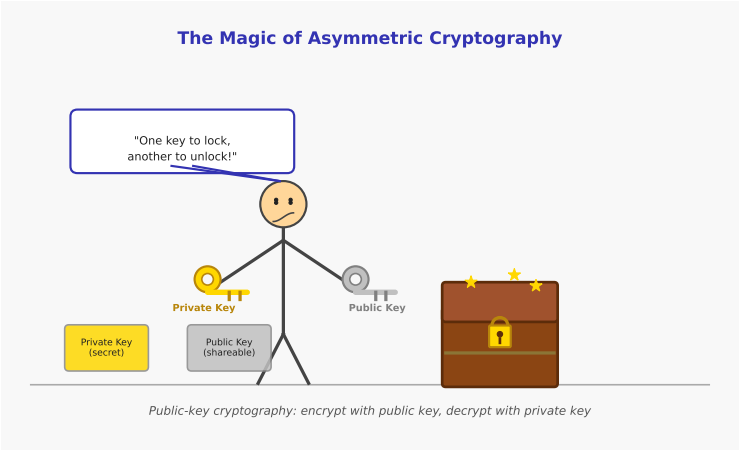
L02: Cryptographic Foundations

What if someone could forge your digital identity on the blockchain?

BSc Blockchain Course

Every Blockchain Transaction Trusts Mathematics Instead of Banks



Public-key cryptography: encrypt with public key, decrypt with private key

Imagine you could prove you own something – a coin, a contract, a vote – without showing an ID card, without a notary, without any institution vouching for you. Just mathematics. That is what cryptography does for blockchains. Today we learn how it works, and what happens if someone breaks the math.

Core tension: every blockchain transaction trusts mathematics instead of banks – but what if someone breaks the math?

Learning Objectives

By the end of this lesson you will be able to:

1. **Describe** the four properties of cryptographic hash functions. *[Understand]*
2. **Explain** how public-key cryptography enables trustless ownership. *[Understand]*
3. **Trace** a Bitcoin transaction from signing to verification. *[Apply]*
4. **Calculate** collision probabilities for different hash output sizes. *[Apply]*
5. **Compare** symmetric vs asymmetric cryptography for blockchain use. *[Analyze]*
6. **Evaluate** whether a blockchain's cryptographic design is quantum-resistant. *[Evaluate]*

Bloom's levels covered: Understand, Apply, Analyze, Evaluate

These objectives map directly to quiz and exercise assessments.

Where Cryptography Fits in the Blockchain Stack

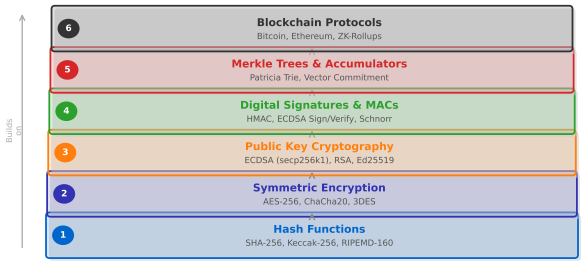
In Lesson 1 you learned *what* a blockchain is. Today we zoom into the *how* – the mathematical building blocks that make every transaction trustworthy without a bank.

Cryptography provides four guarantees:

- **Integrity:** Detect any tampering (hash functions)
- **Authentication:** Prove identity without a third party (digital signatures)
- **Non-repudiation:** You cannot deny having signed
- **Efficiency:** Verify thousands of transactions with a tiny proof (Merkle trees)

Without cryptography: anyone could spend your coins, rewrite transaction history, or impersonate you on the network.

Cryptographic Primitives Stack



- **What you see:** A layered diagram showing how cryptographic primitives stack from hashing at the bottom to applications at the top.
- **Key pattern:** Every higher layer depends on the ones below it – signatures need hashing, wallets need signatures, transactions need wallets.

What If Someone Could Sign Checks in Your Name?

Think about the last time you signed something important – a lease, a bank form, a package receipt. Your handwritten signature says: "I, and only I, authorized this."

Now consider these questions:

- How easy would it be for a skilled forger to **copy** your handwritten signature?
- If someone forged your signature on a check, could you **prove** it was not you?
- What if there were a signature that is **mathematically impossible to forge** – one that proves exactly who signed, what was signed, and that nothing was changed afterward?

That is exactly what a **digital signature** does. It is the cryptographic equivalent of a handwritten signature, except it cannot be copied, forged, or transferred to a different document. Every single Bitcoin and Ethereum transaction uses one.

The stakes are high: In 2013, a bug in Android's random number generator let attackers forge digital signatures and steal $5.7 million in Bitcoin from over 55,000 wallets. One small mathematical mistake – catastrophic, irreversible loss.

Unlike a handwritten signature, a digital signature is bound to the exact message – change one character and the signature becomes invalid.

What Is a Hash Function?

A **hash function** is a mathematical machine that takes any input – a single word, an entire movie file, a blockchain block – and produces a fixed-size "fingerprint."

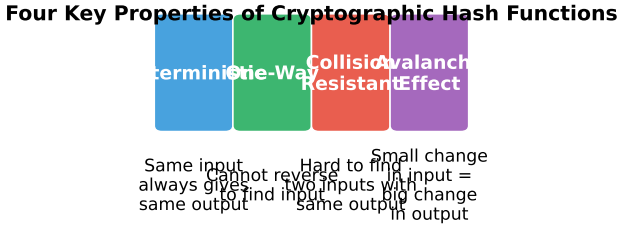
Everyday analogy: Think of a blender. You put in any combination of ingredients. Out comes a smoothie of exactly the same cup size. You cannot "un-blend" the smoothie back into the original ingredients.

