Mid Sem NetSec

Show how the DES block cipher can be used to build a 64-bit hash function. Is the result collision resistant?

Building a 64-bit Hash Function Using DES

DES (Data Encryption Standard) is a **64-bit block cipher** with a **64-bit block size** and **56-bit effective key length**. We can construct a hash function from DES using the **Davies–Meyer construction**, which is a common way to build hash functions from block ciphers.

🔹 Construction (Davies–Meyer using DES)

Let:

- Message M be divided into 56-bit blocks:
  $M = M_1, M_2, \dots, M_n$
- H_0 = Initial 64-bit IV (fixed constant)
- $DES_k(x)$ = DES encryption of block x using key k

Since DES key size is 56 bits, each message block is used as the **key**, and the previous hash value is used as the **plaintext**.

For each block:

$$H_i = DES_{M_i}(H_{i-1}) \oplus H_{i-1}$$

Final hash output:

$$H(M) = H_n$$

So the output hash size = **64 bits**

🔹 How It Works (Intuition)

1. Start with an initial value H_0
2. Encrypt it using first message block as DES key
3. XOR the output with previous hash value
4. Repeat for all message blocks
5. Final 64-bit value is the hash

This ensures:

- Compression (fixed 64-bit output)
- Diffusion via DES
- Dependency on all message blocks

🔹 Is It Collision Resistant?

❌ No, it is NOT collision resistant.

Reasons:

1️⃣ Small Hash Size (64 bits)

By the **birthday attack**:

- Collision complexity ≈ $2^{n/2}$
- Here $n = 64$

So collision can be found in:

$$2^{32}$$

This is computationally feasible today.

2️⃣ DES Itself Is Weak

- Effective key length = 56 bits
- Brute force attack complexity = 2^{56}
- Already practically broken

Example:
DES was broken by the **EFF DES Cracker (1998)** in about 56 hours.

3️⃣ Modern Security Standards

Secure hash functions today require:

- At least **128-bit collision resistance**
- Practical minimum output size = **256 bits** (like SHA-256)

Examples:

- MD5 → 128-bit → broken
- SHA-1 → 160-bit → broken
- SHA-256 → secure

🔹 Final Conclusion

We can construct a 64-bit hash using DES via the **Davies–Meyer construction**:

$$H_i = DES_{M_i}(H_{i-1}) \oplus H_{i-1}$$

But:

- Output size = **64 bits**
- Collision attack complexity = **2^{32}**
- DES itself is insecure

➜ Therefore, the resulting hash function is NOT collision resistant and NOT secure for modern use.

If you want, I can also explain how AES would improve this construction and what output size would be required for real-world security.

An automatic teller machine (ATM) communicates with a central bank computer for PIN verification. A 32-bit CRC code is added to each packet to detect transmission errors and then the link is encrypted using a block cipher in counter mode. Describe an attack that is possible in this setup

🔴 Vulnerability in This ATM Setup

The system does:

1. Append **32-bit CRC** to each packet (for error detection)
2. Encrypt the whole packet using a **block cipher in CTR mode**

At first glance this looks secure — but it is **not**.

🔥 The Core Problem

CTR mode provides:

- ✅ Confidentiality
- ❌ No Integrity / No Authentication

CRC provides:

- ❌ Error detection

- ❌ No cryptographic security
- ❌ Not keyed
- ❌ Linear function

So the system has **no real integrity protection**.

⚡ Why This Is Dangerous

CTR mode encryption works as:

$$C = P \oplus K$$

Where:

- P = plaintext
- K = keystream (from counter)
- C = ciphertext

This makes CTR mode:

🔁 Fully malleable

An attacker can flip ciphertext bits and control plaintext changes.

🚨 Attack: Bit-Flipping Attack (Message Modification)

Step 1: Intercept Packet

Suppose the ATM sends:

```ini
PIN = 1234
Amount = 1000
```

Packet format:

```css
[ Data | CRC ]
```

Encrypted using CTR:

$$C = (Data \| CRC) \oplus K$$

Attacker intercepts ciphertext.

Step 2: Modify Ciphertext

Attacker flips specific bits in ciphertext.

Because:

$$P = C \oplus K$$

If attacker changes ciphertext:

$$C' = C \oplus \Delta$$

Then plaintext becomes:

$$P' = P \oplus \Delta$$

So attacker can carefully modify:

```ini
Amount = 1000  →  9000
```

Step 3: Fix the CRC

Here is the critical weakness:

CRC is linear

CRC satisfies:

$$CRC(x \oplus y) = CRC(x) \oplus CRC(y)$$

So attacker can compute:

$$\Delta_{CRC}$$

and adjust ciphertext so that:

```kotlin
Modified data + Correct modified CRC
```

Result:

- CRC check passes
- Bank accepts modified transaction

💣 What Can Be Achieved?

An attacker could:

- Change withdrawal amount
- Modify destination account number
- Corrupt transaction fields
- Replay modified packets

This is a **forgery attack**.

