CRYPTOGRAPHIC ALGORITHMS: SHA-3

Report

NO.	STUDENT NAME	STUDENT QUID	STUDENT EMAIL	STUDENT WORK
1	HAMS GELBAN	202104047	HG2104047@QU.EDU.QA	3. DEMONSTRATION AND APPLICATION (3.1, 3.2)
2	ROAA NAIM	202106763	RN2106763@QU.EDU.QA	1.2 SHA-3 OVERVIEW, 2.LITERATURE SURVEY
3	SARA SAID	202004852	SS2004852@QU.EDU.QA	ABSTRACT,1.INTRODUCTION, 4. CONCLUSION
4	NOORA AL-EMADI	201901277	NA1901277@QU.EDU.QA	3. DEMONSTRATION AND APPLICATION (3.3, 3.4)
5	VASILIKI GEROKOSTA	202004536	VG2004536@QU.EDU.QA	3. DEMONSTRATION AND APPLICATION (3.5)

SUBMISSION DATE: NOVEMBER 15, 2023

COURSE NUMBER: CMPS310

COURSE DR: Jurlind Budurushi

Cryptographic algorithms: SHA-3

Sara Said[1], Hams Gelban[1], Rouaa Naim[1], Noora Al-Emadi[1], and Vasiliki Gerokosta[1]

Abstract. This research paper explores the Secure Hash Algorithm-3 (SHA-3), its mathematical theory, implementation, and significance in the field of cryptography. SHA-3 is a family of cryptographic hash functions, including SHA3-224, SHA3-256, SHA3-384, and SHA3-512, as well as extendable-output functions SHAKE128 and SHAKE256. It was selected as the successor to the SHA-2 family through the SHA-3 Cryptographic Hash Algorithm Competition organized by the National Institute of Standards and Technology (NIST). The paper discusses the sponge construction employed in SHA-3, outlining its components and operation. A comparison between SHA-3 and SHA-2 is provided, highlighting the unique design principles of each. The efficiency of SHA-3, both in hardware and software implementation, is emphasized, making it suitable for various applications. The paper also touches on the critical role of SHA-3 in enhancing security, particularly in the context of embedded systems. Various practical demonstrations and resources are presented to showcase the real-world applications of SHA-3, including file integrity verification, digital signatures, message digests, file deduplication, password hashing, and its use in cryptocurrencies.

Keywords: SHA-3, Secure Hash Algorithm, Cryptography, Cryptographic Hash Function, Sponge Construction, Collision Resistance, Digital Signatures.

1.Introduction:

In today's era of interconnectivity and data-driven operations, the need for powerful security mechanisms to protect sensitive information has been more critical than ever. Cryptographic algorithms form the foundation of data protection to ensure confidentiality, integrity, and authenticity. The cryptographic algorithm family: secure hash functions have emerged as fundamental components of recent cybersecurity. Notably, the Secure Hash Algorithm 3, SHA-3, is proof of cryptographic excellence.

SHA-3 was carefully crafted and selected as the winner of the SHA-3 Cryptographic Hash Algorithm Competition organized by the National Institute of Standards and Technology (NIST). This competition aimed to find a new hash function standard to replace the older SHA-2 family. Its official standardization in August 2015, through Federal Information Processing Standards Publication 202 (FIPS PUB 202), firmly establishes it as a pivotal element within cryptographic security protocols. SHA-3 indicates a transformative leap in the data hashing domain, offering robust protection against diverse threats.

1.1 Background:

Secure data hashing is required by the vulnerability of data, whether in transit or at rest. Hash functions are fundamental in ensuring the integrity of digital information. They process variable-length input data and produce fixed-length outputs, known as hash values or digests. These hash values serve as unique representations of the input data, playing an essential role in data verification, password storage, digital signatures, and various other security applications.

Over time, advancements in technology revealed vulnerabilities in the earlier members of the Secure Hashing Algorithms family (such as SHA-1 and SHA-2). In response to these challenges, SHA-3 was introduced, setting new standards for security and cryptographic resilience.

1.2 SHA-3 Overview:

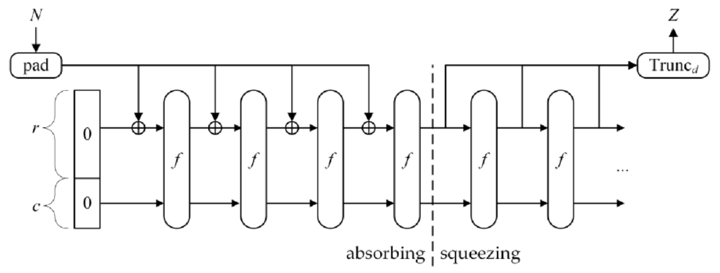
SHA-3 is a family of cryptographic hash functions and extendable-output functions (XOFs). It includes four hash functions: SHA3-224, SHA3-256, SHA3-384, and SHA3-512, along with the XOFs SHAKE128 and SHAKE256. It is based on the KECCAK algorithm. For XOFs, the length of the output can be chosen to meet the requirements of individual applications. The XOFs can be specialized to hash functions, subject to additional security considerations, or used in a variety of other applications.

The sponge construction, a defining feature of SHA-3, specifies functions on binary data with arbitrary output length, it operates on the principles of absorbing and squeezing bits. The function's flexibility allows it to adapt to variable input lengths, making it a versatile choice for applications ranging from message authentication codes to key derivation.

Sponge function: **SPONGE [f, pad, r] (N, d)**

- "f" - The Function: takes strings of a certain length (called "b") and turns them into strings of the same length.
- "pad" - Padding Rule: the mechanism of adding extra characters to a string to make it the right size.
- "r" - Rate and Width

In the SHA-3 hashing process, we start with input "N," applying the padding rule to create a new string, "P." Next, we divide "P" into "r" bits parts. Using an empty container called "S," we iteratively mix each piece of "P" with "S" using the sponge function machine. This process is repeated for all parts of "P." After the mixing, a portion of "S" is extracted, trimmed to "r" bits (if needed), and appended to "Z" until it reaches the desired length, "d," at which point the process concludes, and the final hash (digest) is returned.



In addition to its theoretical foundation, SHA-3 demonstrates practicality across various real-world scenarios. Its applications span from file integrity verification to digital signatures, message digests, file deduplication, and password hashing. Additionally, SHA-3 plays a pivotal role in cryptocurrencies since it ensures transaction verification and maintains the overall security of blockchain technology.

Furthermore, SHA-3 excels in terms of efficiency. It is known for its cost-effective implementation across hardware and software platforms, making it suitable for a wide range of applications, including embedded systems, sensors, consumer electronics, and symmetric key-based message authentication codes (MACs). Additionally, SHA-3 surpasses its predecessors in speed, effectively meeting the demands of modern cryptographic applications.

2. Literature Survey

	SHA1	SHA2 (224, 256, 384, & 512)	SHA3 (224, 256, 384, & 512)
Launch Year	1995	2001	2008
Block Size	512 bits	512 bits 1024 bits	1152 bits 1088 bits 832 bits 576 bits
Output (Hash Value or Message Digest)	160 bits / 20 bytes	256 bits / 32 bytes 512 bits / 64 bytes	224 bits / 28 bytes 256 bits / 32 bytes 384 bits / 48 bytes 512 bits / 64 bytes
Construction System	Merkle–Damgård	Merkle–Damgård	Sponge (Keccak)
Possibility of Collision	Possible – Google found proof of collision in 2017	No proof of collision has been found yet.	Susceptible to collision in squeeze attack.
Weakness	Has only one-use case – password storage. Susceptible to collision Short key length	SHA 256 is slower than its previous versions. Softwares and browsers must be updated to implement SHA2.	Susceptible to collision
Is it Still in Use?	No	Yes	Yes
Utility or Applications	TLS/SSL Certificate Verifying the Integrity of a file	Security application protocols Cryptographic transactions Digital certificates	Can replace SHA2, where necessary.

2.1 Positive Impact

SHA-3 has several positive impacts, making it a significant advancement in the field of cryptography:

1. Enhanced Security and Resistance
    - Collision Attacks: Built to resist collision attacks, preventing different inputs from producing the same hash output.
    - Pre-image Attacks: Denies attackers the ability to find an input corresponding to a specific hash output.
    - Cryptanalysis: Resists various cryptanalysis techniques, maintaining robustness against attempts to analyze and compromise its cryptographic scheme.
    - Length Extension Attacks: Prevents the extension of a hash value without knowledge of the original message.
    - Quantum Attacks (Post-Quantum Resilience): Remains a robust option in the face of potential threats from quantum computing.
2. Versatility and Adaptability
    - With output sizes ranging from 224 to 512 bits, SHA-3 is versatile and adaptable for different security requirements and use cases.
    - Immune to length extension attacks, further enhancing its security.
3. Performance
    - Demonstrating efficient processing on diverse computing platforms, SHA-3 is suitable for both resource-constrained and high-performance environments.
4. Post-Quantum Cryptography
    - Recognized as a strong candidate for post-quantum cryptography, SHA-3 remains robust in the face of potential threats from quantum computing.
5. Continuous Research
    - SHA-3 has undergone extensive public examination and research, resulting in a well-studied cryptographic algorithm.

2.2 Negative Impact

While SHA-3 has many advantages, there are some potential drawbacks and challenges associated with its adoption:

1. Transition and Compatibility
    - Widespread adoption may require transitioning from existing cryptographic algorithms, posing complexities and time-consuming processes.
    - SHA-3 may face vulnerabilities stemming from errors or oversights during its integration into specific systems or applications.
2. Resource Intensiveness
    - The more secure and larger output versions of SHA-3 can be resource-intensive, impacting computational resources.
    - The efficiency of SHA-3 relies on secure and proper implementation within specific use cases. Poor implementation practices may compromise its intended security features.
3. Deterministic Output
    - The deterministic output of SHA-3 may make it vulnerable to certain types of attacks, including side-channel attacks where information could leak during execution.
4. Fast Hardware Computation
    - Enables attackers to quickly test a large number of potential inputs, compromising security.

6. Slow Software Computation
    - Involves intricate mathematical operations, bitwise calculations, and prioritizes security over optimization for speed.
7. Additional attacks SHA-3 may be vulnerable to:
    - Side-Channel Attacks: Vulnerable to side-channel attacks, SHA-3 could leak information during execution, allowing attackers to exploit factors like power consumption or timing variations.
    - Attacks on Broader Cryptographic Systems: Weaknesses in the broader cryptographic system, such as inadequate key management, could be targeted by attackers.

In summary, SHA-3 brings many benefits in terms of security, versatility, and post-quantum resilience, but it also presents challenges related to adoption, resource usage, and staying ahead of evolving threats. Careful consideration and proper implementation are crucial to leveraging its positive impacts while mitigating its potential negative impacts.

3. Demonstration and Application

In this section, we will cover the real-world implementations and applications of SHA-3. To provide a comprehensive view of its versatility, we will explore five key demonstrations showcasing SHA-3's capabilities: Password Hashing, File Duplication Detection, Message Digest Generation, File Checksum Verification, and Digital Signatures. These practical applications highlight SHA-3's strength and reliability in safeguarding data and systems across diverse contexts.

