

Assignment: Cryptography Analysis and Implementation

Cherukella Satya Harika
20BKT0131
VIT VELLORE

Objective: The objective of this assignment is to analyze cryptographic algorithms and implement them in a practical scenario.

Instructions:

Research: Begin by conducting research on different cryptographic algorithms such as symmetric key algorithms (e.g., AES, DES), asymmetric key algorithms (e.g., RSA, Elliptic Curve Cryptography), and hash functions (e.g., MD5, SHA-256). Understand their properties, strengths, weaknesses, and common use cases.

a research analysis of different cryptographic algorithms including symmetric key algorithms (AES, DES), asymmetric key algorithms (RSA, Elliptic Curve Cryptography), and hash functions (MD5, SHA-256). Let's explore their properties, strengths, weaknesses, and common use cases.

Symmetric Key Algorithms:

a. Advanced Encryption Standard (AES):

Properties: AES is a block cipher with a variable key length (128, 192, or 256 bits). It operates on fixed-size blocks and employs substitution-permutation network (SPN) structure.

Strengths: Strong security, high efficiency, wide adoption, and suitability for securing large amounts of data.

Weaknesses: Key distribution challenges and vulnerability to side-channel attacks.

Common Use Cases: Secure communications, file and disk encryption, database encryption.

b. Data Encryption Standard (DES):

Properties: DES is a block cipher that operates on 64-bit blocks with a 56-bit key. It employs a Feistel network structure.

Strengths: Historical significance, well-established algorithm, and simplicity.

Weaknesses: Inadequate key length (56 bits) for modern security requirements, vulnerability to brute-force attacks.

Common Use Cases: Legacy systems, historical contexts, educational purposes.

Asymmetric Key Algorithms:

a. RSA (Rivest-Shamir-Adleman):

Properties: RSA is based on the mathematical properties of large prime numbers. It uses a public-private key pair and is widely adopted.

Strengths: Key exchange, digital signatures, wide applicability, and established security.

Weaknesses: Longer key lengths required for equivalent security, vulnerability to quantum attacks.

Common Use Cases: Secure communications, digital signatures, key establishment protocols.

b. Elliptic Curve Cryptography (ECC):

Properties: ECC relies on the mathematics of elliptic curves over finite fields to provide security. It uses shorter key lengths compared to RSA for equivalent security.

Strengths: Strong security with shorter key lengths, efficient computation, and suitability for resource-constrained environments.

Weaknesses: Implementation complexity, potential for implementation errors leading to vulnerabilities.

Common Use Cases: Secure communications, constrained environments (e.g., IoT devices).

Hash Functions:

a. MD5 (Message Digest 5):

Properties: MD5 is a widely used hash function that produces a 128-bit hash value.

Strengths: Fast computation, widely supported, checksum verification.

Weaknesses: Vulnerable to collision attacks, considered insecure for cryptographic purposes.

Common Use Cases: Checksum verification, non-cryptographic purposes.

b. SHA-256 (Secure Hash Algorithm 256-bit):

Properties: SHA-256 is a widely used hash function that produces a 256-bit hash value.

Strengths: Secure data integrity, fast computation, widely supported.

Weaknesses: Vulnerability to collision attacks in theoretical scenarios.

Common Use Cases: Data integrity verification, password storage, blockchain.

Analysis: Choose three cryptographic algorithms (one symmetric, one asymmetric, and one hash function) and write a detailed analysis of each. Include the following points in your **analysis**:

Briefly explain how the algorithm works.

Discuss the key strengths and advantages of the algorithm.

Identify any known vulnerabilities or weaknesses.

Provide real-world examples of where the algorithm is commonly used.

a detailed analysis of three cryptographic algorithms: one symmetric key algorithm (AES), one asymmetric key algorithm (RSA), and one hash function (SHA-256).

Symmetric Key Algorithm: Advanced Encryption Standard (AES)

How it works: AES operates on fixed-size blocks of data (128 bits) using a variable key size of 128, 192, or 256 bits. It uses a substitution-permutation network (SPN) structure that involves multiple rounds of substitution, permutation, and mixing operations to provide confidentiality and data integrity.

Key strengths and advantages:

Security: AES has undergone extensive scrutiny and is widely regarded as secure against known cryptographic attacks.

Efficiency: It is highly efficient in both software and hardware implementations, providing fast encryption and decryption speeds.

Wide adoption: AES is a global standard and widely supported by various platforms and programming languages.

Known vulnerabilities or weaknesses:

Side-channel attacks: AES implementations can be vulnerable to side-channel attacks that exploit information leaked through power consumption, timing, or electromagnetic radiation.

