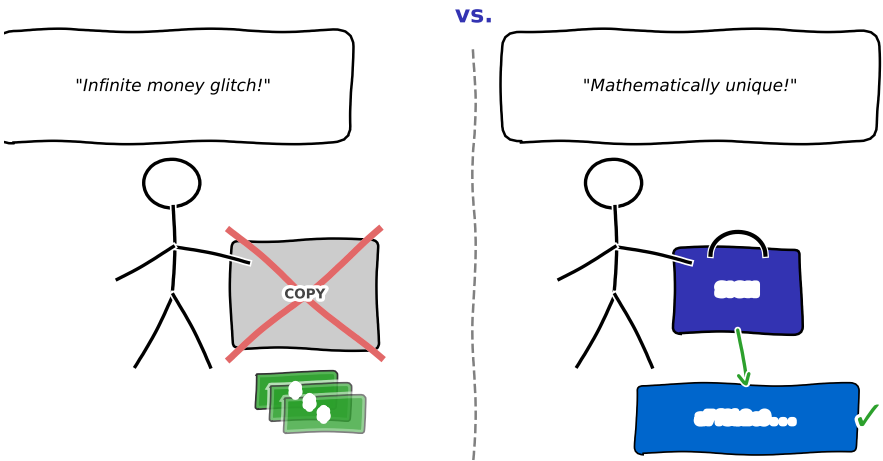
# Lesson 3.1: Cryptographic Foundations for Finance

## Module 3: The Trust Problem

Prof. Dr. Joerg Osterrieder

Digital Finance — BSc Course

# Can You Photocopy Money?




**You can copy a file. You cannot copy a cryptographic signature.**

# Learning Objectives

After completing this lesson, you will be able to:

1. **Explain** the three essential properties of a cryptographic hash function and why each matters for finance [Understand]
2. **Demonstrate** the avalanche effect by tracing how a one-bit input change propagates through SHA-256 [Apply]
3. **Construct** a Merkle tree from a set of transactions and verify a single transaction using an inclusion proof [Apply]
4. **Distinguish** between symmetric and public-key cryptography and explain how digital signatures work [Analyze]
5. **Evaluate** why the double-spend problem makes trustless digital cash fundamentally hard [Evaluate]

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

---

**Objectives follow Bloom's taxonomy: Understand → Apply → Analyze → Evaluate.**

# From Access to Trust

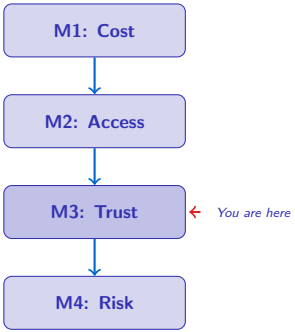
**Module 2 asked:** Who gets access to financial services?

**Module 3 asks:** How can you transact **without needing anyone's permission**?

**The Trust Problem:**

- Physical cash is scarce — you cannot copy a coin
- Digital files are infinitely copyable
- Banks solve this by keeping a **central ledger**
- But what if you want to remove the bank?

**This module's journey:** From cryptographic building blocks (this lesson) to consensus mechanisms, blockchains, and decentralized finance.




**Cryptography provides the mathematical tools to build trust without a central authority.**

## Definition: Cryptographic Hash Function

A **cryptographic hash function** is a mathematical algorithm that takes an input of *any size* and produces a fixed-size output (the "hash" or "digest"). The same input always produces the same output, but you cannot reverse-engineer the input from the output.

**Analogy:** Think of a hash function as a **fingerprint machine**.

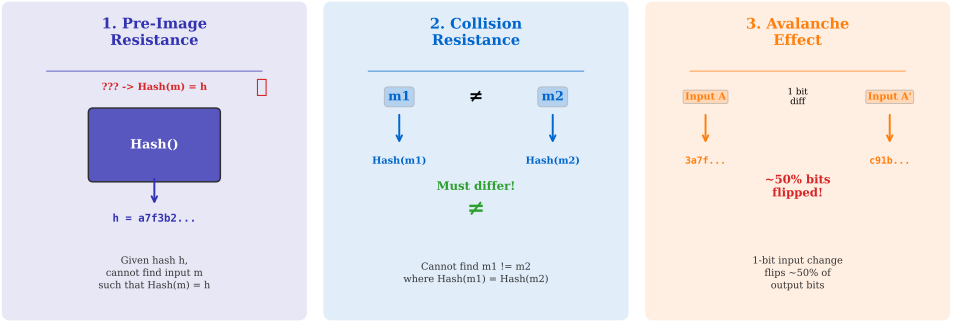
- You feed in a document (any length)
- It produces a unique 64-character "fingerprint"
- Two different documents produce two different fingerprints
- You **cannot** reconstruct the document from the fingerprint

**Example (SHA-256):**

`"Hello"` → `185f8db32271fe...` (64 hex characters)
`"hello"` → `2cf24dba5fb0a3...` (completely different!)

