THE GRAINGER COLLEGE OF ENGINEERING

CS 521

Technological Foundations of Blockchain and Cryptocurrency

Grigore Rosu

Topic 2 – Basic Crypto Primitives

Thanks!

To Professors

David Tse (Stanford)

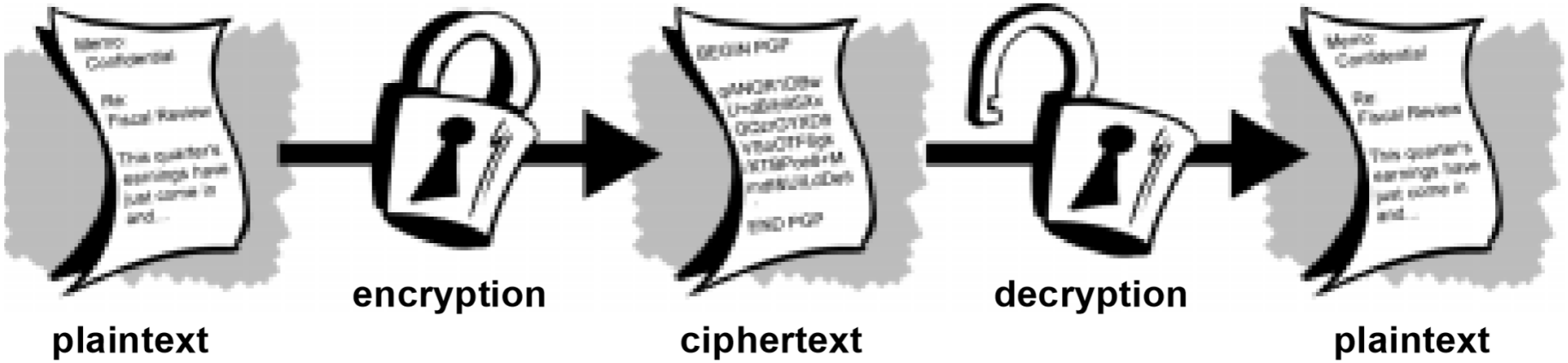
Sriram Viswanath (UT Austin)

Sreeram Kannan (UW – now at EigenLayer)

Some crypto primitives

- Encryption and Signatures
- Cryptographic Hash Functions
- Hash Accumulators
  - Blockchain
  - Merkle trees