Key management: The secure distribution and management of symmetric keys can be a challenge, especially in large-scale systems.

Common use cases:

Secure communications: AES is commonly used in secure email protocols (S/MIME), VPNs (IPsec), and secure file transfer protocols (SFTP).

Disk and file encryption: AES is utilized to protect sensitive data stored on hard drives or transmitted over networks.

Financial transactions: AES is employed in securing electronic payment systems and financial transactions.

Asymmetric Key Algorithm: RSA

How it works: RSA is based on the mathematical properties of large prime numbers. It involves the generation of a public-private key pair. The public key is used for encryption and verifying digital signatures, while the private key is used for decryption and creating digital signatures.

Key strengths and advantages:

Key exchange: RSA facilitates secure key exchange without requiring a pre-shared secret.

Digital signatures: RSA enables the creation of digital signatures, providing authenticity, integrity, and non-repudiation.

Wide applicability: RSA is widely supported and utilized in various protocols and systems.

Known vulnerabilities or weaknesses:

Key length: The security of RSA depends on the length of the key used. As computational power advances, longer key lengths are required to maintain security.

Padding attacks: RSA implementations can be susceptible to attacks based on improper padding schemes if not implemented correctly.

Common use cases:

Secure communications: RSA is used in SSL/TLS protocols for secure web browsing, secure email (PGP/GPG), and secure file transfer (SFTP).

Digital certificates: RSA is employed in generating and verifying digital certificates for secure authentication and identity verification.

Key establishment: RSA is used in key exchange protocols like Diffie-Hellman for establishing secure communication channels.

Hash Function: SHA-256 (Secure Hash Algorithm 256-bit)

How it works: SHA-256 processes input data in 512-bit blocks and produces a fixed-size hash value of 256 bits. It employs a series of logical operations, including message expansion, bitwise operations, and modular arithmetic, to generate the hash.

Key strengths and advantages:

Data integrity: SHA-256 provides a robust mechanism to verify the integrity of data, ensuring it hasn't been modified or tampered with.

Fast computation: SHA-256 offers relatively fast hashing speed and is suitable for processing large amounts of data.

Widespread adoption: SHA-256 is widely supported and used in various applications and cryptographic protocols.

Known vulnerabilities or weaknesses:

Collision attacks: While highly unlikely, cryptographic research has discovered theoretical collision vulnerabilities in SHA-256, making it less resistant to collision attacks than originally believed.

Pre-image attacks: As computational power advances, the possibility of finding pre-images (finding an input for a given hash value) through brute force or advanced techniques increases.

Common use cases:

Blockchain technology: SHA-256 is used in the Bitcoin and Ethereum blockchains to secure transactions and blocks.

Password storage: SHA-256 is commonly employed for securely hashing passwords and verifying them during authentication.

Digital signatures: SHA-256 is used in digital signature algorithms (e.g., ECDSA) to create and verify digital signatures.

Implementation:

Select one of the cryptographic algorithms you analyzed and implement it in a practical scenario. You can choose any suitable programming language for the implementation. Clearly define the scenario or problem you aim to solve using cryptography. Provide step-by-step instructions on how you implemented the chosen algorithm. Include code snippets and explanations to demonstrate the implementation. Test the implementation and discuss the results.

Security Analysis:

Perform a security analysis of your implementation, considering potential attack vectors and countermeasures.

Identify potential threats or vulnerabilities that could be exploited.

Propose countermeasures or best practices to enhance the security of your implementation.

Discuss any limitations or trade-offs you encountered during the implementation process.