Example (SHA-256):

- Input: "Hello" (5 letters)
- Output: `185f8db3...` (64 hex characters, always)
- Input: a 1 GB video file
- Output: still exactly 64 hex characters

Four required properties:

1. Deterministic (same input = same output)
2. One-way (cannot reverse)
3. Collision resistant (hard to find two matching outputs)
4. Avalanche effect (tiny change = completely different output)



`Example: SHA-256("Hello") = 185f8db32271fe25f561a...`

- **What you see:** The four properties of a cryptographic hash function illustrated with input-output examples.
- **Key pattern:** The output is always the same length regardless of input size – 256 bits for SHA-256.
- **Takeaway:** These four properties are what make hash functions useful for blockchain integrity, not just any function that produces a number.

Bitcoin block headers store just 32 bytes (256 bits) of Merkle root – summarizing up to 3,000+ transactions.

The Avalanche Effect: Why One Bit Changes Everything

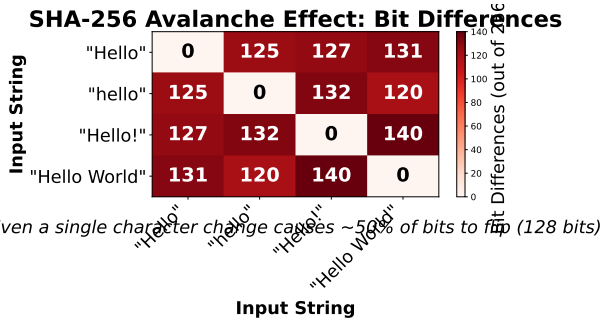
Property 1 – Deterministic: Hash the word "Hello" a million times and you always get the same output. This is what makes verification possible: anyone can re-hash and check.

Property 4 – Avalanche effect: Change *one single character* in the input, and roughly half the output bits flip unpredictably.

Worked example:

- "Hello" produces `185f8db3...`
- "hello" (lowercase h) produces `2cf24dba...`
- "Hellp" (o changed to p) produces `7c1a6bf2...`

All three outputs look completely unrelated. There is no pattern, no way to predict how the output will change. This is why tampering with a single transaction inside a block completely changes the block's hash – making the forgery immediately obvious.



Even a single character change causes ~50% of bits to flip (128 bits)

- **What you see:** Side-by-side hash outputs for nearly identical inputs, with differing bits highlighted.
- **Key pattern:** Roughly 50% of all output bits change even when only one input character differs.
- **Takeaway:** The avalanche effect is what makes blockchain tamper-evident – change one transaction and the entire block hash changes.

The avalanche property is why passwords are stored as hashes: seeing the hash reveals nothing about the password.

Collision Resistance and the Birthday Paradox

Property 3 – Collision resistance: It should be extremely hard to find two different inputs that produce the same hash output. If an attacker could do this, they could swap a legitimate transaction for a fraudulent one with the same hash.

The birthday paradox (everyday version):

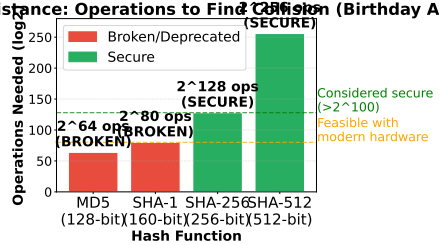
- Put 23 people in a room
- There is a 50% chance two share a birthday
- That seems surprisingly low – hence "paradox"

Applied to hashing: For a hash with an output of n bits, you expect a collision after roughly $2^{n/2}$ attempts.

Worked example (SHA-256):

- Output size: 256 bits
- Collision expected after about 2^{128} attempts
- At 1 billion hashes per second, that takes about 10^{30} years
- The universe is only about 1.4×10^{10} years old

Collision Resistance: Operations to Find Collision (Birthday Attack)



- **What you see:** Collision probability curves for different hash output sizes as the number of attempts grows.
- **Key pattern:** Shorter hashes (128-bit, 160-bit) reach dangerous collision probability far sooner than 256-bit hashes.
- **Takeaway:** SHA-256's 256-bit output gives a collision resistance of 2^{128} – far beyond any foreseeable computing power.

MD5 (128-bit) and SHA-1 (160-bit) are broken – collisions found in seconds and hours respectively. Never use them for security.

How Blockchains Use Hash Functions Everywhere

Hash functions are the most-used cryptographic tool in any blockchain. Here are six places where they appear:

1. **Block linking:** Each block header stores the previous block's hash. Change one old block and every hash after it breaks – a cascade of mismatches visible to all nodes.
2. **Merkle root:** A single 32-byte hash that summarizes all transactions in a block. We cover this in detail later.
3. **Address derivation:** Your public key is hashed to produce your blockchain address. This adds a privacy layer and makes addresses shorter.
4. **Mining puzzle (Proof of Work):** Miners search for a number (called a "nonce") that, when hashed with the block header, produces an output below a target value.
5. **Script hashes:** Bitcoin hides spending conditions behind a hash until the moment you spend.
6. **Commitment schemes:** Hash-locks enable atomic swaps and payment channels (Lightning Network).

