**Final Year B.Tech. (CSE) – VII [ 2024-25]**

**6CS451: Cryptography and Network Security Lab (C&NS Lab)**

**Date: 01/10/2024**

**Assignment 9**

**PRN:** 21510017  **Name:** Onkar Anand Yemul

**9. Calculate the message digest of a text using the SHA-1 algorithm**

**Ans:**

**SHA-1 Algorithm:**

SHA-1 (Secure Hash Algorithm 1) is a cryptographic hash function that produces a 160-bit hash value (20 bytes), often referred to as a **message digest**. It takes an input message of any size and outputs a fixed-size hash, which is commonly represented as a 40-character hexadecimal number. It was developed by the National Security Agency (NSA) and published by NIST in 1993.

**Message Digest of a Text:**

A **message digest** is a fixed-size numerical representation of the contents of a message. For SHA-1, this digest is 160 bits long, and any change in the input message, even a single bit, will result in a drastically different digest (this is known as the **avalanche effect**). The message digest ensures data integrity by allowing anyone to verify that the message hasn't been altered.

**To implement the SHA-1 algorithm without using Python's *hashlib* library, we need to follow the algorithm's steps manually, which involves bitwise operations, padding the input message, and processing it in blocks.**

**Overview of SHA-1 Algorithm:**

1. **Padding the message**: The message is padded so that its length becomes a multiple of 512 bits.
2. **Initialize hash values**: There are five constants (H0, H1, H2, H3, H4) initialized to specific values.
3. **Processing the message in blocks**: The message is processed in chunks of 512 bits, and the hash is updated after each chunk.
4. **Final output**: After all blocks are processed, the hash digest is formed by concatenating the values of H0, H1, H2, H3, and H4.

**Python Code for SHA-1 Implementation without using Python’s built-in hashlib library:**

import struct

# Helper functions for bitwise operations

def left\_rotate(n, b):

    """Left rotate a 32-bit integer n by b bits."""

    return ((n << b) | (n >> (32 - b))) & 0xFFFFFFFF

def sha1\_padding(message):

    """Pad the message to ensure the length is a multiple of 512 bits."""

    original\_byte\_len = len(message)

    original\_bit\_len = original\_byte\_len \* 8

    # Padding with a '1' bit followed by '0's, and add the original message length in bits at the end

    message += b'\x80'  # append the bit '1' (10000000 in binary)

    # Pad with 0s so that the message length is 64 bits short of a multiple of 512

    while (len(message) \* 8) % 512 != 448:

        message += b'\x00'

    # Append the length of the original message in bits (64-bit big-endian integer)

    message += struct.pack('>Q', original\_bit\_len)

    return message

def sha1(message):

    """Calculate the SHA-1 hash of a message."""

    # Initial hash values (first 32 bits of the fractional parts of the square roots of the first 5 primes)

    h0 = 0x67452301

    h1 = 0xEFCDAB89

    h2 = 0x98BADCFE

    h3 = 0x10325476

    h4 = 0xC3D2E1F0

    # Preprocessing: padding the message

    message = sha1\_padding(message)

    # Process the message in successive 512-bit chunks (64 bytes each)

    for i in range(0, len(message), 64):

        chunk = message[i:i + 64]

        # Break chunk into sixteen 32-bit big-endian words

        w = [0] \* 80

        for j in range(16):

            w[j] = struct.unpack('>I', chunk[j\*4:(j\*4)+4])[0]

        # Extend the sixteen 32-bit words into eighty 32-bit words

        for j in range(16, 80):

            w[j] = left\_rotate((w[j-3] ^ w[j-8] ^ w[j-14] ^ w[j-16]), 1)

        # Initialize hash value for this chunk

        a, b, c, d, e = h0, h1, h2, h3, h4

        # Main loop

        for j in range(80):

            if 0 <= j <= 19:

                f = (b & c) | ((~b) & d)

                k = 0x5A827999

            elif 20 <= j <= 39:

                f = b ^ c ^ d

                k = 0x6ED9EBA1

            elif 40 <= j <= 59:

                f = (b & c) | (b & d) | (c & d)

                k = 0x8F1BBCDC

            elif 60 <= j <= 79:

                f = b ^ c ^ d

                k = 0xCA62C1D6

            temp = (left\_rotate(a, 5) + f + e + k + w[j]) & 0xFFFFFFFF

            e = d

            d = c

            c = left\_rotate(b, 30)

            b = a

            a = temp

        # Add this chunk's hash to the result so far

        h0 = (h0 + a) & 0xFFFFFFFF

        h1 = (h1 + b) & 0xFFFFFFFF

        h2 = (h2 + c) & 0xFFFFFFFF

        h3 = (h3 + d) & 0xFFFFFFFF

        h4 = (h4 + e) & 0xFFFFFFFF

    # Produce the final hash value (big-endian)

    return '{:08x}{:08x}{:08x}{:08x}{:08x}'.format(h0, h1, h2, h3, h4)

if \_\_name\_\_ == "\_\_main\_\_":