3.1 Demonstration on Password Hashing

SHA-3 (Secure Hash Algorithm 3) is a cryptographic hash function designed to ensure data integrity and security. When it comes to passwords, SHA-3 plays a crucial role in enhancing security. Passwords are typically hashed using SHA-3 before storage in a database. This process involves taking the user's password, adding a random "salt" value to it, and then applying SHA-3 to generate a fixed-length hash. Salting adds an extra layer of security by making it computationally expensive for attackers to use precomputed tables (rainbow tables) to crack passwords. SHA-3, known for its resistance to cryptographic attacks, ensures that even small changes in the input result in significantly different hash values, making it highly suitable for password hashing and safeguarding user credentials from unauthorized access.
There are various reasons as to why the SHA-3 algorithm is preferred in Password hashing as it provides several advantages, including enhanced security and the prevention of password theft. When a password is hashed, it becomes an inscrutable string of characters, even to potential hackers with access to the hash itself. This means that, even in the worst-case scenario where a hash is compromised, the original password remains concealed.

Password hashing is widely used in real-world applications, particularly in website authentication. When users log into a website, the password they enter is hashed and then compared to the stored hash to verify its accuracy. This method is vastly more secure than storing passwords in plain text.

Use Case: User Registration and Login System with Password Hashing and Salting

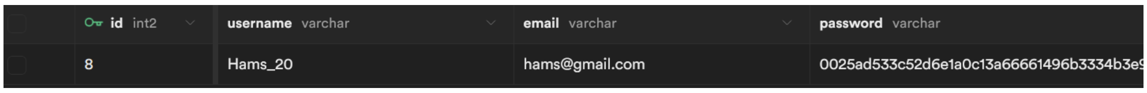
I. Detailed Description of the Use Case:

- User Registration and Signup Functionality:
  - Allow users to sign up by providing a username, password, name, and email.
- Password Hashing:
  - Hash the user's password using SHA-3 with a unique salt for each user.
- Database Setup:
  - Create a database table to store user information, including the username, hashed password, salt, name, and email.
- Login Functionality:
  - Implement a login feature where users can enter their username and password to access their accounts.
- Password Verification:
  - When a user attempts to log in, hash the entered password using SHA-3 and the stored salt. Compare it to the stored hashed password in the database.

II. Steps for Setting Up the Environment:

<u>*Setting up the environment:*</u>
For this part, we used Visual Studio Code as our source-code editor, to be able to work with Python. The choice for the use of Python stems directly from its simplicity and readability, especially for complex algorithms such as SHA3.

1. *Database Setup:*
   - Create a database using Supabase.
   - Create a table to store user information (user_id, username, hashed_password, salt, name, email).
   - Set up a sequence for generating user IDs once it is created.
   - Give the designed website permission to add to the database.



id int2	username varchar	email varchar	password varchar
8	Hams_20	hams@gmail.com	0025ad533c52d6e1a0c13a66661496b3334b3e

2. Programming Environment:

1. Install Visual Studio Code
2. Install the latest version of Python.
3. Select Python interpreter when saving/working with a file.
4. Update PIP: pip is the Python package manager.

```
python -m pip install --upgrade pip.
```

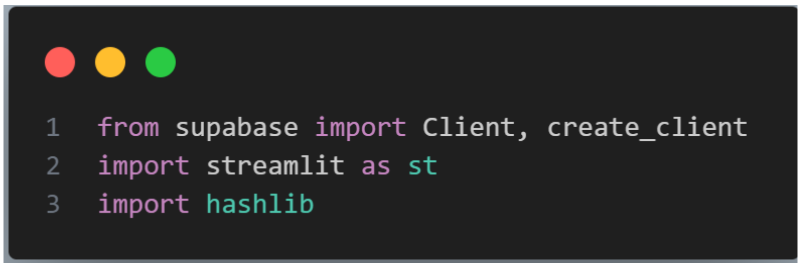
5. Install streamlit to be able to create a simple website visualizing the steps of signing up and login in. Open the terminal on Visual Studio Code and type:

```
pip install streamlit supabase
```

III. Steps for Replicating the Demonstration:

<u>*Steps:*</u>

1. Importing libraries that will be of use for our implementation.



```python
from supabase import Client, create_client
import streamlit as st
import hashlib
```

a. ‘hashlib’ module is used for the SHA-3 hashing. There are many versions of the SHA-3 algorithm that can be used: sha3_224, sha3_512, sha3_384 with the key difference between them being their fixed-length output in bits. However, sha3_256 provides a reasonable length of 256 bits (32 bytes) which contributes to practical representation of a hash value in hexadecimal form.

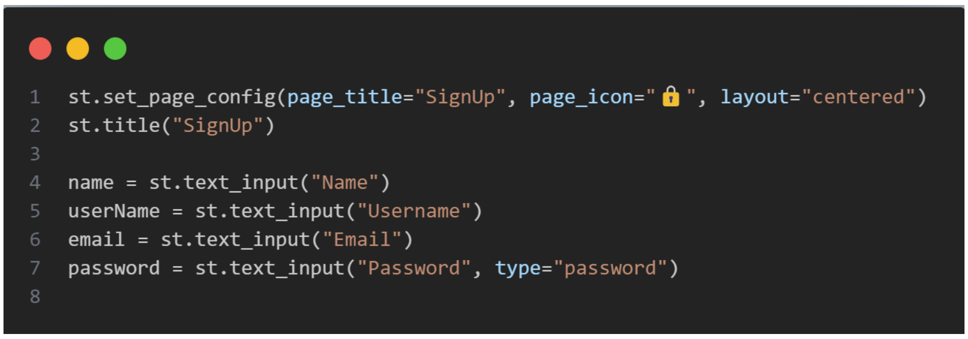
b. ‘supabase’ Client and create_client are from the supabase library, which is a client library for interacting with Supabase, an open-source alternative to Firebase. Client is typically used for creating a client object to interact with Supabase services, while create_client is a function for creating a new client instance or handling client configuration. These are essential for establishing connections and making requests to a Supabase database or authentication service.

c. ‘streamlit’ is a popular Python library used for creating web applications with minimal effort. st is an alias for the streamlit module, and it's commonly used to create interactive and data-driven web applications in a straightforward manner. You can use st to create widgets, display data, and build the user interface of your web application.

d. ‘os’ is another built-in Python module that provides functions for interacting with the operating system. It is used for tasks like file system operations, managing environment variables, and executing system commands. By importing os, we can leverage its functionalities to work with the file system, access environment variables, and perform various OS-related operations within your Python code.

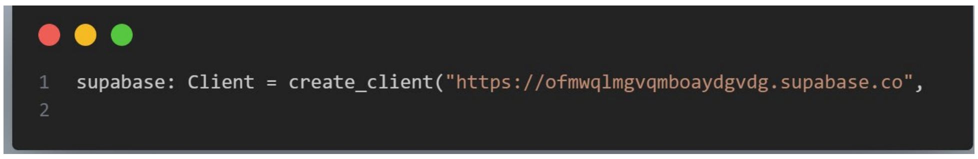
2.User Registration and Signup:

- Implement a user registration interface using a streamlit library.



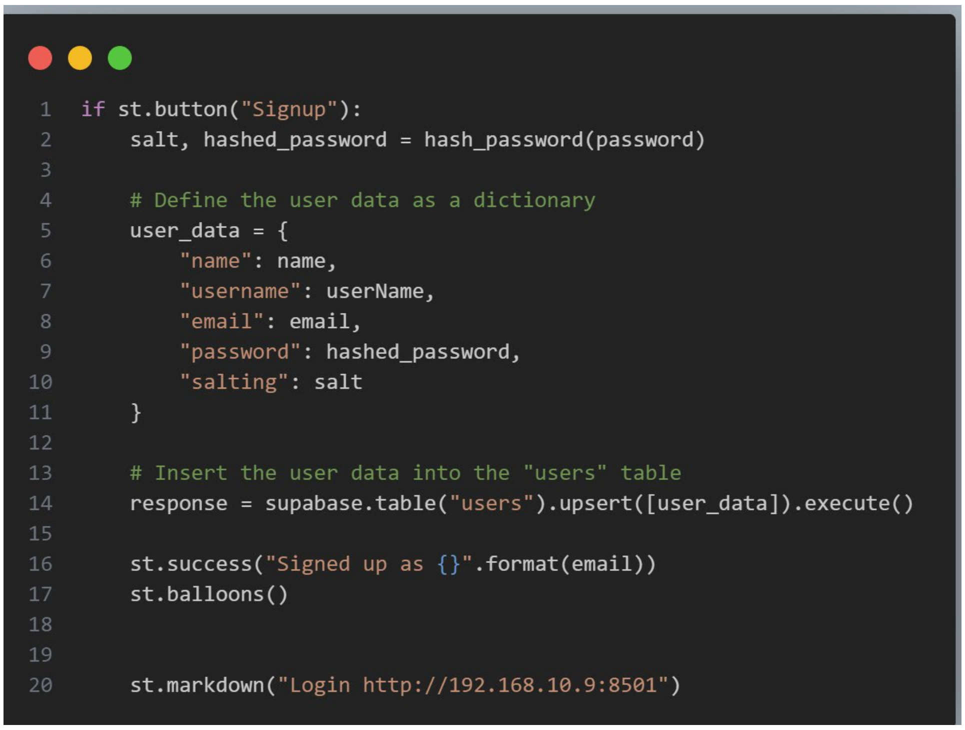
```python
st.set_page_config(page_title="SignUp", page_icon="🔒", layout="centered")
st.title("SignUp")

name = st.text_input("Name")
userName = st.text_input("Username")
email = st.text_input("Email")
password = st.text_input("Password", type="password")

```

- Connect the webpage to the database.



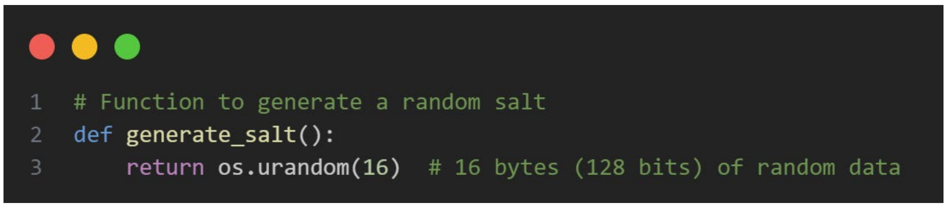
```python
supabase: Client = create_client("https://ofmwqlmgvqmboaydgvdg.supabase.co",

```

- Allow users to input their username, password, name, and email.



