SCHOOL OF COMPUTING, UUM COLLEGE OF ARTS AND SCIENCES

SKIC2113 CRYPTOGRAPHY

(SEMESTER A222)

GROUP ASSIGNMENT 2:

PHASE 4 & 5

TOPIC:

DSA IMPLEMENTATION WITH SHA3-512, RSA AND PKCS1v15

LECTURER:

PROF. MADYA TS. DR. NORLIZA BINTI KATUK

PREPARED BY:

GROUP BLOWFISH

NAME	MATRIC NO.
AINUR HANIM BINTI ABDUL HALIM	288091
TAN ZHI XIAN	284333
NURIN ANDRIANA BINTI MOHD SUBRI	287957
NURIN IZZAH BINTI ISHAK	288063

LINK OF VIDEO DEMONSTRATION: https://youtu.be/Nqv-sietynE

DATE OF SUBMISSION:

19 JULY 2023

TABLE OF CONTENTS

SECTION 1: INTRODUCTION

1.1 Background of Project

The project focuses on creating a DSA (Digital Signature Algorithm) using different algorithms and comparing its performance with the original DSA implementation. Digital signatures play a crucial role in ensuring data integrity, authentication, and non-repudiation *(Asep Saepulrohman, 2021)*. It is commonly employed in various applications that require secure communication, data transmission verification, and electronic document signing.

According to the key aspects, DSA is based on public-key cryptography, which utilizes a key pair consisting of a private key and a corresponding public key *(Harn, 1998)*. The private key is kept secret and is used to generate digital signatures, while the public key is widely distributed and used for verifying the signatures. It generates digital signatures for data using the private key. A digital signature is a cryptographic mechanism that binds the identity of the signer to the data. The recipient can verify the signature using the corresponding public key.

Moreover, DSA utilizes a cryptographic hash function to create a fixed-size hash value of the data to be signed. The hash function condenses the data into a shorter, fixed-length value that uniquely represents the original data. This ensures that even a small change in the data will produce a significantly different hash value. Besides, it also involves the generation of large prime numbers to establish the parameters of the cryptographic scheme. These prime numbers are carefully chosen to ensure the security and strength of the algorithm.

Other than that, DSA relies heavily on the use of random numbers and non-deterministic algorithms. Randomness is crucial for generating unique private keys and random values during the signing process, ensuring the security and unpredictability of the signatures. The security of DSA depends on the length of the key parameters. Longer key lengths provide stronger security but require more computational resources. It is essential to use sufficiently long key lengths to protect against attacks.

In summary, its effectiveness and security have been extensively analyzed and proven, making it a trusted and widely adopted algorithm in the field of cryptography. The algorithm has been subject to rigorous testing and evaluation, including cryptanalysis and formal verification methods, to ensure its resilience against various attacks.

1.2 Objectives

The project aims to achieve four primary objectives, which are:

(a) Develop a DSA using different algorithms for the hash function, public key, and signature components.

(b) Measure the time complexity and memory usage of the newly developed DSA by conducting experiments using different message sizes.

(c) Compare the time and memory usage of the newly developed DSA with the original DSA implementation.

(d) Analyze and interpret the results obtained from the performance evaluation to make informed recommendations about the most suitable algorithmic choices for different scenarios.

The project enables us to deepen our understanding of the practical implementation of DSA and its underlying algorithms. By evaluating the performance of different algorithms within the DSA framework, we gain insights into the real-world applicability of different algorithmic choices and their impact on the overall system's performance.

1.3 Algorithms Chosen for DSA and The Reasons

In our DSA (Digital Signature Algorithm) implementation, the algorithms selected are SHA-3 512 for hashing, RSA for the public key, and pkcs1_15 (PKCS1v15) for the signature.

1.3.1 Hashing - SHA-3 512

For the hashing component, SHA-3 512 has been used in the DSA program. SHA-3 (Secure Hash Algorithm 3) is a family of cryptographic hash functions designed by the National Security Agency (NSA). SHA-3 offers several variants with different output sizes, one of them refers to SHA-3 512.

For further clarification, SHA-3 512 is based on the Keccak algorithm, which was developed by a team of cryptographers led by Guido Bertoni, Joan Daemen, and Gilles Van Assche. It was selected as the winner of the National Institute of Standards and Technology (NIST) hash function competition in 2012 and standardized as SHA-3 *(Nist, 1992)*. Besides, SHA-3 512 operates as a hash function, taking an input message of any length and producing a fixed-size 512-bit (64-byte) hash value *(Wu & Li, 2017)*. The hash value is a unique representation of the input data.

Next, there are some key characteristics and reasons for choosing SHA-3 512 for the DSA program. Firstly, SHA-3 512 provides a high level of security against collision attacks and pre-image attacks *(Karmani et al., 2021)*. Thus, it is computationally infeasible to find two different inputs that produce the same hash value or to find the original input from its hash value.

Secondly, SHA-3 512 has been designed to resist various cryptographic attacks, including differential and linear attacks. These attacks aim to exploit mathematical properties or patterns in the algorithm to deduce information about the input data. However, it offers a strong level of security, making it suitable for secure applications.

Despite producing a longer hash value compared to other variants, SHA-3 512 maintains efficient performance. Its ability to process data in larger chunks allows for faster computation, resulting in high throughput. This efficiency is particularly beneficial for applications that require the hashing of large amounts of data, such as network protocols.

Furthermore, it is a standardized algorithm, widely accepted and used in many cryptographic systems. It has undergone extensive scrutiny and analysis by the cryptographic community, which increases confidence in its security.

In short, choosing SHA-3 512 ensures that the DSA program is utilizing a modern and future-proof cryptographic hash function. It provides a secure and efficient way to generate unique hash values for input data, which is essential for various cryptographic operations.

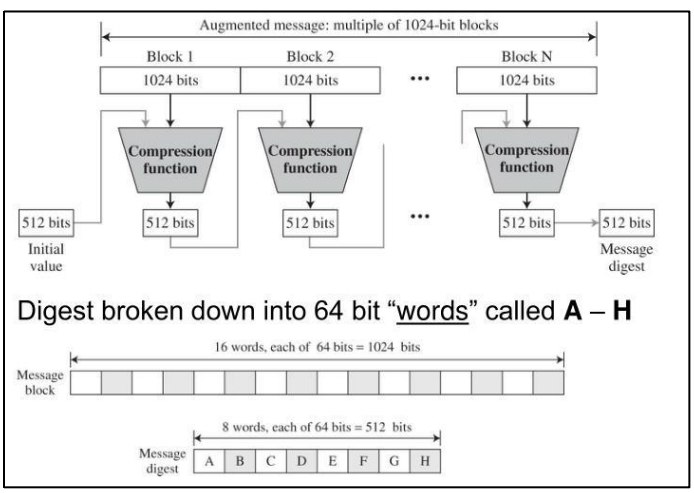


Figure 1: SHA-3 512 Hashing Diagram

1.3.2 Public Key – RSA

In our DSA implementation, we have selected Rivest-Shamir-Adleman (RSA) as the public key algorithm for generating the key pair. RSA is a widely used and well-established asymmetric encryption algorithm that provides strong security and efficient key operations.

RSA works based on the mathematical properties of large prime numbers. The key pair consists of a public key, which is used for encryption and signature verification, and a corresponding private key, which is kept secret and used for decryption and signature generation. The security of RSA is based on the difficulty of factoring large prime numbers. RSA generates a public-private key pair, where the public key is used for encryption or verification, and the private key is used for decryption or signing.

There are several reasons why one might choose RSA over other algorithms like DSA (Digital Signature Algorithm) or ECC (Elliptic Curve Cryptography) for generating a public key in a DSA program. Firstly, RSA's security is based on the computational complexity of factoring large prime numbers. Breaking RSA encryption requires finding the prime factors of a large number, which is currently considered computationally infeasible *(Wiener, 1990)*. This strong security property ensures the confidentiality and integrity of the data being signed and verified.

In addition to its security, RSA's support for longer key lengths enhances its resilience against brute-force attacks and advances in computational power *(Koç et al., 2021)*. Longer key lengths increase the computational effort required to break the encryption, providing a higher level of security. This attribute is particularly valuable when long-term security is a priority.

Next, RSA's widespread support in cryptographic libraries and systems makes it highly compatible and enables seamless integration into various applications and environments. This compatibility ensures that your DSA implementation can be easily utilized alongside existing cryptographic infrastructure. Leveraging the popularity and compatibility of RSA simplifies the deployment and interoperability of your DSA program.

Efficient key exchange is another advantage offered by RSA. It allows for the secure distribution and verification of public keys, enabling the authentication and integrity of digital signatures *(Kaur & Kaur, 2012)*. The efficient key exchange capabilities of RSA facilitate quick and secure establishment of communication channels between parties. This efficiency is crucial for ensuring the timely and secure exchange of information.

In summary, these advantages collectively enhance the security, compatibility, and efficiency of our DSA implementation, making RSA a suitable choice for generating the public key in our program.

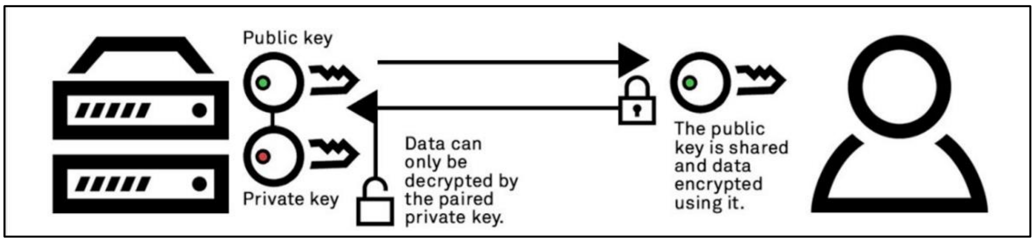


Figure 2: RSA key generation algorithm

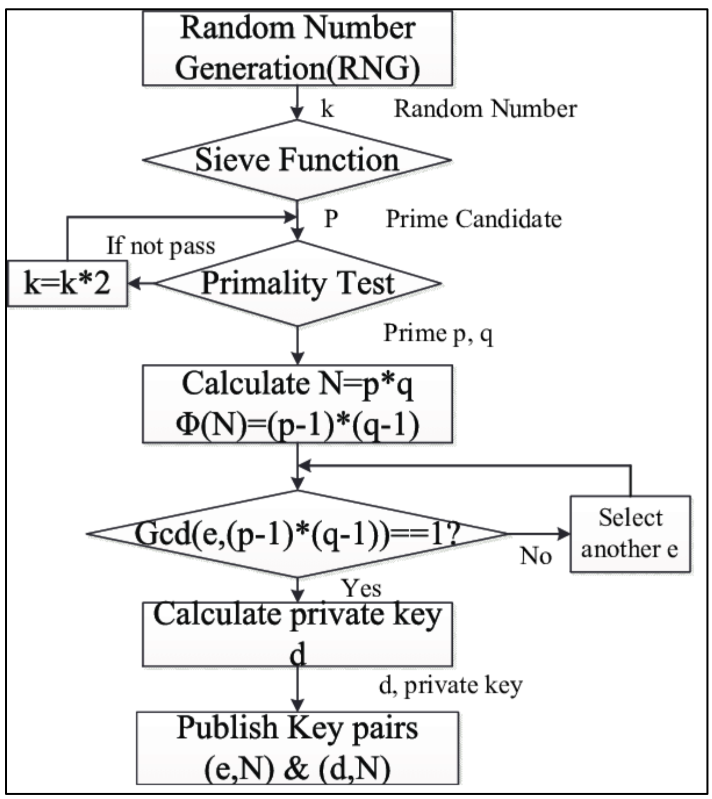


Figure 3: A common flow of RSA key pair generation

1.3.3 Signature – pkcs1_15

For the signature component in our DSA program, we have chosen Public Key Cryptography Standard 1.5 (pkcs1_15), which is a widely used signature scheme following the RSA-based signature padding defined in the PKCS #1 standard.

pkcs1_15 provides a standardized approach for generating digital signatures using RSA. It ensures the integrity and authenticity of the digital signatures by applying appropriate padding to the data being signed before applying the RSA encryption process *(Jager et al., 2018)*. In the context of RSA signatures, PKCS#1 v1.5 defines the padding scheme used when generating signatures using the RSA algorithm. The padding scheme serves multiple purposes, including adding randomness to the signature, ensuring a consistent signature length, and providing security against certain cryptographic attacks. Moreover, padding does not affect the length of the signature. The length of an RSA signature is determined by the key size used. For example, with a 2048-bit RSA key, the length of the signature will be 256 bytes (or 2048 bits). Increasing the length of the signature would require using a longer key, such as a 4096-bit key.

The input message or data is first processed using a cryptographic hash function, such as SHA-3 512, to produce a fixed-length digest of the data and signature. The digest is then padded with additional bytes to meet the desired signature length requirements. The padding scheme includes a predefined structure that adds randomness and specific formatting to the digest. The padded digest is combined with additional information, such as a hash function identifier and other parameters, to create the final data to be signed. The resulting data is encrypted using the RSA private key to generate the signature.

When considering the reasons for choosing pkcs1_15 for generating signatures in a DSA program, several factors come into play. Firstly, pkcs1_15 provides strong security guarantees, preventing unauthorized modification or forgery of digital signatures. The RSA-based padding scheme used in pkcs1_15 adds security layers to the signature generation process. It ensures that the signatures are resistant to known attacks, such as signature malleability, where an attacker can manipulate a valid signature to create a different but still valid signature. By incorporating the security measures of pkcs1_15, the DSA implementation can maintain the integrity and authenticity of the digital signatures, protecting against tampering and fraudulent activities.

Next, pkcs1_15 has been widely adopted and used in various cryptographic applications and protocols. Its usage in industry-standard cryptographic schemes and protocols demonstrates its reliability, effectiveness, and compatibility. By leveraging the established industry adoption of pkcs1_15, the DSA implementation benefits from the collective knowledge, scrutiny, and testing of the broader cryptographic community. It ensures that the signature generation process follows widely accepted practices and aligns with industry standards and recommendations.