Bitcoin uses SHA-256 (applied twice, written SHA-256d) for mining, Merkle trees, and block headers. **Ethereum** uses Keccak-256 for everything: state storage, receipts, addresses, and signatures.

Hashing is the single most frequently executed operation in blockchain software – optimized down to hardware chips.

How Do Digital Signatures Prove Ownership?

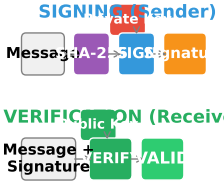
A **digital signature** is the blockchain equivalent of a handwritten signature – but mathematically unforgeable.

Three guarantees:

1. **Authentication:** Only the person with the private key (a secret number) can produce a valid signature.
2. **Integrity:** The signature is locked to the exact message. Change one character and the signature becomes invalid.
3. **Non-repudiation:** The signer cannot later deny having signed.

In a Bitcoin transaction:

- Alice creates a message: "Send 0.5 BTC to Bob"
- Alice signs with her private key
- The network verifies with Alice's public key
- No secret is shared; anyone can verify



Private key signs, public key verifies. Only the owner can sign, anyone can verify.

- **What you see:** The three-step flow: (1) hash the message, (2) sign the hash with the private key, (3) anyone verifies with the public key.
- **Key pattern:** The private key signs; the public key verifies. Information flows in one direction only.
- **Takeaway:** Verification is non-interactive – no communication with the signer is needed. Any node can check any signature.

Private key = proof of ownership. Public key = proof of identity. Digital signature = proof of authorization.

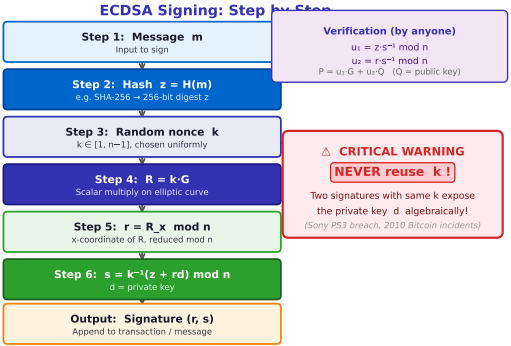
ECDSA Signing: A Step-by-Step Walkthrough

ECDSA (Elliptic Curve Digital Signature Algorithm) is the specific signature scheme used by Bitcoin and Ethereum. Here is the process in plain language.

Signing (what Alice does):

1. Hash the transaction message to get a fixed-size number
2. Pick a fresh random number (the "ephemeral key")
3. Multiply that random number by a special point on the elliptic curve to get point R
4. Combine R, the message hash, and Alice's private key using modular arithmetic to produce the signature value s
5. The final signature is the pair (r, s) – just 64 bytes

Critical rule: The random number in step 2 must be different for *every* signature. In 2010, Sony used the *same* random number twice for the PlayStation 3. An attacker extracted Sony's entire private signing key from just two signatures.



- **What you see:** A step-by-step diagram of the ECDSA signing algorithm from message input to signature output.
- **Key pattern:** The random ephemeral key is the most dangerous step – reuse it once and the private key is exposed.
- **Takeaway:** Modern wallets use a deterministic method (RFC 6979) to compute the ephemeral key from the message, eliminating the

Verification: How Anyone Can Check a Signature

Verification (what every node does):
Given: the transaction message, the signature (r, s), and Alice's public key.

1. Check that the signature values are within the valid range
2. Compute the inverse of s (a standard modular arithmetic step)
3. Multiply the message hash and the r value by this inverse
4. Use those results to compute a new point on the elliptic curve
5. If the x-coordinate of that new point matches r: **signature is valid**
6. If it does not match: **signature is rejected**

Why this works (the intuition):
The verification equation rebuilds the original random point R using *only* public information (message + public key + signature). If the signer used the correct private key, the reconstruction matches. If not, it produces a completely different point.

Performance numbers:

- ECDSA verification: about 200 microseconds per signature
- A full Bitcoin block (about 2,500 transactions): verified in under 1 second
- Schnorr signatures (Bitcoin Taproot upgrade): batch verification is 4 times faster

Verification requires only the public key – no secret information. This is why "trustless" verification is possible.

Tracing a Bitcoin Transaction: From Key to Confirmation

Let us follow a single Bitcoin payment from Alice to Bob, step by step, seeing every cryptographic primitive in action.

1. **Key generation:** Alice's wallet picks 256 random bits as her private key. It multiplies by the generator point on the secp256k1 curve to get her public key. The public key is hashed (SHA-256, then RIPEMD-160) to produce her Bitcoin address.
2. **Transaction creation:** Alice specifies: "Send 0.5 BTC from my address to Bob's address." The wallet constructs a transaction message containing the input (where Alice's coins came from) and the output (Bob's address and amount).
3. **Signing:** Alice's wallet hashes the transaction and signs it with her private key using ECDSA. The 64-byte signature is attached to the transaction.
4. **Broadcast:** The signed transaction is sent to nearby Bitcoin nodes, which relay it across the entire peer-to-peer network.
5. **Verification:** Every node that receives the transaction checks: (a) is the ECDSA signature valid? (b) does Alice have enough coins?
6. **Inclusion in a block:** A miner bundles verified transactions into a block, builds a Merkle tree of their hashes, and solves the Proof-of-Work puzzle (finding a nonce that makes the block hash small enough).
7. **Confirmation:** After six more blocks are built on top, Alice's transaction is considered final – irreversible.

