## 1. Hash Function:

The provided `custom_hash` function implements a simple hash algorithm with the following components:

**Mathematical Principles:**

- **Hashing**: The function computes a hash value based on the input data (`data: bytes`). The input is processed byte by byte to compute the hash.
- **Prime multiplier (31)**: A prime number is chosen as a multiplier. This helps to minimize hash collisions by distributing hash values more evenly across possible results. Using primes in hashing functions is a well-established practice to avoid patterns and make the output harder to predict.
- **Modulus operation (2^32)**: The modulus operation ensures that the resulting hash value is within the limits of a 32-bit integer. This is necessary to prevent overflow and restrict the output size, ensuring it remains within the bounds of the modulus (`2^32`), which gives a range of values from `0` to `2^32 - 1`.

**Accumulation**: The hash is computed by iterating through each byte of the input data, updating the `hash_val` with the formula:
ini

**Code:** `hash_val = (hash_val * prime + byte) % modulus`

- This operation ensures that the final hash value is influenced by each byte in the input, and that small changes in the input data will result in large changes in the hash value.

**Security Guarantees:**

- **Preimage Resistance**: Given a hash value, it is computationally hard to find an input that hashes to this value. While this simple hash function doesn't offer cryptographic-level security, its design aims to produce a unique output for different inputs, making it hard (but not impossible) to reverse-engineer the original data.
- **Collision Resistance**: It's difficult to find two distinct inputs that produce the same hash value, though the function's simple design and modulus operation might allow for some potential collisions due to the limited output space (`2^32`).
- **Uniformity**: The use of a prime multiplier ensures that the hash values are distributed fairly uniformly over the output space.

## 2. Symmetric Encryption (Fernet):

The provided encryption and decryption use the **Fernet** class from the `cryptography` library, which implements symmetric encryption.

**Mathematical Principles:**

- **Symmetric Encryption**: In symmetric encryption, the same key is used for both encryption and decryption. In this case, Fernet uses **AES (Advanced Encryption Standard)** in CBC (Cipher Block Chaining) mode, along with **HMAC (Hash-based Message Authentication Code)** for integrity verification.
- **Key Generation**: `Fernet.generate_key()` creates a secure key for encryption. This key is 32 bytes long, which is used to encrypt and decrypt the data. The key must remain secret; if an attacker gets the key, they can both encrypt and decrypt messages.
- **Encryption**: The message is encrypted using the Fernet object and its `encrypt()` method. The encryption process involves multiple steps, including the generation of an initialization vector (IV) for the AES encryption, and HMAC to verify the integrity of the ciphertext.
- **Decryption**: The encrypted message is decrypted using the `decrypt()` method, and the original message is recovered if the correct key is provided. The decryption process involves reversing the steps of the encryption, including verifying the authenticity of the ciphertext via HMAC.

**Security Guarantees:**

- **Confidentiality**: The encrypted message is secure against unauthorized access because only someone with the correct key can decrypt it. Even if an attacker intercepts the ciphertext, they cannot decrypt it without the key.
- **Integrity**: The use of HMAC ensures the integrity of the encrypted message. If the ciphertext is altered in any way (for example, during transmission), the decryption will fail.
- **Authenticity**: Only the holder of the key can encrypt and decrypt messages. Since the key is randomly generated and securely managed, it ensures that the encrypted message originates from a legitimate source and not an imposter.

## Conclusion:

- **Hash Function**: The custom hash function offers basic security guarantees like preimage resistance and collision resistance. However, it doesn't provide cryptographic strength, and its security is weak compared to modern cryptographic hash functions (like SHA-256).
- **Symmetric Encryption**: The Fernet encryption ensures strong confidentiality, integrity, and authenticity guarantees. The use of AES and HMAC provides robust security for the encrypted messages, making it secure against attacks such as eavesdropping and tampering.