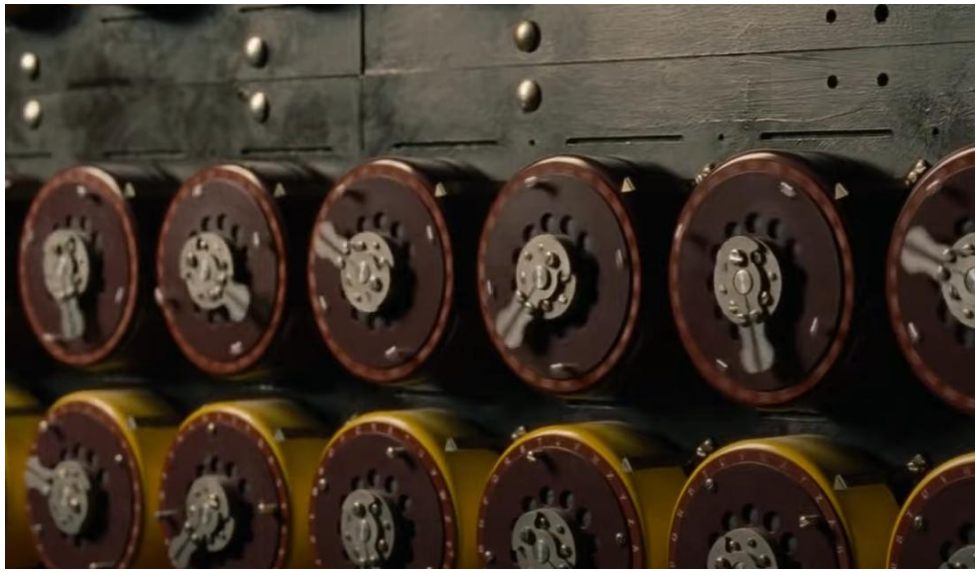
Basic Encryption



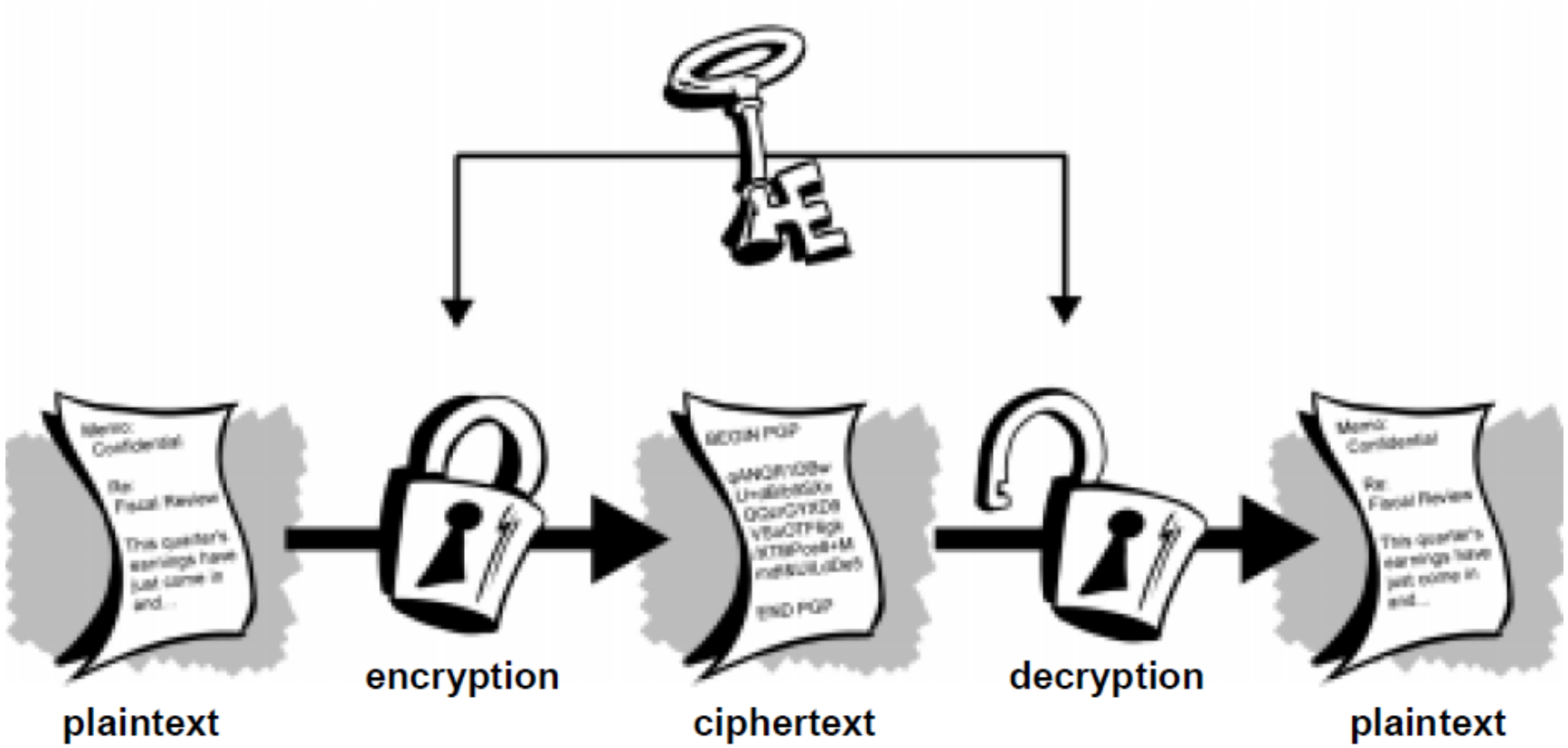
Cypher: Offset the Alphabet
Key: 4

Scene from “Breaking the Enigma Code”

https://youtu.be/zZuqLLdx2YQ



Symmetric (aka Secret-Key) Cryptography



How Secret-Key Cryptography Works

- **Single Shared Key** – Both sender and receiver use the same secret key for encryption and decryption
- **Key Distribution** – The shared key must be securely exchanged between parties before communication
- **Fast Performance** – Symmetric algorithms are computationally efficient, ideal for encrypting large amounts of data
- **Common Algorithms** – Examples include AES (Advanced Encryption Standard), DES, and 3DES

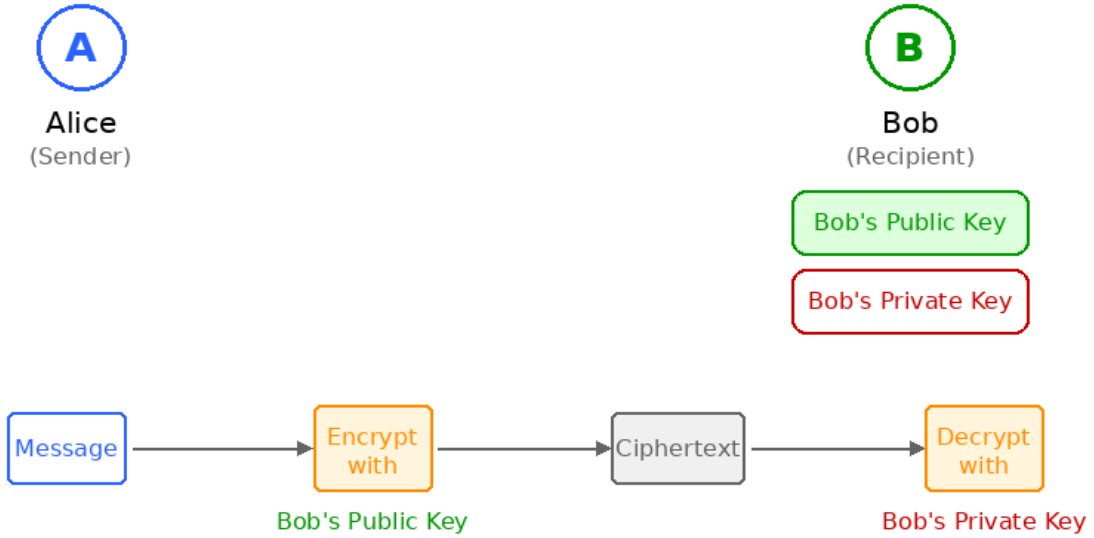
Pros and Cons of Secret-Key Cryptography

- ✓ High performing – fast, especially if the data is not going to be transmitted
- ✓ Can be implemented in hardware and software
- ✗ Secure key distribution is difficult, requires trust and secrecy between the parties as well as trust for the “distribution mechanism” if the parties are not in the same location