**A hash function is a one-way "fingerprint" — easy to compute, impossible to reverse.**

# Three Essential Properties of Cryptographic Hash Functions




- **What you see:** Three visual panels showing pre-image resistance (lock icon), collision resistance (two distinct inputs), and avalanche effect (1-bit change)
- **Key pattern:** Pre-image = one-way function, collision = uniqueness guarantee, avalanche = total output change from tiny input change
- **Takeaway:** These three properties together make reversing or forging hash outputs computationally infeasible

**These three properties together make hash functions the backbone of digital integrity verification.**

# Property 1: Pre-Image Resistance

## Definition: Pre-Image Resistance

Given a hash output $h$, it is computationally infeasible to find *any* input $m$ such that $\text{Hash}(m) = h$.

**Why this matters for finance:**

- A bank publishes the hash of a transaction: `a7f3b2...`
- An attacker who sees this hash **cannot** reconstruct the transaction details
- This allows you to **prove** you know something without revealing it

**Analogy:** Imagine locking a letter in a safe that anyone can inspect from outside, but nobody can open. Pre-image resistance means you cannot figure out what the letter says by looking at the safe.

`Hash(???) = a7f3b2c9...` ⇐ **Cannot reverse!**

---

**Pre-image resistance ensures that seeing the hash output reveals nothing about the input.**

# Property 2: Collision Resistance

## Definition: Collision Resistance

It is computationally infeasible to find two different inputs $m_1 \neq m_2$ such that $\text{Hash}(m_1) = \text{Hash}(m_2)$.

**Why this matters for finance:**

- If two different transactions produce the same hash, an attacker could **swap one for the other** without detection
- Example: Replace "Pay Alice \$100" with "Pay Eve \$10,000" — same hash, no alarm
- Collision resistance prevents this substitution attack

**How strong is SHA-256?**

- SHA-256 produces $2^{256}$ possible output values
- That is approximately $1.16 \times 10^{77}$ — more than the number of atoms in the observable universe ($\approx 10^{80}$)
- Finding a collision by brute force would take longer than the age of the universe

---

**Collision resistance guarantees that every unique input maps to a unique hash — no two documents can masquerade as each other.**

# Property 3: The Avalanche Effect

## Definition: Avalanche Effect

A tiny change in the input (even one bit) produces a **completely different** hash output. On average, changing one input bit flips approximately 50% of the output bits.

**SHA-256 Avalanche Effect: Small Input Change, Massive Output Change**

# Why Finance Needs Hash Functions

**Hash functions solve critical financial problems:**

| Financial Need | How Hash Functions Help |
|---|---|
| **Data integrity** | Hash a contract; any tampering changes the hash |
| **Transaction verification** | Hash each transaction; compare hashes to detect fraud |
| **Password storage** | Store hash of password, not password itself |
| **Blockchain** | Each block contains the hash of the previous block, creating a tamper-evident chain |
| **Digital signatures** | Sign the hash of a document (fast) instead of the entire document (slow) |
| **Commitment schemes** | Publish hash of a bid; reveal the bid later; prove it was not changed |

**Key insight:** Hash functions are the "integrity glue" of digital finance — they let you detect any unauthorized change.

**Nearly every security protocol in digital finance relies on hash functions at its foundation.**

# What Is SHA-256?

## Definition: SHA-256

**SHA-256** (Secure Hash Algorithm, 256-bit) is a cryptographic hash function designed by the U.S. National Security Agency (NSA). It produces a 256-bit (32-byte) hash, typically displayed as 64 hexadecimal characters.