Furthermore, the adherence to the well-defined PKCS #1 standard ensures clarity and consistency in the signature generation process across different systems and implementations. This standardization facilitates proper interoperability, promotes widespread adoption of the DSA implementation, and enables seamless integration with other cryptographic systems. By following a well-established standard like pkcs1_15, the DSA program ensures compatibility with various cryptographic libraries and systems, enhancing its versatility and practicality.

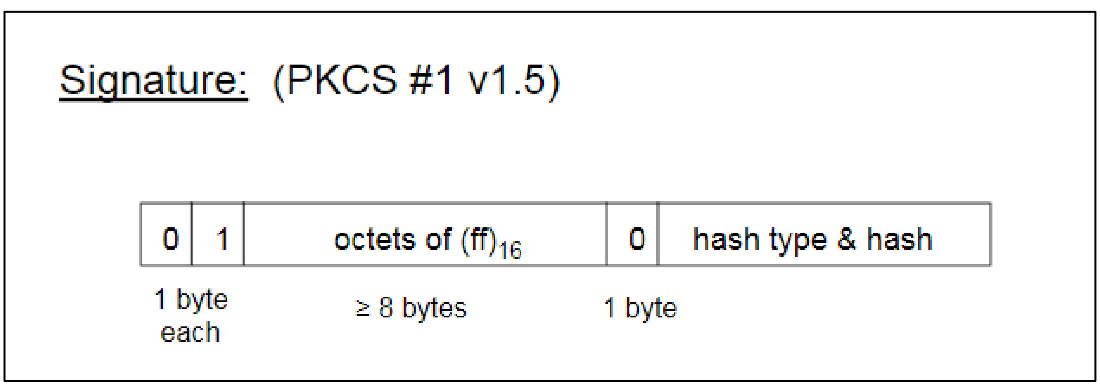


Figure 4: pkcs1_15 on signature

SECTION 2: EXPERIMENT

2.1 Machine Specification

In order to assess potential variations in the output of the DSA program, particularly regarding memory usage and time complexity, we acquired two machines from our group (Group Blowfish). Subsequently, an analysis was conducted to compare the results based on the differing machine specifications.

Table 1: Comparison of 2 machines used to run the DSA program

Component	Machine A	Machine B
Brand	ASUS	ACER
Model	ROG Strix G713IH	Swift SF314-42
CPU	AMD Ryzen 7 4800H with Radeon Graphics	AMD Ryzen 5 4500U with Radeon Graphics
GPU	NVIDIA GeForce GTX 1650	Graphics AMD Radeon (TM) Graphics
Memory RAM	8 GB DDR4	8 GB DDR4
Storage ROM	512 GB SSD	475 GB SSD
Operating System	Windows 11	Windows 11
Python Version	3.11.3	3.11.3
Python Libraries	cryptography 3.4.8 pycryptodome 3.18.0 pycrytodomex 3.18.0 pycryptodome-test-vectors 1.0.12 memory-profiler 0.61.0 exceptiongroup 1.0.4	cryptography 41.0.2 dsa 13.3.0 memory-profiler 0.61.0 profiler 0.1.0
Python IDEs	Visual Studio 2022	PyCharm Community Edition 2023.1.1

2.1.1 Cryptography and Memory Profiling Libraries Utilized in Python

Python, a popular programming language, offers a wide range of libraries and frameworks to enhance software development. Among these, cryptography and memory profiling are crucial aspects that we utilized in our project as it contributes to security and performance optimization.

Cryptography

Cryptography is the practice of securing communication and data by employing various mathematical algorithms and protocols. In Python, the cryptography library provides comprehensive cryptographic functionalities. It serves as a "one-stop-shop" for all cryptographic operations, including key generation, encryption, decryption, digital signatures, and more. The library supports a plethora of cryptographic algorithms such as RSA, AES, SHA, HMAC, and others. By utilizing cryptography, developers can ensure data confidentiality, integrity, and authenticity in their applications.

PyCryptodome

PyCryptodome is a Python library that emerged as a successor to the PyCrypto library. It provides an extensive set of cryptographic primitives and algorithms. With PyCryptodome, developers have access to symmetric and asymmetric encryption algorithms, hash functions, key derivation functions, digital signatures, and more. It offers a high level of flexibility and allows for the implementation of complex cryptographic protocols. PyCryptodome's rich feature set empowers developers to build secure and robust systems by leveraging proven cryptographic techniques.

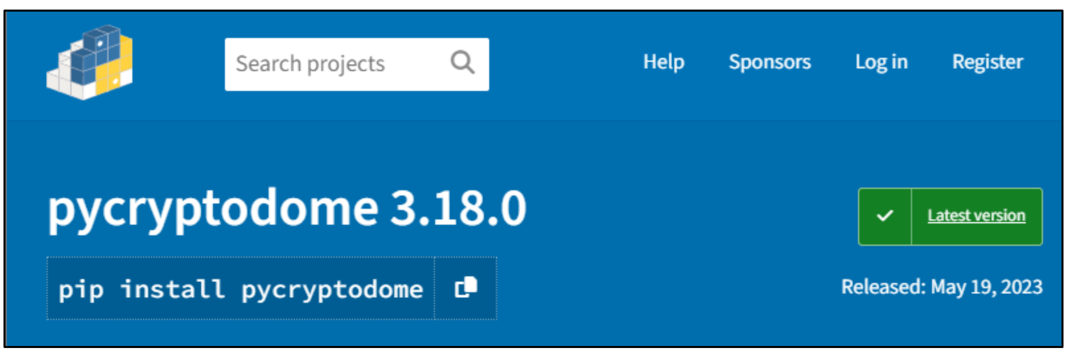


Figure 5: pycryptodome with latest version

PyCryptodomex

PyCryptodomex, another distribution of the PyCryptodome library, is designed as a drop-in replacement for the older PyCrypto library. PyCryptodomex shares the same feature set as PyCryptodome, providing compatibility with both Python 2 and Python 3. This alternative distribution ensures seamless transition and avoids potential conflicts when migrating from

PyCrypto to the improved PyCryptodome. By utilizing PyCryptodome or PyCryptodomex, developers can harness the power of cryptographic algorithms and strengthen the security of their applications.

Memory Profiling with memory_profiler

Memory profiling is a vital aspect of optimizing software performance and identifying memory-intensive sections of code. Python offers the memory_profiler library, which enables developers to monitor memory usage in their programs. With memory_profiler, developers can profile the memory consumption of specific functions or lines of code. By measuring memory usage, they gain insights into how memory consumption evolves during program execution. This information allows for the detection of memory leaks, inefficient memory allocation, and optimization opportunities. memory_profiler provides decorators and functions, such as the “@profile” decorator and the “memory_usage” function, facilitating memory profiling effortlessly.

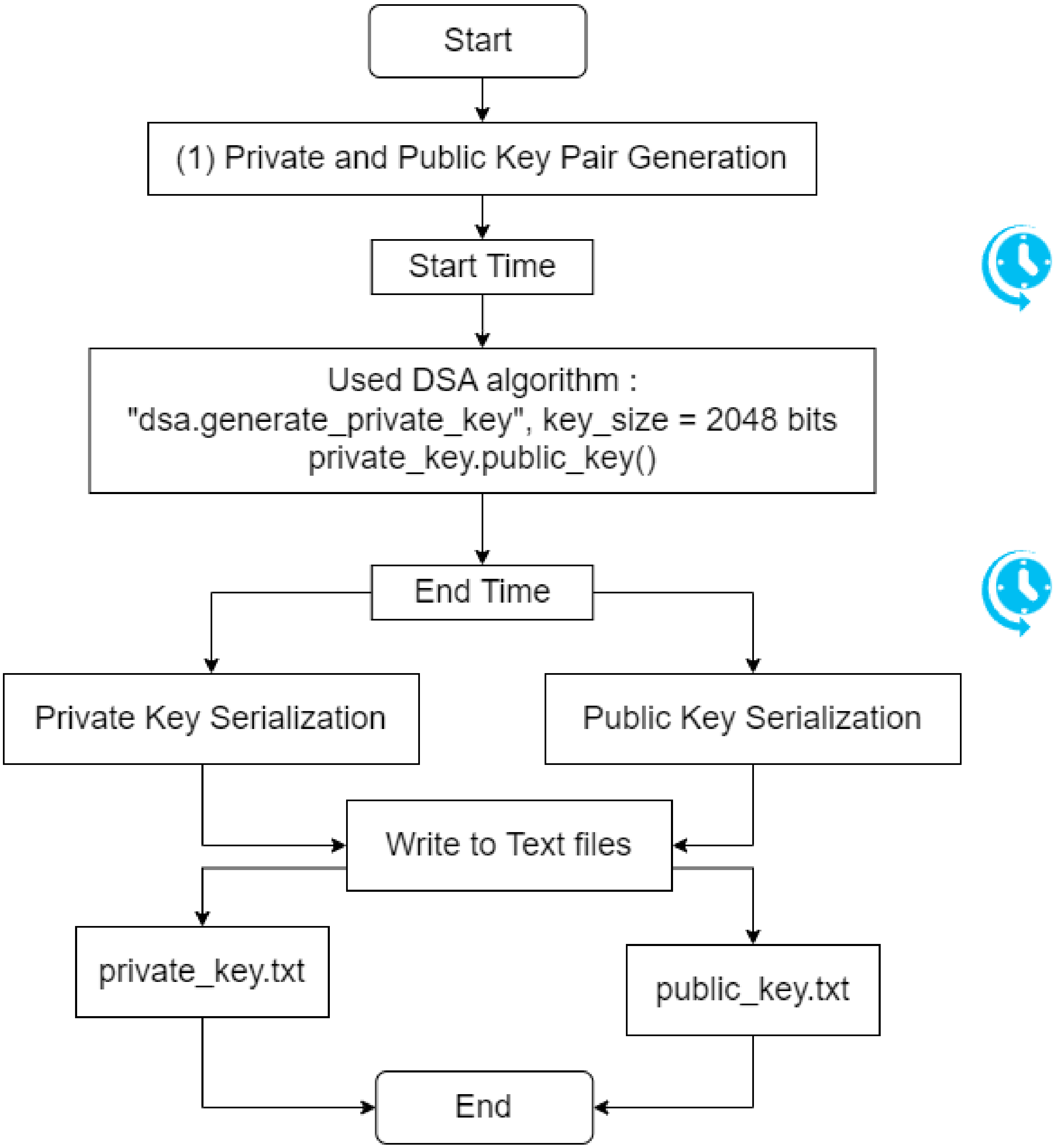
```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    81     44.6 MiB     44.6 MiB           1   @profile
    82                                         def verify_signature(public_key, message, signature):
    83     44.6 MiB      0.0 MiB           1       print_box("Validating Signature")
    84     44.6 MiB      0.0 MiB           1       print("<Importing public key for validating signature>")
    85
    86                                             # Read the signature from the text file
    87     44.6 MiB      0.0 MiB           1       with open('signature.txt', 'rb') as file:
    88     44.6 MiB      0.0 MiB           1           signature = file.read().strip()
    89
    90     44.6 MiB      0.0 MiB           1       message_hash = hashlib.sha3_512(message.encode('utf-8')).
    91     44.6 MiB      0.0 MiB           1       start_time = time.time()
    92
    93     44.6 MiB      0.0 MiB           1       try:
    94     44.6 MiB      0.0 MiB           2           public_key.verify(
    95     44.6 MiB      0.0 MiB           1               signature,
    96     44.6 MiB      0.0 MiB           1               message.encode('utf-8'),
    97     44.6 MiB      0.0 MiB           1               padding.PKCS1v15(),
    98     44.6 MiB      0.0 MiB           1               hashes.SHA3_512()
    99                                                 )
   100     44.6 MiB      0.0 MiB           1           end_time = time.time()
   101     44.6 MiB      0.0 MiB           1           execution_time = end_time - start_time
   102     44.6 MiB      0.0 MiB           1           return True, execution_time
   103                                             except Exception:
   104                                                 return False, None
```

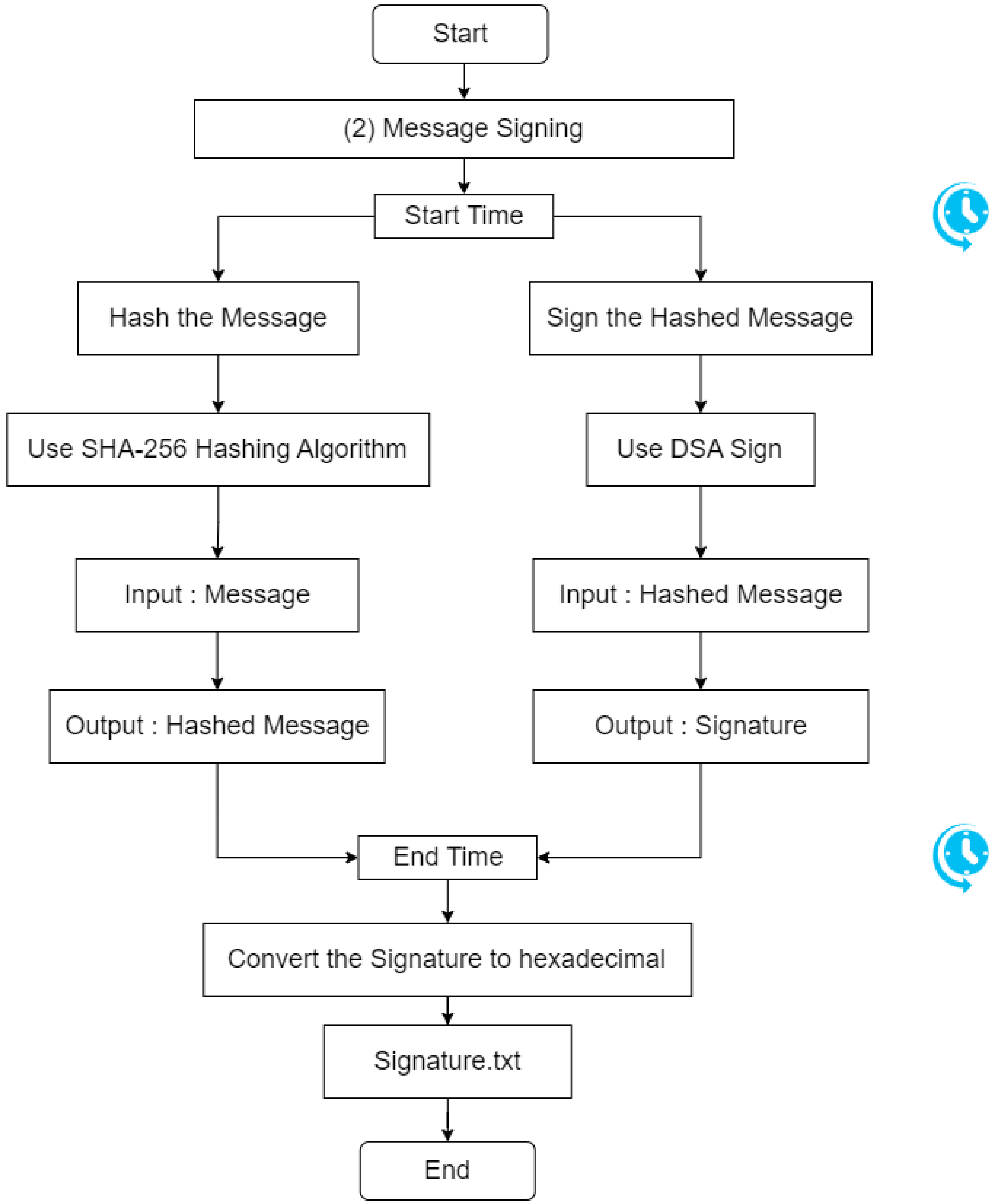
Figure 6: Memory Profiler works in Python