Every step uses cryptography: key generation, address derivation, signing, verification, Merkle tree, and mining.

Public Key Cryptography: The Lock-and-Key Analogy

Everyday analogy: Imagine a special mailbox. Anyone can drop a letter through the slot (encrypt or verify), but only the person with the unique key can open it (decrypt or sign).

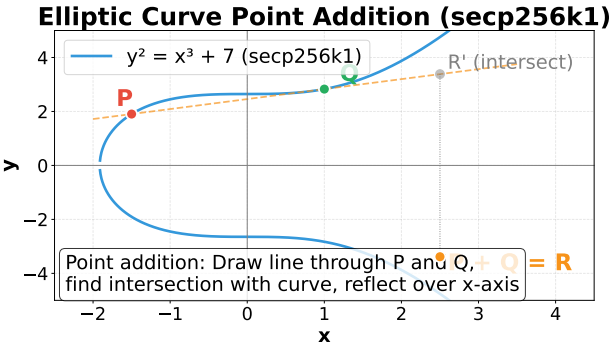
Two mathematically linked keys:

- **Private key:** A secret 256-bit number (32 bytes). Used to *sign* transactions. Must never be shared.
- **Public key:** Derived from the private key by multiplying it by a fixed "generator point" on an elliptic curve. Used to *verify* signatures. Can be shared freely.

The trapdoor:

- Easy (forward): Private key to public key takes microseconds
- Hard (backward): Public key back to private key requires roughly 2^{128} operations – infeasible

Why elliptic curves? They give the same security as older methods (RSA) but with keys 10 times smaller and faster computation.



- **What you see:** An elliptic curve showing point addition: drawing a line through two points to find a third.
- **Key pattern:** Multiplying a point by a large number (the private key) is easy; recovering the number from the result is effectively impossible.
- **Takeaway:** This "easy forward, impossible backward" property is the mathematical foundation of all blockchain ownership.

Bitcoin and Ethereum both use the secp256k1 curve. Satoshi chose it because its parameters are transparent – no hidden constants.

Key Generation: From Random Bits to a Blockchain Address

Step 1 – Generate the private key:

- Draw 256 cryptographically random bits
- Check: is the resulting number between 1 and the curve's group order?
- If yes, you have a valid private key (32 bytes)

Step 2 – Compute the public key:

- Multiply private key by the generator point on the curve
- Result: a point with x and y coordinates
- Compressed format: just the x-coordinate + one bit (33 bytes)

Step 3 – Derive the address:

- Bitcoin: SHA-256, then RIPEMD-160, then Base58Check encoding – gives a 34-character address starting with "1"
- Ethereum: Keccak-256 of the public key, take the last 20 bytes – gives a 42-character hex address starting with "0x"

Entropy sources: Hardware random number generators, operating system entropy pools, or BIP-39 seed phrases

Private key -> Public key (easy) | Public key -> Private key (impossible)

Key Generation Process

- **What you see:** The three-stage pipeline: random bits become a private key, then a public key, then an address.
- **Key pattern:** Each step is a one-way function – you cannot go backward from address to public key, or from public key to private key.
- **Takeaway:** The quality of the initial randomness determines everything. Weak randomness = predictable keys = stolen funds.

Merkle Trees: Summarizing Thousands of Transactions in 32 Bytes

Problem: A Bitcoin block can contain 3,000+ transactions. How do you check whether a specific transaction is included without downloading and verifying all of them?

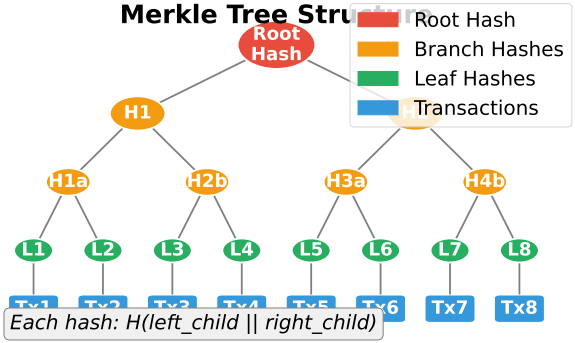
Solution: the Merkle tree.

1. Hash each transaction individually (the "leaves")
2. Pair adjacent hashes and hash them together
3. Repeat until one hash remains: the **Merkle root**
4. Store only the root in the block header (32 bytes)

Worked example with 4 transactions:

- Leaf hashes: H(Tx1), H(Tx2), H(Tx3), H(Tx4)
- Level 1: H(H(Tx1) + H(Tx2)), H(H(Tx3) + H(Tx4))
- Root: H(Level1-left + Level1-right)
- Result: one 32-byte root captures all 4 transactions

Invented by Ralph Merkle in 1979. Now used in Git, BitTorrent, ZFS, and every blockchain.



- **What you see:** A binary tree of hashes with transactions at the bottom and the single Merkle root at the top.
- **Key pattern:** Changing any single transaction changes its leaf hash, which changes its parent, which cascades all the way to the root.
- **Takeaway:** The Merkle root acts as a tamper-proof summary – one number that represents the entire set of transactions.

Merkle Proofs: Verify Without Downloading the Full Block

Merkle proof (inclusion proof): To prove a transaction is in a block, you need only a handful of hashes – not the entire block.

