Hash Functions and Data Integrity

MAT364 - Cryptography Course

Instructor: Adil Akhmetov

University: SDU

Week 6

Press Space for next page →

What Are Cryptographic Hash Functions?

Definition

Hash function maps input of arbitrary length to a fixed-length output (digest).

Cryptographic vs Non-cryptographic

- Cryptographic: SHA-256, SHA-3, BLAKE2/BLAKE3
- Non-cryptographic: CRC32, Adler32 (checksums only)

Core Properties

- **Preimage resistance**: Given h, hard to find m such that H(m) = h
- **Second-preimage resistance**: For given m, hard to find m′ ≠ m with H(m′) = H(m)
- **Collision resistance**: Hard to find any m, m′ with H(m) = H(m′)
- **Avalanche effect**: Small input change → large output change

Use Cases

- Data integrity and file verification
- Digital signatures and certificates
- Password storage and KDFs
- Blockchain and Merkle trees

Key idea: Hashes ensure integrity; paired with keys (HMAC) they ensure authenticity.

Hash Families and Constructions

Common Algorithms

- **MD5** (128-bit) – broken (collisions practical)
- **SHA-1** (160-bit) – deprecated (chosen-prefix collisions)
- **SHA-2** (224/256/384/512) – widely used, secure
- **SHA-3/Keccak** – sponge construction, secure
- **BLAKE2/BLAKE3** – fast modern alternatives

Security Levels

- n-bit hash ≈ **birthday bound** 2^(n/2) for collisions
- Choose 256-bit hashes for modern security margins

Constructions

- **Merkle–Damgård** (MD5, SHA-1, SHA-2)
  - Iterative compression; length-extension caveat
- **Sponge** (Keccak/SHA-3)
  - Absorb-squeeze model; resistant to length-extension

Length-Extension Attacks

- If API exposes raw H(m) with MD construction, attacker may compute H(m || x)
- Use **HMAC** or SHA-3 to avoid this

Hashing in Practice

Python Examples

```python
import hashlib

def sha256_hex(data: bytes) -> str:
    return hashlib.sha256(data).hexdigest()

def file_sha256(path: str) -> str:
    h = hashlib.sha256()
    with open(path, 'rb') as f:
        for chunk in iter(lambda: f.read(1 << 20), b''):
            h.update(chunk)
    return h.hexdigest()

print(sha256_hex(b"hello"))
```

Message Authentication: HMAC

Why HMAC?

- Hash alone gives integrity, not authenticity
- **HMAC(K, m)** uses a secret key K to prevent forgeries
- Secure even if underlying hash has length-extension

Properties

- Deterministic and fast
- Resistant to length-extension
- Standardized (RFC 2104)

Example (Python)

```python
import hmac, hashlib

def hmac_sha256_hex(key: bytes, message: bytes) → str
    return hmac.new(key, message, hashlib.sha256).hexd

print(hmac_sha256_hex(b'secret', b'hello'))
```

Typical Uses

- API request signing
- Token validation
- Secure cookies and sessions

Best practice: Prefer HMAC over raw hashes for authenticity; include nonces/timestamps to prevent replay.

Password Storage and KDFs

Goals

- Slow down brute-force and credential stuffing
- Use **salt** to prevent rainbow tables
- Optionally use **pepper** stored separately

Options

- PBKDF2 (widely supported)
- bcrypt (adaptive, 60-char hashes)
- scrypt (memory-hard)
- Argon2id (modern recommendation)

PBKDF2 Example (Python)

```
import os, hashlib

def hash_password(password: str, rounds: int = 200_000)
    salt = os.urandom(16)
    key = hashlib.pbkdf2_hmac('sha256', password.encode
    return salt + key

def verify_password(password: str, stored: bytes, round
    salt, key = stored[:16], stored[16:]
    candidate = hashlib.pbkdf2_hmac('sha256', password,
    return hmac.compare_digest(candidate, key)
```

Guidance

- Target ≥100ms per hash on server
- Unique random salt per password
- Use constant-time compare

Data Integrity in Systems

Common Patterns

- File integrity: publish SHA-256 sums
- Package managers: signed manifests (hash + signature)
- Backups: deduplication via chunk hashes
- Databases: row/version checksums

Merkle Trees

- Each leaf is a hash of data block
- Internal nodes hash children
- **Merkle root** authenticates entire dataset
- Used in blockchains and git

Example: Merkle Root

```python
import hashlib

def h(x: bytes) -> bytes:
    return hashlib.sha256(x).digest()

def merkle_root(leaves: list[bytes]) -> bytes:
    level = [h(x) for x in leaves]
    while len(level) > 1:
        if len(level) % 2 == 1:
            level.append(level[-1])  # duplicate last
        level = [h(level[i] + level[i+1]) for i in rang
    return level[0]
```

Applications

- Blockchain transaction inclusion proofs
- Large-file integrity with partial verification

Security and Attacks

Threats

- Collision attacks (birthday bound)
- Length-extension (MD construction)
- Timing side-channels in comparisons
- Poor randomness for salts/keys

Mitigations

- Use modern hashes (SHA-256/3, BLAKE2/3)
- HMAC or SHA-3 to avoid extension
- Constant-time comparisons
- CSPRNG for salts/keys

Birthday Paradox (Quick Math)

Let output size be n bits. Collision work ≈ 2^(n/2).

Examples:

- 128-bit hash → ~2^64 work (insufficient long-term)
- 256-bit hash → ~2^128 work (modern standard)

When to Use What

- Integrity only: SHA-256
- Integrity + authenticity: HMAC-SHA-256
- Passwords: bcrypt/scrypt/Argon2id

🎯 Student Task: Verify Integrity

Task

1. Compute SHA-256 of a file you choose
2. Tamper with one byte and recompute
3. Explain observed avalanche effect
4. Create HMAC over the file using a secret key

Deliverables

- Original and modified hashes
- HMAC value and key length used
- Short paragraph on integrity vs authenticity

✅ Solution Sketch

Expected Outcomes

- Modified file hash is completely different
- HMAC changes when key or data changes
- Explanation distinguishes integrity (hash) vs authenticity (HMAC)

Example Commands (bash)

```
shasum -a 256 myfile.bin
python - <<'PY'
import hmac, hashlib
key=b'secretkey'; data=b'example'
print(hmac.new(key, data, hashlib.sha256).hexdigest())
PY
```

Real-World Applications

Where Hashes Power Systems

- **Git** object IDs (SHA-1/SHA-256)
- **TLS** cert chains and signatures
- **Package registries** (npm, PyPI) integrity
- **Blockchain** linking and Merkle proofs

Best Practices Recap

- Prefer SHA-256/3 or BLAKE2/3
- Use HMAC for MACs; avoid raw-hash MACs
- Never store plain passwords; use KDFs
- Document and publish expected hashes

Questions?

Let's discuss hash functions! 💬

Next Week: We will dive deep into RSA and asymmetric cryptography details.

Assignment: Build a small tool to compute file hashes and HMACs.