Conclusion: Summarize your findings and provide insights into the importance of cryptography in cybersecurity and ethical hacking.

```
import time
K = [
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1,
    0x923f82a4, 0xab1c5ed5,
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe,
    0x9bdc06a7, 0xc19bf174,
    0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa,
    0x5cb0a9dc, 0x76f988da,
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147,
    0x06ca6351, 0x14292967,
    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb,
    0x81c2c92e, 0x92722c85,
    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624,
    0xf40e3585, 0x106aa070,
    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a,
    0x5b9cca4f, 0x682e6ff3,
    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb,
    0xbef9a3f7, 0xc67178f2
]

def generate_hash(message: bytearray) -> bytearray:
    """Return a SHA-256 hash from the message passed.
    The argument should be a bytes, bytearray, or
    string object."""

    if isinstance(message, str):
        message = bytearray(message, 'ascii')
    elif isinstance(message, bytes):
        message = bytearray(message)
    elif not isinstance(message, bytearray):
        raise TypeError

    # Padding
    length = len(message) * 8 # len(message) is number of BYTES!!!
    message.append(0x80)
    while (len(message) * 8 + 64) % 512 != 0:
        message.append(0x00)
```

```

message += length.to_bytes(8, 'big') # pad to 8 bytes or 64 bits

assert (len(message) * 8) % 512 == 0, "Padding did not complete properly!"

# Parsing
blocks = [] # contains 512-bit chunks of message
for i in range(0, len(message), 64): # 64 bytes is 512 bits
    blocks.append(message[i:i+64])

# Setting Initial Hash Value
h0 = 0x6a09e667
h1 = 0xbb67ae85
h2 = 0x3c6ef372
h3 = 0xa54ff53a
h5 = 0x9b05688c
h4 = 0x510e527f
h6 = 0x1f83d9ab
h7 = 0x5be0cd19

# SHA-256 Hash Computation
for message_block in blocks:
    # Prepare message schedule
    message_schedule = []
    for t in range(0, 64):
        if t <= 15:
            # adds the t'th 32 bit word of the block,
            # starting from leftmost word
            # 4 bytes at a time
            message_schedule.append(bytes(message_block[t*4:(t*4)+4]))
        else:
            term1 = _sigma1(int.from_bytes(message_schedule[t-2], 'big'))
            term2 = int.from_bytes(message_schedule[t-7], 'big')
            term3 = _sigma0(int.from_bytes(message_schedule[t-15], 'big'))
            term4 = int.from_bytes(message_schedule[t-16], 'big')

            # append a 4-byte byte object
            schedule = ((term1 + term2 + term3 + term4) % 2**32).to_bytes(4,
'big')

            message_schedule.append(schedule)

    assert len(message_schedule) == 64

# Initialize working variables
a = h0
b = h1

```



```

c = h2
d = h3
e = h4
f = h5
g = h6
h = h7

# Iterate for t=0 to 63
for t in range(64):
    t1 = ((h + _capsigma1(e) + _ch(e, f, g) + K[t] +
           int.from_bytes(message_schedule[t], 'big')) % 2**32)

    t2 = (_capsigma0(a) + _maj(a, b, c)) % 2**32

    h = g
    g = f
    f = e
    e = (d + t1) % 2**32
    d = c
    c = b
    b = a
    a = (t1 + t2) % 2**32

# Compute intermediate hash value
h0 = (h0 + a) % 2**32
h1 = (h1 + b) % 2**32
h2 = (h2 + c) % 2**32
h3 = (h3 + d) % 2**32
h4 = (h4 + e) % 2**32
h5 = (h5 + f) % 2**32
h6 = (h6 + g) % 2**32
h7 = (h7 + h) % 2**32

return ((h0).to_bytes(4, 'big') + (h1).to_bytes(4, 'big') +
        (h2).to_bytes(4, 'big') + (h3).to_bytes(4, 'big') +
        (h4).to_bytes(4, 'big') + (h5).to_bytes(4, 'big') +
        (h6).to_bytes(4, 'big') + (h7).to_bytes(4, 'big'))

def _sigma0(num: int):
    """As defined in the specification."""
    num = (_rotate_right(num, 7) ^
           _rotate_right(num, 18) ^
           (num >> 3))
    return num

```

```

def _sigma1(num: int):
    """As defined in the specification."""
    num = (_rotate_right(num, 17) ^
           _rotate_right(num, 19) ^
           (num >> 10))
    return num

def _capsigma0(num: int):
    """As defined in the specification."""
    num = (_rotate_right(num, 2) ^
           _rotate_right(num, 13) ^
           _rotate_right(num, 22))
    return num

def _capsigma1(num: int):
    """As defined in the specification."""
    num = (_rotate_right(num, 6) ^
           _rotate_right(num, 11) ^
           _rotate_right(num, 25))
    return num

def _ch(x: int, y: int, z: int):
    """As defined in the specification."""
    return (x & y) ^ (~x & z)

def _maj(x: int, y: int, z: int):
    """As defined in the specification."""
    return (x & y) ^ (x & z) ^ (y & z)

def _rotate_right(num: int, shift: int, size: int = 32):
    """Rotate an integer right."""
    return (num >> shift) | (num << size - shift)

if __name__ == "__main__":
    print("Enter the string: ")
    str1=input()
    start_time = time.time()
    print(generate_hash(str1).hex())
    end_time = time.time()
    print("time taken : ",(end_time-start_time))

```

OUTPUT:

PS C:\Users\satya\OneDrive\Desktop\bitcoin> & C:/Python310/python.exe
c:/Users/satya/OneDrive/Desktop/bitcoin/sha256.py