    # Input text

    text = input("Enter the text to calculate SHA-1 hash: ")

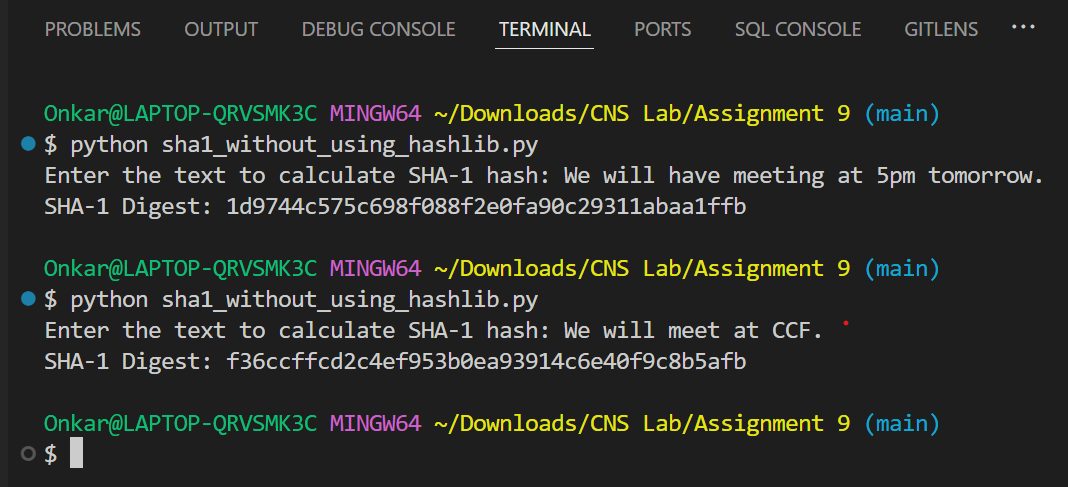
    # Convert the text to bytes and compute SHA-1 hash

    sha1\_digest = sha1(text.encode('utf-8'))

    # Output the SHA-1 hash

    print(f"SHA-1 Digest: {sha1\_digest}")

**Output:**



**To calculate the message digest of a text using the SHA-1 algorithm in Python, you can use the hashlib library, which provides easy access to various hash algorithms, including SHA-1.**

1. **hashlib library**:

* The hashlib library provides various cryptographic hashing algorithms including SHA-1, SHA-256, MD5, etc.

1. **SHA-1 Hash Object**:

* hashlib.sha1() creates a new SHA-1 hash object.

1. **Updating the Hash**:

* The update() method takes the input text (which is first encoded into bytes) and updates the hash object with that data.

1. **Getting the Digest**:

* The hexdigest() method returns the hash value as a hexadecimal string.

**Python Code for SHA-1 Message Digest Calculation using hashlib library:**

import hashlib

def calculate\_sha1(text):

    # Create a new SHA-1 hash object

    sha1\_hash = hashlib.sha1()

    # Encode the input text to bytes and update the hash object

    sha1\_hash.update(text.encode('utf-8'))

    # Get the hexadecimal representation of the digest

    digest = sha1\_hash.hexdigest()

    return digest

if \_\_name\_\_ == "\_\_main\_\_":

    # Input text

    text = input("Enter the text to calculate SHA-1 hash: ")

