, \ldots$ and look for their UTXOs. But k_0 is also exactly what can *spend* everything. So a phone app that shows your balance must either hold the master secret (bad) or connect with the hardware wallet constantly (impractical).

Goal: split the seed into two.

- k_0 (on the hardware wallet) can generate every sk_i (and hence every pk_i), and is the only thing that can sign.
- k_{pub} (on the phone/laptop) can generate every *address* pk_i, for balance-scanning, and *nothing else*.

8.6 Splitting the spending key from the viewing key

This is what **hierarchical-deterministic (HD) wallets** do (BIP32). Present the construction in the notes' notation:

$$k_0 \in \{0,1\}^{256}, \qquad (k_1, k_2) \leftarrow \text{HMAC}(k_0, \texttt{"init"}), \qquad k_{\text{pub}} = (k_2,\ h = g^{k_1}).$$

For every $i = 1, 2, \ldots$:

$$sk_i \ \leftarrow\ k_1 + \text{HMAC}(k_2, i),$$
$$pk_i \ \leftarrow\ g^{sk_i} \ =\ g^{k_1} \cdot g^{\text{HMAC}(k_2,i)} \ =\ h \cdot g^{\text{HMAC}(k_2,i)}.$$

The last line is the trick: the *public* value pk_i can be computed from $k_{\text{pub}} = (k_2, h)$ *alone*, with no secret. But sk_i needs k_1, which k_{pub} does not contain. Therefore:

- k_{pub} generates all addresses for viewing; it cannot sign.
- k_0 is the sole signing capability and stays on the hardware wallet.

Remark 8.1 (Why the viewing key is not free of risk)**.** k_{pub} leaks *privacy* completely: whoever has it sees your entire balance and transaction history across all derived addresses. An attacker holding $k_{\text{pub}} = (k_2, h)$ *and* any single child secret sk_j recovers the master offset

$$k_1 \;=\; sk_j - \text{HMAC}(k_2, j),$$

and with k_1 can compute *every* sk_i, total loss.

9 Summary

The lecture in five points:

1. **The ledger is a set of UTXOs.** A transaction destroys some UTXOs (inputs) and creates others (outputs); every full node stores the UTXO set and checks three things per transaction, the inputs exist, value is conserved (the surplus is the fee), and each input's script runs to `true`. Consensus (Lecture 3) is assumed: ≈one valid block every 10 minutes.
2. **Every UTXO is locked by a script.** Bitcoin Script is a non-Turing-complete stack machine, run as ScriptSig | ScriptPK, valid iff it ends with `true`. P2PKH is the standard lock; hashlocks, timelocks, and multisig are richer ones; **P2SH** moves the choice of conditions (hidden until spend time) to the payee.
3. **Two changes were to the *format*, not the lock.** Signature malleability let anyone rename a transaction; **SegWit** (2017) segregates signatures out of the hashed body (fixing it for SegWit inputs) and discounts their weight; **Taproot** (2021) adds Schnorr signatures and MAST so a cooperatively-spent complex policy looks like a single-key payment on-chain.
4. **Script enables real applications** with no extra machinery: 2-of-3 co-signed custody, escrow with a non-custodial judge, and a CLTV-based inheritance vault.
5. **Key management** derives an unlimited supply of keys from one backed-up seed, keeps the signing seed on a hardware wallet, and hands a phone a *viewing key* that can enumerate addresses but not spend.

Looking ahead. Lecture 3 removes the assumption we leaned on throughout: it explains *which* block gets appended and *why* a confirmed transaction cannot later be undone, the Byzantine Generals Problem, the Dolev–Strong protocol, and Nakamoto consensus built on the proof-of-work puzzle from Lecture 1.