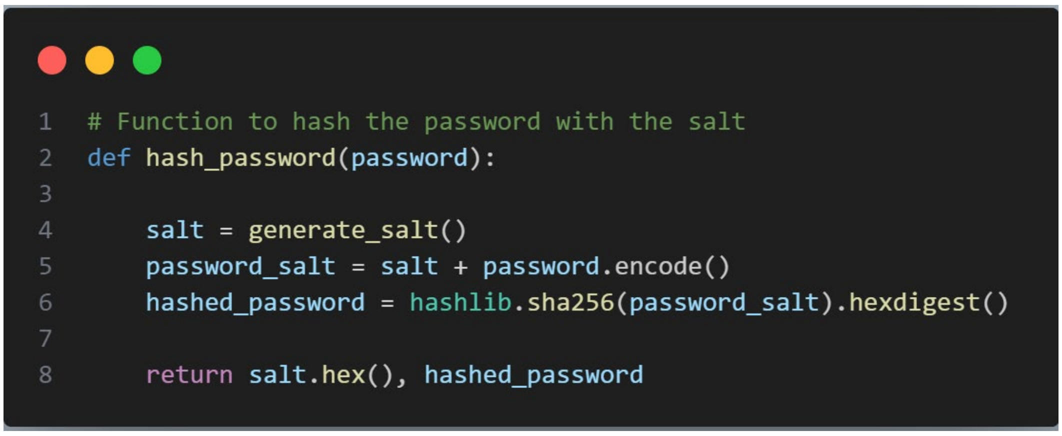
```python
if st.button("Signup"):
    salt, hashed_password = hash_password(password)

    # Define the user data as a dictionary
    user_data = {
        "name": name,
        "username": userName,
        "email": email,
        "password": hashed_password,
        "salting": salt
    }

    # Insert the user data into the "users" table
    response = supabase.table("users").upsert([user_data]).execute()

    st.success("Signed up as {}".format(email))
    st.balloons()


    st.markdown("Login http://192.168.10.9:8501")
```

- Generate a random salt.



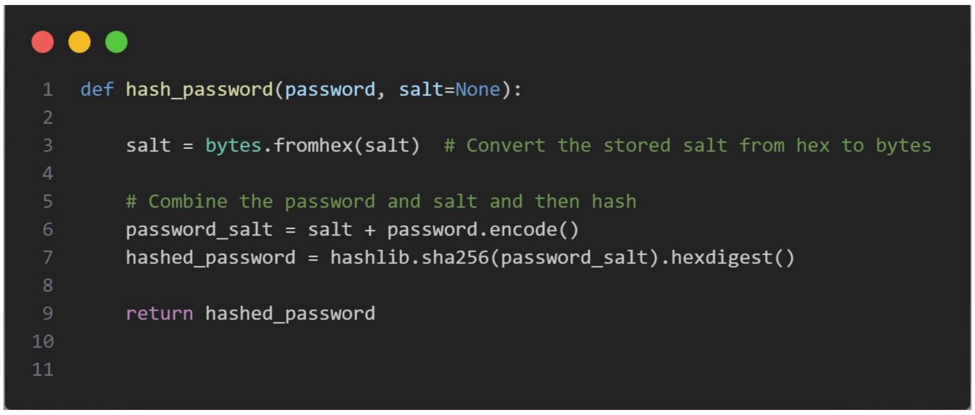
```python
# Function to generate a random salt
def generate_salt():
    return os.urandom(16)  # 16 bytes (128 bits) of random data
```

- Hash the password with the salt using SHA-3:



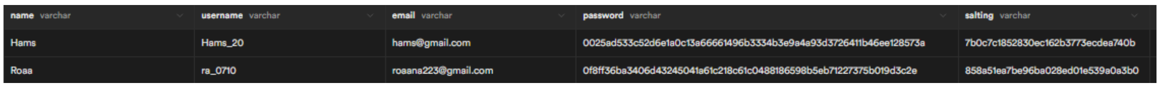
```python
# Function to hash the password with the salt
def hash_password(password):

    salt = generate_salt()
    password_salt = salt + password.encode()
    hashed_password = hashlib.sha256(password_salt).hexdigest()

    return salt.hex(), hashed_password
```

- Use Function that generates the hash value of a given message. Analytically, ‘message is converted to bytes using ‘. encode(‘utf-8’)’, ‘sha3_256’ calculates the hash of the message and ‘.hexdigest()’ method is called on the hash object to retrieve the hexadecimal representation of the hash which is returned from the function.



```python
def hash_password(password, salt=None):

    salt = bytes.fromhex(salt)  # Convert the stored salt from hex to bytes

    # Combine the password and salt and then hash
    password_salt = salt + password.encode()
    hashed_password = hashlib.sha256(password_salt).hexdigest()

    return hashed_password


```

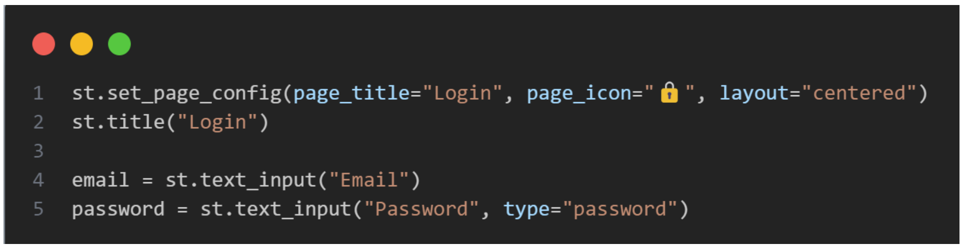
- Store the username, hashed password, salt, name, and email in the database.



name varchar	username varchar	email varchar	password varchar	salting varchar
Hams	Hams_20	hams@gmail.com	0025ad533c52d6e1a0c13a66661496b3334b3e9a4a93d3726411b46ee128573a	7b0c7c1852830ec162b3773ecdea740b
Roaa	ra_0710	roaana223@gmail.com	0f8ff36ba3406d43245041a61c218c61c0488186598b5eb71227375b019d3c2e	858a51ea7be96ba028ed01e539a0a3b0

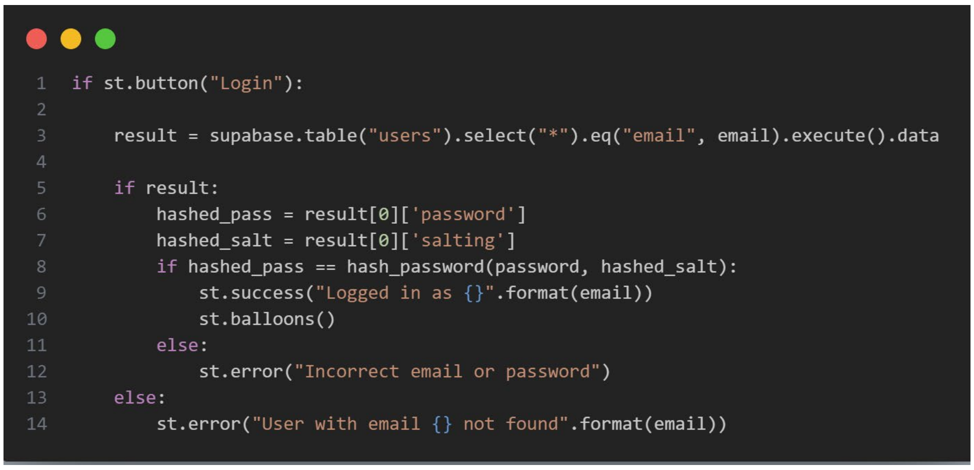
3.Login Functionality:

- Create a login interface.



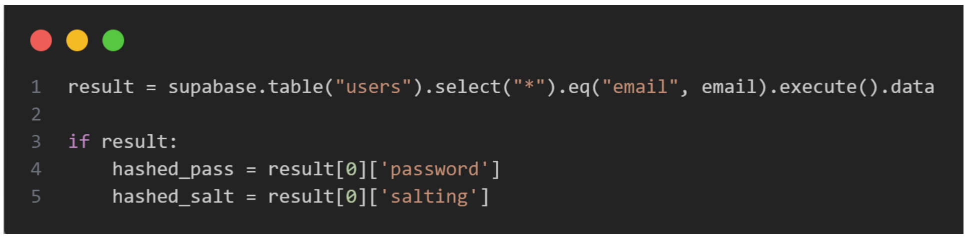
```python
st.set_page_config(page_title="Login", page_icon="🔒", layout="centered")
st.title("Login")

email = st.text_input("Email")
password = st.text_input("Password", type="password")
```

- Allow users to input their username and password.



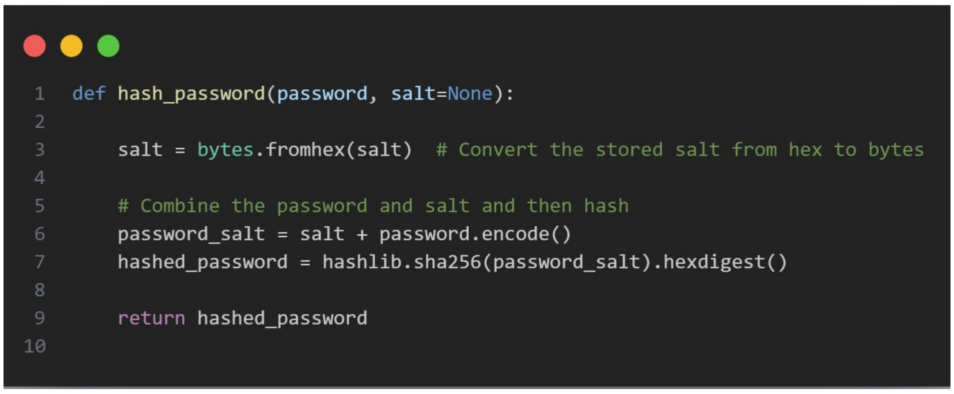
```python
if st.button("Login"):

    result = supabase.table("users").select("*").eq("email", email).execute().data

    if result:
        hashed_pass = result[0]['password']
        hashed_salt = result[0]['salting']
        if hashed_pass == hash_password(password, hashed_salt):
            st.success("Logged in as {}".format(email))
            st.balloons()
        else:
            st.error("Incorrect email or password")
    else:
        st.error("User with email {} not found".format(email))
```

- Retrieve the stored salt from the database based on the entered username.



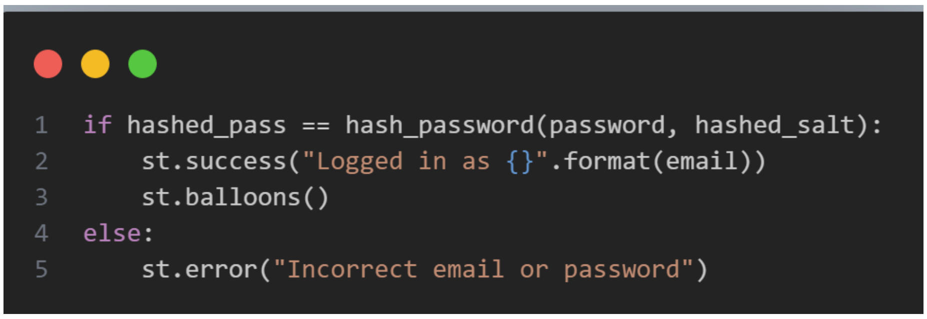
```python
result = supabase.table("users").select("*").eq("email", email).execute().data

if result:
    hashed_pass = result[0]['password']
    hashed_salt = result[0]['salting']
```

- Hash the entered password with the stored salt using SHA-3.