**Key facts:**

- Part of the SHA-2 family (published 2001)
- Input: any data of any size (even an empty string)
- Output: always exactly 256 bits (64 hex characters)
- Used in: Bitcoin mining, Transport Layer Security / Secure Sockets Layer (TLS/SSL) certificates, digital signatures, file integrity checks

**Try it yourself:**

```
echo -n "Digital Finance" | sha256sum
→ a hash string you can verify on any computer
```

**Deterministic:** The same input always produces the same output, on any computer, anywhere in the world.

**SHA-256 is the most widely used hash function in blockchain and financial cryptography.**

# What Is a Merkle Tree?

## Definition: Merkle Tree

A **Merkle tree** (or hash tree) is a data structure in which every leaf node contains the hash of a data block, and every non-leaf node contains the hash of its two children. The single hash at the top is called the **Merkle root**.

**Analogy:** Imagine a tournament bracket.

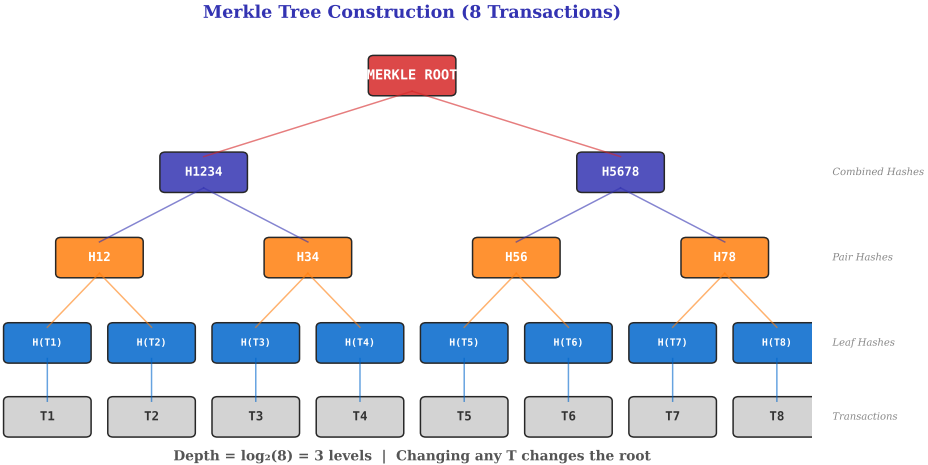
- Each player (data block) gets a "score" (hash)
- Pairs of scores are combined and hashed to produce a "round winner"
- This continues until you have a single champion: the **Merkle root**
- If any player's score changes, the entire bracket changes up to the root

**Why it matters:**

- Verifying **one** transaction in a block of 1,000 transactions requires only $\log_2(1{,}000) \approx 10$ hashes, not all 1,000
- This is what makes blockchain verification **efficient**

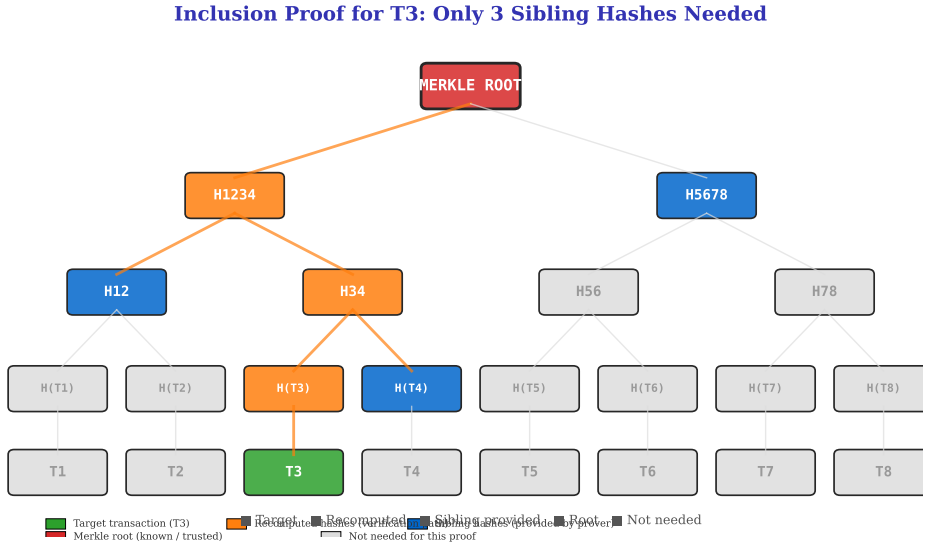
**Merkle trees enable efficient verification: check one item without downloading all the data.**