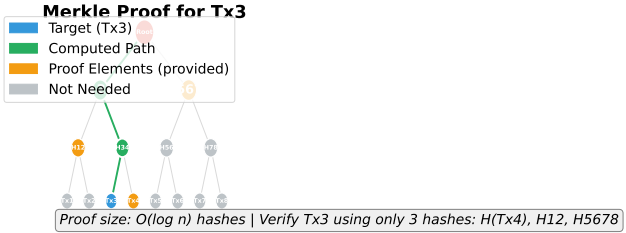
How it works:

1. Provide the sibling hashes along the path from the transaction to the root
2. The verifier re-hashes bottom-up
3. If the computed root matches the block header's root: proven

Efficiency at scale:

Transactions	Full block	Proof size
1,000	250 KB	320 bytes
10,000	2.5 MB	416 bytes
100,000	25 MB	544 bytes

Worked example: With 1,000 transactions, the proof is just $\lceil \log_2(1000) \rceil = 10$ hashes $= 10 \times 32$ bytes $= 320$ bytes. That is a 780-fold compression compared to downloading the full block.



- **What you see:** A Merkle tree with one path highlighted, showing the sibling hashes needed for a proof.
- **Key pattern:** The number of hashes needed grows logarithmically – doubling the transactions adds only one more hash to the proof.
- **Takeaway:** This is how mobile wallets (SPV clients) can verify transactions by downloading only 80-byte block headers + tiny Merkle proofs.

SPV = Simplified Payment Verification. Satoshi described it in the Bitcoin whitepaper (Section 8) as the method for lightweight clients.

When Cryptographic Assumptions Turned Out to Be Wrong

Cryptography is not permanent. Algorithms that were considered safe for decades have been broken as computing power and mathematical techniques advanced.

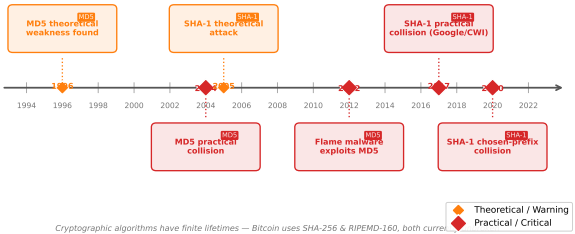
Broken algorithms – a timeline:

Algorithm	Broken	Attack
MD5	2004	Collisions in seconds
SHA-1	2017	Google's SHAttered
DES (56-bit)	1997	Brute force in 22 hours
RSA-512	1999	Factored by a network

The SHAttered attack (2017):

- Google found two different PDF files with the same SHA-1 hash
- Cost: about 110 GPU-years of computation
- Impact: SSL certificate forgery became possible
- Lesson: deprecate early – do not wait for production breaks

Historical Cryptographic Breaks



- **What you see:** A timeline of cryptographic algorithm breaks, showing the year each algorithm was compromised.
- **Key pattern:** There is a consistent gap of 10–20 years between an algorithm's widespread adoption and its first practical break.
- **Takeaway:** "Secure today" does not mean "secure forever." Blockchain systems must plan for algorithm migration.

Will Quantum Computers Break Blockchain Cryptography?

A sufficiently powerful quantum computer could break the mathematics that protects every blockchain private key. Here is what we know.

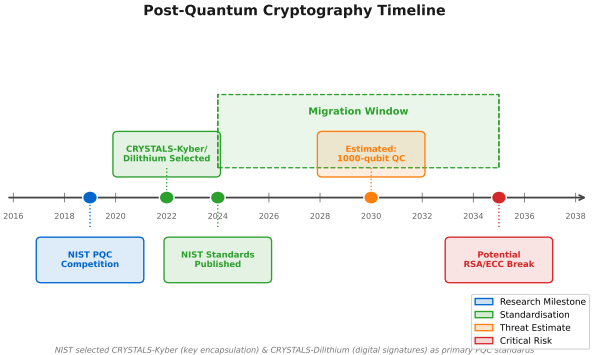
What quantum computers can do:

Primitive	Classical	Quantum
ECDSA keys	2^{128} ops	Broken entirely
SHA-256	2^{256} ops	Halved to 2^{128}
AES-256	2^{256} ops	Halved to 2^{128}

Timeline estimates:

- Today: about 1,000 noisy qubits
- Needed: about 4,000 logical (error-corrected) qubits
- Expert consensus: 10–20 years until cryptographically relevant quantum computers

Good news: Hash functions are only weakened, not broken. Bitcoin addresses that have never spent (public key not revealed) have an extra layer of protection.



- **What you see:** A timeline showing current qubit counts, projected milestones, and the threshold for breaking current cryptography.
- **Key pattern:** We are roughly 10–20 years away from the "danger zone" – enough time to prepare, but not enough to ignore.
- **Takeaway:** Post-quantum migration is an active research area. NIST finalized new quantum-resistant standards in 2024.

When the Math Is Fine but the Code Is Not

The cryptographic algorithms themselves are sound. The danger is in *how they are implemented*. Every major blockchain theft from cryptographic failure was caused by an implementation mistake, not a broken algorithm.