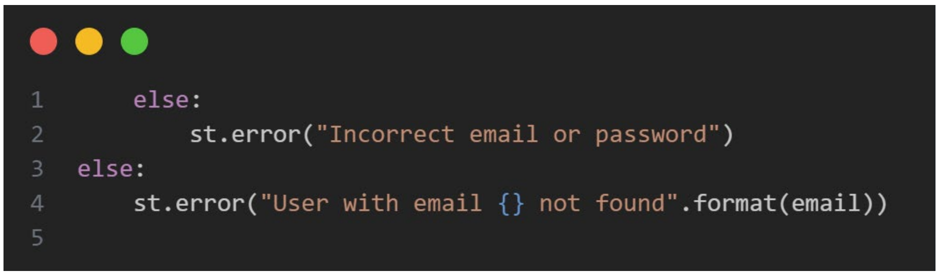
```python
def hash_password(password, salt=None):

    salt = bytes.fromhex(salt)  # Convert the stored salt from hex to bytes

    # Combine the password and salt and then hash
    password_salt = salt + password.encode()
    hashed_password = hashlib.sha256(password_salt).hexdigest()

    return hashed_password

```

- Compare the computed hash value with the stored hashed password.



```python
if hashed_pass == hash_password(password, hashed_salt):
    st.success("Logged in as {}".format(email))
    st.balloons()
else:
    st.error("Incorrect email or password")
```

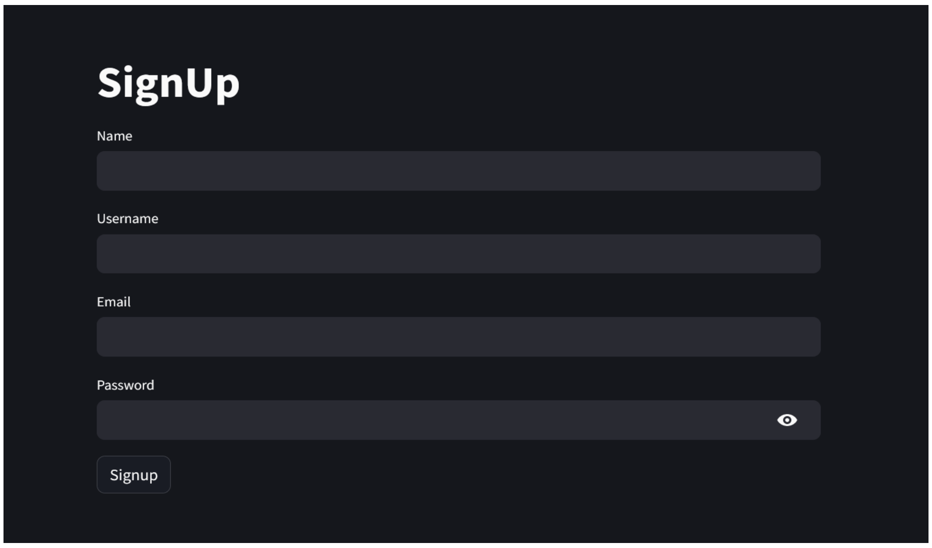
- Grant access if the hashes match; otherwise, deny access.

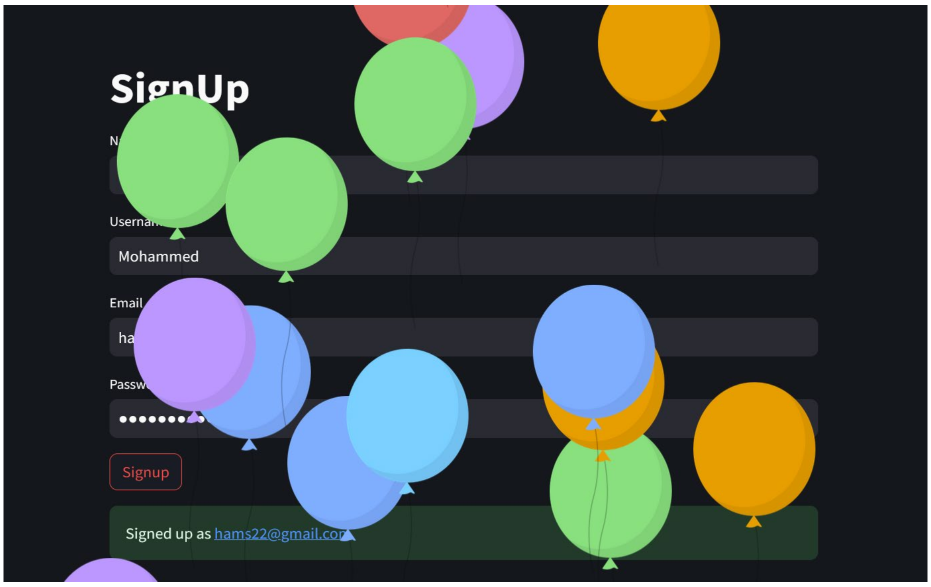


```
    else:
        st.error("Incorrect email or password")
else:
    st.error("User with email {} not found".format(email))

```

4. User-Friendly Interface:

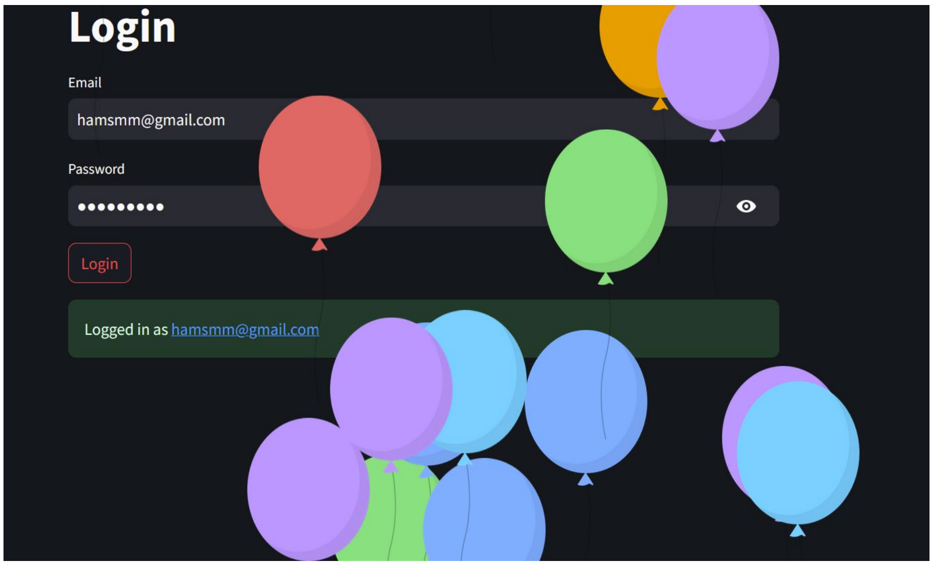
- Create user-friendly interfaces for registration, login, and password recovery.





Signed up as hams22@gmail.com

    ○ Provide clear feedback to users on the success or failure of each action.


Login

Email

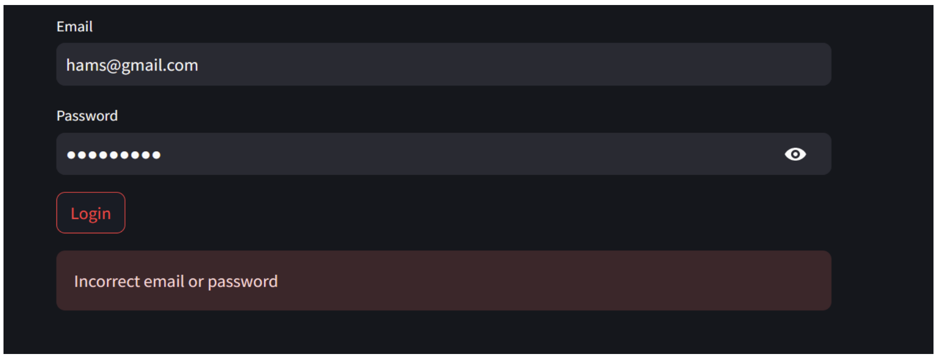
hamsmm@gmail.com

Password

•••••••••

Login

Logged in as hamsmm@gmail.com


Email

hams@gmail.com

Password

•••••••••

Login

Incorrect email or password

3.2 Demonstration of File Duplication

File deduplication is a data optimization technique that plays a crucial role in efficiently managing and conserving storage resources. It involves identifying and eliminating duplicate copies of files or data in a storage system. By doing so, organizations can save storage space, reduce data redundancy, and improve data management. File deduplication typically works by using hash functions, such as SHA-3, to generate unique fingerprints (hashes) for each file or data chunk. These fingerprints are compared to identify duplicates, and redundant copies are replaced with references to the original data, thus conserving storage space. SHA-3, known for its cryptographic strength, ensures that the generated file fingerprints are unique and collision-resistant, making it a reliable choice for file deduplication to avoid data loss while achieving efficient storage utilization.

I. Detailed Description of the Use Case:

The use case focuses on demonstrating the concept of file deduplication, a data optimization technique.

- Storage Efficiency:
    - Highlight the benefits of file deduplication in conserving storage space and reducing data redundancy.
- Hashing for Deduplication:
    - Show how cryptographic hash functions like SHA-3 are used to generate unique fingerprints (hashes) for files.
- Duplicate Identification:
    - Explain the process of identifying duplicate files by comparing their hashes.
- Space Savings:
    - Illustrate how deduplication replaces redundant copies with references to the original data,

leading to efficient storage utilization.

- Data Integrity:
  - Emphasize the importance of using strong hash functions like SHA-3 to ensure the uniqueness and integrity of file fingerprints.

II. Steps for Setting Up the Environment:

1. **Select a Storage System**:
   - Choose a storage system (e.g., local file system, network-attached storage, or cloud storage), we choose a local file system.

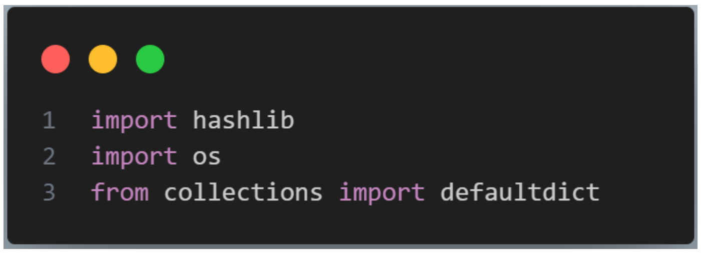
2. Programming Environment:

   - Install Visual Studio Code
   - Install the latest version of Python.
   - Select Python interpreter when saving/working with a file.
   - Update PIP: pip is the Python package manager.
     - python -m pip install --upgrade pip.

III. Steps for Replicating the Demonstration:

Steps:

1. Importing libraries that will be of use for our implementation



```
import hashlib
import os
from collections import defaultdict
```

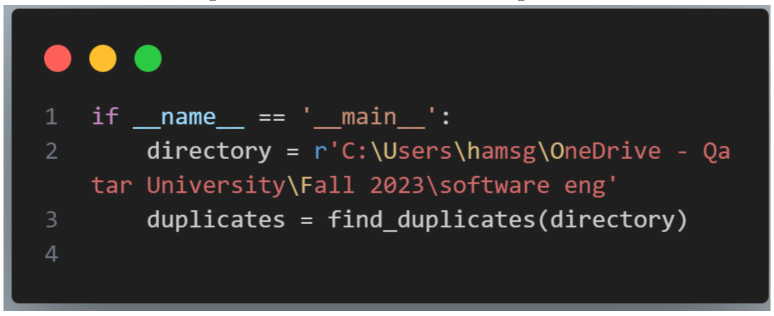
a. ‘hashlib’ module is used for the SHA-3 hashing. There are many versions of the SHA-3 algorithm that can be used: sha3_224, sha3_512, sha3_384 with the key difference between them being their fixed-length output in bits. However, sha3_256 provides a reasonable length of 256 bits (32 bytes) which contributes to practical representation of a hash value in hexadecimal form.

b. ‘os’ is another built-in Python module that provides functions for interacting with the operating system. It is used for tasks like file system operations, managing environment variables, and executing system commands. By importing os, we can leverage its functionalities to work with the file system, access environment variables, and perform various OS-related operations within your Python code.

c. ‘defaultdict’: Picture a dictionary that's ready for the unexpected. This subclass in collections ensures that even if a key doesn't exist, there's a default value waiting to step in.