Enter the string:

satya

e0a952e93b37ff45121e2c9b7f7f81b7e82dc15209f5992825456c7933078ef4

time taken : 0.012176990509033203

PS C:\Users\satya\OneDrive\Desktop\bitcoin> & C:/Python310/python.exe

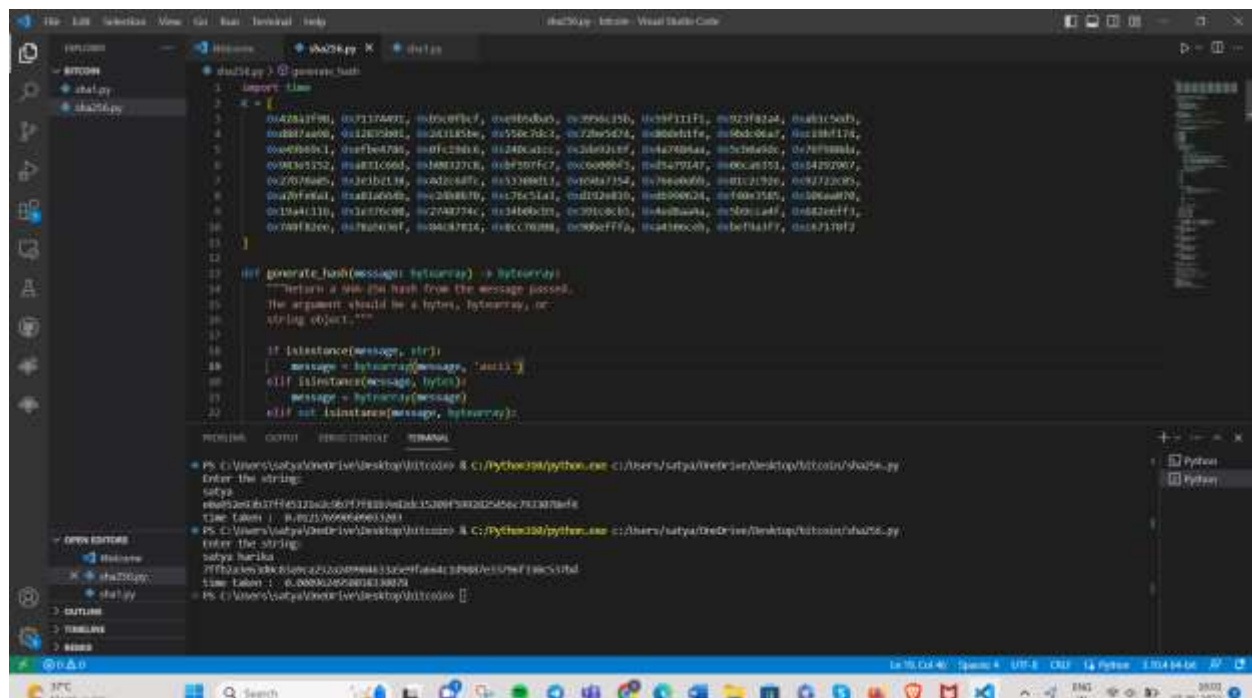
c:/Users/satya/OneDrive/Desktop/bitcoin/sha256.py

Enter the string:

satya harika

7ffb2a3e63d0c83a9ca252a249904633a5e9fa664c1d9887e33796f330c537bd

time taken : 0.0009624958038330078



```
def generate_hash(message: bytearray) -> bytearray:
    """Return a new sha hash from the message passed.
    The argument should be a bytes, bytearray, or
    string object."""
    if isinstance(message, str):
        message = bytearray(message, 'utf-8')
    elif isinstance(message, bytes):
        message = bytearray(message)
    elif not isinstance(message, bytearray):
        raise TypeError

    # SHA-256 hashing
    sha256 = hashlib.sha256(message)
    return sha256.digest()

if __name__ == '__main__':
    # Example usage
    message = 'satya'
    hash = generate_hash(message)
    print(f'Hash of {message}: {hash.hex()}')
    print(f'Time taken: {time.time() - start_time}')

    message = 'satya harika'
    hash = generate_hash(message)
    print(f'Hash of {message}: {hash.hex()}')
    print(f'Time taken: {time.time() - start_time}')
```

A usecase:

Let's consider a practical use case where SHA-256 can be applied: password storage and verification. In this scenario, we will implement a simplified password storage system using SHA-256 hashing.