Real-world incidents:

Incident	Year	Mistake	Impact
PlayStation 3 hack	2010	Used same random number for two signatures	Sony's private signing key recovered
Android Bitcoin wallets	2013	Weak random number generator in Java	$5.7M stolen from 55,000+ wallets
Blockchain.info	2014	RNG failed during server crash	Private keys reconstructable
Slope Wallet	2022	Seed phrases logged in plain text	$8M stolen

The golden rule: Never write your own cryptography code. Use audited, battle-tested libraries:

- `libsecp256k1` (Bitcoin Core's library)
- `NaCl / libsodium` (general purpose)
- `cryptography` (Python)

Every incident above was preventable with standard practices. Cryptography is unforgiving – one mistake, permanent loss.

The Discrete Log Problem: Why Your Private Key Is Safe

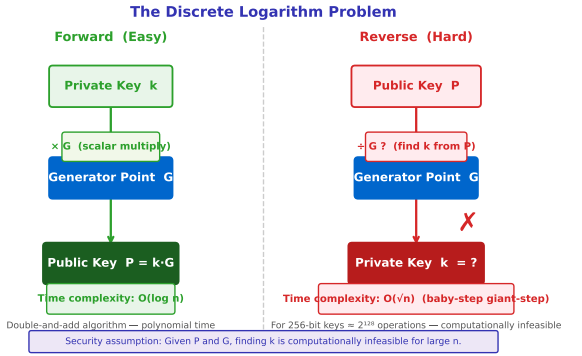
The security of every blockchain key pair rests on one hard mathematical problem: given the public key (a point on the curve), find the private key (the number used to get there).

Everyday analogy:

- Imagine a clock with 10^{77} hours on its face
- Someone tells you: "I started at 12 o'clock, moved the hand forward some number of hours, and ended up here"
- You see where the hand is, but how many hours did they add?
- With a normal 12-hour clock, you could try all 12 positions
- With 10^{77} positions, you would need more time than the universe has existed

Numbers for secp256k1:

- The curve has about 10^{77} valid points
- Best known attack: about 2^{128} operations
- At 1 billion operations per second: 10^{30} years
- Age of the universe: about 10^{10} years



- **What you see:** A visualization of the discrete logarithm problem showing the difficulty of reversing point multiplication on an elliptic curve.
- **Key pattern:** Forward computation (private key to public key) is fast; reverse computation grows exponentially with key size.

Which Hash Algorithm for Which Blockchain?

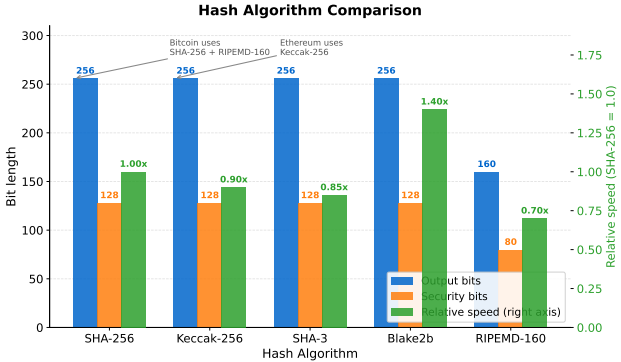
Different blockchains chose different hash functions. The choice affects security margins, performance, and compatibility.

Algorithm	Output	Used by
SHA-256	256 bit	Bitcoin (PoW, Merkle)
RIPEMD-160	160 bit	Bitcoin (addresses)
Keccak-256	256 bit	Ethereum
SHA-3	256 bit	General use
Blake2b	256 bit	Zcash

Common confusion: Keccak-256 vs SHA-3.
Ethereum adopted Keccak-256 in 2013. The SHA-3 standard was finalized in 2015 with a minor change to the padding scheme. They are *not* the same function. This difference has caused real bugs in implementations.

Security levels:

- MD5 (128-bit): broken – do not use
- SHA-1 (160-bit): deprecated
- SHA-256 (256-bit): 128-bit security – safe
- SHA-256d (double): safe + length-extension resistance



- **What you see:** A comparison of hash algorithms by output size, speed, and security status.
- **Key pattern:** Larger output sizes correlate with higher collision resistance, but also with slower computation.
- **Takeaway:** SHA-256 (Bitcoin) and Keccak-256 (Ethereum) are both considered safe for the foreseeable future against classical computers

How Is a Blockchain Address Derived from a Public Key?

Your blockchain address is **not** your public key. It is a hash of your public key, with some extra steps for error detection.

Why hash the public key?

- Shorter and more human-friendly
- Adds a quantum resistance layer: your public key is not revealed until you spend
- Checksums catch typos before you lose money

Bitcoin (P2PKH):

1. SHA-256 the public key
2. RIPEMD-160 the result (compress to 20 bytes)
3. Add a version prefix and a 4-byte checksum
4. Base58Check encode – produces a 34-character string starting with "1"

Ethereum:

1. Keccak-256 the public key
2. Take the last 20 bytes
3. Prefix with "0x" – produces a 42-character hex string

Address Derivation

Key Differences:

Bitcoin: Double hash + checksum | Ethereum: Single hash, no checksum (EIP-55 for case)

- **What you see:** The derivation pipelines for Bitcoin and Ethereum addresses, from public key to final address format.
- **Key pattern:** Both paths are one-way: you cannot recover a public key from an address (until the address is used to spend).
- **Takeaway:** Always double-check addresses before sending. There is no "undo" if funds go to the wrong address.

An address is a hashed public key with a checksum. Sending to an address does NOT reveal your public key until you spend from it.