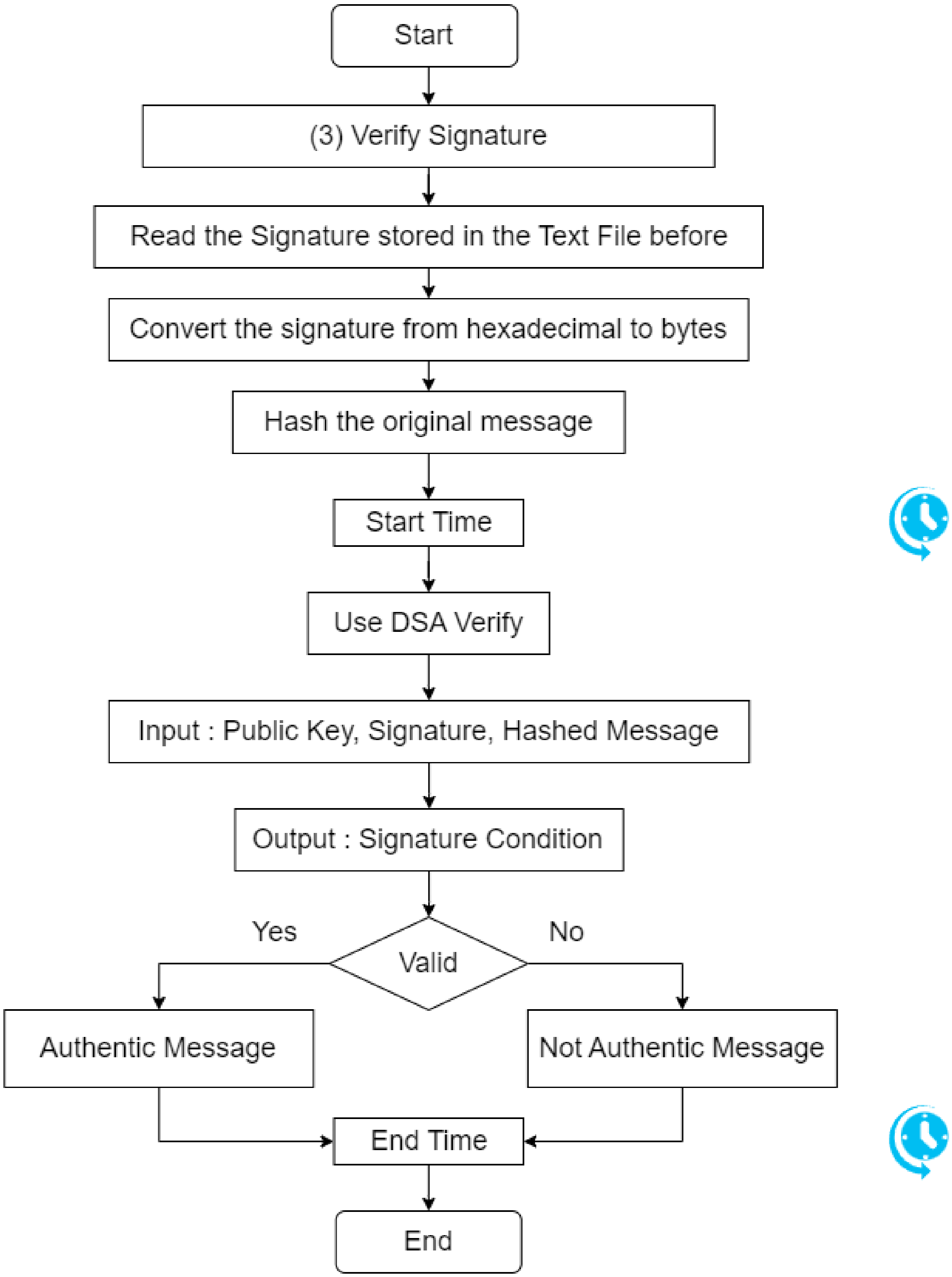
2.2 Flowchart

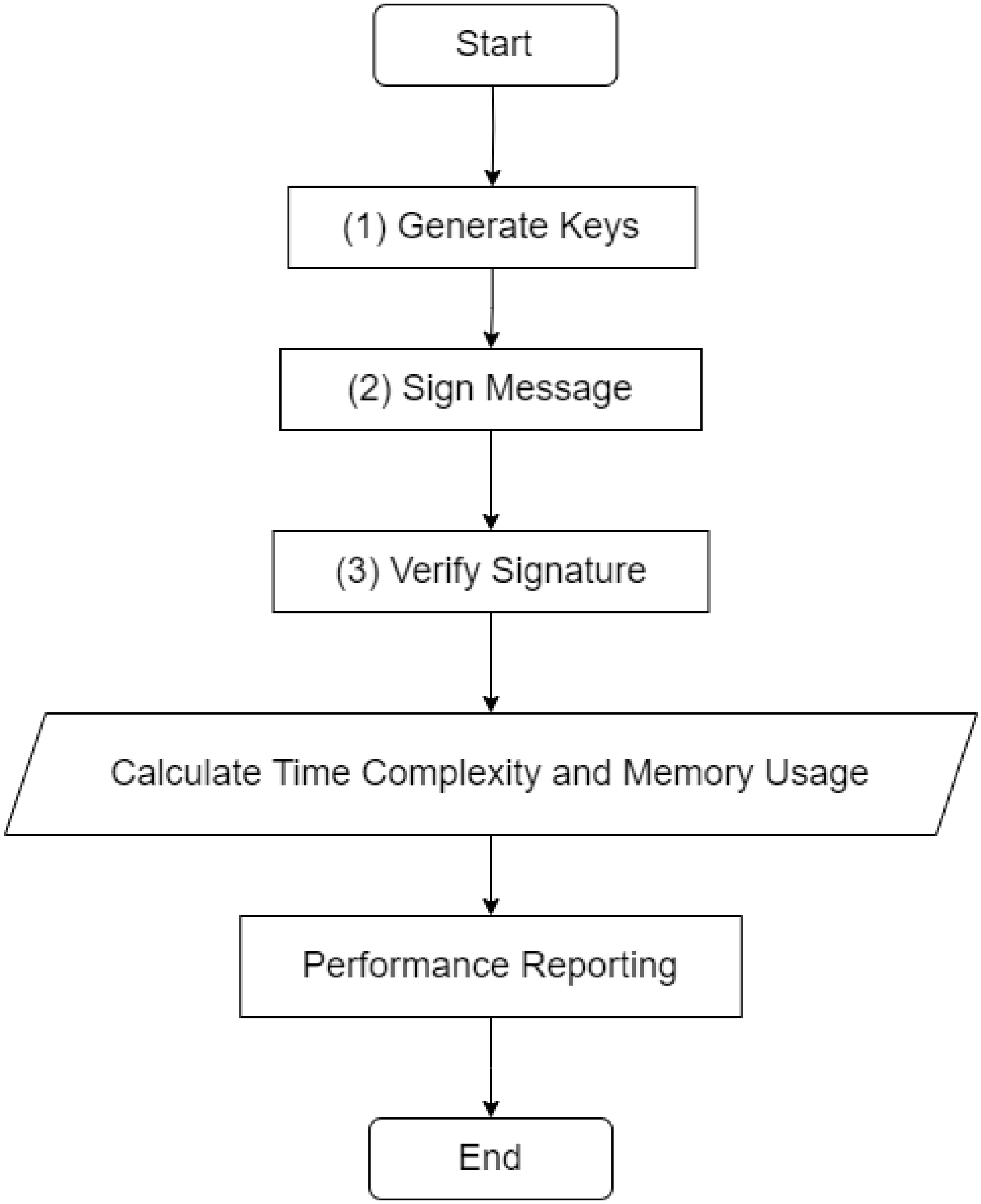
Flowcharts are graphical representations that aid in visualizing complex processes, making it easier to comprehend the logic and sequence of operations. The flowchart for the DSA (Digital Signature Algorithm) program is created using tools which is Draw.io.