# Building a Merkle Tree: Step by Step




- **What you see:** Four transactions at the bottom, paired hashes in the middle, and a single Merkle root at the top
- **Key pattern:** Each parent node is Hash(left child —— right child), forming a binary tree with $\log_2(n)$ levels

# Merkle Inclusion Proof: Verifying One Transaction



# Merkle Trees in Finance

| Application | How Merkle Trees Help |
|---|---|
| **Blockchain** | Each block header contains a Merkle root of all transactions; lightweight nodes verify individual transactions without downloading the full block |
| **Proof of Reserves** | An exchange publishes a Merkle tree of all customer balances; each customer can verify their balance is included without seeing others' balances |
| **Audit trails** | Regulators verify the integrity of a large dataset by checking the Merkle root instead of every record |
| **Data synchronization** | Two databases identify exactly which records differ by comparing Merkle trees (used in distributed ledgers) |

**Key insight:** Merkle trees convert the problem of "verify everything" into "verify $\log_2(n)$ hashes."

**The logarithmic scaling of Merkle proofs is what makes blockchains practical at scale.**

# The Key Distribution Problem

**Before public-key cryptography, there was a fundamental problem:**

- **Symmetric encryption:** Alice and Bob share the *same* secret key
- To communicate securely, they must first *agree* on a key
- But how do you agree on a secret key if your communication channel is insecure?

**The chicken-and-egg problem:**


**Solution:** Public-key cryptography (1976) eliminated the need to share secret keys.

---

**The key distribution problem was the central unsolved challenge in cryptography until 1976.**

# What Is Public-Key Cryptography?

## Definition: Public-Key Cryptography

**Public-key cryptography** (asymmetric cryptography) uses a *pair* of mathematically related keys: a **public key** that anyone can see, and a **private key** that only the owner knows. Data encrypted with one key can only be decrypted with the other.

**Analogy: The mailbox model.**

- Your **public key** is like your mailbox slot — anyone can drop a letter in
- Your **private key** is the key to the mailbox — only you can open it and read the letters
- Anyone can **send** you an encrypted message (using your public key)
- Only you can **read** it (using your private key)

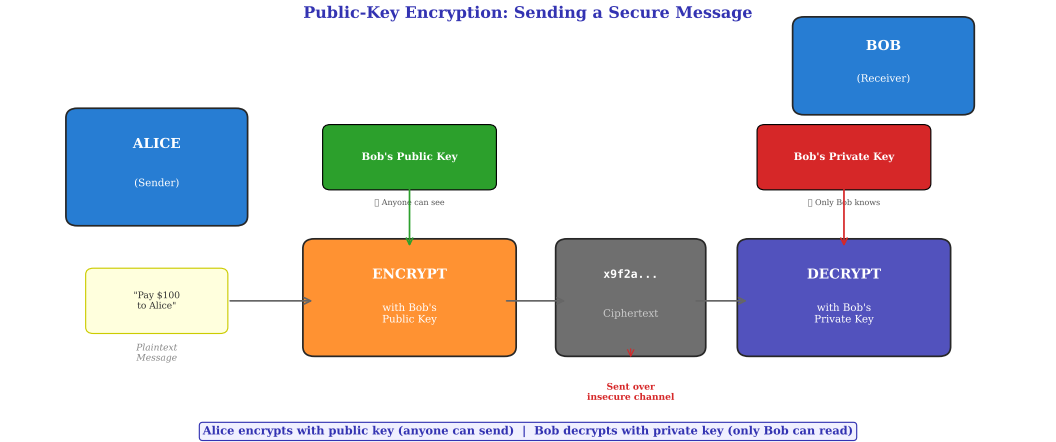
**Key properties:**

- Public key is derived from the private key, but you **cannot** compute the private key from the public key
- This one-way relationship relies on hard mathematical problems (e.g., elliptic curve discrete logarithm)

---

**Public-key cryptography solves the key distribution problem: no shared secret needed.**

# Public-Key Encryption: How It Works




- **What you see:** Alice encrypts a message with Bob's public key; only Bob's private key can decrypt it
- **Key pattern:** Public key encrypts (anyone can do this), private key decrypts (only the owner)
- **Takeaway:** No shared secret needed beforehand — the public key can be broadcast openly

# What Is a Digital Signature?

## Definition: Digital Signature

A **digital signature** is a cryptographic proof that a specific private key holder approved a specific message. It provides **authentication** (who signed it), **integrity** (the message was not altered), and **non-repudiation** (the signer cannot deny signing).