Encryption vs Signing: A Common Confusion

Many people think blockchain "encrypts" transactions. It does not. Blockchain uses **signing**, not encryption. These are two different operations with different goals.

Encryption (confidentiality):

- Goal: keep a message **secret**
- Sender uses recipient's public key to scramble
- Only recipient's private key can unscramble
- Example: sending a secret message

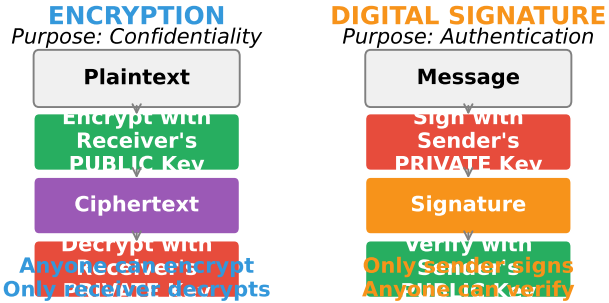
Signing (authentication):

- Goal: prove **who sent it**
- Sender uses own private key to sign
- Anyone with sender's public key can verify
- Example: authorizing a Bitcoin transaction

Blockchain uses signing because:

- Transactions are *public* – anyone can read them
- The goal is authentication, not secrecy
- Exception: privacy coins (Zcash, Monero) add encryption-like constructs

Encryption vs Digital Signatures



- **What you see:** Side-by-side diagrams showing the key usage difference: encryption uses the recipient's key; signing uses the sender's key.
- **Key pattern:** The private and public keys swap roles depending on

Multisig and Schnorr: When One Key Is Not Enough

Problem: If your private key is stolen or lost, your funds are gone forever. Single points of failure are dangerous.

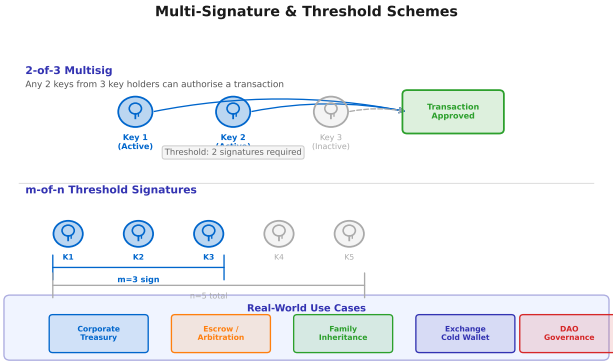
Solution: multi-signature schemes.
An "m-of-n" multisig requires *m* out of *n* designated key holders to sign before funds can move.

Common configurations:

- **2-of-3:** Corporate treasury – two directors must approve
- **2-of-2:** Lightning channel – both parties must agree
- **3-of-5:** Exchange cold storage – majority of custodians

Schnorr signatures (Bitcoin Taproot, 2021):

- Multiple signers can combine into a single 64-byte signature
- On-chain, a 3-of-5 multisig looks identical to a single-key payment
- Result: better privacy, lower fees, faster verification



- **What you see:** Different multisig configurations showing how multiple keys combine to authorize a transaction.
- **Key pattern:** Multisig eliminates single points of failure – losing one key does not mean losing funds.
- **Takeaway:** Nearly all institutional crypto custody uses 2-of-3 or 3-of-5 multisig. If you hold significant value, you should too.

How to Evaluate a Blockchain's Cryptographic Security

When someone proposes a new blockchain or cryptocurrency, use this framework to assess whether its cryptographic foundations are sound.

Five questions to ask:

1. **Hash function:** Does it use a well-studied algorithm (SHA-256, Keccak-256) with at least 128-bit collision resistance? If it uses a custom or obscure hash, that is a red flag.
2. **Signature scheme:** Does it use a standard scheme (ECDSA, EdDSA, Schnorr) with audited implementation? Roll-your-own crypto is the most common source of catastrophic vulnerabilities.
3. **Key size:** Is the key size adequate? For elliptic curves, 256 bits gives 128-bit security. For RSA, you need at least 2,048 bits for equivalent protection.
4. **Randomness:** How are keys generated? Hardware random number generators or deterministic derivation (BIP-39/BIP-32) are safe. Software-only random number generators have a history of failures.
5. **Quantum readiness:** Is there a documented migration path to post-quantum algorithms? NIST finalized standards in 2024 (CRYSTALS-Dilithium, FALCON, SPHINCS+), so "no plan" is no longer acceptable.

Scoring: 5/5 = strong cryptographic foundation. 4/5 = investigate the weak point. 3/5 or below = avoid.

This framework applies to any blockchain, token, or wallet you encounter. The math must be sound at every layer.

What 128-Bit Security Actually Means in Practice

Both Bitcoin (secp256k1) and Ethereum (secp256k1) provide 128-bit security. But what does that number actually mean?

128-bit security means: An attacker needs at least 2^{128} operations to break the system.

Putting 2^{128} in perspective:

- 2^{128} is approximately 3.4×10^{38}
- If 1 billion computers each tried 1 billion keys per second: 10^{18} attempts per second
- Time needed: $\frac{3.4 \times 10^{38}}{10^{18}} = 3.4 \times 10^{20}$ seconds, which is about 10^{13} years
- The observable universe is about 1.4×10^{10} years old
- So you would need roughly **1,000 times the age of the universe**

Security level comparison:

Algorithm	Key/Output	Security bits	Status
MD5	128 bit	64 (broken)	**Do not use**
SHA-1	160 bit	80 (broken)	**Deprecated**
SHA-256 / secp256k1	256 bit	128	**Secure**
AES-256	256 bit	256	**Post-quantum safe**

Quantum computers halve effective security: SHA-256 drops to 128-bit (still safe). ECDSA signatures are fully broken by quantum.