2.2.1 Phase 3 - Original DSA Program

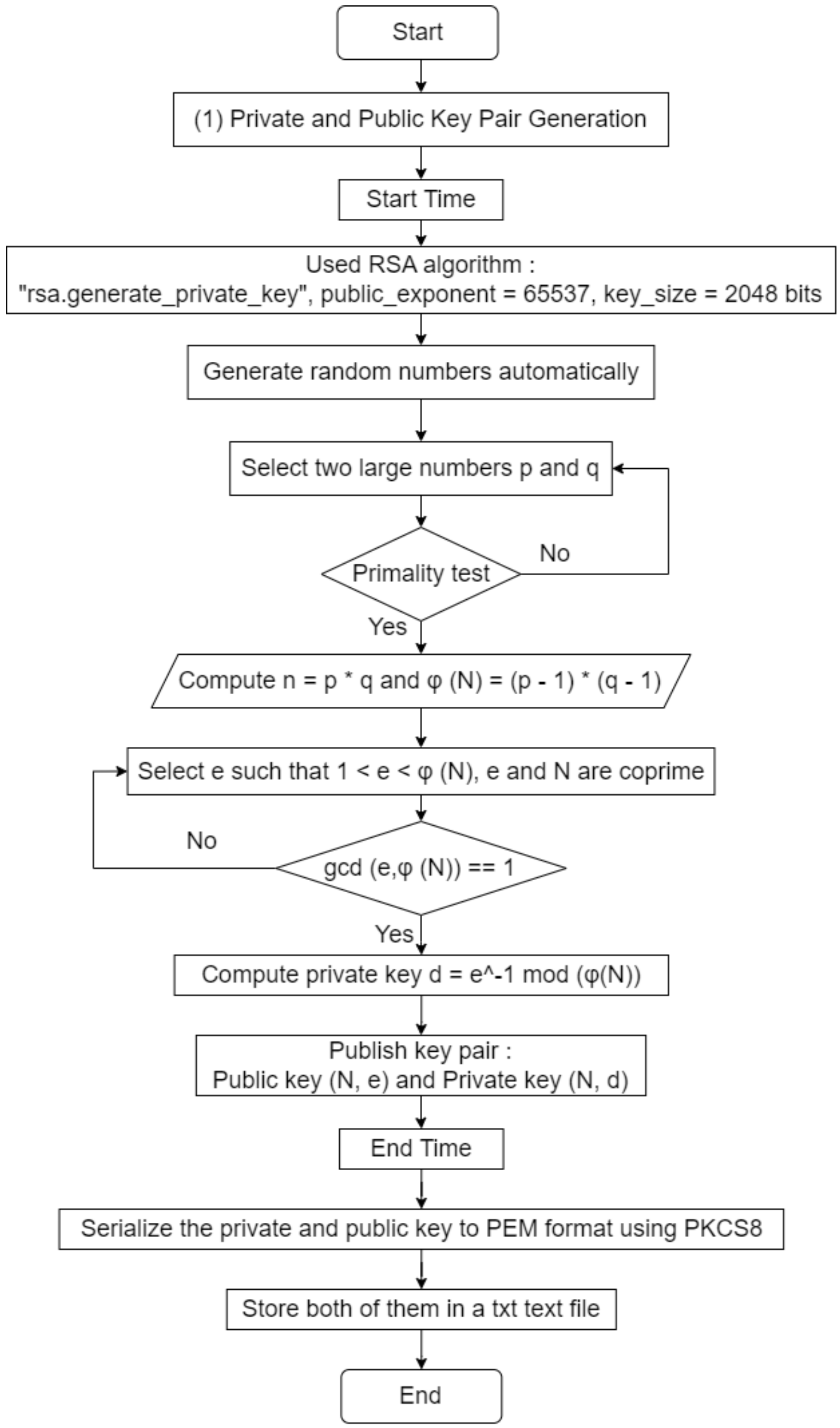


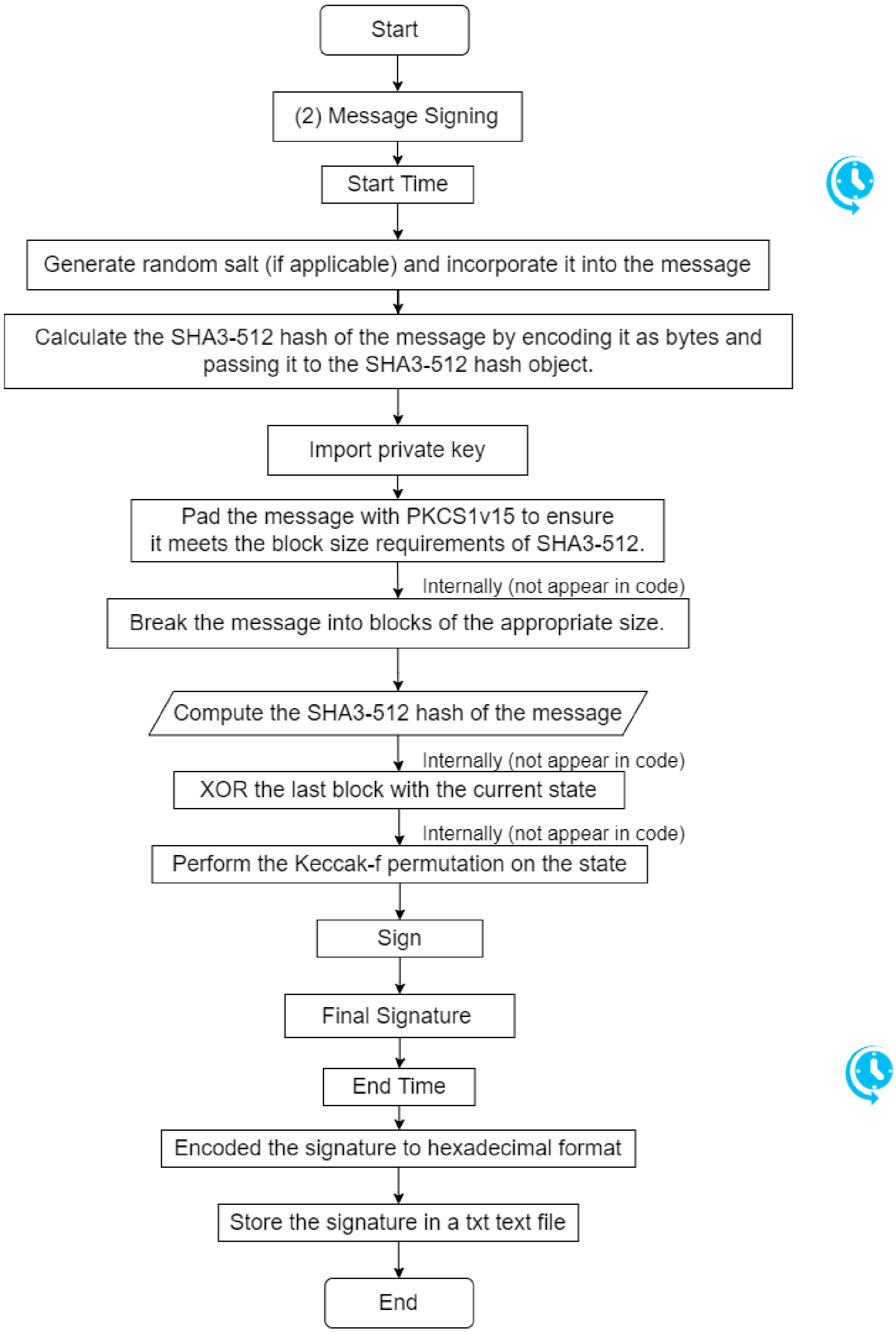

Start
(2) Message Signing
Start Time
Hash the Message
Sign the Hashed Message
Use SHA-256 Hashing Algorithm
Use DSA Sign
Input : Message
Input : Hashed Message
Output : Hashed Message
Output : Signature
End Time
Convert the Signature to hexadecimal
Signature.txt
End


Start
(3) Verify Signature
Read the Signature stored in the Text File before
Convert the signature from hexadecimal to bytes
Hash the original message
Start Time
Use DSA Verify
Input : Public Key, Signature, Hashed Message
Output : Signature Condition
Yes
Valid
No
Authentic Message
Not Authentic Message
End Time
End


Start
(1) Generate Keys
(2) Sign Message
(3) Verify Signature
Calculate Time Complexity and Memory Usage
Performance Reporting
End

2.2.2 Phase 4 - Modified DSA Program




Start
(2) Message Signing
Start Time
Generate random salt (if applicable) and incorporate it into the message
Calculate the SHA3-512 hash of the message by encoding it as bytes and passing it to the SHA3-512 hash object.
Import private key
Pad the message with PKCS1v15 to ensure it meets the block size requirements of SHA3-512.
Internally (not appear in code)
Break the message into blocks of the appropriate size.
Compute the SHA3-512 hash of the message
Internally (not appear in code)
XOR the last block with the current state
Internally (not appear in code)
Perform the Keccak-f permutation on the state
Sign
Final Signature
End Time
Encoded the signature to hexadecimal format
Store the signature in a txt text file
End

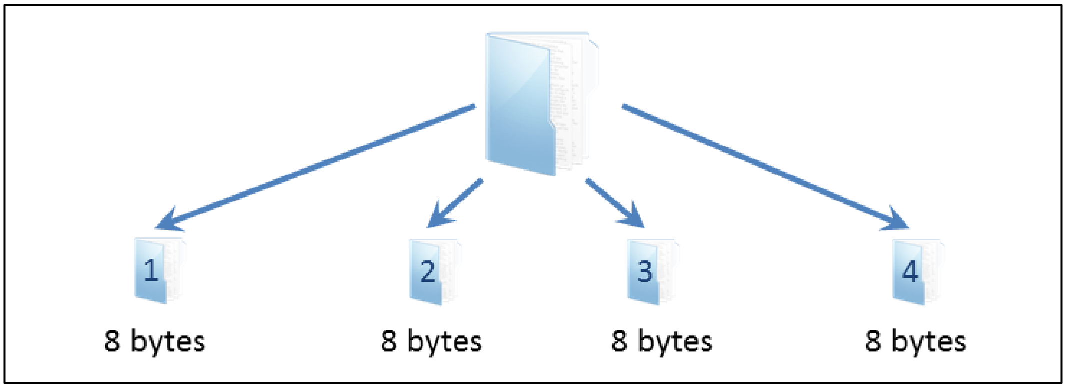


Figure 7: Example of padding works

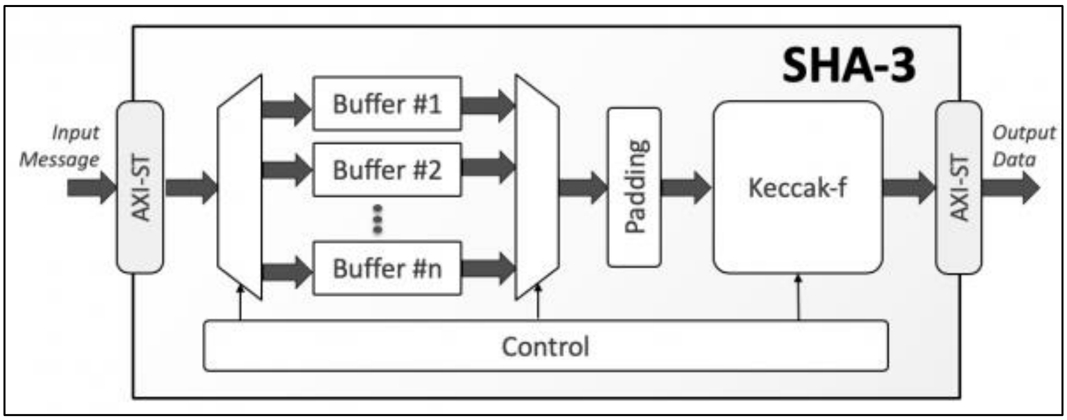
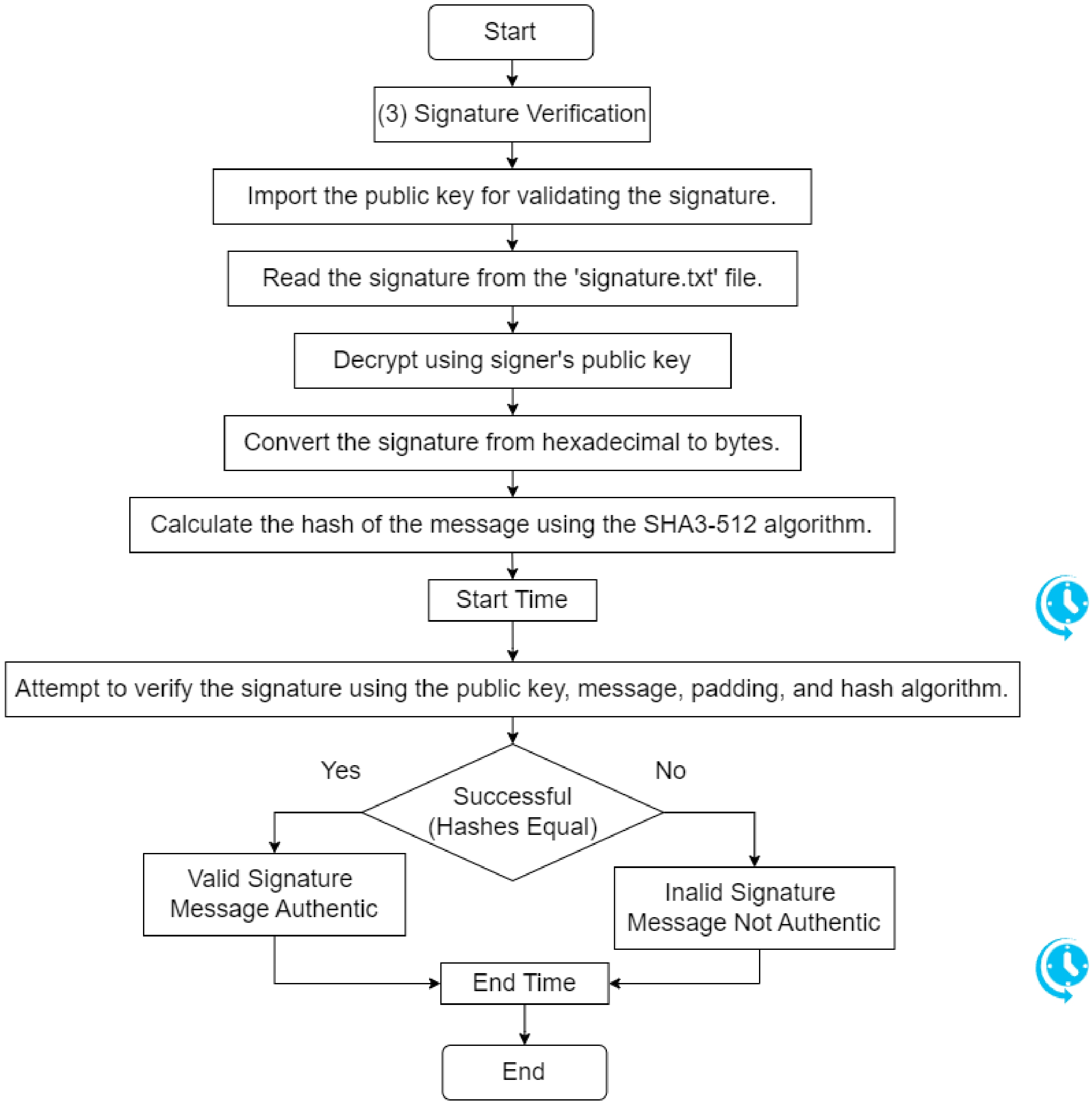
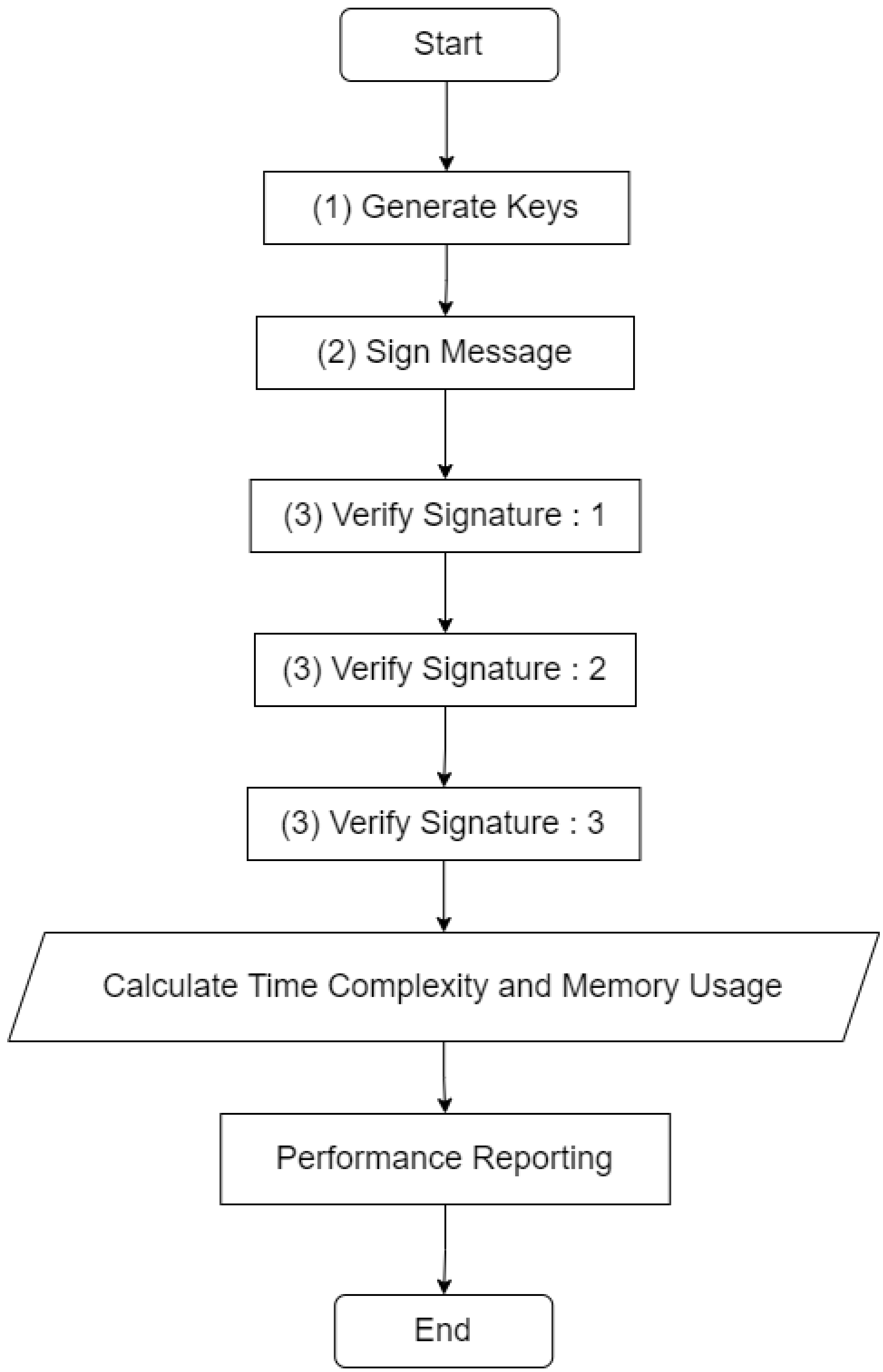