**Analogy: A tamper-evident wax seal.**

- A handwritten signature can be forged; a digital signature **cannot**
- It binds the signer's identity (private key) to the specific message
- If even one bit of the message changes, the signature becomes invalid

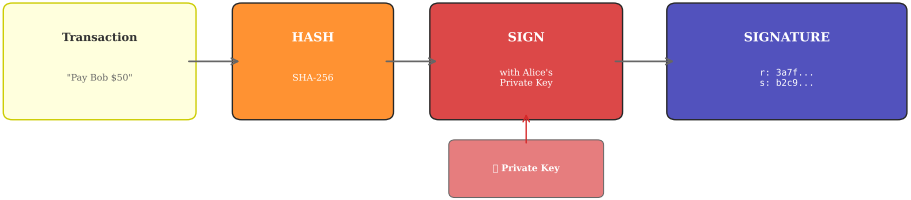
**In finance, digital signatures replace:**

- Handwritten signatures on contracts
- Physical authorization stamps at banks
- In-person identity verification for transactions

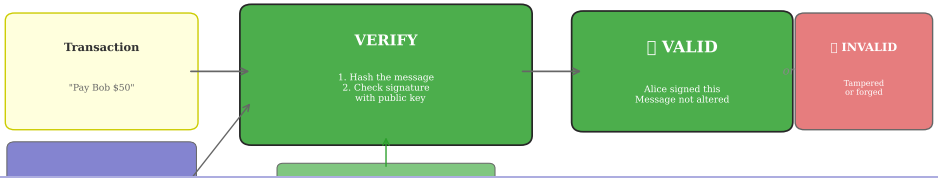
**Digital signatures provide cryptographic proof of authorship and integrity — stronger than any handwritten signature.**

# Digital Signature: Sign and Verify

**Step 1: Alice SIGNS the Transaction**




**Step 2: Anyone VERIFIES Using Alice's Public Key**



# What Is ECDSA?

## Definition: ECDSA

The **Elliptic Curve Digital Signature Algorithm (ECDSA)** is a digital signature scheme based on the mathematics of elliptic curves. It provides the same security as older algorithms (like RSA) but with **much smaller key sizes**, making it efficient for resource-constrained environments.

**Comparison of key sizes for equivalent security:**

| Algorithm | Key Size | Security Level |
|---|---|---|
| RSA | 3,072 bits | 128-bit |
| ECDSA | 256 bits | 128-bit |

**Where ECDSA is used:**

- **Bitcoin and Ethereum:** Every transaction is signed with ECDSA (secp256k1, the specific elliptic curve used by Bitcoin)
- **TLS certificates:** Securing HTTPS connections for online banking
- **Mobile payments:** Small key sizes fit resource-limited devices

**ECDSA achieves strong security with small keys — ideal for blockchain transactions and mobile devices.**

# Key Pairs in Practice: Your Blockchain Identity

**In blockchain, your key pair is your identity:**

| | **Private Key** | **Public Key / Address** |
|---|---|---|
| **What is it?** | A random 256-bit number | Derived from the private key via elliptic curve math |
| **Who knows it?** | Only you | Everyone |
| **Purpose** | Sign transactions | Receive funds, verify signatures |
| **If lost?** | Funds are **permanently inaccessible** | Can be regenerated from private key |
| **If stolen?** | Attacker controls all your funds | No risk |

**Critical:** There is no "forgot my password" for private keys. There is no bank to call. This is the trade-off of decentralization: **full control = full responsibility**.

**In decentralized finance, your private key is your identity, your password, and your bank vault key — all in one.**

# What Is the Double-Spend Problem?

## Definition: Double-Spend Problem

The **double-spend problem** is the risk that a unit of digital currency is spent *more than once*. Because digital data can be copied, a malicious user could send the same digital coin to two different people simultaneously.