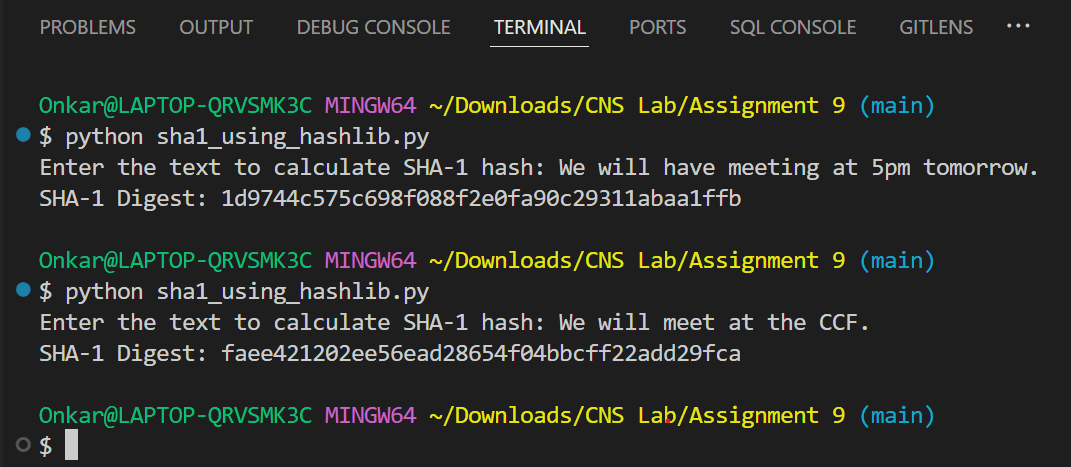
    # Calculate SHA-1 message digest

    sha1\_digest = calculate\_sha1(text)

    # Output the result

    print(f"SHA-1 Digest: {sha1\_digest}")

**Output:**

****

**Advantages of SHA-1:**

* **Speed and Efficiency**: SHA-1 was designed to be computationally efficient and can process large amounts of data quickly.
* **Widespread Use**: It has been widely adopted and used for various cryptographic applications, including digital signatures, certificates, and integrity checks.
* **Fixed-Length Output**: Regardless of the input size, the output is always 160 bits, making it convenient to use in various security protocols.

**Disadvantages of SHA-1:**

* **Weakness to Collisions**: SHA-1 is vulnerable to **collision attacks**, where two different inputs produce the same hash output. This reduces its effectiveness in ensuring data integrity and security.
* **Security Deprecation**: Due to these vulnerabilities, SHA-1 is no longer considered secure for cryptographic purposes. Most modern systems and protocols, including browsers and SSL certificates, have moved to stronger hash functions like SHA-256 or SHA-3.

**Importance of SHA-1:**

* **Legacy Systems**: Despite its vulnerabilities, SHA-1 was used for many years in security applications such as digital signatures and certificates.
* **Data Integrity**: SHA-1 can still be used to check the integrity of data, ensuring that files have not been altered during transmission.

**Security Risks and Vulnerabilities of SHA-1:**

* **Collision Attacks**: The primary vulnerability is the possibility of collision attacks. This means that an attacker could potentially create two different messages with the same hash, compromising the authenticity of the data.
* **Birthday Attack**: A specific type of attack known as a **birthday attack** makes it easier to find collisions in SHA-1 due to its 160-bit length, reducing the security level.
* **Deprecation in Modern Systems**: Due to these weaknesses, SHA-1 has been deprecated in most cryptographic protocols like TLS (Transport Layer Security) and digital certificates, where stronger algorithms like SHA-256 are preferred.

While SHA-1 played a significant role in the development of cryptographic standards, its vulnerabilities, especially to collision attacks, have made it unsuitable for modern security applications. Understanding SHA-1’s purpose and limitations is important, especially when dealing with legacy systems or understanding the evolution of cryptographic hash functions.

**Practical Applications of SHA-1:**

1. **Digital Signatures**: SHA-1 was commonly used in creating digital signatures to ensure the authenticity and integrity of documents. It would generate a hash of the message, which is then signed by a private key.
2. **File Integrity Verification**: SHA-1 was used to generate checksums for files to verify that files were not altered during transfer or storage. The recipient could compare the hash of the received file with the original hash to ensure integrity.
3. **Version Control Systems**: In systems like Git, SHA-1 hashes were used to identify commits, ensuring the integrity and tracking of changes in code repositories.
4. **SSL Certificates**: Until 2017, SHA-1 was used in SSL/TLS certificates for secure web communications. The hash was part of the process to ensure a website's identity and secure data transmission.
5. **Password Hashing**: SHA-1 was once used for hashing passwords in databases, providing a layer of security by storing a hashed version of the password instead of the plaintext.

Despite these applications, most systems have transitioned to more secure alternatives due to SHA-1's vulnerabilities.