2. File Collection:

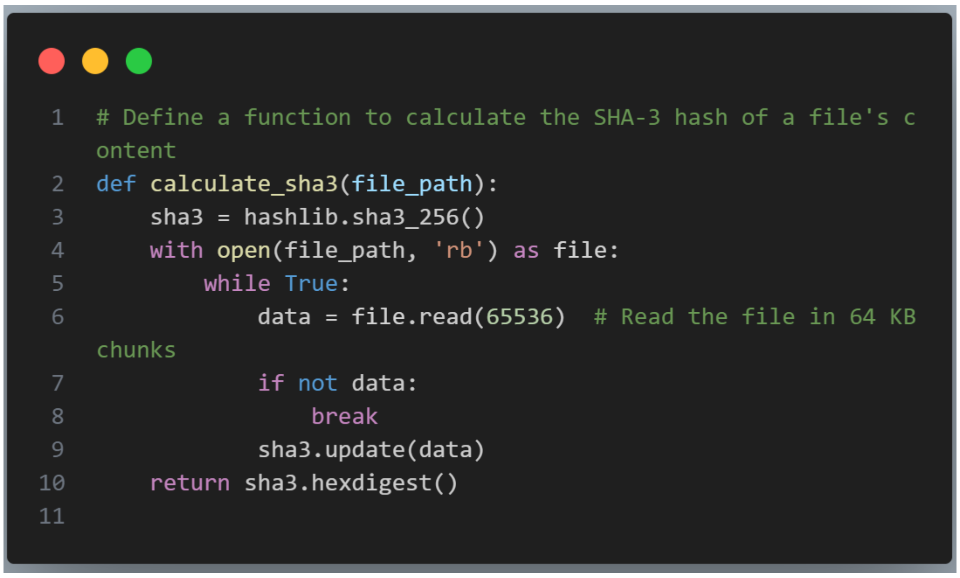
- Gather a set of sample files to demonstrate deduplication. Include some duplicate files.



```python
if __name__ == '__main__':
    directory = r'C:\Users\hamsg\OneDrive - Qatar University\Fall 2023\software eng'
    duplicates = find_duplicates(directory)

```

3. **Hashing Files**:

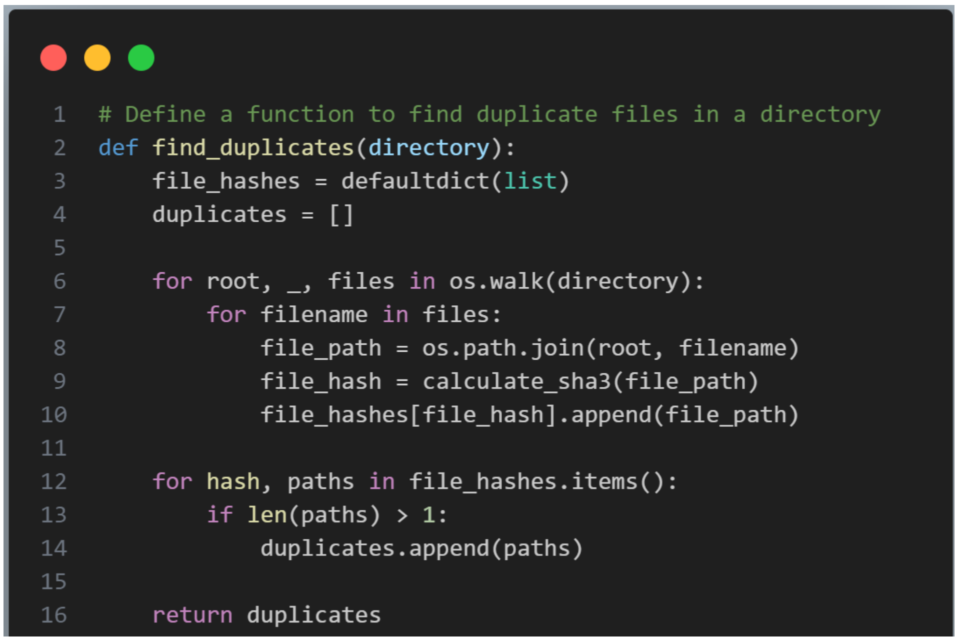
- Implement a script or use a deduplication tool to calculate the SHA-3 hashes for all files.



```python
# Define a function to calculate the SHA-3 hash of a file's content
def calculate_sha3(file_path):
    sha3 = hashlib.sha3_256()
    with open(file_path, 'rb') as file:
        while True:
            data = file.read(65536)  # Read the file in 64 KB chunks
            if not data:
                break
            sha3.update(data)
    return sha3.hexdigest()

```

4. **Identify Duplicates**:

- Compare the generated file hashes to identify duplicate files. Deduplication tools typically maintain a record of file hashes to detect duplicates efficiently.

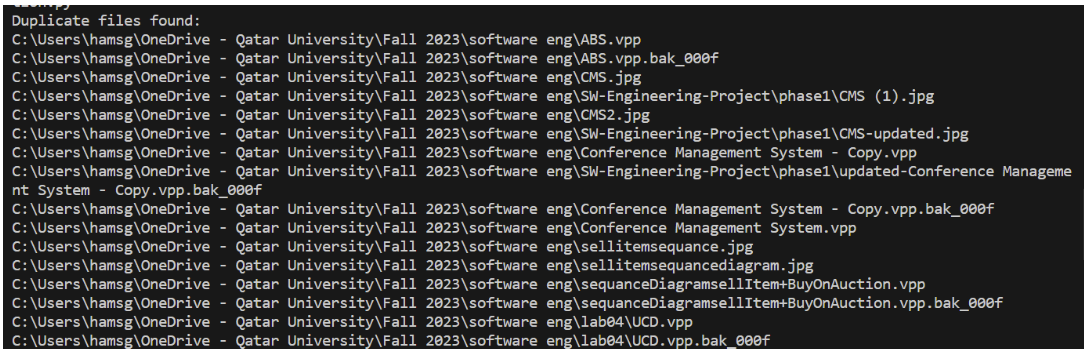


```python
# Define a function to find duplicate files in a directory
def find_duplicates(directory):
    file_hashes = defaultdict(list)
    duplicates = []

    for root, _, files in os.walk(directory):
        for filename in files:
            file_path = os.path.join(root, filename)
            file_hash = calculate_sha3(file_path)
            file_hashes[file_hash].append(file_path)

    for hash, paths in file_hashes.items():
        if len(paths) > 1:
            duplicates.append(paths)

    return duplicates
```

5. **Data Integrity Check**:
   - Verify that the deduplication process has not caused any data loss or corruption. The use of strong hash functions like SHA-3 helps ensure data integrity.

6. **Output the file name:**

   - List all the duplicated files that have been found.



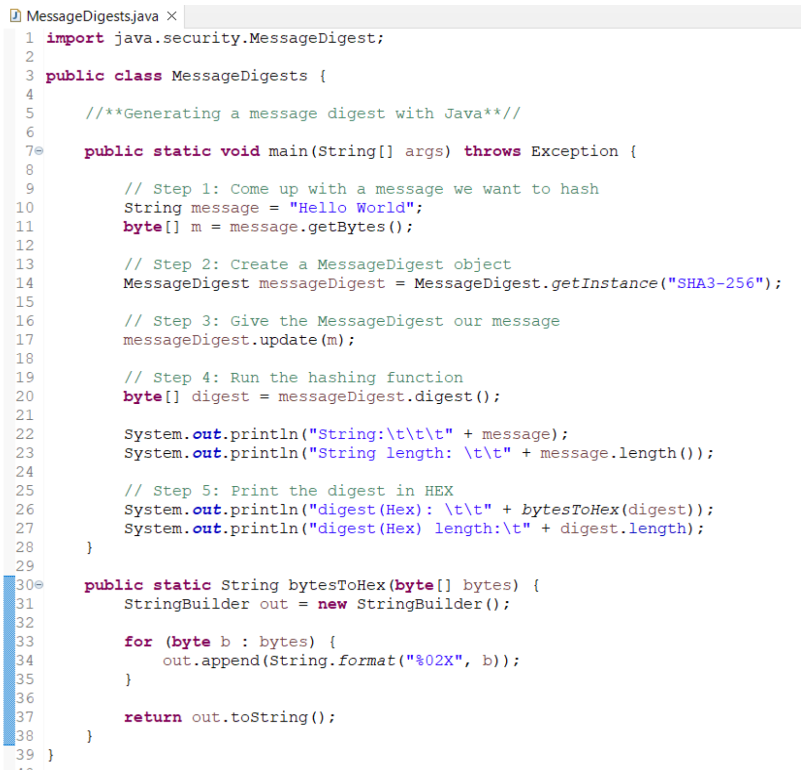
```
Duplicate files found:
C:\Users\hamsg\OneDrive - Qatar University\Fall 2023\software eng\ABS.vpp
C:\Users\hamsg\OneDrive - Qatar University\Fall 2023\software eng\ABS.vpp.bak_000f
C:\Users\hamsg\OneDrive - Qatar University\Fall 2023\software eng\CMS.jpg
C:\Users\hamsg\OneDrive - Qatar University\Fall 2023\software eng\SW-Engineering-Project\phase1\CMS (1).jpg
C:\Users\hamsg\OneDrive - Qatar University\Fall 2023\software eng\CMS2.jpg
C:\Users\hamsg\OneDrive - Qatar University\Fall 2023\software eng\SW-Engineering-Project\phase1\CMS-updated.jpg
C:\Users\hamsg\OneDrive - Qatar University\Fall 2023\software eng\Conference Management System - Copy.vpp
C:\Users\hamsg\OneDrive - Qatar University\Fall 2023\software eng\SW-Engineering-Project\phase1\updated-Conference Manageme
nt System - Copy.vpp.bak_000f
C:\Users\hamsg\OneDrive - Qatar University\Fall 2023\software eng\Conference Management System - Copy.vpp.bak_000f
C:\Users\hamsg\OneDrive - Qatar University\Fall 2023\software eng\Conference Management System.vpp
C:\Users\hamsg\OneDrive - Qatar University\Fall 2023\software eng\sellitemsequance.jpg
C:\Users\hamsg\OneDrive - Qatar University\Fall 2023\software eng\sellitemsequancediagram.jpg
C:\Users\hamsg\OneDrive - Qatar University\Fall 2023\software eng\sequanceDiagramsellItem+BuyOnAuction.vpp
C:\Users\hamsg\OneDrive - Qatar University\Fall 2023\software eng\sequanceDiagramsellItem+BuyOnAuction.vpp.bak_000f
C:\Users\hamsg\OneDrive - Qatar University\Fall 2023\software eng\lab04\UCD.vpp
C:\Users\hamsg\OneDrive - Qatar University\Fall 2023\software eng\lab04\UCD.vpp.bak_000f
```

3.3 Demonstration on Message Digests

Message Digests is a string created by using SHA3 algorithm (one way function). It helps to ensure integrity of message between two parties.

