Immutable Records

[Group Assignment - 3]

CS6/745: Modern Cryptography

Submitted to: Dr. Yuliang Zheng

Submitted by:

1. Joyanta Mondal (jmondal)
2. Shouzab Khan (Skhan6)
3. Pooja Srinivas (psriniva)
4. Rajendra Mohan (rnavulur)

March 24, 2024

INTRODUCTION:

In the era of digital communication, ensuring the integrity and immutability of information shared on public platforms is crucial. Discussion forums, social networks and many websites as such are constantly updated, making it possible for content to be altered or deleted after initial posting. In this assignment, we explore the implementation of a strategy to ensure the authenticity and immutability of data on such platforms.

The report is structured as follows:

Part I focuses on the core implementation of the hashing strategy. And Part II enhances the security of the system by introducing digital signatures. Using an algorithm like RSA, each hash value and its corresponding timestamp are signed, creating a time-signature pair. Afterwards, we discuss the immutability of the system, compare Part I and II, and share some vulnerabilities with pros and cons.

To work with our assignment, we use PyAscon and the cryptography library in Python version 3.12.

Part I:

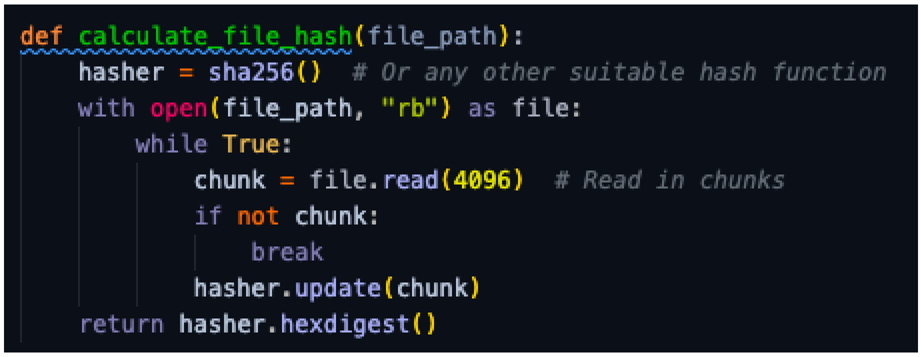
In the part, we implement a strategy for creating an immutable record of content snapshots using Merkle trees and hash chaining. It operates on a directory structure where each subdirectory represents a snapshot of data at a given time. Key design concepts include:

Merkle Hashing: Each file's content is hashed (using SHA-256). Then, a Merkle tree is built by recursively hashing pairs of hashes (or subdirectory hashes), ultimately creating a single Merkle root that represents the cryptographic fingerprint of the entire snapshot.

Hash Chaining: To preserve the chronological order of snapshots, the Merkle root of each snapshot is combined with the previous day's hash and fed into the Ascon-Hash function. This creates a chain where any modification to a past snapshot would irrevocably alter subsequent hashes.

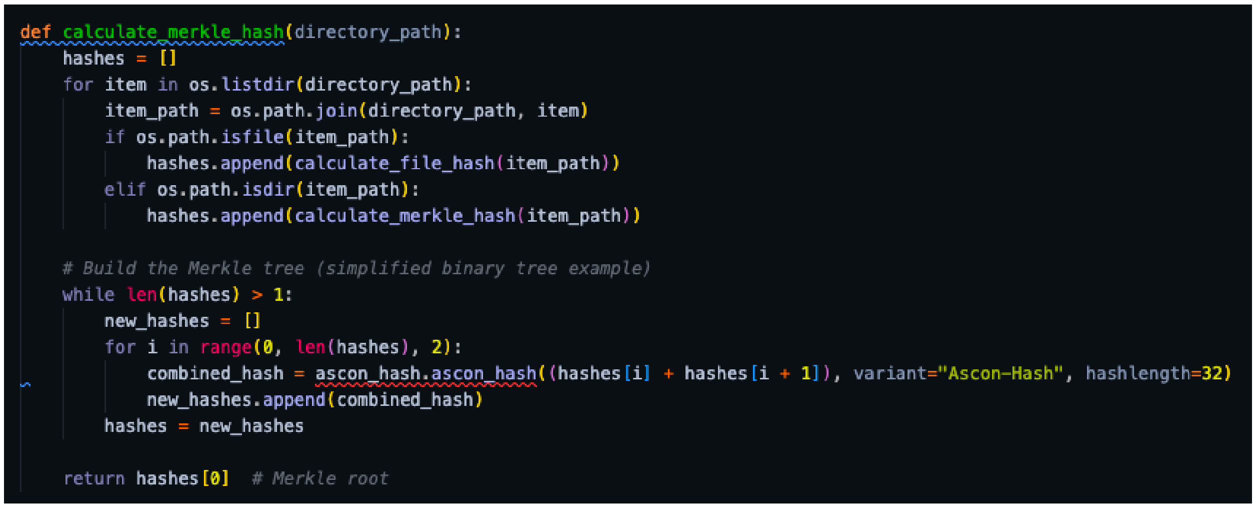
We implement these by several functions:

calculate_file_hash: Calculates the SHA-256 hash of a single file by reading it in chunks.



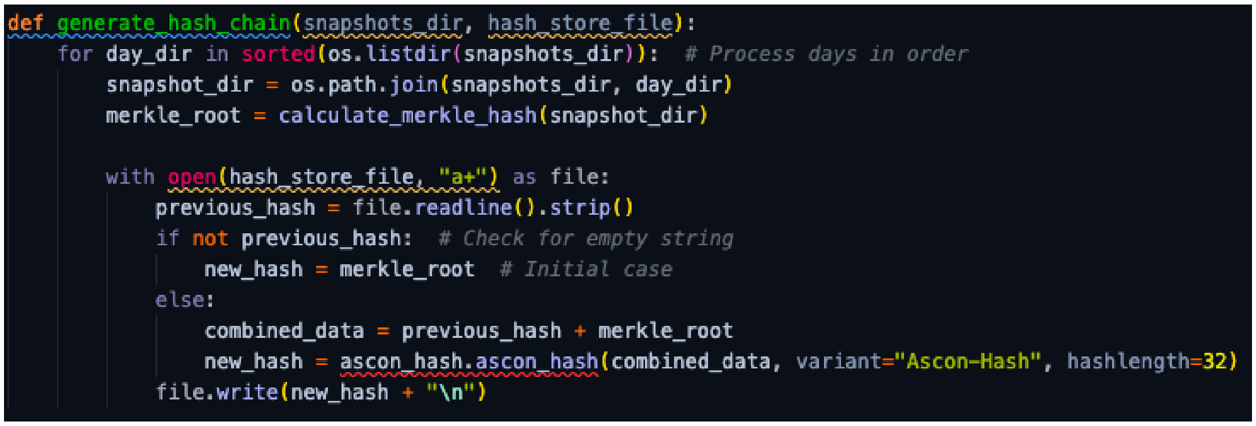
```python
def calculate_file_hash(file_path):
    hasher = sha256()  # Or any other suitable hash function
    with open(file_path, "rb") as file:
        while True:
            chunk = file.read(4096)  # Read in chunks
            if not chunk:
                break
            hasher.update(chunk)
    return hasher.hexdigest()
```

calculate_merkle_hash: Traverses a directory, calculates hashes for files, and recursively constructs a Merkle tree, returning the Merkle root.



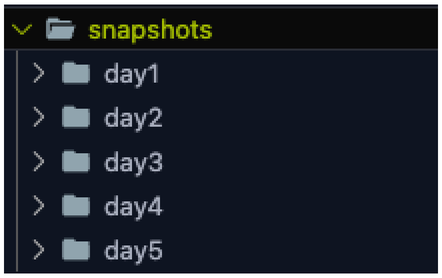
```python
def calculate_merkle_hash(directory_path):
    hashes = []
    for item in os.listdir(directory_path):
        item_path = os.path.join(directory_path, item)
        if os.path.isfile(item_path):
            hashes.append(calculate_file_hash(item_path))
        elif os.path.isdir(item_path):
            hashes.append(calculate_merkle_hash(item_path))

    # Build the Merkle tree (simplified binary tree example)
    while len(hashes) > 1:
        new_hashes = []
        for i in range(0, len(hashes), 2):
            combined_hash = ascon_hash.ascon_hash((hashes[i] + hashes[i + 1]), variant="Ascon-Hash", hashlength=32)
            new_hashes.append(combined_hash)
        hashes = new_hashes

    return hashes[0]  # Merkle root
```