Figure 8: SHA-3 secure hash crypto

X	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E	20
3	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D	30
4	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C	40
5	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B	50
6	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	52	5A	60
7	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69	70
8	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78	80
9	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87	90
A	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96	A0

Figure 9: Hexadecimal table


Start
(3) Signature Verification
Import the public key for validating the signature.
Read the signature from the 'signature.txt' file.
Decrypt using signer's public key
Convert the signature from hexadecimal to bytes.
Calculate the hash of the message using the SHA3-512 algorithm.
Start Time
Attempt to verify the signature using the public key, message, padding, and hash algorithm.
Yes
No
Successful
(Hashes Equal)
Valid Signature
Message Authentic
Inalid Signature
Message Not Authentic
End Time
End


Start
(1) Generate Keys
(2) Sign Message
(3) Verify Signature : 1
(3) Verify Signature : 2
(3) Verify Signature : 3
Calculate Time Complexity and Memory Usage
Performance Reporting
End

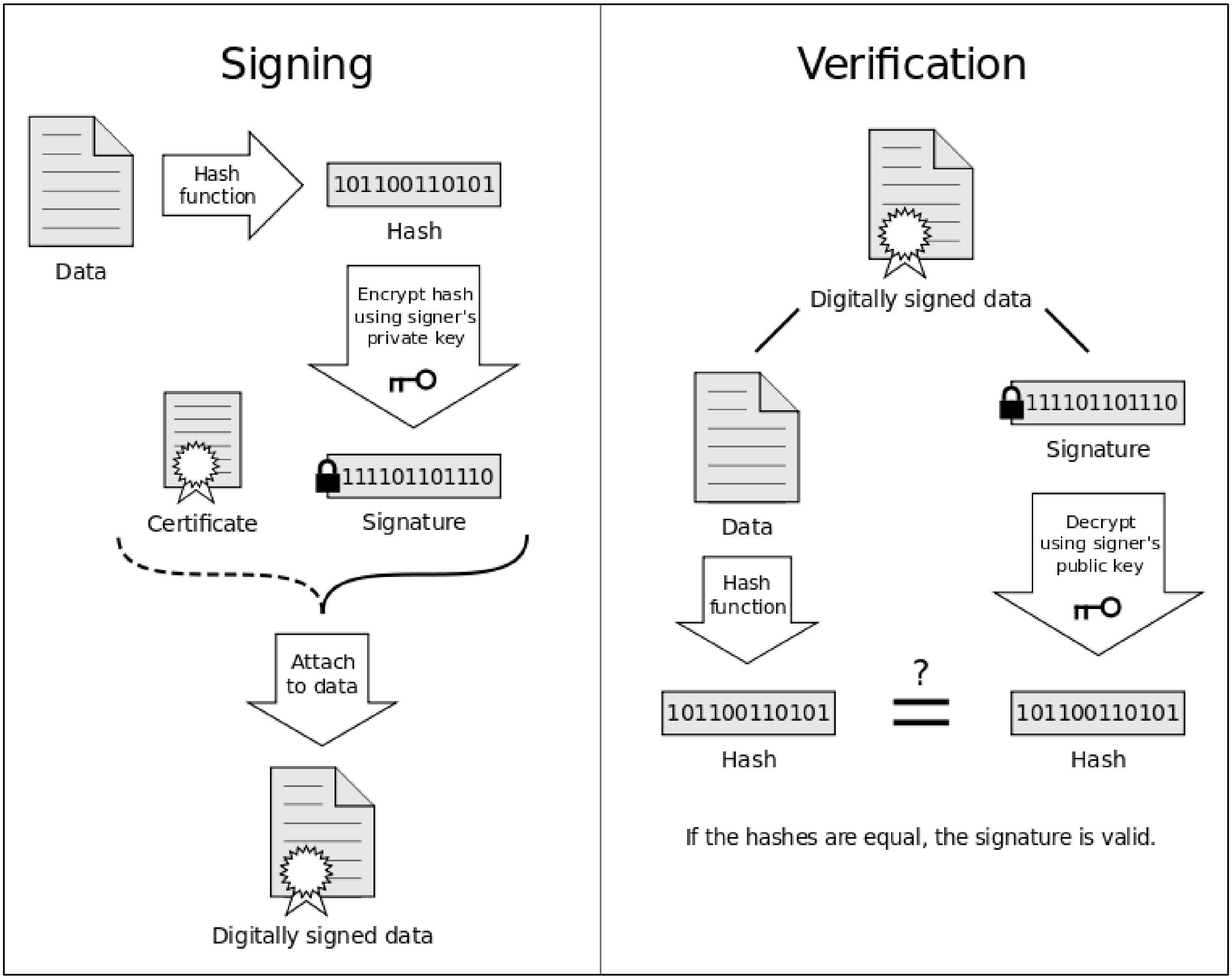


Figure 10: Graphical demonstration of Signing and Verification process

2.3 Source Code

2.3.1 Phase 3 - Original DSA Program

```
from multiprocessing import freeze_support
from cryptography.hazmat.primitives.asymmetric import utils
import time
import hashlib
from cryptography.hazmat.primitives.asymmetric import dsa
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives import serialization
from memory_profiler import profile
from memory_profiler import memory_usage
import base64

# Box-like structure function
def print_box(message):
    print("=" * 80)
    print(f"| {message.center(76)} |")
    print("=" * 80)

# Function to generate a DSA key pair, serialize them to PEM format,
# and write them to text files.
@profile() # Decorator for memory profiling
def generate_keys():
    print_box("Generating Key Pair")
    # Generate a DSA private key
    private_key = dsa.generate_private_key(key_size=2048)
    # Derive the corresponding public key
    public_key = private_key.public_key()

    # Serialize the private key to PEM format
    private_key_pem = private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
```

```
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.NoEncryption()
    )

    # Write the serialized private key to a text file
    with open('private_key.txt', 'wb') as file:
        file.write(private_key_pem)

    # Serialize the public key to PEM format
    public_key_pem = public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )

    # Write the serialized public key to a text file
    with open('public_key.txt', 'wb') as file:
        file.write(public_key_pem)

    print("Private key object                  :", private_key)
    print("Public key object                   :", public_key)
    return private_key, public_key

# Function to sign a message using the provided private key.
@profile()
def sign_message(private_key, message):
    print_box("Signing Message")
    message_hash = hashlib.sha256(message.encode('utf-8')).digest()
    start_time_hash = time.time()  # Start measuring time for hashing
    print("<Importing private key for sign>")

    signature = private_key.sign(
        message_hash,
```

```
        utils.Prehashed(hashes.SHA256())
    ) # Sign the hashed message using the private key

    end_time_hash = time.time()  # End measuring time for hashing
    # The execution time for hashing the message
    execution_time_hash = end_time_hash - start_time_hash

    execution_time = execution_time_hash
    print("Original message hash              :", message_hash)
    print("Signature of original message hash :", signature)

    # The hexadecimal representation of the signature
    signature_hex = signature.hex()

    # Store the signature in a text file
    with open('signature.txt', 'w') as file:
        file.write(signature_hex)

    return signature_hex, execution_time

# Function to verify the signature of a message using the
# provided public key.
@profile()
def verify_signature(public_key, message, signature_hex):
    print_box("Validating Signature")
    print("<Importing public key for validating signature>")

    # Read the signature from the text file
    with open('signature.txt', 'r') as file:
        signature_hex = file.read().strip()

    # Convert the signature from hexadecimal to bytes
```

```python
    signature = bytes.fromhex(signature_hex)

    # Hash the original message
    message_hash = hashlib.sha256(message.encode('utf-8')).digest()
    start_time = time.time()

    try:
        public_key.verify(
            signature,
            message_hash,
            utils.Prehashed(hashes.SHA256())
        ) # Verify the signature using the public key
        end_time = time.time()
        execution_time = end_time - start_time
        return True, execution_time
    except Exception:
        return False, None

if __name__ == '__main__':
    freeze_support()  # Add freeze_support() when using multiprocessing on Windows

    # Measure execution time and memory usage
    def measure_time_and_memory(func, *args):
        start_time = time.time()
        mem_usage = memory_usage((func, args, {}))
        end_time = time.time()
        execution_time = end_time - start_time
        max_mem_usage = max(mem_usage)
        # The maximum memory usage during function execution
        return execution_time, max_mem_usage

    # Generate keys
```

```
    start_time = time.time()
    private_key, public_key = generate_keys()
    end_time = time.time()
    key_pair_time, key_pair_mem = measure_time_and_memory(generate_keys)
    print(f"Time taken for key generation     : {end_time - start_time} seconds")
    print("Key pair generation memory usage  :", key_pair_mem, "KB")
    print()


    # Messages of different sizes
    messages = [
        "Hello, World!",
        "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer pretium, lacus a consequat suscipit, magna sem sagittis erat, vitae posuere ex ex sed justo. Curabitur commodo ex lectus, in consequat nisi pellentesque a. Suspendisse id malesuada lectus, nec elementum neque. Nullam venenatis eros sit amet odio pharetra, id varius tellus auctor. Curabitur sit amet ex eget leo laoreet pulvinar. Nulla facilisi. Fusce ac mauris ut mauris semper aliquam in et est. Donec condimentum neque a interdum consectetur. Nulla facilisi. Donec venenatis dapibus mauris at iaculis.",
        "Curiosity is a remarkable human trait that drives us to explore, question, and seek knowledge about the world around us. It is the spark that ignites our imagination, propelling us on a lifelong journey of discovery. From our earliest years, curiosity is inherent in our nature, driving us to touch, taste, and explore our environment. As we grow, this innate curiosity blossoms, shaping our identities and influencing the paths we choose to follow. One of the most significant aspects of curiosity is its ability to fuel learning. When we approach new ideas, subjects, or experiences with a curious mindset, we are open to the possibilities they hold. Curiosity compels us to ask questions, seek answers, and engage in critical thinking. It challenges us to delve deeper, to unravel the complexities of the world and expand our knowledge. By nurturing curiosity, we embrace a lifelong pursuit of learning, continuously evolving and growing intellectually. Curiosity also plays a pivotal role in fostering creativity. When we allow our curiosity to roam freely, we invite inspiration and innovation into our lives. Curiosity compels us to explore uncharted territories, pushing the boundaries of what is known and comfortable. It encourages us to challenge assumptions, think outside the box, and connect seemingly unrelated ideas. By embracing our natural curiosity, we unlock the potential to create, invent, and bring forth new ideas that can shape the world around us. Beyond its influence on personal growth, curiosity has far-reaching implications for society as a whole. Curiosity serves as a catalyst for progress and advancement. Throughout history, great discoveries and inventions have been born out of a relentless pursuit of answers driven by curiosity. From the exploration of distant lands to scientific breakthroughs, every leap forward is the result of individuals and communities driven by an insatiable curiosity about the unknown. Moreover, curiosity fosters empathy and understanding. When we approach others with genuine curiosity, we open the doors to deeper connections and meaningful relationships. By seeking to understand different perspectives, cultures, and experiences, we break down barriers and bridge divides. Curiosity allows us to appreciate the richness and diversity of the human tapestry, promoting tolerance and acceptance in an increasingly interconnected world. However, despite its profound potential, curiosity can be stifled if not nurtured and encouraged. In a society that often values conformity and certainty, curiosity can be viewed as a disruption or a distraction. The pressure to conform and the fear of the unknown can dampen the flames of curiosity, leading to missed opportunities and limited growth. As a society, we must recognize the transformative power of curiosity and create environments that foster its growth in individuals of all ages. To cultivate curiosity, education plays a vital role.
```

```
Schools should embrace inquiry-based learning, encouraging students to ask questions, explore their interests, and pursue independent research. Teachers can serve as guides, nurturing curiosity by creating a safe space for exploration and discovery. By incorporating real-world applications and promoting interdisciplinary approaches, education can unleash the full potential of curiosity, preparing students to become lifelong learners and active contributors to society. In addition to education, parents and caregivers also have a crucial role to play in cultivating curiosity. By encouraging exploration, providing opportunities for hands-on learning, and celebrating curiosity-driven achievements, parents can foster a lifelong love of learning in their children. They can instill a sense of wonder, inspire critical thinking, and nurture the natural inquisitiveness that lies within each child. In conclusion, curiosity is a powerful force that drives human progress, fuels creativity, and fosters understanding. It is the key to unlocking new knowledge, inspiring innovation, and forging connections. By nurturing and embracing curiosity, both as individuals and as a society, we unlock a world of possibilities, paving the way for a future built on continuous growth, discovery, and enlightenment. Let us celebrate curiosity and embark on a journey of endless exploration and learning, for it is through curiosity that we truly come alive."
    ]


    for i, message in enumerate(messages):
        print(f"Message {i + 1}: {message}")
        print()

        # Sign the message
        start_time = time.time()
        signature_hex, sign_execution_time = sign_message(private_key, message)
        end_time = time.time()
        sign_time, sign_mem = measure_time_and_memory(sign_message, private_key, message)
        print(f"Time taken for signing        : {end_time - start_time} seconds")
        print("Message signing memory usage :", sign_mem, "KB")
        print()

        # Verify the signature
        start_time = time.time()
        is_valid, verify_execution_time = verify_signature(public_key, message, signature_hex)
        end_time = time.time()
        validate_time, validate_mem = measure_time_and_memory(
            verify_signature, public_key, message, signature_hex)
        print(f"Time taken for verification         : {end_time - start_time} seconds")
        print("Signature validation memory usage   :", validate_mem, "KB")
```

```
        print()

        # Output the result
        if is_valid:
            print("Condition        : The message is authentic.")
            print("Original Message :", message)
            print()
        else:
            print("The message is not authentic.")
            print("Signature is invalid!")

        # Time Complexity Report
        print("----------------------------------------------------------------------------")
        print("|                           Performance Report                             |")
        print("|" + "-" * 74 + "|")
        print("|{:^44s}|{:^29s}|".format("Algorithm", "Time Complexity"))
        print("|" + "-" * 74 + "|")
        print("| {:42s} | {:19.6f} seconds |".format("Public and Private Key Pair Generation"
                                                     , key_pair_time))
        print("| {:42s} | {:19.6f} seconds |".format("Encryption : Message Signing"
                                                     , sign_time))
        print("| {:42s} | {:19.6f} seconds |".format("Decryption : Signature Validation"
                                                     , validate_time))
        overall_time_complexity = key_pair_time + sign_time + validate_time
        print("| {:42s} | {:19.6f} seconds |".format("Signature Execution"
                                                     , sign_execution_time))
        print("| {:42s} | {:19.6f} seconds |".format("Overall Process"
                                                     , overall_time_complexity))
        print("|" + "-" * 74 + "|")
        print()

        # Memory Usage Report
```

```
    print("|" + "-" * 74 + "|")
    print("| {:^42s} | {:^27s} |".format("Algorithm", "Memory Usage"))
    print("|" + "-" * 74 + "|")
    print("| {:42s} | {:24.6f} KB |".format("Public and Private Key Pair Generation"
                                            , key_pair_mem))
    print("| {:42s} | {:24.6f} KB |".format("Encryption : Message Signing"
                                            , sign_mem))
    print("| {:42s} | {:24.6f} KB |".format("Decryption : Signature Validation"
                                            , validate_mem))
    overall_memory_usage = key_pair_mem +sign_mem +validate_mem
    print("| {:42s} | {:24.6f} KB |".format("Signature Execution"
                                            , sign_mem))
    print("| {:42s} | {:24.6f} KB |".format("Overall Process"
                                            , overall_memory_usage))
    print("|" + "-" * 74 + "|")
    print()
```