**Why physical cash does not have this problem:**

- If you hand a $10 bill to Alice, you no longer have it
- Physical transfer is **inherently exclusive** — the bill cannot be in two places

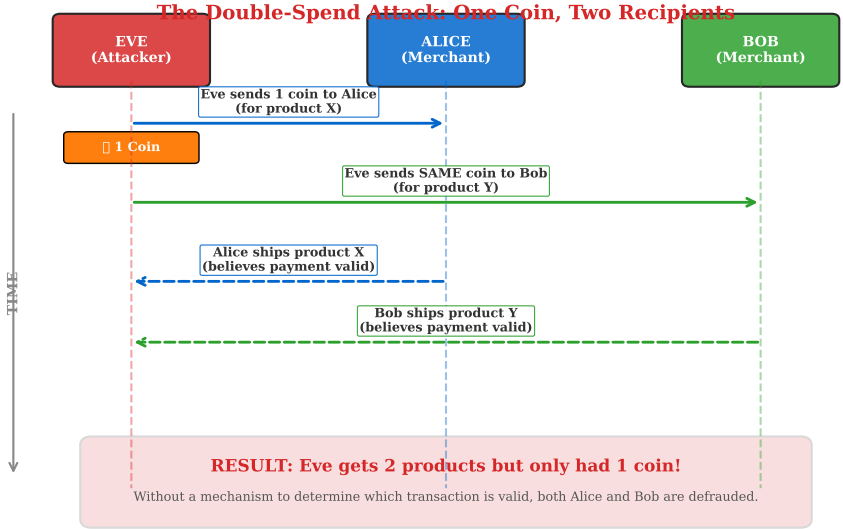
**Why digital cash has this problem:**

- A digital file ("coin") is just data — it can be copied perfectly
- Without a mechanism to prevent it, you could send the same coin to Alice and Bob simultaneously
- Both recipients would believe they received a valid payment

---

**The double-spend problem is the fundamental challenge that separates digital money from physical money.**

# Double-Spend Attack: How It Works



# Traditional Solution: The Trusted Third Party

**Banks solve double-spending with a central ledger:**

1. Alice asks the bank to transfer $10 to Bob
2. The bank checks Alice's account balance: does she have $10?
3. If yes: debit Alice, credit Bob, update the ledger
4. If no: reject the transaction
5. The bank's ledger is the **single source of truth**

**This works, but requires:**

- **Trust:** You must trust the bank to maintain the ledger honestly
- **Availability:** The bank must be online 24/7
- **Access:** You need a bank account (excludes the unbanked)
- **Fees:** The bank charges for this service (Module 1 revisited)
- **Censorship risk:** The bank can freeze your account

**The question:** Can we prevent double-spending **without** a trusted third party?

**Banks are trusted third parties that prevent double-spending — but at the cost of centralization, fees, and access barriers.**

# Why Cryptography Alone Cannot Solve Double-Spending

**We now have powerful tools:**

- **Hash functions** — detect any tampering
- **Merkle trees** — efficiently verify data inclusion
- **Digital signatures** — prove who authorized a transaction

**But none of these solve the ordering problem:**

- Alice signs a transaction sending 1 coin to Bob — valid signature
- Alice signs a transaction sending the *same* coin to Charlie — also a valid signature
- Both signatures are mathematically correct
- **Who gets the coin?** Cryptography alone cannot answer this

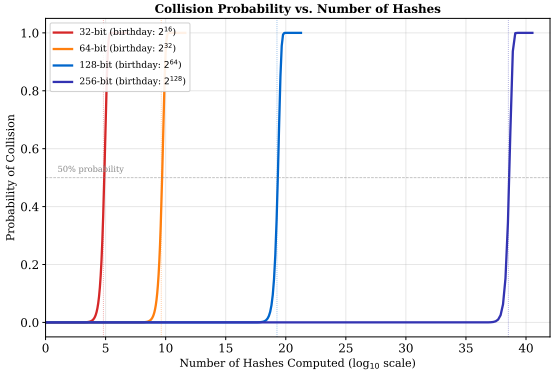
**What is missing:**

- A way to establish a **global ordering** of transactions
- A way to **agree** on which transaction came first
- This is the **consensus problem** — the topic of the next lesson

---

**Cryptography provides integrity and authentication, but solving double-spending also requires consensus on transaction ordering.**

# How Secure Are Hash Functions? The Birthday Paradox




**Birthday Attack Comparison**

| Hash Size | Birthday Bound | Time at 10^18/s | Status |
|---|---|---|---|
| 32-bit | 2^16 (65,536) | < 1 ms | BROKEN |
| 64-bit | 2^32 (4.3 billion) | < 1 sec | BROKEN |
| 128-bit | 2^64 (1.8 x 10^19) | ~18 sec | Weak |
| 256-bit (SHA-256) | 2^128 (3.4 x 10^38) | ~10^13 years | SECURE |