**SHA-512 Algorithm:**

SHA-512 (Secure Hash Algorithm 512-bit) is part of the SHA-2 family and produces a 512-bit message digest. It’s a cryptographic hash function designed to provide higher security by generating a unique, fixed-length hash value from input data. SHA-512 is widely used for its robust security features.

**Overview of SHA-512 Algorithm**

SHA-512 is a cryptographic hash function that converts any input (such as a message or file) into a fixed 512-bit hash. The algorithm processes the input data in blocks and applies multiple rounds of complex operations to produce a unique hash value.

**Key Steps in the SHA-512 Algorithm:**

1. **Padding the Message**:  
   The input message is padded to ensure its length is congruent to 896 bits modulo 1024. Padding involves adding a single '1' bit followed by enough '0' bits, so that the message length becomes 1024 bits less than a multiple of 1024. The last 128 bits are used to store the original length of the message.
2. **Breaking the Message into Blocks**:  
   The padded message is divided into 1024-bit blocks for processing. Each block will undergo the hashing process individually.
3. **Initialize Hash Values**:  
   The algorithm uses eight 64-bit initial hash values, which are constant and specified by the SHA-512 standard. These values form the basis of the hash computation.
4. **Processing Each Block**:  
   For each 1024-bit block:
   * **Message Expansion**: The 1024-bit block is expanded into 80 64-bit words. These words are used in the main hashing loop.
   * **Compression Function**: The main loop processes the block with bitwise operations, modular additions, and logical functions (such as AND, XOR, OR). A set of 80 constant values is used along with the expanded message words.
   * The hash values are updated after each round using these operations.
5. **Update Hash Values**:  
   After processing each block, the intermediate hash values are updated. These values accumulate the results of each block’s computations.
6. **Concatenate Final Hash**:  
   After all the blocks have been processed, the final 512-bit hash is produced by concatenating the updated hash values from all rounds.

**Python Code for SHA-512 Implementation using Python’s built-in hashlib library:**

import hashlib

# Function to hash a message using SHA-512

def sha512\_encrypt(message):

    sha512\_hash = hashlib.sha512()

    sha512\_hash.update(message.encode('utf-8'))  # Convert the message to bytes

    return sha512\_hash.hexdigest()

# Function to verify the hash (like a decryption process)

def verify\_hash(original\_message, provided\_hash):

    original\_hash = sha512\_encrypt(original\_message)

    return original\_hash == provided\_hash

# Menu-driven system

def menu():

    while True:

        print("\n===== SHA-512 Hashing System =====")

        print("1. Encrypt a message using SHA-512")

        print("2. Verify a message against a given hash")

        print("3. Exit")

        choice = input("Enter your choice (1/2/3): ")

        if choice == '1':

            message = input("Enter the message to hash: ")

            hashed\_message = sha512\_encrypt(message)

            print(f"\nSHA-512 Hash: {hashed\_message}")

        elif choice == '2':

            original\_message = input("Enter the original message: ")

            provided\_hash = input("Enter the hash to verify against: ")

            if verify\_hash(original\_message, provided\_hash):

                print("\nVerification successful! The message matches the provided hash.")

            else:

                print("\nVerification failed! The message does not match the provided hash.")

        elif choice == '3':

            print("Exiting the program...")