2.3.2 Phase 4 - Modified DSA Program

```
# Import freeze_support for Windows multiprocessing
from multiprocessing import freeze_support
# Import time for measuring execution time
import time
# Import hashlib for hashing functions
import hashlib
# Import RSA for key generation
from cryptography.hazmat.primitives.asymmetric import rsa
# Import hashes for cryptographic hashes
from cryptography.hazmat.primitives import hashes
# Import serialization for key serialization
from cryptography.hazmat.primitives import serialization
# Import padding for signature padding
from cryptography.hazmat.primitives.asymmetric import padding
# Import profile from memory_profiler for memory profiling
from memory_profiler import profile
# Import memory_usage from memory_profiler for memory profiling
from memory_profiler import memory_usage
import base64


# Box-like structure function
def print_box(message):
    print("=" * 80)
    print(f"| {message.center(76)} |")
    print("=" * 80)

# Generates a private-public key pair and saves them to files.
@profile
def generate_keys():
    print_box("Generating Key Pair")
```

```python
    # Use RSA algorithm
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048
    )
    public_key = private_key.public_key()

    # Serialize the private key to PEM format
    private_key_pem = private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.NoEncryption()
    )

    # Write the serialized private key to a text file
    with open('private_key.txt', 'wb') as file:
        file.write(private_key_pem)

    # Serialize the public key to PEM format
    public_key_pem = public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )

    # Write the serialized public key to a text file
    with open('public_key.txt', 'wb') as file:
        file.write(public_key_pem)

    print("Private key object                 :", private_key)
    print("Public key object                  :", public_key)
    return private_key, public_key
```

```
# Signs a message using a private key.
@profile
def sign_message(private_key, message):
    print_box("Signing Message")
    # Use SHA3-512 algorithm
    message_hash = hashlib.sha3_512(message.encode('utf-8')).digest()
    start_time_hash = time.time()  # Start measuring time for hashing
    print("<Importing private key for sign>")

    signature = private_key.sign(
        message.encode('utf-8'),
        padding.PKCS1v15(),
        hashes.SHA3_512()
    )

    end_time_hash = time.time()  # End measuring time for hashing
    execution_time_hash = end_time_hash - start_time_hash
    # Calculate the time excluding file execution
    execution_time = execution_time_hash
    print("Original message hash              :", message_hash)
    print("Signature of original message hash :", signature)

    # Convert the signature to hexadecimal format
    signature_hex = signature.hex()

    # Store the signature in a text file
    with open('signature.txt', 'w') as file:
        file.write(signature_hex)
    # The function returns the signature and the execution time for the hashing process.
    return signature_hex, execution_time

# Verifies the authenticity of a signed message using a public key.
```

```
@profile
def verify_signature(public_key, message, signature_hex):
    print_box("Validating Signature")
    print("<Importing public key for validating signature>")

    # Read the signature from the text file
    with open('signature.txt', 'r') as file:
        signature_hex = file.read().strip()

    # Convert the signature from hexadecimal to bytes
    signature = bytes.fromhex(signature_hex)

    message_hash = hashlib.sha3_512(message.encode('utf-8')).digest()
    start_time = time.time()

    # If the signature is valid, the function returns True along
    # with the execution time for the verification process.
    # Otherwise, it returns False and None.
    try:
        public_key.verify(
            signature,
            message.encode('utf-8'),
            padding.PKCS1v15(),
            hashes.SHA3_512()
        )
        end_time = time.time()
        execution_time = end_time - start_time # Calculate time without file execution
        return True, execution_time
    except Exception:
        return False, None

if __name__ == '__main__':
```

```
    freeze_support()  # Add freeze_support() when using multiprocessing on Windows

    # Measure execution time and memory usage
    # The function takes the target function (func) and its arguments (args) as input.
    # It measures the time taken to execute the function and records the memory usage
    # using memory_usage from memory_profiler.
    def measure_time_and_memory(func, *args):
        start_time = time.time()
        mem_usage = memory_usage((func, args, {}))
        end_time = time.time()
        execution_time = end_time - start_time
        max_mem_usage = max(mem_usage)

        return execution_time, max_mem_usage

    # Generate keys
    start_time = time.time()
    private_key, public_key = generate_keys()
    end_time = time.time()
    key_pair_time, key_pair_mem = measure_time_and_memory(generate_keys)
    print(f"Time taken for key generation     : {end_time - start_time} seconds")
    print("Key pair generation memory usage  :", key_pair_mem, "KB")
    print()

    # Messages of different sizes
    messages = [
        "Hello, World!",
        "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer pretium, lacus a consequat suscipit, magna sem sagittis erat, vitae posuere ex ex sed justo. Curabitur commodo ex lectus, in consequat nisi pellentesque a. Suspendisse id malesuada lectus, nec elementum neque. Nullam venenatis eros sit amet odio pharetra, id varius tellus auctor. Curabitur sit amet ex eget leo laoreet pulvinar. Nulla facilisi. Fusce ac mauris ut mauris semper aliquam in et est. Donec condimentum neque a interdum consectetur. Nulla facilisi. Donec venenatis dapibus mauris at iaculis.",
        "Curiosity is a remarkable human trait that drives us to explore, question, and seek knowledge about the world around us. It is the spark that ignites our imagination, propelling us on a lifelong journey of discovery. From our earliest years, curiosity is inherent in our nature, driving us to touch, taste, and explore our environment. As we grow, this innate curiosity blossoms, shaping our identities and influencing the paths we choose to follow. One of the most significant
```

```
aspects of curiosity is its ability to fuel learning. When we approach new ideas, subjects, or experiences with a curious mindset, we are open to the possibilities they hold. Curiosity compels us to ask questions, seek answers, and engage in critical thinking. It challenges us to delve deeper, to unravel the complexities of the world and expand our knowledge. By nurturing curiosity, we embrace a lifelong pursuit of learning, continuously evolving and growing intellectually. Curiosity also plays a pivotal role in fostering creativity. When we allow our curiosity to roam freely, we invite inspiration and innovation into our lives. Curiosity compels us to explore uncharted territories, pushing the boundaries of what is known and comfortable. It encourages us to challenge assumptions, think outside the box, and connect seemingly unrelated ideas. By embracing our natural curiosity, we unlock the potential to create, invent, and bring forth new ideas that can shape the world around us. Beyond its influence on personal growth, curiosity has far-reaching implications for society as a whole. Curiosity serves as a catalyst for progress and advancement. Throughout history, great discoveries and inventions have been born out of a relentless pursuit of answers driven by curiosity. From the exploration of distant lands to scientific breakthroughs, every leap forward is the result of individuals and communities driven by an insatiable curiosity about the unknown. Moreover, curiosity fosters empathy and understanding. When we approach others with genuine curiosity, we open the doors to deeper connections and meaningful relationships. By seeking to understand different perspectives, cultures, and experiences, we break down barriers and bridge divides. Curiosity allows us to appreciate the richness and diversity of the human tapestry, promoting tolerance and acceptance in an increasingly interconnected world. However, despite its profound potential, curiosity can be stifled if not nurtured and encouraged. In a society that often values conformity and certainty, curiosity can be viewed as a disruption or a distraction. The pressure to conform and the fear of the unknown can dampen the flames of curiosity, leading to missed opportunities and limited growth. As a society, we must recognize the transformative power of curiosity and create environments that foster its growth in individuals of all ages. To cultivate curiosity, education plays a vital role. Schools should embrace inquiry-based learning, encouraging students to ask questions, explore their interests, and pursue independent research. Teachers can serve as guides, nurturing curiosity by creating a safe space for exploration and discovery. By incorporating real-world applications and promoting interdisciplinary approaches, education can unleash the full potential of curiosity, preparing students to become lifelong learners and active contributors to society. In addition to education, parents and caregivers also have a crucial role to play in cultivating curiosity. By encouraging exploration, providing opportunities for hands-on learning, and celebrating curiosity-driven achievements, parents can foster a lifelong love of learning in their children. They can instill a sense of wonder, inspire critical thinking, and nurture the natural inquisitiveness that lies within each child. In conclusion, curiosity is a powerful force that drives human progress, fuels creativity, and fosters understanding. It is the key to unlocking new knowledge, inspiring innovation, and forging connections. By nurturing and embracing curiosity, both as individuals and as a society, we unlock a world of possibilities, paving the way for a future built on continuous growth, discovery, and enlightenment. Let us celebrate curiosity and embark on a journey of endless exploration and learning, for it is through curiosity that we truly come alive."
    ]

    # The script iterates over each message in the messages list.

    # Because we need the same key pair for testing purpose, so we only loop

    # it after the generate key pair function.

    # Note : Different message for sign and verify with same key pair

    for i, message in enumerate(messages):

        print(f"Message {i + 1}: {message}")
```

```python
    print()

    # Sign the message
    start_time = time.time()
    signature_hex, sign_execution_time = sign_message(private_key, message)
    end_time = time.time()
    sign_time, sign_mem = measure_time_and_memory(sign_message, private_key, message)
    print("<Results Exclude File Execution>")
    print(f"Time taken for signing       : {end_time - start_time} seconds")
    print("Message signing memory usage :", sign_mem, "KB")
    print()

    # Verify the signature
    start_time = time.time()
    is_valid, verify_execution_time = verify_signature(public_key, message, signature_hex)
    end_time = time.time()
    validate_time, validate_mem = measure_time_and_memory(
        verify_signature, public_key, message, signature_hex)
    print("<Results Exclude File Execution>")
    print(f"Time taken for verification         : {end_time - start_time} seconds")
    print("Signature validation memory usage   :", validate_mem, "KB")
    print()

    # Output the result whether the message is authentic or not.
    if is_valid:
        print("Condition        : The message is authentic.")
        print("Original Message :", message)
        print()
    else:
        print("The message is not authentic.")
        print("Signature is invalid!")

    # Prints the performance report, including time complexity and memory usage for each
```

```python
        # step and the overall process (include file reading and writing).
        print("---------------------------------------------------------------------------")
        print("|                           Performance Report                             |")
        print("|" + "-" * 74 + "|")
        print("|{:^44s}|{:^29s}|".format("Algorithm", "Time Complexity"))
        print("|" + "-" * 74 + "|")
        print("| {:42s} | {:19.6f} seconds |".format("Public and Private Key Pair Generation"
                                                  , key_pair_time))
        print("| {:42s} | {:19.6f} seconds |".format("Encryption : Message Signing"
                                                  , sign_time))
        print("| {:42s} | {:19.6f} seconds |".format("Decryption : Signature Validation"
                                                  , validate_time))
        overall_time_complexity = key_pair_time + sign_time + validate_time
        print("| {:42s} | {:19.6f} seconds |".format("Signature Execution"
                                                  , sign_execution_time))
        print("| {:42s} | {:19.6f} seconds |".format("Overall Process"
                                                  , overall_time_complexity))
        print("|" + "-" * 74 + "|")
        print()

        print("|" + "-" * 74 + "|")
        print("| {:^42s} | {:^27s} |".format("Algorithm", "Memory Usage"))
        print("|" + "-" * 74 + "|")
        print("| {:42s} | {:24.6f} KB |".format("Public and Private Key Pair Generation"
                                             , key_pair_mem))
        print("| {:42s} | {:24.6f} KB |".format("Encryption : Message Signing"
                                             , sign_mem))
        print("| {:42s} | {:24.6f} KB |".format("Decryption : Signature Validation"
                                             , validate_mem))
        overall_memory_usage = key_pair_mem + sign_mem + validate_mem
        print("| {:42s} | {:24.6f} KB |".format("Signature Execution"
                                             , sign_mem))
```

```
        print("| {:42s} | {:24.6f} KB |".format("Overall Process"
                                                , overall_memory_usage))
        print("|" + "-" * 74 + "|")
        print()
```

The programs share a common structure and perform key generation, message signing, and signature verification operations. The main difference lies in the cryptographic algorithm used.