Advanced Topics: What Comes Next in Blockchain Cryptography

Today you learned the core cryptographic primitives. Here is a preview of three advanced topics that are actively shaping the future of blockchain.

1. Zero-Knowledge Proofs (ZKPs):

- Prove a statement is true *without revealing why* it is true
- Example: prove you are over 18 without showing your birthday
- Used by: Zcash (privacy), zk-rollups on Ethereum (scaling)
- We will revisit ZKPs in Lesson 9 (Layer-2 solutions) and Lesson 10 (Privacy)

2. HD Wallets (BIP-32 / BIP-39 / BIP-44):

- Generate millions of key pairs from a single 24-word seed phrase
- Each address is different (privacy), but all are backed up by one phrase
- Covered in detail in Lesson 3 (Bitcoin)

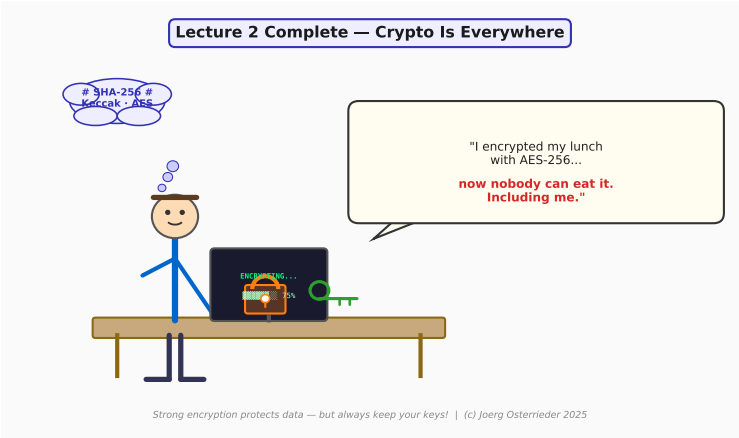
3. Post-Quantum Cryptography:

- NIST finalized three quantum-resistant signature schemes in 2024
- Bitcoin and Ethereum migration plans are active research areas
- No urgency today, but preparation is essential

Next lesson: *Bitcoin Deep Dive* – how Bitcoin assembles all the primitives from today into a working payment system. UTXO model, transaction structure, Script language, and mining mechanics.

Lesson 3 builds directly on today's cryptographic primitives – signatures, Merkle trees, and address derivation all appear in Bitcoin transactions.

One Last Thought...



Now you understand why "trust the math" is not just a slogan – it is a precise engineering statement. The math works, but only if the implementation is flawless. One reused random number, one weak key generator, and the entire house of cards collapses.

Cryptography: because "trust me" is not a security model.

Key Takeaways

1. **Hash functions:** Four properties – deterministic, one-way, collision-resistant, avalanche effect. SHA-256 (Bitcoin) and Keccak-256 (Ethereum) are the workhorses.
2. **Collision resistance:** With a 256-bit hash, you need about 2^{128} attempts to find a collision – far more than all the computing power on Earth could produce in the lifetime of the universe.
3. **Digital signatures:** The private key signs, the public key verifies. ECDSA produces a 64-byte signature that anyone can check without any secret information.
4. **The trapdoor:** Private key to public key is fast; public key to private key is computationally infeasible. This one-way property underpins all blockchain ownership.
5. **Merkle trees:** Logarithmic-size proofs let mobile wallets verify transactions without downloading entire blocks – 320 bytes instead of 250 KB for a 1,000-transaction block.
6. **Implementation matters:** Every major crypto theft was caused by implementation errors (reused random numbers, weak generators), not by broken algorithms.
7. **Quantum horizon:** Quantum computers will eventually break ECDSA but only halve hash security. Post-quantum standards exist; migration planning should start now.

Review question: If a blockchain uses a 128-bit hash function, how many attempts would it take to find a collision? Is that safe enough?

Summary / Next Lesson Preview

Blockchain replaces institutional trust with mathematical trust. Hash functions ensure data integrity; digital signatures prove ownership without revealing secrets; Merkle trees enable efficient verification of large datasets. Together, these three primitives make it possible for strangers to transact without any intermediary. But the math is only as reliable as its implementation – weak randomness, reused nonces, and unaudited code have cost millions. Looking ahead, quantum computing will eventually force a migration to new signature schemes, but hash functions will remain safe with their security merely halved, not broken.

Key Vocabulary:

- Hash function (SHA-256, Keccak-256)
- Collision resistance
- Avalanche effect
- Public key / Private key
- Digital signature (ECDSA)
- Elliptic curve (secp256k1)
- Merkle tree / Merkle root
- Merkle proof (SPV)
- Address derivation
- Multisig / Schnorr

Next lesson: *L03: Bitcoin* – how Bitcoin combines these cryptographic primitives into a functioning decentralized payment system. UTXOs, transaction scripts, mining, and the halving cycle.

Try this before Lesson 3: hash the word "blockchain" with SHA-256, then change one letter and compare the two outputs.