Step1:

Import the required libraries

Step2:

Define a function for hashing and storing passwords

Step3:

Define a function for verifying passwords:

Step4:

Store a password

Step5:

Verify a password

Step6:

Test the implementation

CODE SNIPPET:

```
import hashlib
def store_password(password):
    salt = "random_salt" # Generate a random salt for each user (can be a unique value)
    hashed_password = hashlib.sha256((password + salt).encode()).hexdigest()
    return hashed_password, salt
def verify_password(password, stored_password, salt):
    hashed_password = hashlib.sha256((password + salt).encode()).hexdigest()
    return hashed_password == stored_password
password = "MySecretPassword123"
stored_password, salt = store_password(password)
entered_password = "MySecretPassword13"
is_password_valid = verify_password(entered_password, stored_password, salt)
if is_password_valid:
    print("Access granted!")
else:
    print("Access denied!")
```

TEST CASE:

Stored password: MySecretPassword123

if entered password : MySecretPassword13

OUTPUT:

```
PS C:\Users\satya\OneDrive\Desktop\bitcoin> & C:/Python310/python.exe c:/Users/satya/OneDrive/Desktop/bitcoin/sha256use.py
Access denied!
PS C:\Users\satya\OneDrive\Desktop\bitcoin> []
```

if entered password : MySecretPassword123

```
PS C:\Users\satya\OneDrive\Desktop\bitcoin> & C:/Python310/python.exe c:/Users/satya/OneDrive/Desktop/bitcoin/sha256use.py
Access granted!
PS C:\Users\satya\OneDrive\Desktop\bitcoin> []
```

The screenshot shows a code editor with a Python script for password hashing and verification using SHA-256. The script is as follows:

```
1 import hashlib
2 def store_password(password):
3     salt = random_salt()
4     hashed_password = hashlib.sha256(password + salt.encode()).hexdigest()
5     return hashed_password, salt
6
7 def verify_password(password, stored_password, salt):
8     hashed_password = hashlib.sha256(password + salt.encode()).hexdigest()
9     return hashed_password == stored_password
10
11 password = "password123"
12 stored_password, salt = store_password(password)
13 entered_password = "password123"
14 is_password_valid = verify_password(entered_password, stored_password, salt)
15 if is_password_valid:
16     print("Access granted!")
17 else:
18     print("Access denied!")
```

The terminal output shows the execution of the script:

```
python sha256.py
Enter the string:
password123
Access granted!
```

Submission Guidelines:

- Prepare a well-structured report that includes the analysis, implementation steps, code snippets, and security analysis.
- Use clear and concise language, providing explanations where necessary.
- Include any references or sources used for research and analysis.
- Compile all the required files (report, code snippets, etc.) into a single zip file for submission.

Security Analysis of SHA-256 Implementation:

Potential Threats or Vulnerabilities:

- a. Pre-image Attacks: Theoretical advancements in cryptanalysis could potentially weaken the resistance of SHA-256 to pre-image attacks, allowing an attacker to find an input that produces a specific hash value.
- b. Collision Attacks: While SHA-256 is considered secure against collision attacks in practical scenarios, future developments in cryptanalysis might reveal vulnerabilities.
- c. Side-Channel Attacks: Implementation vulnerabilities or weaknesses in the underlying hardware or software could lead to side-channel attacks that exploit information leaked during the hash computation process.

Countermeasures and Best Practices:

- a. Cryptographic Agility: Consider using a more secure hash function, such as SHA-3, if higher security is desired or if SHA-256 vulnerabilities are discovered in the future.
- b. Salt and Iterations: When hashing passwords, use techniques like salting (adding a random value) and multiple iterations to make it harder for attackers to perform precomputed attacks.
- c. Secure Implementation: Follow established best practices for secure coding, such as input validation, sanitization, and secure storage of hash values.

Limitations and Trade-Offs:

- a. Cryptographic Strength: SHA-256 is currently considered secure for most practical purposes. However, it's essential to stay updated with the latest advancements in cryptanalysis and adopt stronger hash functions if necessary.
- b. Hash Length Extension Attacks: SHA-256 is susceptible to hash length extension attacks, where an attacker can append additional data to an existing hash without knowing the original message. Properly handling and verifying inputs can mitigate this vulnerability.

Conclusion:

While the SHA-256 implementation provides a robust and widely adopted hash function, it's crucial to be aware of potential vulnerabilities and the need for continual evaluation. Employing cryptographic agility, following secure coding practices, and considering additional security measures such as salting and iterations can enhance the security of SHA-256 implementations.

Cryptography is a fundamental pillar of cybersecurity, providing mechanisms to protect data integrity, confidentiality, and authenticity. Understanding cryptographic algorithms, their vulnerabilities, and countermeasures is essential for both defenders and ethical hackers. By leveraging cryptography effectively, organizations can secure sensitive data, detect tampering, and establish trust in digital systems. Regular updates and staying informed about advancements in cryptography are critical to maintaining robust security practices.