We can generate a message digest using MassegeDigest class in Java from java. Security package.
Steps:

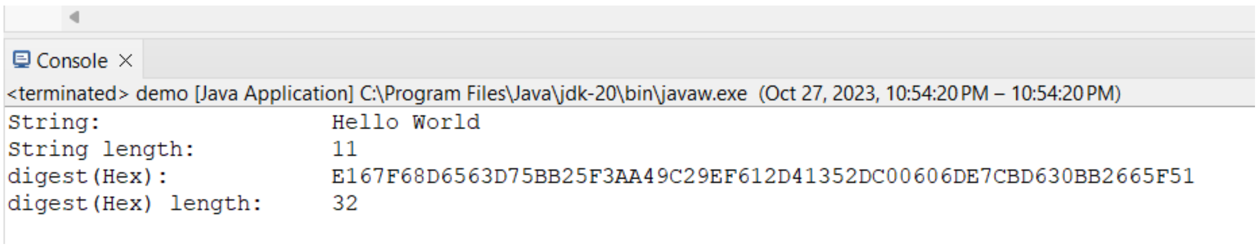
1. Create a message.
2. Create an object for MessageDigest.
   - Using a method called *getInstance()* from MessageDigest class.
3. Pass the message to MessageDigest object.
   - *messageDigest.update(message)*
4. Call the digest method from class MessageDigest.
   - *messageDigest.digest(),* it returns the digest in byte array form.
5. Print Digest in hex.
   - Convert the message digest from byte to hex.



MessageDigests.java

```java
import java.security.MessageDigest;

public class MessageDigests {

    //**Generating a message digest with Java**//

    public static void main(String[] args) throws Exception {

        // Step 1: Come up with a message we want to hash
        String message = "Hello World";
        byte[] m = message.getBytes();

        // Step 2: Create a MessageDigest object
        MessageDigest messageDigest = MessageDigest.getInstance("SHA3-256");

        // Step 3: Give the MessageDigest our message
        messageDigest.update(m);

        // Step 4: Run the hashing function
        byte[] digest = messageDigest.digest();

        System.out.println("String:\t\t\t" + message);
        System.out.println("String length: \t\t" + message.length());

        // Step 5: Print the digest in HEX
        System.out.println("digest(Hex): \t\t" + bytesToHex(digest));
        System.out.println("digest(Hex) length:\t" + digest.length);
    }

    public static String bytesToHex(byte[] bytes) {
        StringBuilder out = new StringBuilder();

        for (byte b : bytes) {
            out.append(String.format("%02X", b));
        }

        return out.toString();
    }
}
```

Output:



Console ×

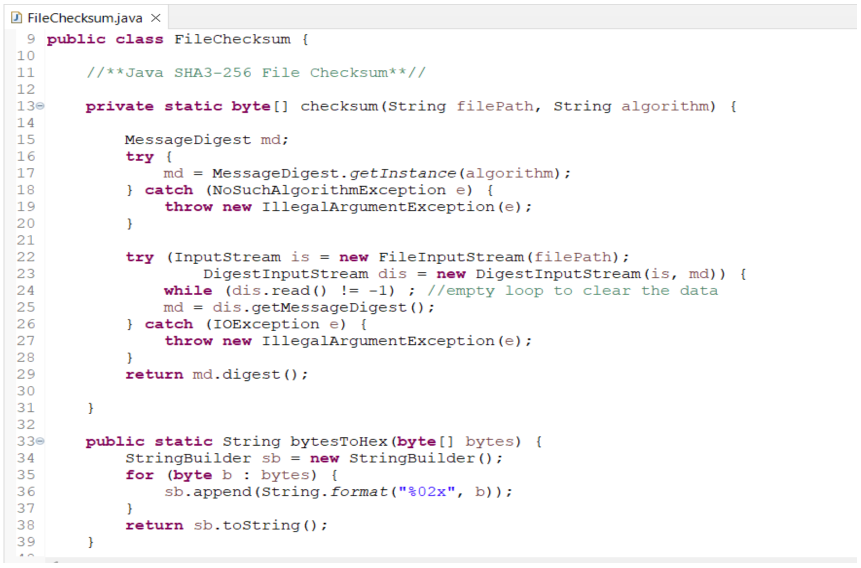
```
<terminated> demo [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe  (Oct 27, 2023, 10:54:20 PM – 10:54:20 PM)
String:                 Hello World
String length:          11
digest(Hex):            E167F68D6563D75BB25F3AA49C29EF612D41352DC00606DE7CBD630BB2665F51
digest(Hex) length:     32
```

3.4 Demonstration on File Checksum

File Checksum is a hash created by using SHA3 algorithm. It is used to verify if a file or data has been modified (ensure integrity). For example, if a file has been modified the file checksum will be changed, and then we will know if the file has been modified or it is not the same as original file.

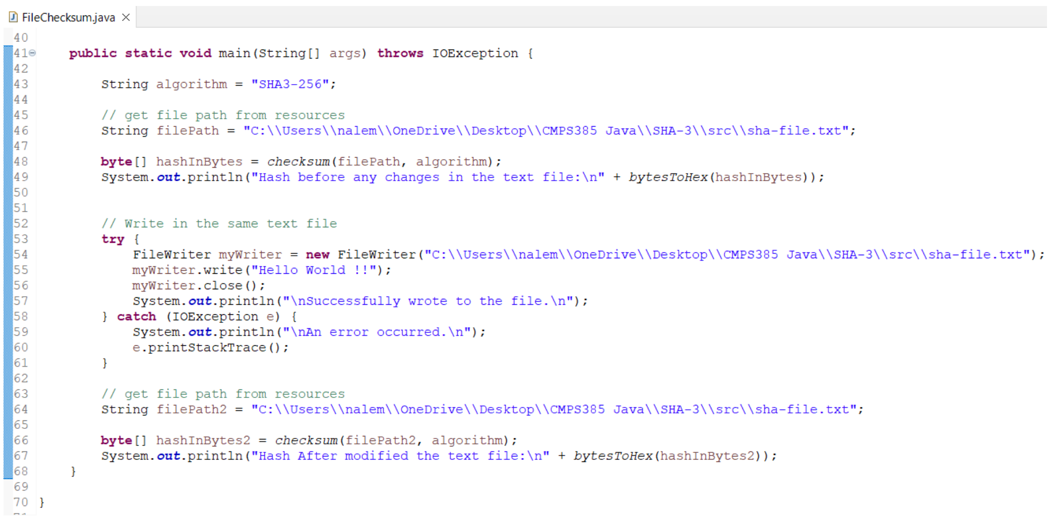
We can create a file checksum using MessageDigest class in Java from java.security package.
Steps:

1- Create an object of MessageDigest.
   - Using a method called *getInstance()* from MessageDigest class
2- Read the file.
3- Create a hash using SHA3 algorithm.
   - Using a method called *digest(),* it returns the hash in byte array form
4- Print hash in hex.
   - Convert the hash from byte to hex



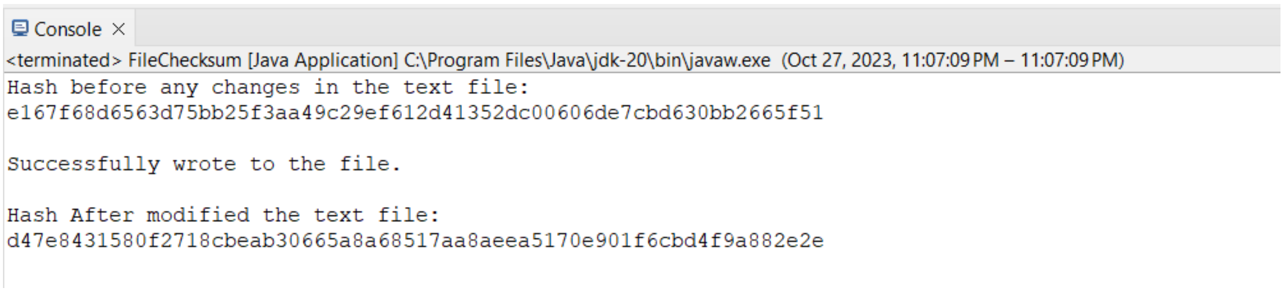
FileChecksum.java ×

```java
public class FileChecksum {

    //**Java SHA3-256 File Checksum**//

    private static byte[] checksum(String filePath, String algorithm) {

        MessageDigest md;
        try {
            md = MessageDigest.getInstance(algorithm);
        } catch (NoSuchAlgorithmException e) {
            throw new IllegalArgumentException(e);
        }

        try (InputStream is = new FileInputStream(filePath);
                DigestInputStream dis = new DigestInputStream(is, md)) {
            while (dis.read() != -1) ; //empty loop to clear the data
            md = dis.getMessageDigest();
        } catch (IOException e) {
            throw new IllegalArgumentException(e);
        }
        return md.digest();

    }

    public static String bytesToHex(byte[] bytes) {
        StringBuilder sb = new StringBuilder();
        for (byte b : bytes) {
            sb.append(String.format("%02x", b));
        }
        return sb.toString();
    }
```



```java
    public static void main(String[] args) throws IOException {

        String algorithm = "SHA3-256";

        // get file path from resources
        String filePath = "C:\\Users\\nalem\\OneDrive\\Desktop\\CMPS385 Java\\SHA-3\\src\\sha-file.txt";

        byte[] hashInBytes = checksum(filePath, algorithm);
        System.out.println("Hash before any changes in the text file:\n" + bytesToHex(hashInBytes));


        // Write in the same text file
        try {
            FileWriter myWriter = new FileWriter("C:\\Users\\nalem\\OneDrive\\Desktop\\CMPS385 Java\\SHA-3\\src\\sha-file.txt");
            myWriter.write("Hello World !!");
            myWriter.close();
            System.out.println("\nSuccessfully wrote to the file.\n");
        } catch (IOException e) {
            System.out.println("\nAn error occurred.\n");
            e.printStackTrace();
        }

        // get file path from resources
        String filePath2 = "C:\\Users\\nalem\\OneDrive\\Desktop\\CMPS385 Java\\SHA-3\\src\\sha-file.txt";

        byte[] hashInBytes2 = checksum(filePath2, algorithm);
        System.out.println("Hash After modified the text file:\n" + bytesToHex(hashInBytes2));
    }

}
```

output:



```
<terminated> FileChecksum [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe (Oct 27, 2023, 11:07:09 PM – 11:07:09 PM)
Hash before any changes in the text file:
e167f68d6563d75bb25f3aa49c29ef612d41352dc00606de7cbd630bb2665f51

Successfully wrote to the file.

Hash After modified the text file:
d47e8431580f2718cbeab30665a8a68517aa8aeea5170e901f6cbd4f9a882e2e
```

3.5 Demonstration on Digital Signatures

Digital signatures play a crucial role in the establishment of authenticity and non-repudiation when working with any form of data. Using digital signatures, a private key is generated, meaning that only its holder can access the data and thus ensure authenticity. Non-repudiation is satisfied when documents that have been digitally signed attest to the identity and purpose of the signer. The choice of hash function is crucial for digital signatures since the hash function's security determines the signature's overall level of security. Thanks to its standardization and cryptographic capabilities, SHA-3 is seen as a promising candidate for digital signatures.