The code begins by importing the necessary libraries and defining a utility function, print_box(), to display a box-like structure for better visualization. The programs utilize the cryptography library for cryptographic operations and the memory_profiler library for memory profiling.

The generate_keys() function generates a key pair (private and public keys) using either DSA or RSA algorithm. The private key is serialized in PEM format and stored in a text file named 'private_key.txt', while the public key is serialized and stored in a text file named 'public_key.txt'. The function returns the generated private and public keys.

The sign_message() function signs a message using the private key. It takes the private key and the message as input and computes the SHA3-512 hash of the message. The private key is used to sign the hashed message, and the resulting signature is stored in a text file named 'signature.txt'. The function also returns the signature and the execution time for the hashing process.

The verify_signature() function verifies the authenticity of a signed message using the public key. It takes the public key, the message, and the signature as inputs. The function reads the signature from the 'signature.txt' file, converts it to bytes, and verifies the signature using the provided public key. If the signature is valid, the function returns True along with the execution time for the verification process. Otherwise, it returns False and None.

The main part of the code consists of a loop that iterates over a list of messages. For each message, it calls the 3 main functions to run. At the same time, the time execution will be calculated without file or (in the above part) with file process (in the last part of performance report table).

2.4 Output

In order to provide a comprehensive view of the output generated by the code snippets, we present screenshots showcasing the execution results and performance reports.

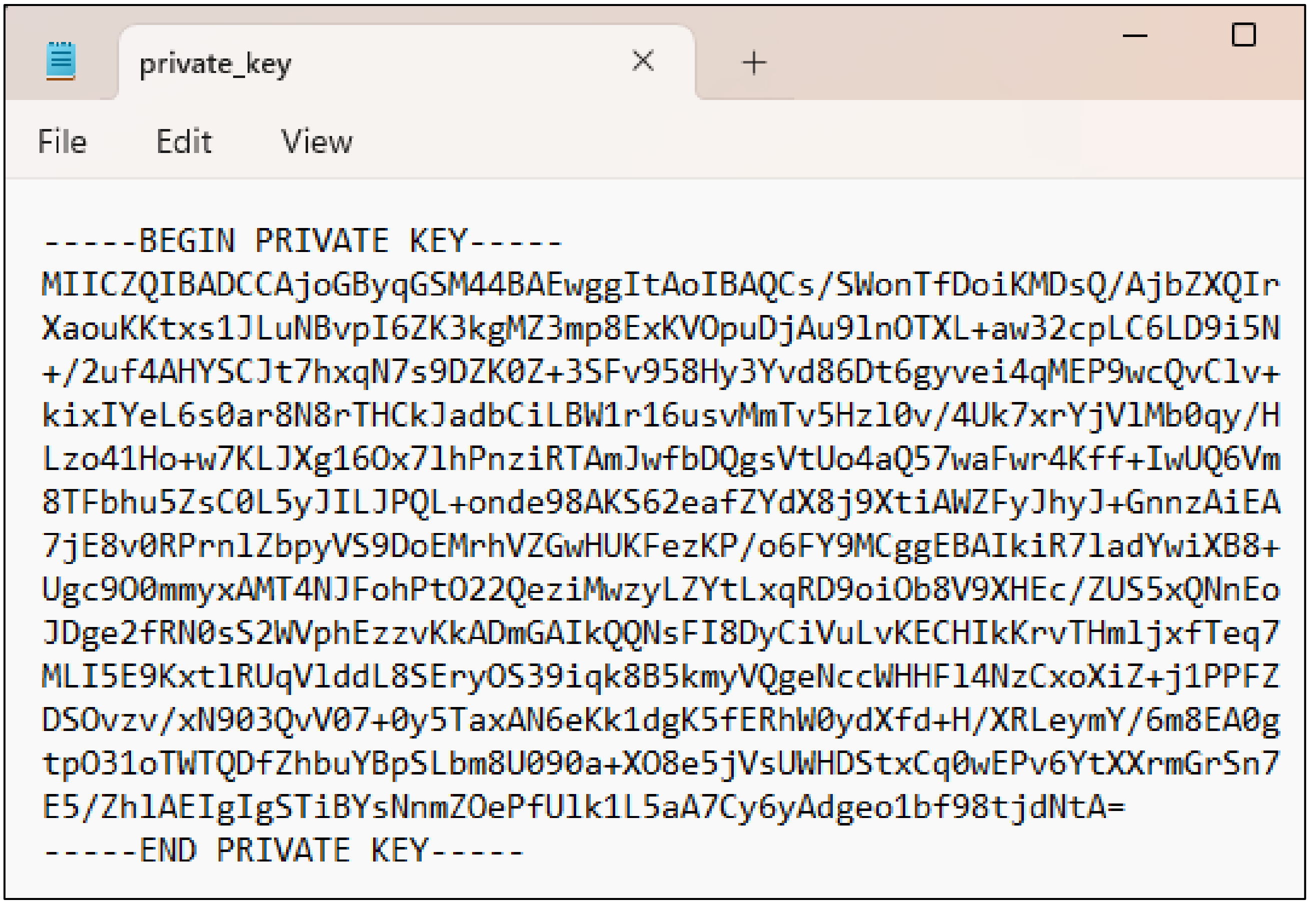
2.4.1 Phase 3 - Original DSA Program

```
================================================================================
|                              Generating Key Pair                             |
================================================================================
Private key object                 : <cryptography.hazmat.backends.openssl.dsa._DSAPrivateKey object at 0x0000
01FE637B3520>
Public key object                  : <cryptography.hazmat.backends.openssl.dsa._DSAPublicKey object at 0x00000
1FE637B3460>
Filename: C:\Users\summe\source\repos\Blowfish_Coding Phase 3_Simple DSA Program\Blowfish_Coding Phase 3_Simpl
e DSA Program\Blowfish_Coding_Phase_3_Simple_DSA_Program.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    20     44.1 MiB     44.1 MiB           1   @profile() # Decorator for memory profiling
    21                                         def generate_keys():
    22     44.1 MiB      0.0 MiB           1       print_box("Generating Key Pair")
    23                                             # Generate a DSA private key
    24     44.1 MiB      0.0 MiB           1       private_key = dsa.generate_private_key(key_size=2048)
    25                                             # Derive the corresponding public key
    26     44.1 MiB      0.0 MiB           1       public_key = private_key.public_key()
    27
    28                                             # Serialize the private key to PEM format
    29     44.1 MiB      0.0 MiB           2       private_key_pem = private_key.private_bytes(
    30     44.1 MiB      0.0 MiB           1           encoding=serialization.Encoding.PEM,
    31     44.1 MiB      0.0 MiB           1           format=serialization.PrivateFormat.PKCS8,
    32     44.1 MiB      0.0 MiB           1           encryption_algorithm=serialization.NoEncryption()
    33                                             )
    34
    35                                             # Write the serialized private key to a text file
    36     44.1 MiB      0.0 MiB           1       with open('private_key.txt', 'wb') as file:
    37     44.1 MiB      0.0 MiB           1           file.write(private_key_pem)
    38
    39                                             # Serialize the public key to PEM format
    40     44.1 MiB      0.0 MiB           2       public_key_pem = public_key.public_bytes(
    41     44.1 MiB      0.0 MiB           1           encoding=serialization.Encoding.PEM,
    42     44.1 MiB      0.0 MiB           1           format=serialization.PublicFormat.SubjectPublicKeyInfo
    43                                             )
    44
    45                                             # Write the serialized public key to a text file
    46     44.1 MiB      0.0 MiB           1       with open('public_key.txt', 'wb') as file:
    47     44.1 MiB      0.0 MiB           1           file.write(public_key_pem)
    48
    49     44.1 MiB      0.0 MiB           1       print("Private key object                 :", private_key)
    50     44.1 MiB      0.0 MiB           1       print("Public key object                  :", public_key)
    51     44.1 MiB      0.0 MiB           1       return private_key, public_key


Time taken for key generation     : 0.770902156829834 seconds
Key pair generation memory usage  : 44.15234375 KB
```

Figure 11: Ouput of Phase 3 by Machine A for generating key pair function on Message 1, the time do not include file execution



-----BEGIN PRIVATE KEY-----
MIICZQIBADCCAjoGByqGSM44BAEwggItAoIBAQCs/SWonTfDoiKMDsQ/AjbZXQIr
XaouKKtxs1JLuNBvpI6ZK3kgMZ3mp8ExKVOpuDjAu9lnOTXL+aw32cpLC6LD9i5N
+/2uf4AHYSCJt7hxqN7s9DZK0Z+3SFv958Hy3Yvd86Dt6gyvei4qMEP9wcQvClv+
kixIYeL6s0ar8N8rTHCkJadbCiLBW1r16usvMmTv5Hzl0v/4Uk7xrYjVlMb0qy/H
Lzo41Ho+w7KLJXg16Ox7lhPnziRTAmJwfbDQgsVtUo4aQ57waFwr4Kff+IwUQ6Vm
8TFbhu5ZsC0L5yJILJPQL+onde98AKS62eafZYdX8j9XtiAWZFyJhyJ+GnnzAiEA
7jE8v0RPrnlZbpyVS9DoEMrhVZGwHUKFezKP/o6FY9MCggEBAIkiR7ladYwiXB8+
Ugc9O0mmyxAMT4NJFohPtO22QeziMwzyLZYtLxqRD9oiOb8V9XHEc/ZUS5xQNnEo
JDge2fRN0sS2WVphEzzvKkADmGAIkQQNsFI8DyCiVuLvKECHIkKrvTHmljxfTeq7
MLI5E9KxtlRUqVlddL8SEryOS39iqk8B5kmyVQgeNccWHHFl4NzCxoXiZ+j1PPFZ
DSOvzv/xN903QvV07+0y5TaxAN6eKk1dgK5fERhW0ydXfd+H/XRLeymY/6m8EA0g
tpO31oTWTQDfZhbuYBpSLbm8U090a+XO8e5jVsUWHDStxCq0wEPv6YtXXrmGrSn7
E5/ZhlAEIgIgSTiBYsNnmZOePfUlk1L5aA7Cy6yAdgeo1bf98tjdNtA=
-----END PRIVATE KEY-----

Figure 12: Private key stored in the txt

public_key

File Edit View

-----BEGIN PUBLIC KEY-----
MIIDRzCCAjoGByqGSM44BAEwggItAoIBAQCs/SWonTfDoiKMDsQ/AjbZXQIrXaou
KKtxs1JLuNBvpI6ZK3kgMZ3mp8ExKVOpuDjAu9lnOTXL+aw32cpLC6LD9i5N+/2u
f4AHYSCJt7hxqN7s9DZK0Z+3SFv958Hy3Yvd86Dt6gyvei4qMEP9wcQvClv+kixI
YeL6s0ar8N8rTHCkJadbCiLBW1r16usvMmTv5Hzl0v/4Uk7xrYjVlMb0qy/HLzo4
1Ho+w7KLJXg16Ox7lhPnziRTAmJwfbDQgsVtUo4aQ57waFwr4Kff+IwUQ6Vm8TFb
hu5ZsC0L5yJILJPQL+onde98AKS62eafZYdX8j9XtiAWZFyJhyJ+GnnzAiEA7jE8
v0RPrnlZbpyVS9DoEMrhVZGwHUKFezKP/o6FY9MCggEBAIkiR7ladYwiXB8+Ugc9
O0mmyxAMT4NJFohPtO22QeziMwzyLZYtLxqRD9oiOb8V9XHEc/ZUS5xQNnEoJDge
2fRN0sS2WVphEzzvKkADmGAIkQQNsFI8DyCiVuLvKECHIkKrvTHmljxfTeq7MLI5
E9KxtlRUqVlddL8SEryOS39iqk8B5kmyVQgeNccWHHFl4NzCxoXiZ+j1PPFZDSOv
zv/xN903QvV07+0y5TaxAN6eKk1dgK5fERhW0ydXfd+H/XRLeymY/6m8EA0gtpO3
1oTWTQDfZhbuYBpSLbm8U090a+XO8e5jVsUWHDStxCq0wEPv6YtXXrmGrSn7E5/Z
hlADggEFAAKCAQBX9NtSLF0PMvacR2RbqlCsSaGkzv3NqsIigoX1POXyQx2l6ZGn
0ZnMqaTx0jK4xCGAwdGIa8ggi+F3s3E2HSuG87z7p5SsLQtIReNCgimo2EXjJdVO
HCbrlK4D6S0H75Ou7P8YGyHt/UZrOnOxfD5bkAFay+D41LM2lhs1R4Md/a02/zAL
6jE9TxQuUGwyWleamDispubif3Q7kvT21Znazh3e32rsCYVS5ZyIKQ0n6mQFQfJX
c7eP8JeVgar7OpqEGwoyWDGMu50Px1RJWP1yMhxFmh4AiXefO/gw9Y9Pe6fBEA9t
oJukCWxUrER8jYT46VKzivYicHwnC05hPFWB
-----END PUBLIC KEY-----

Figure 13: Public key stored in the txt

signature

File Edit View

30440220251d8bfc6a683a1ad1f3798b9b24639fb1ca5f12a6e2357d00dc4856397f60
cf022078d931b13fdcd78d765443db01f68bd8217656ae27e4ecf1288085abc7f5d971