🎯 Why This Works

Because:

Mechanism	Protects
CRC	Random noise errors
CTR mode	Confidentiality only
Neither	Message authenticity

There is **no Message Authentication Code (MAC)**.

🔐 What Should Have Been Used?

Instead of:

```scss
Encrypt( Data || CRC )
```

They should use:

Option 1: Encrypt-then-MAC

```ini
Ciphertext = Encrypt(Data)
Tag = MAC(Ciphertext)
```

Option 2: Use Authenticated Encryption

- AES-GCM
- AES-CCM

These provide:

- Confidentiality
- Integrity
- Authentication

✅ Final Answer (Exam-Style Summary)

The system is vulnerable to a **bit-flipping attack** because CTR mode encryption is malleable and CRC is linear and not cryptographically secure.

An attacker can modify ciphertext bits to alter the decrypted message and recompute the corresponding CRC difference, causing the modified packet to pass CRC verification.

Thus, the scheme does **not provide integrity or authentication**, and forged transactions are possible.

Consider the problem of mitigation of DDoS attack on a web server. We discussed a solution to this problem using DLP in the class. Propose a solution using Integer Factorization problem.

a) How do you generate the challenge for the attacker?

b) How do you generate different challenges for the different attackers?

c) How do you verify the response given by the attacker?

d) Explain the total solution along with its merits and demerits

Mitigating DDoS Using the Integer Factorization Problem

(Computational Puzzle-Based Defense)

The idea is to force every client to solve a **computationally expensive problem** before the server allocates heavy resources. Since integer factorization is hard, bots cannot flood the server cheaply.

This is essentially a **Proof-of-Work (PoW)** mechanism based on the difficulty of factoring large composite numbers.

(a) How do you generate the challenge?

The server creates a challenge using the **Integer Factorization Problem**.

Step 1: Generate two primes

Choose large primes:

$$p, q$$

Step 2: Compute composite number

$$N = p \times q$$

Step 3: Send challenge

The challenge to client is:

```css
Given N, compute its prime factors p and q
```

Since integer factorization is hard (basis of RSA security), solving it requires significant computation.

Optional: Include timestamp + nonce to avoid reuse.

(b) How do you generate different challenges for different attackers?

To prevent precomputation and sharing of solutions:

Method 1: Use unique nonce per client

For each connection:

$$N_i = p_i \times q_i$$

where p_i, q_i are freshly generated primes.

Each client receives a different N_i.

Method 2: Bind challenge to client data

Compute:

$$N_i = H(IP \parallel Timestamp) \oplus BaseNumber$$

Then embed this into a composite number construction.

Now challenge depends on:

- Client IP
- Timestamp
- Server secret

So attackers cannot reuse answers.

(c) How do you verify the response?

Client returns:

```css
(p, q)
```

Server verifies by:

$$p \times q = N$$

Verification is **very fast** compared to factoring.

Complexity:

- Factoring → Hard (sub-exponential)
- Multiplication check → $O(n^2)$, very fast

If correct:

- Server allows connection
- Otherwise drop request

(d) Total Solution Explanation

🔹 Complete Flow

1. Client requests connection
2. Server detects high load
3. Server sends challenge: factor N
4. Client solves factorization
5. Client sends p, q
6. Server verifies
7. If correct → service granted

🔹 Why This Mitigates DDoS

In DDoS:

- Attacker sends millions of requests
- Server resources get exhausted

With factorization puzzle:

- Each request costs attacker heavy CPU
- Server only performs cheap verification

So attack becomes expensive.

This is similar in spirit to **Proof-of-Work systems like** Bitcoin, where computational work prevents spam.

🔹 Merits

✅ 1. Asymmetric cost

- Client work = very high
- Server verification = very low

✅ 2. Reduces bot flooding

Bots must spend real computation.

✅ 3. No heavy cryptographic state

No complex key management required.

✅ 4. Adjustable difficulty

Server can increase bit-length of N during high load.

🔹 Demerits

❌ 1. Legitimate clients also suffer

Mobile devices may struggle.

❌ 2. Powerful botnets still possible

If attacker has massive distributed power, attack still works.

❌ 3. Prime generation cost on server

Server must generate large primes.

❌ 4. Latency increases

Extra round-trip before service.

❌ 5. Not suitable for real-time services

🔹 Security Basis

Security depends on hardness of:

- Integer Factorization Problem

- Same assumption used in RSA

Factoring large (e.g., 1024-bit) composite numbers is computationally hard with classical computers.

✅ Final Summary (Exam-Ready)

A DDoS mitigation scheme can be built using a factorization-based computational puzzle.

The server sends a composite number $N = p \times q$ to the client and asks for its prime factors.

Different challenges are generated per client using fresh primes or client-bound nonces.

Verification is done by checking $p \times q = N$, which is efficient.

The scheme shifts computational burden to the client, limiting large-scale request flooding. However, it increases latency and affects legitimate users.