There are various reasons as to why the SHA-3 algorithm is preferred in digital signatures as opposed to other secure hashing algorithms. Most importantly, SHA-3 has undergone extensive testing and evaluation by many cryptanalysts and since its design is based upon solid mathematical principles it is determined to be a secure choice for digital signatures. As mentioned previously, SHA-3 has very high resistance in terms of collisions; the possibility of different messages producing the same hash value is extremely low. Furthermore, the algorithm prevents signature forgery (replicating a signature to impersonate another individual) as a consequence of the inability to computationally reverse the hash function and find the original message from its hash value. Due to SHA-3's powerful avalanche effect, a slight modification to the input message can produce a noticeably different hash value. Because of this, it is more difficult for attackers to change the message and maintain the same hash. Lastly, SHA-3 was designed to provide great performance both on security as well as computational efficiency.

In this case, we need to utilize the SHA-3 algorithm to hash the desired message. Along with the SHA-3 algorithm we should implement the DSA algorithm, which is used to generate digital signatures, as well as authenticate the sender of the message, and prevent message modifications.

Setting up the environment:
For this part, we used Visual Studio Code as our source-code editor, to be able to work with Python. The choice for the use of Python stems directly from its simplicity and readability, especially for complex algorithms such

as DSA. Additionally, Python offers a wide range of cryptographic libraries such as ‘cryptography’ and ‘pycryptodome’ that we had to take into consideration for the functionality of our code.

Analytically,

1. Install Visual Studio Code
2. Install latest version of Python
3. Select Python interpreter when saving/working with a file.
4. Update PIP; pip is the Python package manager

   ```
   python -m pip install --upgrade pip
   ```

5. Install PyCryptodome to be able to use cryptographic functions. Open the terminal on Visual Studio Code and type:

   ```
   pip install pycryptodome
   ```

Replicating the demonstration

The following steps of the code can be implemented for any Secure Hash Algorithm, but it is preferred to use the SHA-3 for additional security, as the generated hash digest is of longer length, thus making it more resistant to collision attacks.

As already stated, our code for the demonstration consists of two separate algorithms: SHA-3 and DSA, that are interconnected and work together to provide a digital signature. By using the digital signature, we ensure that if the file that contains the message is modified by an illegitimate user, a message will appear, that proves that the signature does in fact, not match the original. While the SHA-3 algorithm is used to hash the document to ensure data integrity, the DSA algorithm creates and verifies the digital signatures to provide authenticity and non-repudiation for the signed document.

Steps:

1. Importing libraries that will be of use for our implementation

   ```
   from hashlib import sha3_256

   from Crypto.Util.number import *

   from random import *
   ```

   a. ‘hashlib’ module is used for the SHA-3 hashing. There are many versions of the SHA-3 algorithm that can be used: sha3_224, sha3_512, sha3_384 with the key difference between them being their fixed-length output in bits. However, sha3_256 provides a reasonable length of 256 bits (32 bytes) which contributes to practical representation of a hash value in hexadecimal form.
   b. ‘Crypto.Util.number’ module provides multiple cryptographic functionalities such as data encryption/decryption and digital signatures. Two of the functions that can be accessed by this module and we have used in the code are “getPrime( )” which generates a prime number of the specified number of bits, and “inverse(a, m)” that calculates the modular inverse of a modulo m. The inverse function is commonly used for encryption and decryption.
   c. ‘random’ module includes functions such as ‘randint()’ that allows us to generate keys of random integer values.

2. Function that generates the hash value of a given message. Analytically, ‘message is converted to bytes using ‘.encode(‘utf-8’)’, ‘sha3_256’ calculates the hash of the message and ‘.hexdigest()’ method is called on the hash object to retrieve the hexadecimal representation of the hash which is returned from the function.

   ```
   def sh3Alg(message):

       hashed=sha3_256(message.encode('utf-8')).hexdigest()

       return hashed
   ```

3. This function uses the ‘getPrime(5)’ function, which is commonly used for DSA, to generate a small prime number 'q' with 5 bits. Then ‘p’ is initialized as None and goes into a loop that generates the prime number 'p' using ten bits. The loop terminates only when a value for 'p' is found for which the

condition (p - 1) % q == 0 is satisfied. Then ‘p’ and ‘q’ which are the prime divisors get printed. The user is asked to enter the private key ‘h’ which is also used to calculate ‘g’. Based on DSA parameter generation processes, ‘g’ is computed by the following: g = h^((p-1)/q) % p. Lastly, all parameters are returned by the function. It is worth noting that ‘p’, ‘g’, ‘q’ are public parameters while ‘h’ is private.

```
def paramGeneration():
q=getPrime(5)
p=getPrime(10)
while((p-1)%q!=0):
p=getPrime(10)
q=getPrime(5)
print('prime divisor (q)',q)
print('prime divisor (p)',p)
flag=True
while(flag):
h=int(input("Enter an integer between 1 and p-1(h): "))
if(1<h<(p-1)):
g=1
while(g==1):
g=pow(h,int((p-1)/q))%p
flag=False
else:
print("wrong entry")
    print("value of g is : ",g)
    return(p,q,g)
```

4. The following function generates two random keys, a public and private one using the ‘random’ library.

```
def per_user_key(p,q,g):
    x=randint(1,q-1)
    print("randomly chosen x(Private key) is",x)
    y=pow(g,x)%p
    print("randomly chosen y(public key): ",y)
    return(x,y)
```

5. In this case the function generates the signature of a given ‘name’ file. The content of the file is read and its hash function is generated. It returns a tuple ‘(r,s,k)’ that expresses the DSA signature components.

```
def signature(name,p,q,g,x):
    with open(name) as file:
        text=file.read()
        hash_component=sh3Alg(text)
        print("hash of document sent is: ", hash_component)
    r=0
    s=0
    while(s==0 or r==0):
        k=randint(1,q-1)
        r=((pow(g,k))%p)%q
        i=inverse(k,q)
        hashed=int(hash_component,16)
        s=(i*(hashed+(x*r)))%q
    return(r,s,k)
```

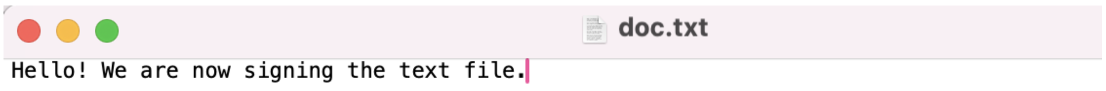
6. Lastly, this function represents the verification process for the signature. It opens the text file and reads its content and then computes the hash by calling the ‘sh3Alg’. ‘u1’ and ‘u2’ are generated to calculate v. If ‘r’ is equal to ‘v’ it implies that the original signature and the one entered are the same and thus the entity accessing and modifying the file is authorized. Otherwise, a warning is printed that states that the signature is invalid.

```
def verification(name,p,q,g,r,s,y):
    with open(name) as file:
        text=file.read()
        hash_component=sh3Alg(text)
        print("hash of document is: ",hash_component)
    w=inverse(s,q)
    print("value of w is: ",w)
    hashed=int(hash_component,16)
    u1=(hashed*w)%q
    u2=(r*w)%q
    v=((pow(g,u1)*pow(y,u2))%p)%q
    print("value of u1: ",u1)
    print("value of u2: ",u2)
    print("value of v: ",v)
    if(v==r):
        print("signature is valid")
    else:
        print("signature is invalid")
```

7. The last piece of our code, is calling all the functions that were previously created, to generate all the necessary parameters and print them on the terminal.

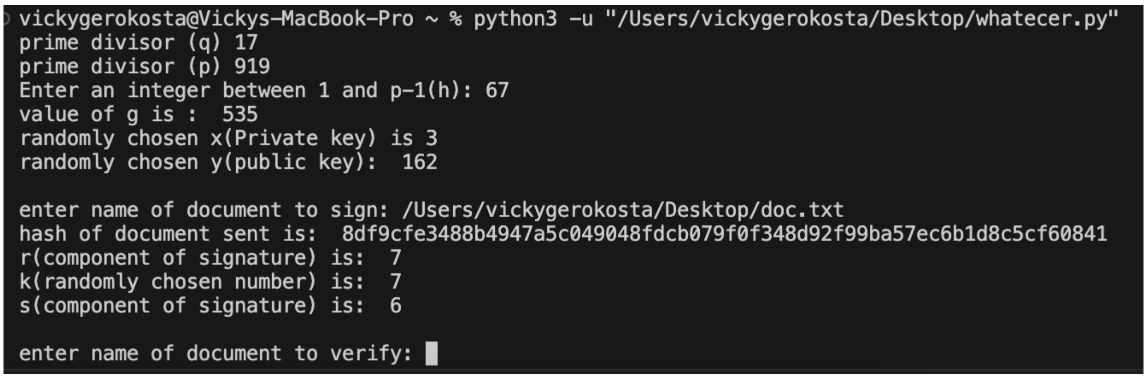
```
global_var=paramGeneration()

keys=per_user_key(global_var[0],global_var[1],global_var[2])

print()

file_name=input("enter name of document to sign: ")

components=signature(file_name,global_var[0],global_var[1],global_va
r[2],keys[0])

print("r(component of signature) is: ",components[0])

print("k(randomly chosen number) is: ",components[2])

print("s(component of signature) is: ",components[1])

print()

file_name=input("enter name of document to verify: ")

verification(file_name,global_var[0],global_var[1],global_var[2],com
ponents[0],components[1],keys[1])
```

- To test our program, we have created a text file and added its content.