**At 10^18 hashes/sec, breaking SHA-256 would take ~1000x the age of the universe**

- **What you see:** Log-scale graph showing collision probability rising with number of hash attempts, with SHA-256 threshold marked
- **Key pattern:** Due to birthday paradox, 50% collision probability at $2^{128}$ hashes (square root of output space)
- **Takeaway:** Even at $10^{18}$ hashes/sec, finding a collision would take $10^{13}$ years (1000x age of universe)

**Birthday paradox:** In a room of 23 people, there is a >50% chance two share a birthday.
**For SHA-256:** You need approximately $2^{128}$ ($3.4 \times 10^{38}$) hashes before a 50% collision probability.

# Putting It All Together: The Cryptographic Toolkit

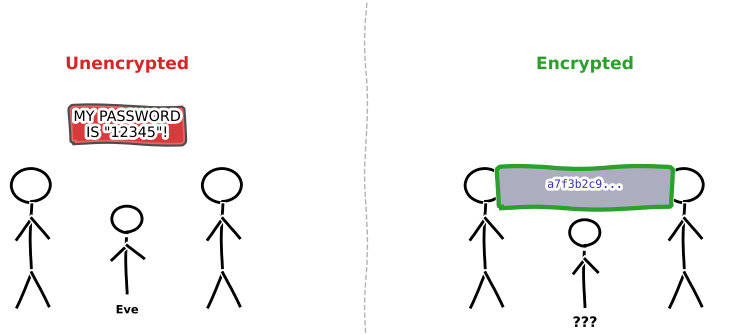
| Tool | What It Does | Financial Application |
|---|---|---|
| **Hash function** | Creates a unique fingerprint of any data | Tamper detection, blockchain linking |
| **Merkle tree** | Organizes hashes into a tree for efficient proof | Lightweight transaction verification |
| **Public key** | Receives encrypted messages; verifies signatures | Blockchain address (receive funds) |
| **Private key** | Decrypts; creates digital signatures | Authorize transactions |
| **Digital signature** | Proves authorship and integrity | Sign every blockchain transaction |

**All of these tools together still cannot prevent double-spending.**

The missing ingredient — **consensus** — is the subject of Lesson 3.2.

**Hash functions, Merkle trees, and digital signatures are necessary but not sufficient for trustless digital money.**

# One Last Thought...




**Cryptography: Making secrets boring to everyone except the intended recipient.**

**Sometimes the best way to remember a concept is to laugh about it.**

# Key Takeaways

1. **Hash functions** produce fixed-size fingerprints with three key properties: pre-image resistance, collision resistance, and the avalanche effect
2. **SHA-256** is the dominant hash function in blockchain finance, producing a 256-bit output
3. **Merkle trees** organize hashes into a tree, enabling verification of one transaction out of thousands with only $\log_2(n)$ hashes
4. **Public-key cryptography** uses key pairs (public + private) to encrypt, decrypt, and sign without sharing secrets
5. **ECDSA** provides strong digital signatures with small key sizes, used in Bitcoin and Ethereum
6. The **double-spend problem** is the core challenge of digital money — you can copy a file, but you cannot copy a coin
7. Cryptography alone provides integrity and authentication, but **consensus mechanisms** (next lesson) are needed to prevent double-spending

**These cryptographic building blocks are the foundation for every topic in Module 3.**

# Summary / Next Lesson Preview

**This lesson:** We built the cryptographic toolkit — hash functions, Merkle trees, public-key cryptography, and digital signatures — and identified the double-spend problem as the barrier to trustless digital money.

**Key vocabulary:**

- Cryptographic hash function
- SHA-256
- Pre-image resistance
- Collision resistance
- Avalanche effect
- Merkle tree / Merkle root
- Inclusion proof
- Public-key cryptography
- Private key / Public key
- Digital signature
- ECDSA
- Double-spend problem

**Next lesson (M3L2):** *Consensus Mechanisms* — How do thousands of strangers agree on a single transaction history without trusting each other? We explore Proof of Work, Proof of Stake, and the Byzantine Generals Problem.

---

**Review: Can you explain all three properties of hash functions and why digital signatures alone do not prevent double-spending?**