Asymmetric (aka Public-Key) Cryptography

CONFIDENTIALITY

(Encrypt with recipient's PUBLIC key)

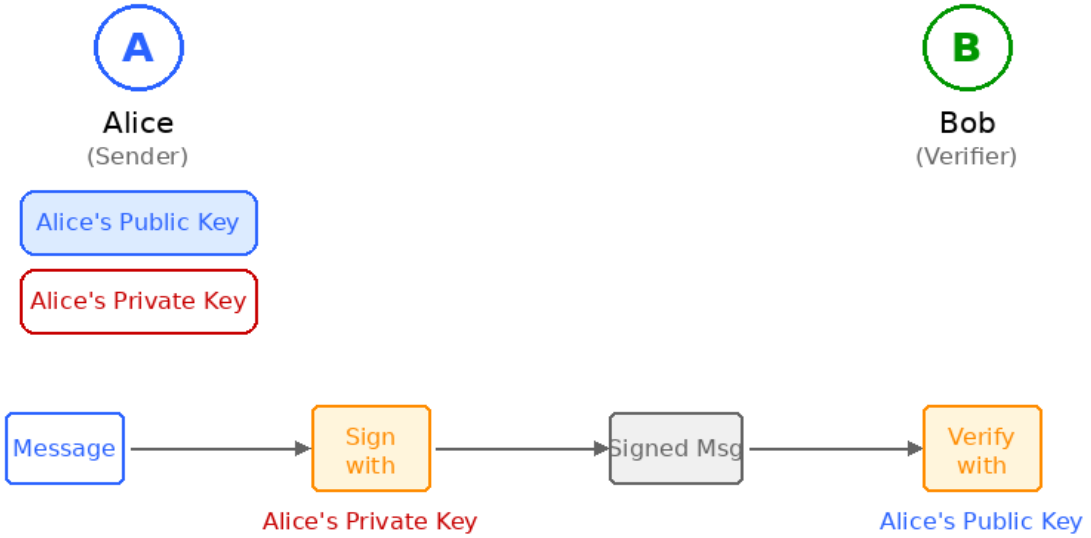


✓ Only Bob can read the message

Goal: Keep message secret

AUTHENTICATION

(Sign with sender's PRIVATE key)



✓ Anyone can verify it came from Alice

Goal: Prove sender identity

How Public-Key Cryptography Works

- **Key Pair** – Each party has a public key (shared openly) and a private key (kept secret)
- **No Key Exchange Required** – Public keys can be freely distributed; only the private key must remain secret
- **Asymmetric Encryption** – Data encrypted with one key can only be decrypted with the other key in the pair
  - **Confidentiality** – Encrypt with recipient's public key; only they can decrypt with their private key
  - **Authentication** – Encrypt with your private key; anyone can verify it came from you using your public key
- **Common Algorithms** – Examples include RSA, Elliptic Curve Cryptography (ECC), and Diffie-Hellman

RSA Cryptosystem

1. Key Generation

Choose large primes p and q
typically 1024+ bits each

Compute $n = p \times q$
n is public, 2048+ bits

Compute $\varphi(n) = (p-1)(q-1)$
Euler's totient = count of integers < n coprime to n

Choose e with $1 < e < \varphi(n)$, $\gcd(e, \varphi(n)) = 1$
Common e = 65537 (Fermat prime $2^{16}+1$)

Compute $d = e^{-1} \bmod \varphi(n)$
Extended Euclidean Algorithm: since $\gcd(e, \varphi(n)) = 1$,
Bézout's identity guarantees $\exists\, d, k$: $e \cdot d + \varphi(n) \cdot k = 1$
Repeat division: $r_0 = \varphi(n)$, $r_1 = e$, $r_{i+1} = r_{i-1} \bmod r_i$ until $r_k = 0$
Back-substitute to find d from the quotients
O(log n) steps — very fast even for 2048-bit numbers