generate_hash_chain: Iterates through the snapshot directories, calculates Merkle roots, and builds the hash chain by combining each root with the previous hash. The chain is stored in the hash_store_file.



```python
def generate_hash_chain(snapshots_dir, hash_store_file):
    for day_dir in sorted(os.listdir(snapshots_dir)):  # Process days in order
        snapshot_dir = os.path.join(snapshots_dir, day_dir)
        merkle_root = calculate_merkle_hash(snapshot_dir)

        with open(hash_store_file, "a+") as file:
            previous_hash = file.readline().strip()
            if not previous_hash:  # Check for empty string
                new_hash = merkle_root  # Initial case
            else:
                combined_data = previous_hash + merkle_root
                new_hash = ascon_hash.ascon_hash(combined_data, variant="Ascon-Hash", hashlength=32)
            file.write(new_hash + "\n")
```

We create snapshots of random images in a directory (divided into subdirectory and 1 image in each subdirectory).



After executing the code, it creates a hash and saves it in a file hash_chain.txt.

Part II:

In this part, we ensure data integrity and traceability through a carefully orchestrated process that includes snapshot generation, Merkle root calculation, timestamping, and digital signing.

1. Merkle Tree Calculation:

For each snapshot, a Merkle root hash is calculated. The Merkle tree structure enables efficient verification that none of the individual files within the snapshot have been altered since its creation.

calculate_file_hash: Hashes individual files using SHA-256 for content-based fingerprinting.

calculate_merkle_hash: Constructs a Merkle tree for each "snapshot" directory. A Merkle tree is a data structure where each non-leaf node is the hash of its child nodes. The final hash at the top (the Merkle root) represents the collective fingerprint of the entire directory.

2. Creating the Chain

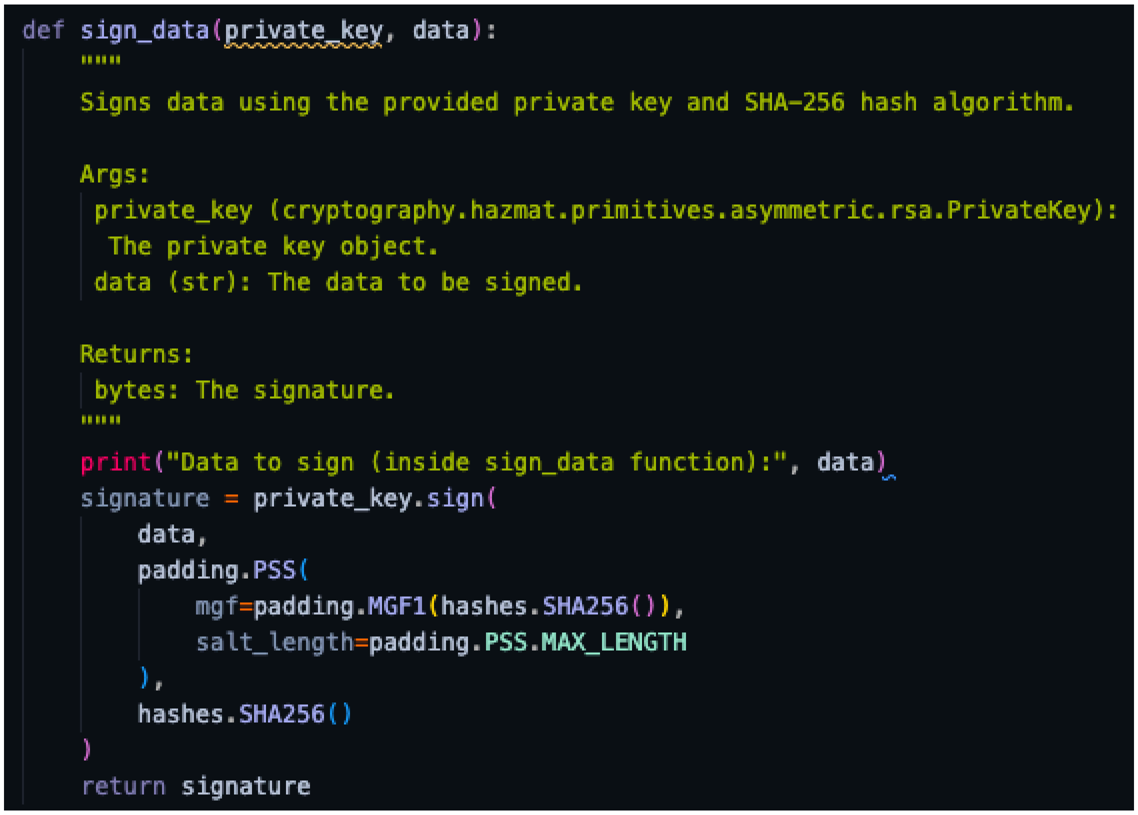
generate_hash_chain: The heart of the chain generation process. Here's how a new link is added:

Read Previous Link: Reads the last entry (if any) from the hash_store_file.

Calculate New Merkle Root: Calculates the Merkle root for the current data snapshot.

Timestamp Generation: A precise timestamp is generated alongside the Merkle root.

Data Signing: To guarantee authenticity and non-repudiation, the Merkle Hash digitally signed using a private key. This creates a secure signature that can only be produced by the holder of the private key.

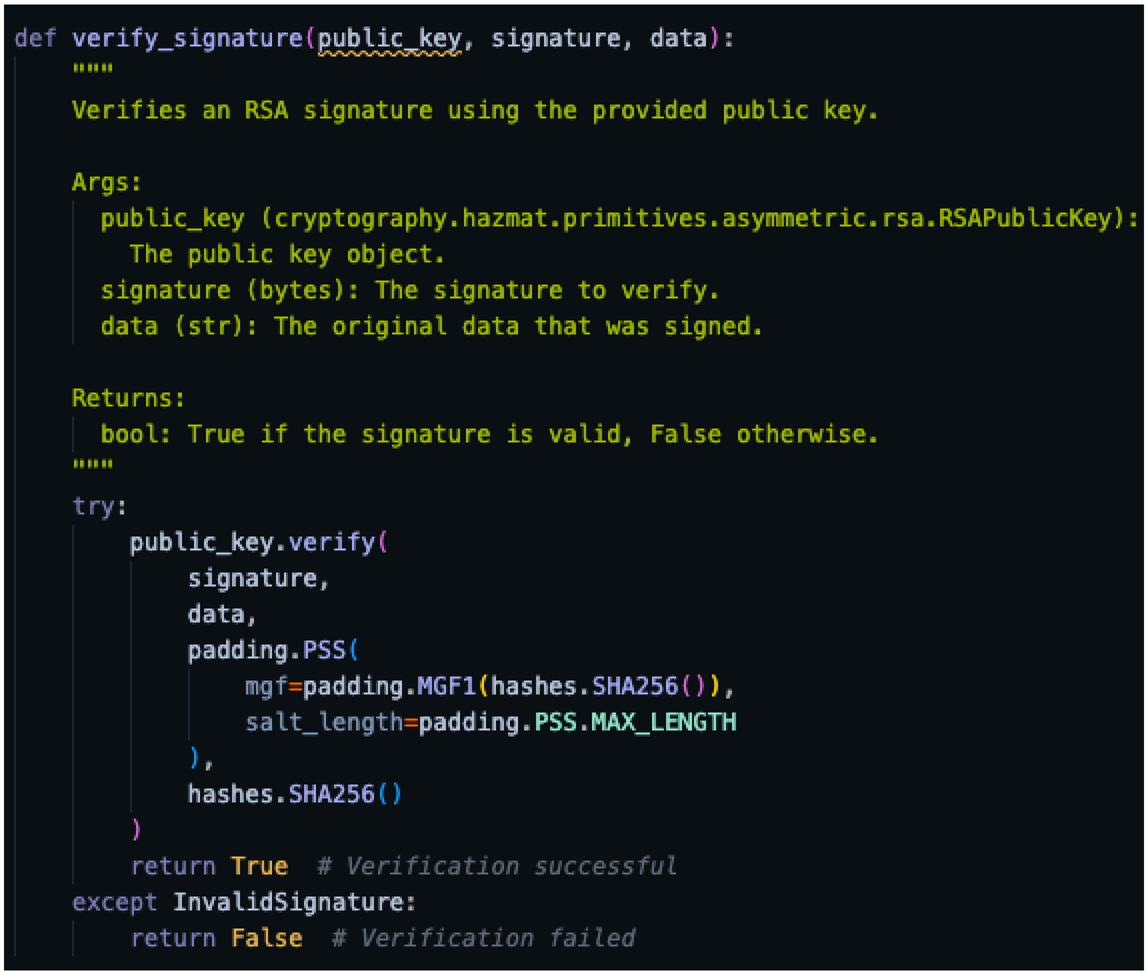


```python
def sign_data(private_key, data):
    """
    Signs data using the provided private key and SHA-256 hash algorithm.

    Args:
     private_key (cryptography.hazmat.primitives.asymmetric.rsa.PrivateKey):
      The private key object.
     data (str): The data to be signed.

    Returns:
     bytes: The signature.
    """
    print("Data to sign (inside sign_data function):", data)
    signature = private_key.sign(
        data,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
    return signature
```