            break

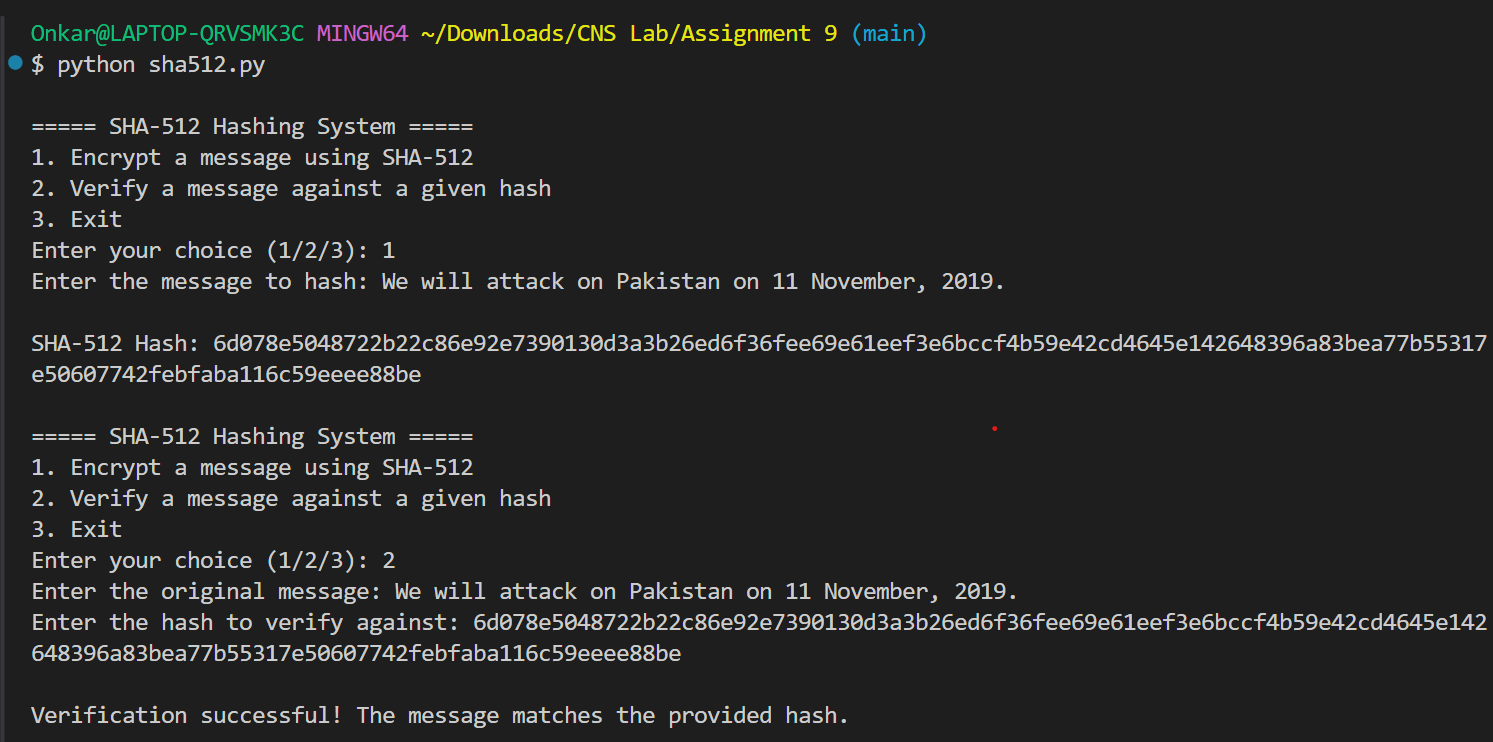
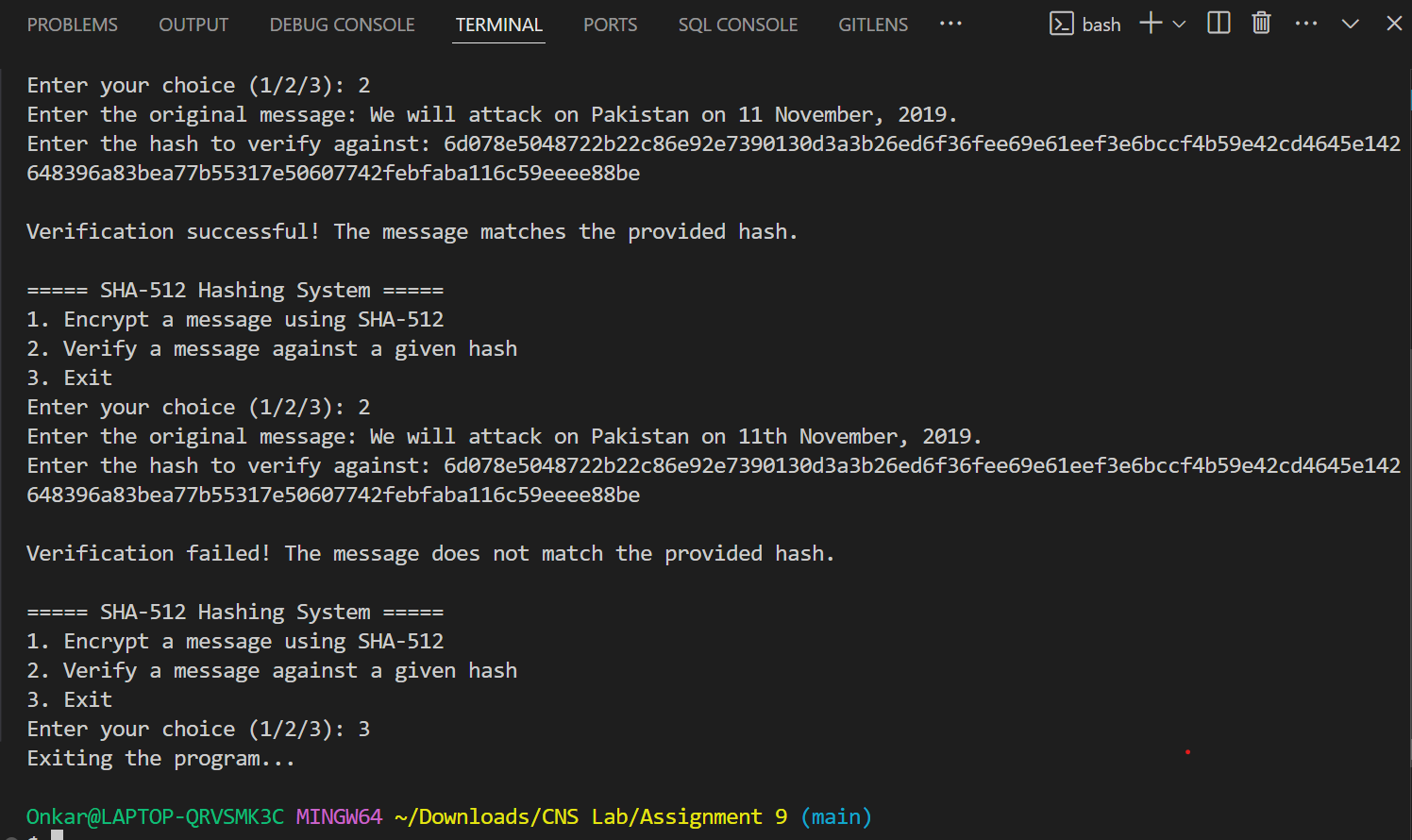
        else:

            print("Invalid choice. Please choose a valid option.")

if \_\_name\_\_ == "\_\_main\_\_":

    menu()

**Output:**

**Advantages of SHA-512:**

1. **High Security**: SHA-512 produces a 512-bit hash, making it much harder to break with brute force attacks compared to smaller hash sizes like SHA-1.
2. **Collision Resistance**: It offers strong resistance to collision attacks, meaning it’s very unlikely two different inputs will produce the same hash.
3. **Efficiency**: Despite its large output size, SHA-512 is designed to be computationally efficient on modern hardware.
4. **Compatibility with SHA-2 Family**: SHA-512 shares its core design with other SHA-2 algorithms, making it easier to switch between different levels of security.

**Disadvantages of SHA-512:**

1. **Higher Computational Cost**: Because of its larger size, SHA-512 may require more processing power and memory, making it slower on less powerful devices.
2. **Large Hash Size**: The 512-bit hash is larger, which may not be necessary for all applications, particularly when storage or bandwidth is a concern.
3. **Overkill for Small Applications**: In some use cases, the high security provided by SHA-512 might be unnecessary, and a smaller hash size like SHA-256 might suffice.

**Importance of SHA-512:**

SHA-512 is critical in contexts where strong security is essential, particularly in environments that demand protection against sophisticated attacks. Its high bit-length and resistance to common cryptographic attacks make it crucial for protecting sensitive data.

**Practical Applications of SHA-512:**

1. **Digital Signatures and Certificates**: SHA-512 is often used in creating digital signatures and securing SSL/TLS certificates to verify the authenticity and integrity of data.
2. **File Integrity**: It’s used in verifying file integrity by generating checksums to ensure that files have not been tampered with during transfer or storage.
3. **Cryptocurrency**: SHA-512 is used in blockchain technology to secure transactions and validate blocks.
4. **Password Hashing**: It’s commonly used in securely hashing passwords in databases, making stored passwords difficult to reverse-engineer.
5. **Secure Communication**: SHA-512 plays a role in securing communications over networks by being part of cryptographic protocols such as TLS.

**Security Risks and Vulnerabilities:**

* **Larger Hash Size Overhead**: While SHA-512 is more secure than smaller hashes, the added size may introduce performance issues for systems that don’t need this level of security.
* **Quantum Computing Threat**: In the future, quantum computing might pose a threat to even robust algorithms like SHA-512, necessitating the development of quantum-resistant algorithms.

Despite its high security, SHA-512 is still vulnerable to advances in technology, but for now, it remains one of the strongest cryptographic hash functions available.