Public Key
(n, e)
share openly

Private Key
(n, d)
keep secret

2. Confidentiality

Encrypt Message m → $c = m^e \bmod n$ → Ciphertext c
public key

Decrypt Ciphertext c → $m = c^d \bmod n$ → Message m
private key

3. Authentication

Sign Message m → $s = m^d \bmod n$ → Signature s
private key

Verify Signature s → $m = s^e \bmod n$ → Message m ✓
public key

Why RSA Works

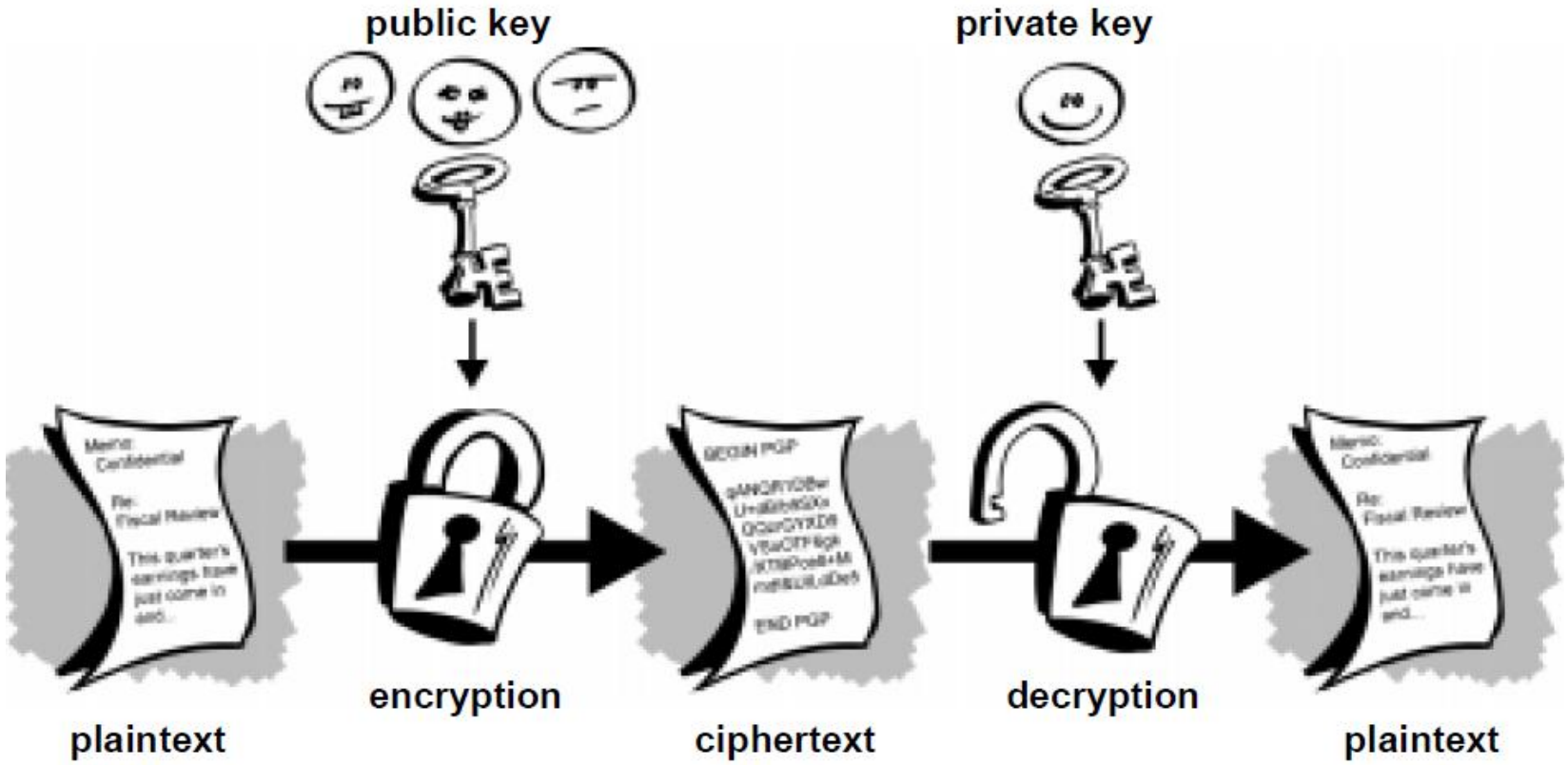
Security: Factoring $n = p \times q$ is computationally infeasible for large primes

Euler's Theorem: If $\gcd(m, n) = 1$, then $m^{\varphi(n)} \equiv 1 \pmod n$

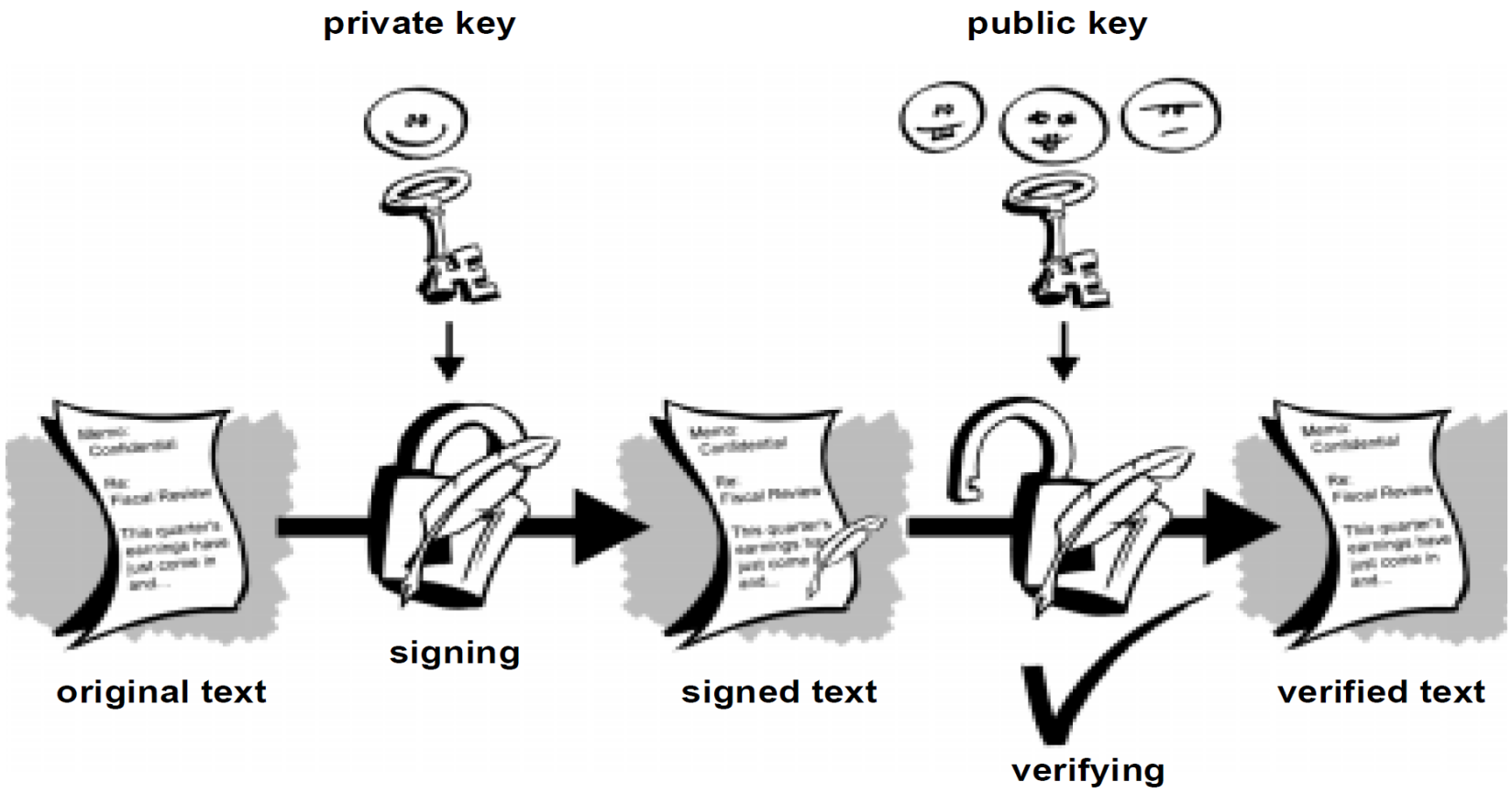
Confidentiality: $e \cdot d \equiv 1 \pmod{\varphi(n)} \Longrightarrow c^d = (m^e)^d = m^{(e \cdot d)} = m \cdot (m^{\varphi(n)})^k \equiv m \pmod n$

