

## Proof of Work Example

### Overview

Main purpose in this notebook:

- Build a minimal PoW blockchain implementation.
- Mine and verify blocks under a difficulty rule.
- Show how hash linkage gives tamper evidence at chain level.

Logic of the notebook:

1. Define hash and difficulty primitives.
2. Build one block and verify validity conditions.
3. Tamper with contents to show why verification fails.
4. Link multiple blocks to show how local validity becomes chain integrity.

### Core Building Blocks

Core condition:

$$\text{hash}(\text{header}) < \text{target},$$

or equivalently in this toy version: hash starts with a required number of leading zeros.

Interpretation:

- Mining = costly search for a nonce that satisfies the rule.
- Verification = cheap recomputation of hash and rule checks.

```

import hashlib
import time
import pandas as pd

def sha256_hex(s: str) -> str:
    return hashlib.sha256(s.encode("utf-8")).hexdigest()

def header_hash(header: dict) -> str:
    serialized = (
        f"{header['prev_hash']}|{header['merkle_root']}|{header['timestamp']}|{header['nonce']}
    )
    return sha256_hex(serialized)

def meets_difficulty(hash_hex: str, difficulty: int) -> bool:
    return hash_hex.startswith("0" * difficulty)

```

## Mine and Verify a Block

```

def mine_block(prev_hash: str, tx_summary: str, difficulty: int, max_tries: int = 5_000_000)
    body = tx_summary
    merkle_root = sha256_hex(body)
    timestamp = int(time.time())
    for nonce in range(max_tries):
        header = {
            "prev_hash": prev_hash,
            "merkle_root": merkle_root,
            "timestamp": timestamp,
            "nonce": nonce,
            "difficulty": difficulty,
        }
        if meets_difficulty(header_hash(header), difficulty):
            return {"header": header, "body": body}
    raise RuntimeError("Increase max_tries or lower difficulty.")

```

```

def verify_block(block: dict, required_difficulty: int) -> bool:
    header = block["header"]
    body = block["body"]
    recomputed_hash = header_hash(header)
    body_matches_root = sha256_hex(body) == header["merkle_root"]
    difficulty_matches = header["difficulty"] == required_difficulty
    return difficulty_matches and meets_difficulty(recomputed_hash, required_difficulty) and

required_difficulty = 4
block1 = mine_block("0" * 64, "Alice pays Bob 1 coin", required_difficulty)

```

```

{'hash': '000042f8d803e496c1c2310c994978b7171f81c81f96eeb7491e78ebbe68de0c',
 'valid': True,
 'nonce': 2841}

```

Tamper test:

```

{'tampered_valid': False}

```

Key result:

- A body change without re-mining fails verification.
- Verification fails because the body no longer matches the stored Merkle root in the header.

## Build and Verify a Chain

Single-block validity is not enough for ledger security. The key property is **linked validity over many blocks**:

- each block must be valid on its own,
- and each block must reference the exact hash of its predecessor.

As chain length grows, rewriting old history requires re-mining a longer suffix.

```
# Build a longer chain to make linkage effects clearer.
tx_stream = [
    "Alice pays Bob 1.0 coin",
    "Bob pays Carol 0.2 coins",
    "Carol pays Dave 0.1 coins",
    "Dave pays Erin 0.05 coins",
    "Erin pays Frank 0.03 coins",
    "Frank pays Grace 0.02 coins",
    "Grace pays Heidi 0.01 coins",
    "Heidi pays Ivan 0.01 coins",
]

chain = []
prev = block0
for tx in tx_stream:
    b = link_block(prev, tx, required_difficulty)
    chain.append(b)
    prev = b

{
    "chain_length": len(chain),
    "chain_valid": verify_chain(chain, required_difficulty, first_prev_hash=header_hash(block0))
}
```

```
{'chain_length': 8, 'chain_valid': True}
```

Chain-verification logic:

- Check each block is individually valid (difficulty + body consistency).
- Check each block points to the exact hash of its predecessor.
- So any change in an earlier block propagates forward as broken links.

Security implication:

- In a longer chain, tampering with block  $k$  requires re-mining blocks  $k, k + 1, \dots, T$ .
- That cumulative work requirement is what gives confirmation depth economic meaning.

## Visualize Chain Linkage

```
rows = []
for i, b in enumerate(chain, start=1):
    linked_ok = True if i == 1 else (b["header"]["prev_hash"] == header_hash(chain[i-2]["header"]))
    rows.append({
        "block": i,
        "nonce": b["header"]["nonce"],
        "hash_prefix": header_hash(b["header"])[:12],
        "prev_hash_prefix": b["header"]["prev_hash"][:12],
        "linked_ok": linked_ok,
    })
pd.DataFrame(rows)
```

	block	nonce	hash_prefix	prev_hash_prefix	linked_ok
0	1	6786	000079661ca7	52cae89ca662	True
1	2	130165	00001d55a7e3	000079661ca7	True
2	3	137529	000068679128	00001d55a7e3	True
3	4	1038	00001f7c5d37	000068679128	True
4	5	128125	0000ea957c20	00001f7c5d37	True
5	6	65967	0000a15de3ab	0000ea957c20	True
6	7	191138	00004600a357	0000a15de3ab	True
7	8	169413	0000f51b7736	00004600a357	True

### Interpretation:

- `linked_ok=True` across all rows means hash pointers are internally consistent.
- Nonces vary because each block solves a fresh PoW search problem.
- Even in this toy model, deeper chains make retroactive editing more expensive.

## Takeaways

- PoW security comes from costly search plus cheap verification.
- Hash linkage creates chain-level tamper evidence.
- If an old block changes, all later links break unless the attacker re-mines the suffix.
- This notebook is a mechanics demonstration, not a full protocol implementation or economic equilibrium model.