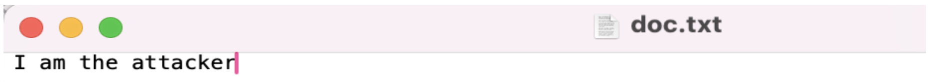
```
Hello! We are now signing the text file.
```

- After running the program in Python:



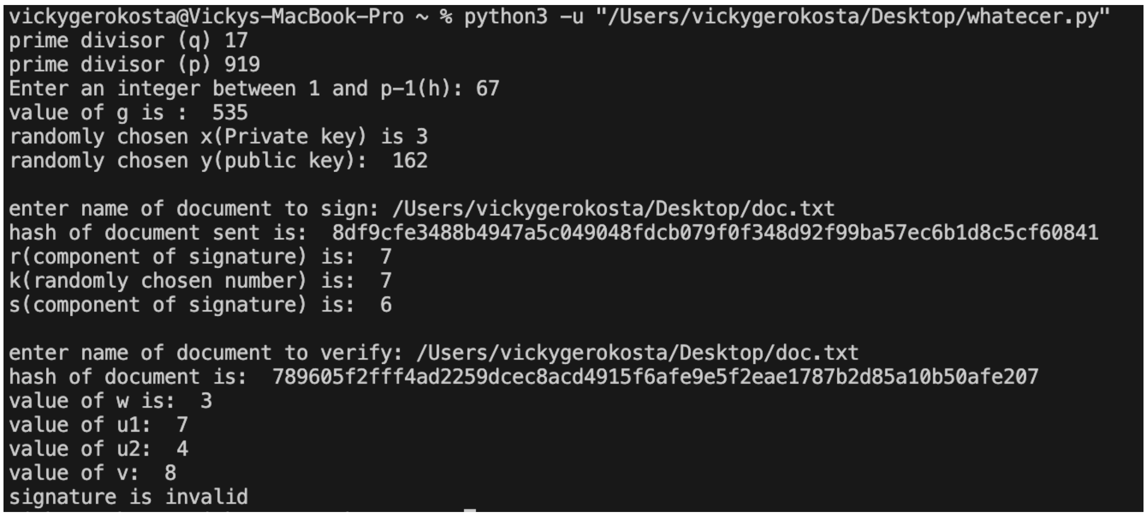
```
vickygerokosta@Vickys-MacBook-Pro ~ % python3 -u "/Users/vickygerokosta/Desktop/whatecer.py"
prime divisor (q) 17
prime divisor (p) 919
Enter an integer between 1 and p-1(h): 67
value of g is :  535
randomly chosen x(Private key) is 3
randomly chosen y(public key):  162

enter name of document to sign: /Users/vickygerokosta/Desktop/doc.txt
hash of document sent is:  8df9cfe3488b4947a5c049048fdcb079f0f348d92f99ba57ec6b1d8c5cf60841
r(component of signature) is:  7
k(randomly chosen number) is:  7
s(component of signature) is:  6

enter name of document to verify: 
```

- Now, to check the verification process, we have to act like the attacker and modify the file.



```
I am the attacker
```

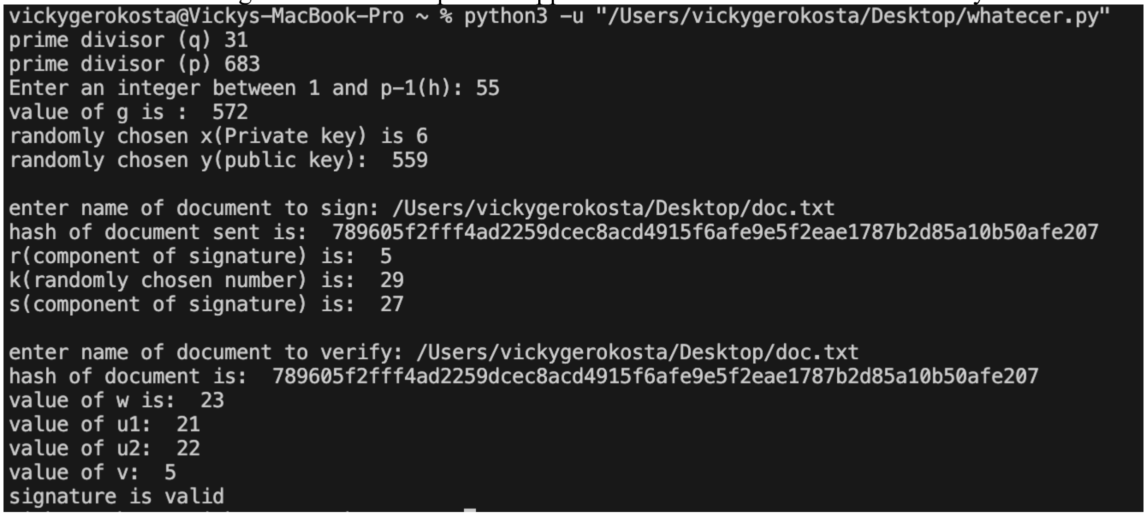
- After entering the file name again:



```
vickygerokosta@Vickys-MacBook-Pro ~ % python3 -u "/Users/vickygerokosta/Desktop/whatecer.py"
prime divisor (q) 17
prime divisor (p) 919
Enter an integer between 1 and p-1(h): 67
value of g is :  535
randomly chosen x(Private key) is 3
randomly chosen y(public key):  162

enter name of document to sign: /Users/vickygerokosta/Desktop/doc.txt
hash of document sent is:  8df9cfe3488b4947a5c049048fdcb079f0f348d92f99ba57ec6b1d8c5cf60841
r(component of signature) is:  7
k(randomly chosen number) is:  7
s(component of signature) is:  6

enter name of document to verify: /Users/vickygerokosta/Desktop/doc.txt
hash of document is:  789605f2fff4ad2259dcec8acd4915f6afe9e5f2eae1787b2d85a10b50afe207
value of w is:  3
value of u1:  7
value of u2:  4
value of v:  8
signature is invalid
```

As we can notice, the hash values of the document before and after modification are different, and the signatures do indeed not match, hence the 'signature is invalid'.

- The following would be the output that appears in the terminal if we do not modify the text file.



```
vickygerokosta@Vickys-MacBook-Pro ~ % python3 -u "/Users/vickygerokosta/Desktop/whatecer.py"
prime divisor (q) 31
prime divisor (p) 683
Enter an integer between 1 and p-1(h): 55
value of g is :  572
randomly chosen x(Private key) is 6
randomly chosen y(public key):  559

enter name of document to sign: /Users/vickygerokosta/Desktop/doc.txt
hash of document sent is:  789605f2fff4ad2259dcec8acd4915f6afe9e5f2eae1787b2d85a10b50afe207
r(component of signature) is:  5
k(randomly chosen number) is:  29
s(component of signature) is:  27

enter name of document to verify: /Users/vickygerokosta/Desktop/doc.txt
hash of document is:  789605f2fff4ad2259dcec8acd4915f6afe9e5f2eae1787b2d85a10b50afe207
value of w is:  23
value of u1:  21
value of u2:  22
value of v:  5
signature is valid
```

4. Conclusion

In conclusion, SHA-3, the Secure Hash Algorithm-3, is a groundbreaking achievement in cryptographic hash functions. Its historical journey, from the need for an improved successor to SHA-2 to its selection as the winner of the SHA-3 Cryptographic Hash Algorithm competition, showcases its significance in enhancing data security. SHA-3 operates on innovative sponge construction, offering a robust foundation for secure data processing.

Compared to SHA-2, SHA-3 distinguishes itself through its unique design principles and enhanced security features. It is particularly resistant to different cryptographic attacks, making it a compelling choice for modern cryptographic applications. The efficiency of SHA-3 in both hardware and software implementation, solidifies its position as a cost-effective solution for a wide range of applications.

Moreover, SHA-3's practical utility shows through its role in real-world scenarios, from file integrity verification and digital signatures to file deduplication, password hashing, and its integration into the world of cryptocurrencies. The adoption of SHA-3 in these contexts reflects its importance in ensuring data authenticity, security, and efficiency.

As the field of data security continues to evolve, SHA-3 represents a significant milestone in safeguarding digital assets and maintaining trust in data transactions. Its unique characteristics, efficient implementation, and real-world applications make it a cornerstone in contemporary cryptography, addressing the ever-growing need for secure and reliable data processing and protection. SHA-3 serves as a testament to the continuous progress in the field of cryptographic hash functions and the relentless pursuit of data security excellence.

References

1. Bitkan - buy Bitcoin, Ethereum and altcoins with ease. BitKan.com. (n.d.). https://bitkan.com/learn/where-is-sha-3-used-how-does-sha-3-work-6052

2. Cernera, G. (2022, April 22). Generating a message digest with Java, explained step-by-step with pictures. Medium. https://gregorycernera.medium.com/generating-a-message-digest-with-java-explained-step-by-step-with-pictures-b9be6c6e5036

3. Chemejon. (2022, June 10). In-depth visual breakdown of the SHA-3 cryptographic hashing algorithm - jon's blog. Jon's Blog - A Collection of Things I Have Learned. https://chemejon.io/sha-3-explained/

4. Dworkin, M. J. (2015). SHA-3 standard: Permutation-based hash and extendable-output functions. SHA-3 STANDARD: PERMUTATION-BASED HASH AND EXTENDABLE OUTPUT FUNCTIONS. https://doi.org/10.6028/nist.fips.202

5. Gupta, L. (2023, March 14). Java file checksum - MD5 and SHA-256 hash examples. HowToDoInJava. https://howtodoinjava.com/java/java-security/sha-md5-file-checksum-hash/

6. Hombergs, T. (2021, November 14). Creating hashes in Java. Reflectoring. https://reflectoring.io/creating-hashes-in-java/
7. Java create and write to files. (n.d.). https://www.w3schools.com/java/java_files_create.asp
8. Java cryptography - message digest. Online Tutorials, Courses, and eBooks Library. (n.d.). https://www.tutorialspoint.com/java_cryptography/java_cryptography_message_digest.htm
9. Java SHA3-512 hash using Apache Commons codec. Java SHA3 512 Hash using Apache Commons Codec. (n.d.). https://simplesolution.dev/java-sha3-512-hash-using-apache-commons-codec/
10. Jones, S. (2019a, May 20). What you need to know about SHA-3 for embedded system security. Electronic Design. https://www.electronicdesign.com/technologies/embedded/article/21808025/what-you-need-to-know-about-sha3-for-embedded-system-security
11. Mkyong, Suraj, Author mkyong 5 years ago Reply to Suraj This is Hashing, Raj, P., Lionel, Anu, Pancho, Ayoub, Author mkyong 5 years ago Reply to Ayoub article is updated, Savani, Chhorn, Dannie, Aks1316, Newson, S., Author mkyong 5 years ago Reply to Steve Newson Java 9, Sandeep, Davidovi?, Z., Husta, G., Tereschuk, Y., … Rishabh. (2020a, June 16). Java SHA-256 and SHA3-256 hashing example. Mkyong.com. https://mkyong.com/java/java-sha-hashing-example/

12. National Institute of Standards and Technology. (2015, August 4). SHA-3 standard: Permutation-based hash and extendable-output functions. CSRC. https://csrc.nist.gov/pubs/fips/202/final

13. Oursky. (2022, April 7). Password hashing and Salting explained. Authgear. https://www.authgear.com/post/password-hashing-salting

14. P, K. S., K, M., & Kumar, S. (2015). IDENTIFICATION OF DUPLICATE FILES USING SHA-3 ALGORITHM, 10(0973–4562), 8112–8116.
15. Secure hash algorithms. Secure Hash Algorithms - Practical Cryptography for Developers. (n.d.). https://cryptobook.nakov.com/cryptographic-hash-functions/secure-hash-algorithms

16. Why aren't we using SHA-3? CSO Online. (2018, February 21). https://www.csoonline.com/article/564619/why-arent-we-using-sha3.html
17. *Digital Signature Algorithm (DSA) in Cryptography: How It Works & More*. (n.d.). https://www.simplilearn.com/tutorials/cryptography-tutorial/digital-signature-algorithm#:~:text=Parameter%20Generation,as%20per%20Original%20DSS%20length.
18. *Digital Signature Algorithm (DSA) In Python From Scratch*. (n.d.). https://sefiks.com/2023/06/14/digital-signature-algorithm-dsa-in-python-from-scratch/.
19. *hashlib — Secure hashes and message digests*. (n.d.). https://docs.python.org/3/library/hashlib.html.
20. *SHA3 in Python*. (n.d.). https://www.geeksforgeeks.org/sha3-in-python/.