Authenticity: $s^e = (m^d)^e = m^{(d \cdot e)} \equiv m \pmod n$; only private key holder can produce valid s

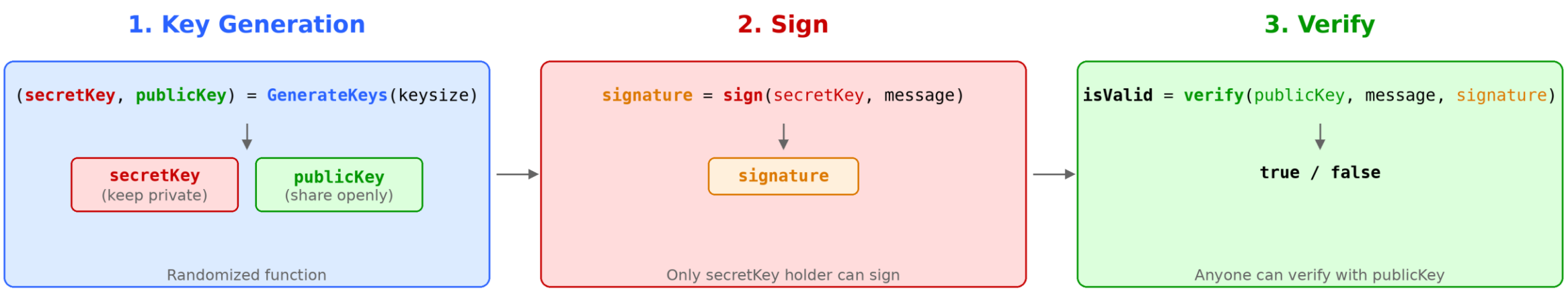
Confidentiality – Encrypt with Recipient's Public Key



Authentication – Digital Signatures



Digital Signatures API



Key Properties

Security Guarantees:

- Authenticity: Only the secret key holder can create valid signatures
- Integrity: Any modification to the message invalidates the signature
- Non-repudiation: Signer cannot deny having signed the message

Common Algorithms:

- RSA-PSS: RSA with probabilistic padding
- ECDSA: Elliptic Curve Digital Signature Algorithm
- EdDSA: Edwards-curve DSA (Ed25519)

Unforgeable Signatures

- **Existential Unforgeability** – Computationally infeasible to forge a valid signature on any message without the secret key
- **Adaptive Chosen Message Attack** – Secure even if adversary can request signatures on chosen messages before attempting forgery
- **Computational Hardness** – Based on hard math problems: discrete log (ECDSA), integer factorization (RSA)
- **Common Algorithms**
  - **ECDSA** – Elliptic Curve Digital Signature Algorithm; used in Bitcoin and Ethereum
  - **EdDSA** – Edwards-curve DSA (Ed25519); faster and used in newer protocols
  - **Schnorr** – Provably secure with signature aggregation; adopted by Bitcoin (Taproot)

Pros and Cons of Public-Key Cryptography

- ✓ **No Pre-Shared Secret** – Communicate securely without prior key exchange arrangement
- ✓ **Scalable Key Management** – N users need only N key pairs (vs N^2 keys for symmetric)
- ✓ **Digital Signatures** – Enables authentication, integrity, and non-repudiation
- ✓ **Decentralized Identity** – Public keys can serve as pseudonymous identities (addresses)
- ✗ **Computationally Expensive** – 100-1000x slower than symmetric encryption
- ✗ **Key-Identity Binding** – Public key alone doesn't prove real-world identity of holder
- ✗ **Secret Key Protection** – Loss or theft of private key compromises all security
- ✗ **Quantum Vulnerability** – RSA/ECDSA broken by quantum computers (Shor's algorithm)