Record Storage: Finally, the timestamp, merkle hash, and the digital signature are persistently stored in a folder. This creates an immutable record that associates the data snapshot with a point in time and cryptographically proves its origin.

3. Verification

verify_signature: This function can be used within the generate_hash_chain loop. It employs the public key to:



```python
def verify_signature(public_key, signature, data):
    """
    Verifies an RSA signature using the provided public key.

    Args:
      public_key (cryptography.hazmat.primitives.asymmetric.rsa.RSAPublicKey):
        The public key object.
      signature (bytes): The signature to verify.
      data (str): The original data that was signed.

    Returns:
      bool: True if the signature is valid, False otherwise.
    """
    try:
        public_key.verify(
            signature,
            data,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )
        return True  # Verification successful
    except InvalidSignature:
        return False  # Verification failed
```

Verify Previous Link: Checks if the signature of the previous link is valid, ensuring that the previous Merkle root and timestamp have not been tampered with.

Discussion:

1. Why the system is immutable:

The core principle behind the immutability of this system lies in the combination of cryptographic hashes and the linked chain structure:

Hash Properties: Cryptographic hash functions like SHA-256 have the following characteristics:

One-Way: Computationally infeasible to derive the original input data from its hash.

Deterministic: The same input always produces the same hash.

Avalanche Effect: Even a tiny change in the input data results in a drastically different hash.

Merkle Tree: Any change to a file deep within a snapshot directory will cascade upwards, ultimately altering the Merkle root hash.

Timestamp and Signature: Each link in the chain includes a timestamp and a digital signature created with your private key. This does the following:

Tamper Detection: If the timestamp, Merkle root, or signature are modified after they were signed, the verification process will fail.

Non-Repudiation: Only the holder of the private key could have created a valid signature, which establishes the origin and authenticity of each link.

2. Comparing Solutions: Part I vs. Part II

Feature	Part I (Simple Hashing)	Part II (Cryptographic Chain)
Tamper Detection	Limited - Detects changes within individual files	Thorough - Detects changes to files, timestamps, or the chain structure itself
Immutability	Weak - No way to prove when changes occurred	Strong - Chain structure enforces the order of updates
Authenticity	None - Cannot reliably determine who made changes	Ensures only the private key holder can create valid links
Non-repudiation	None - No way to prove origin	Provides a strong guarantee of who created each chain entry

3. Vulnerabilities

Even with strong cryptographic foundations, the system has potential vulnerabilities:

Private Key Compromise: If someone gains unauthorized access to the private key, they could forge new links in the chain, effectively rewriting history. Stringent key protection mechanisms are crucial.

Collisions in Hash Functions: While extremely unlikely with modern hash functions, a collision (two different inputs producing the same hash) could theoretically allow for tampering that goes undetected.

4. Pros and Cons of Shortened Snapshot Intervals

Pros:

Finer-grained Detection: Shorter intervals make it possible to pinpoint the time window of potential tampering more accurately.

Faster Reaction Time: The system admin can be alerted to potential issues sooner, allowing for a faster response.

Cons:

Increased Overhead: More frequent snapshots mean more computation (hashing) and storage space required for the chain data.