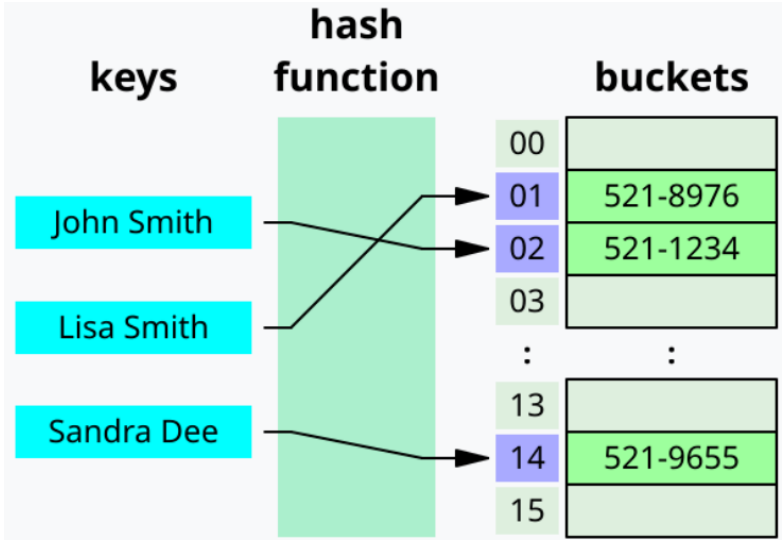
Decentralized Identity Management

- **Public Key as Identity** – Your public key *is* your identity; called an *address* in blockchain terminology
- **Multiple Identities** – Generate as many (`publicKey`, `secretKey`) pairs as you want
  - Publish `publicKey` as your address; sign transactions with `secretKey`
- **Self-Issued** – No central authority needed; anyone can create an identity at any time
- **Verifiable** – Others can verify messages came from you using your public key
- **Pseudonymous** – Identity not linked to real-world name; privacy by default

(Digital identity use case by Erdion Hoxha, project in CS521 in 2025)

Hash Functions

A function that maps data of **arbitrary size** to a **fixed-size** output: H(x) → y



- **Defining Properties**
  - **Arbitrary sized inputs** – Accepts data of any length
  - **Fixed size deterministic output** – Same input always yields same hash
  - **Efficiently computable** – Fast to compute for any given input
  - **Minimize collisions** – Different inputs should produce different outputs
- **Canonical Application – Hash Tables**
  - Map keys to buckets; store and retrieve data records efficiently

Example: Division Hashing

$$y = x \bmod 2^{256}$$

- **Satisfies Basic Properties**
  - **Uniform output** – Output evenly distributed across 2^{256} possible values
  - **Simple deterministic function** – Same input always gives same output
  - **Collision resistant** – Hard to find two inputs that hash to the same output
- **Not Sufficient for Cryptography**
  - **Easily reversible** – Given y, trivial to find an x such that H(x) = y
  - **Collisions easy to construct** – An adversary can deliberately craft collisions
  - Cryptography needs ***extra properties*** beyond what hash tables require →

Cryptographic Hash Functions

Extra Security Properties

- **Adversarial collision resistance**

  Infeasible to find any two inputs $x \neq x'$ where $H(x) = H(x')$, even for a determined attacker (birthday paradox: $\sim 2^{n/2}$ attempts for n-bit hash)

- **One-way function (preimage resistance)**

  Given y, infeasible to find any x such that $H(x) = y$

- **Specialized one-way (2nd preimage resistance)**

  Given x, infeasible to find $x' \neq x$ such that $H(x') = H(x)$

- **Avalanche effect (puzzle friendliness)**

  Tiny change in input → completely different output; no shortcut to find inputs that produce a target output pattern

Canonical Applications

- **Message digest**

  Compact fingerprint to verify data integrity

- **Commitments**

  Commit to a value without revealing it; reveal later to prove no change

- **Puzzle generation**

  Create problems that are hard to solve but easy to verify

- **Mining process**

  Find nonce such that H(block + nonce) < target – basis of proof-of-work in Bitcoin

Hashing Algorithms

SHA-256	SHA-2 family | 256-bit output | NSA 2001 | Used in Bitcoin, TLS, SSH	✓ **No Collisions (yet)**
SHA-512	SHA-2 family | 512-bit output | NSA 2001 | Higher security margin	✓ **No Collisions (yet)**
SHA-1	160-bit | NSA 1995 | Derived from MD4 | Predecessor to SHA-2	✗ **Broken (Google, 2017)**
MD5	128-bit | Rivest 1991 | Derived from MD4 | Chained compression	✗ **Collisions found! (2004)**

MD4 (1990) → MD5 (1991) & SHA-1 (1995) → **SHA-2 (2001)** → SHA-3/Keccak (2015, backup not replacement)

Why This Matters for Blockchain

- **Bitcoin uses SHA-256** – Proof-of-work mining, transaction IDs, Merkle trees, address generation
- **Ethereum uses Keccak-256** – A SHA-3 variant for addresses and state storage
- **Scale** – Bitcoin network computes ~600 *quintillion* SHA-256 hashes per second (6×10^{20} H/s) – and still no collision found
- **If SHA-256 broke** – An attacker could forge transactions, rewrite block history, or steal funds

Putting It Together: Hash-then-Sign

Digital signatures combine hashing and asymmetric cryptography into one workflow:

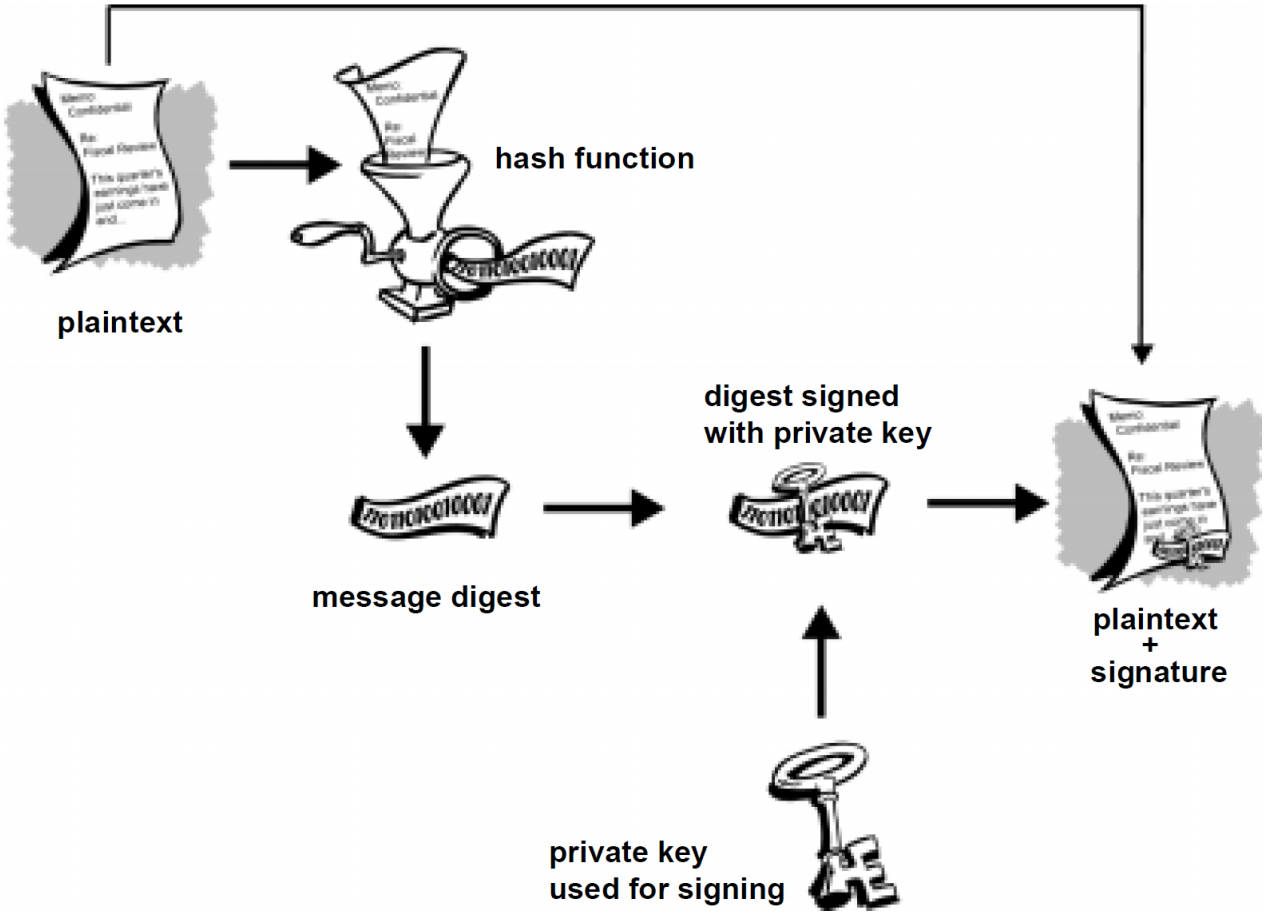
- **Step 1 – Hash the plaintext**
  - Compute a fixed-size message digest of the document
- **Step 2 – Sign the digest**
  - Encrypt the digest with the sender's private key
- **Step 3 – Send plaintext + signature**
  - Receiver can verify by hashing the plaintext and comparing against the decrypted signature

Why hash first?

- Signing is slow on large data – hashing reduces it to a small fixed-size digest first

Blockchain connection

- Every Bitcoin transaction is hash-then-signed – this proves ownership and authorizes transfers on-chain



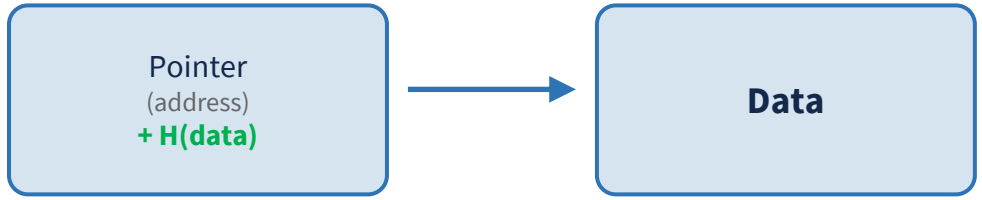
Hash Pointers

A hash pointer stores the **location of information** plus a **hash of that information**, enabling tamper detection.

Regular Pointer

Retrieve information – used to build linked lists, binary trees, etc.

Hash Pointer



Retrieve information **and verify it has not changed**

If data changes → H(data) ≠ stored hash → tamper detected!

Data Structures with Hash Pointers

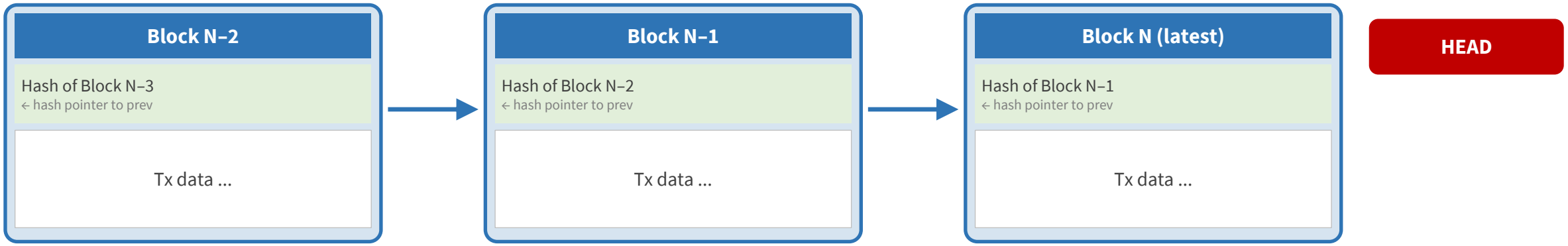
- Regular pointers build data structures (linked lists, binary trees, etc.)
- Hash pointers can also build data structures – but with tamper detection built in

Crucially Useful for Blockchains

- Blockchain = hash-pointer-based data structure
- Two key applications:
  - **Linked list of blocks** – the chain itself
  - **Merkle trees** – efficient data verification inside each block

Blockchain: A Linked List with Hash Pointers

Each block has a **header** (hash pointer to previous block) + **data** (e.g., transactions).

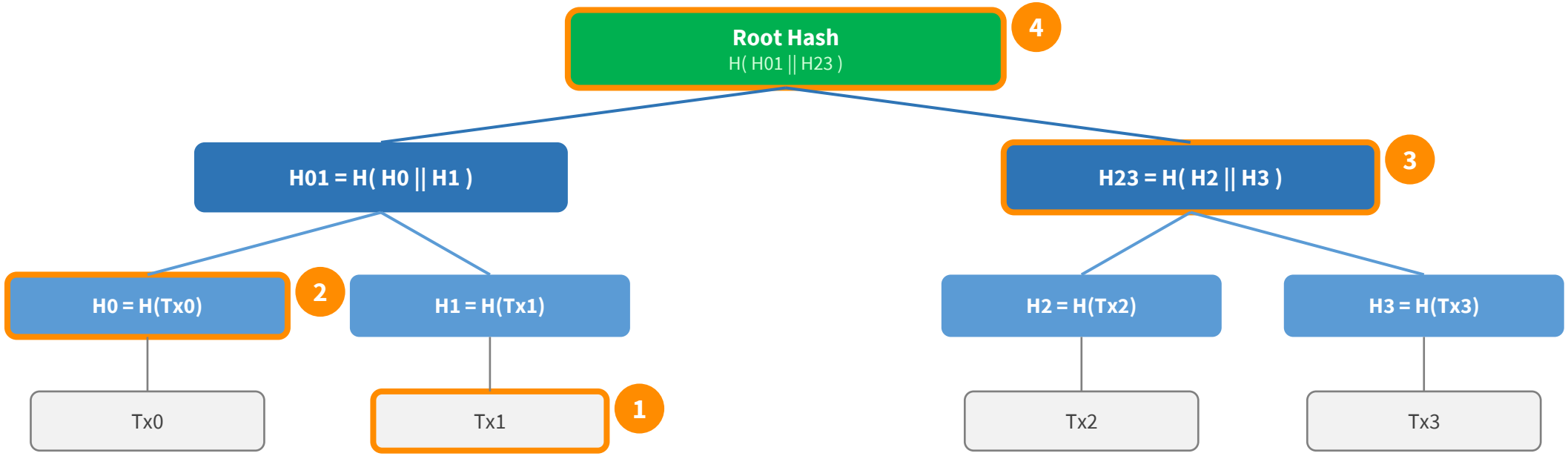


Tamper-Evident Log

- If an attacker modifies data in Block N–2, its hash changes
- Block N–1's stored hash no longer matches → tampering detected
- This cascades: changing *any* block invalidates every block after it
- **Knowing only the head is enough to detect tampering anywhere in the chain**

Hence the name: **block chain → blockchain**

Merkle Trees: Binary Trees of Hash Pointers



- Binary tree of hash pointers – only need to **retain the root hash** to detect tampering in any leaf
- **Proof of Membership**: to verify Tx1 is in the tree, provide H0, H23, and root – verifier recomputes upward in O(log n)
- **Proof of Non-membership**: with a sorted Merkle tree, show that a value falls between two consecutive leaves – also O(log n)
- **Blockchain use**: each block's data section organizes transactions into a Merkle tree – the root hash goes in the block header

Merkle Trees Inside Blocks

Block = Header + Data

- **Header** contains:
  - Hash pointer to the previous block
  - **Merkle root hash** of all transactions
  - Nonce, timestamp, difficulty
- **Data** = block-specific information (transactions organized as a Merkle tree)

Why Merkle trees in blocks?

- A lightweight node can verify any single transaction belongs to a block by downloading only O(log n) hashes instead of the entire block
- Bitcoin blocks contain ~2,000 transactions – proof of membership requires only ~11 hashes
- This enables **SPV (Simplified Payment Verification)** – mobile wallets rely on this