Figure 14: Signature store in the txt

```
Message 1: Hello, World!
```

```
================================================================================
|                               Signing Message                                |
================================================================================
<Importing private key for sign>
Original message hash              : b'\xdf\xfd`!\xbb+\xd5\xb0\xafgb\x90\x80\x9e\xc3\xa51\x91\xdd\x81\xc7\xf7\nK(h\x
8a6!\x82\x98o'
Signature of original message hash : b'0D\x02 \x1d\x8c\x7f!3\xe1^O\xaa\xb14.\xaec\xee\x11\x9a\xec\xa6B\x9a\x8ec~\xd5
\xf5>\x05\xc1\xa1T\xcd\x02 x\x85\x0c\x95+H\x8f\x80"\xec^\xad\x17\x88\x9cbLl.\xc5\xd9\xcet/w\xde0\xec;j\x10R'
Filename: C:\Users\summe\source\repos\Blowfish_Coding Phase 3_Simple DSA Program\Blowfish_Coding Phase 3_Simple DSA
Program\Blowfish_Coding_Phase_3_Simple_DSA_Program.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    54     44.2 MiB     44.2 MiB           1   @profile()
    55                                         def sign_message(private_key, message):
    56     44.2 MiB      0.0 MiB           1       print_box("Signing Message")
    57     44.2 MiB      0.0 MiB           1       message_hash = hashlib.sha256(message.encode('utf-8')).digest()
    58     44.2 MiB      0.0 MiB           1       start_time_hash = time.time()  # Start measuring time for hashing

    59     44.2 MiB      0.0 MiB           1       print("<Importing private key for sign>")
    60
    61     44.2 MiB      0.0 MiB           2       signature = private_key.sign(
    62     44.2 MiB      0.0 MiB           1           message_hash,
    63     44.2 MiB      0.0 MiB           1           utils.Prehashed(hashes.SHA256())
    64                                             ) # Sign the hashed message using the private key
    65
    66     44.2 MiB      0.0 MiB           1       end_time_hash = time.time()  # End measuring time for hashing
    67                                             # The execution time for hashing the message
    68     44.2 MiB      0.0 MiB           1       execution_time_hash = end_time_hash - start_time_hash
    69
    70     44.2 MiB      0.0 MiB           1       execution_time = execution_time_hash
    71     44.2 MiB      0.0 MiB           1       print("Original message hash              :", message_hash)
    72     44.2 MiB      0.0 MiB           1       print("Signature of original message hash :", signature)
    73
    74                                             # The hexadecimal representation of the signature
    75     44.2 MiB      0.0 MiB           1       signature_hex = signature.hex()
    76
    77                                             # Store the signature in a text file
    78     44.2 MiB      0.0 MiB           1       with open('signature.txt', 'w') as file:
    79     44.2 MiB      0.0 MiB           1           file.write(signature_hex)
    80
    81     44.2 MiB      0.0 MiB           1       return signature_hex, execution_time


Time taken for signing       : 0.009687423706054688 seconds
Message signing memory usage : 44.19921875 KB
```

Figure 15: Output of Phase 3 by Machine A for signing the Message 1

```
Time taken for signing       : 0.009687423706054688 seconds
Message signing memory usage : 44.19921875 KB
```

Figure 16: This time excluded the file execution, time writing in the txt is not counted

```
================================================================================
|                             Validating Signature                             |
================================================================================
<Importing public key for validating signature>
Filename: C:\Users\summe\source\repos\Blowfish_Coding Phase 3_Simple DSA Program\Blowfish_Coding Phase 3_Simple DSA
Program\Blowfish_Coding_Phase_3_Simple_DSA_Program.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    85     44.2 MiB     44.2 MiB           1   @profile()
    86                                         def verify_signature(public_key, message, signature_hex):
    87     44.2 MiB      0.0 MiB           1       print_box("Validating Signature")
    88     44.2 MiB      0.0 MiB           1       print("<Importing public key for validating signature>")
    89
    90                                             # Read the signature from the text file
    91     44.2 MiB      0.0 MiB           1       with open('signature.txt', 'r') as file:
    92     44.2 MiB      0.0 MiB           1           signature_hex = file.read().strip()
    93
    94                                             # Convert the signature from hexadecimal to bytes
    95     44.2 MiB      0.0 MiB           1       signature = bytes.fromhex(signature_hex)
    96
    97                                             # Hash the original message
    98     44.2 MiB      0.0 MiB           1       message_hash = hashlib.sha256(message.encode('utf-8')).digest()
    99     44.2 MiB      0.0 MiB           1       start_time = time.time()
   100
   101     44.2 MiB      0.0 MiB           1       try:
   102     44.2 MiB      0.0 MiB           2           public_key.verify(
   103     44.2 MiB      0.0 MiB           1               signature,
   104     44.2 MiB      0.0 MiB           1               message_hash,
   105     44.2 MiB      0.0 MiB           1               utils.Prehashed(hashes.SHA256())
   106                                                 ) # Verify the signature using the public key
   107     44.2 MiB      0.0 MiB           1           end_time = time.time()
   108     44.2 MiB      0.0 MiB           1           execution_time = end_time - start_time
   109     44.2 MiB      0.0 MiB           1           return True, execution_time
   110                                             except Exception:
   111                                                 return False, None

Time taken for verification         : 0.021506547927856445 seconds
Signature validation memory usage   : 44.22265625 KB

Condition        : The message is authentic.
Original Message : Hello, World!
```

Figure 17: Output of Phase 3 by Machine A for signature verification on Message 1

```
Time taken for verification         : 0.021506547927856445 seconds
Signature validation memory usage   : 44.22265625 KB
```

Figure 18: This time excluded the file execution, time reading from the txt is not counted.

```
Condition        : The message is authentic.
Original Message : Hello, World!
```

Figure 19: The message verification shows that the message is authentic as the signature is valid.

```
Message 2: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer pretium, lacus a consequat suscipit, mag
na sem sagittis erat, vitae posuere ex ex sed justo. Curabitur commodo ex lectus, in consequat nisi pellentesque a.
Suspendisse id malesuada lectus, nec elementum neque. Nullam venenatis eros sit amet odio pharetra, id varius tellus
 auctor. Curabitur sit amet ex eget leo laoreet pulvinar. Nulla facilisi. Fusce ac mauris ut mauris semper aliquam i
n et est. Donec condimentum neque a interdum consectetur. Nulla facilisi. Donec venenatis dapibus mauris at iaculis.
```

```
================================================================================
|                               Signing Message                                |
================================================================================
<Importing private key for sign>
Original message hash             : b'\xb7\xdc\xd4\x9e\xa15\xdd\xde\x9a\x902\x9c*\x02\xe8-\xe2(\xefL\xb1\xf9)\x18\x
a4\xfe,\x12\xe8\xd0\xc3\xb9'
Signature of original message hash : b"0E\x02!\x00\x91\x0c\xbd \xa7qY\xec{\x04\xbb\xa0\xd6-\x11}l\xa9X\x11\xf4\xf3Fr
\xf7]\x1e:/\x8a\x05\t\x02 h\xc7TQP\x0c'\xd7J\xbfE\xfd(9\xcfeI7=\xf8\xb3\xb2g\xa9\x8d\xacm}\xc0\x00\xe7\x06"
Filename: C:\Users\summe\source\repos\Blowfish_Coding Phase 3_Simple DSA Program\Blowfish_Coding Phase 3_Simple DSA
Program\Blowfish_Coding_Phase_3_Simple_DSA_Program.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    54     44.2 MiB     44.2 MiB           1   @profile()
    55                                         def sign_message(private_key, message):
    56     44.2 MiB      0.0 MiB           1       print_box("Signing Message")
    57     44.2 MiB      0.0 MiB           1       message_hash = hashlib.sha256(message.encode('utf-8')).digest()
    58     44.2 MiB      0.0 MiB           1       start_time_hash = time.time()  # Start measuring time for hashing

    59     44.2 MiB      0.0 MiB           1       print("<Importing private key for sign>")
    60
    61     44.2 MiB      0.0 MiB           2       signature = private_key.sign(
    62     44.2 MiB      0.0 MiB           1           message_hash,
    63     44.2 MiB      0.0 MiB           1           utils.Prehashed(hashes.SHA256())
    64                                             ) # Sign the hashed message using the private key
    65
    66     44.2 MiB      0.0 MiB           1       end_time_hash = time.time()  # End measuring time for hashing
    67                                             # The execution time for hashing the message
    68     44.2 MiB      0.0 MiB           1       execution_time_hash = end_time_hash - start_time_hash
    69
    70     44.2 MiB      0.0 MiB           1       execution_time = execution_time_hash
    71     44.2 MiB      0.0 MiB           1       print("Original message hash              :", message_hash)
    72     44.2 MiB      0.0 MiB           1       print("Signature of original message hash :", signature)
    73
    74                                             # The hexadecimal representation of the signature
    75     44.2 MiB      0.0 MiB           1       signature_hex = signature.hex()
    76
    77                                             # Store the signature in a text file
    78     44.2 MiB      0.0 MiB           1       with open('signature.txt', 'w') as file:
    79     44.2 MiB      0.0 MiB           1           file.write(signature_hex)
    80
    81     44.2 MiB      0.0 MiB           1       return signature_hex, execution_time


Time taken for signing       : 0.007771492004394531 seconds
Message signing memory usage : 44.23046875 KB
```

Figure 20: Output of Phase 3 by Machine A for signing the Message 2

```
Time taken for signing       : 0.007771492004394531 seconds
Message signing memory usage : 44.23046875 KB
```

Figure 21: This time excluded the file execution, time writing in the txt is not counted

```
================================================================================
|                             Validating Signature                             |
================================================================================
<Importing public key for validating signature>
Filename: C:\Users\summe\source\repos\Blowfish_Coding Phase 3_Simple DSA Program\Blowfish_Coding Phase 3_Simple DSA Program\Blowfish_Coding_Phase_3_Simple_DSA_Program.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    85     44.3 MiB     44.3 MiB           1   @profile()
    86                                         def verify_signature(public_key, message, signature_hex):
    87     44.3 MiB      0.0 MiB           1       print_box("Validating Signature")
    88     44.3 MiB      0.0 MiB           1       print("<Importing public key for validating signature>")
    89
    90                                             # Read the signature from the text file
    91     44.3 MiB      0.0 MiB           1       with open('signature.txt', 'r') as file:
    92     44.3 MiB      0.0 MiB           1           signature_hex = file.read().strip()
    93
    94                                             # Convert the signature from hexadecimal to bytes
    95     44.3 MiB      0.0 MiB           1       signature = bytes.fromhex(signature_hex)
    96
    97                                             # Hash the original message
    98     44.3 MiB      0.0 MiB           1       message_hash = hashlib.sha256(message.encode('utf-8')).digest()
    99     44.3 MiB      0.0 MiB           1       start_time = time.time()
   100
   101     44.3 MiB      0.0 MiB           1       try:
   102     44.3 MiB      0.0 MiB           2           public_key.verify(
   103     44.3 MiB      0.0 MiB           1               signature,
   104     44.3 MiB      0.0 MiB           1               message_hash,
   105     44.3 MiB      0.0 MiB           1               utils.Prehashed(hashes.SHA256())
   106                                                 ) # Verify the signature using the public key
   107     44.3 MiB      0.0 MiB           1           end_time = time.time()
   108     44.3 MiB      0.0 MiB           1           execution_time = end_time - start_time
   109     44.3 MiB      0.0 MiB           1           return True, execution_time
   110                                             except Exception:
   111                                                 return False, None


Time taken for verification         : 0.012009620666503906 seconds
Signature validation memory usage   : 44.265625 KB

Condition        : The message is authentic.
Original Message : Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer pretium, lacus a consequat susci
pit, magna sem sagittis erat, vitae posuere ex ex sed justo. Curabitur commodo ex lectus, in consequat nisi pellente
sque a. Suspendisse id malesuada lectus, nec elementum neque. Nullam venenatis eros sit amet odio pharetra, id variu
s tellus auctor. Curabitur sit amet ex eget leo laoreet pulvinar. Nulla facilisi. Fusce ac mauris ut mauris semper a
liquam in et est. Donec condimentum neque a interdum consectetur. Nulla facilisi. Donec venenatis dapibus mauris at
iaculis.
```

Figure 22: Output of Phase 3 by Machine A for signature verification on Message 2


```
Time taken for verification         : 0.012009620666503906 seconds
Signature validation memory usage   : 44.265625 KB

Condition        : The message is authentic.
Original Message : Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer pretium, lacus a consequat susci
pit, magna sem sagittis erat, vitae posuere ex ex sed justo. Curabitur commodo ex lectus, in consequat nisi pellente
sque a. Suspendisse id malesuada lectus, nec elementum neque. Nullam venenatis eros sit amet odio pharetra, id variu
s tellus auctor. Curabitur sit amet ex eget leo laoreet pulvinar. Nulla facilisi. Fusce ac mauris ut mauris semper a
liquam in et est. Donec condimentum neque a interdum consectetur. Nulla facilisi. Donec venenatis dapibus mauris at
iaculis.
```


Figure 23: This time excluded the file execution, time reading from the txt is not counted. The message verification shows that the message is authentic as the signature is valid.

Message 3: Curiosity is a remarkable human trait that drives us to explore, question, and seek knowledge about the world around us. It is the spark that ignites our imagination, propelling us on a lifelong journey of discovery. From our earliest years, curiosity is inherent in our nature, driving us to touch, taste, and explore our environment. As we grow, this innate curiosity blossoms, shaping our identities and influencing the paths we choose to follow. One of the most significant aspects of curiosity is its ability to fuel learning. When we approach new ideas, subjects, or experiences with a curious mindset, we are open to the possibilities they hold. Curiosity compels us to ask questions, seek answers, and engage in critical thinking. It challenges us to delve deeper, to unravel the complexities of the world and expand our knowledge. By nurturing curiosity, we embrace a lifelong pursuit of learning, continuously evolving and growing intellectually. Curiosity also plays a pivotal role in fostering creativity. When we allow our curiosity to roam freely, we invite inspiration and innovation into our lives. Curiosity compels us to explore uncharted territories, pushing the boundaries of what is known and comfortable. It encourages us to challenge assumptions, think outside the box, and connect seemingly unrelated ideas. By embracing our natural curiosity, we unlock the potential to create, invent, and bring forth new ideas that can shape the world around us. Beyond its influence on personal growth, curiosity has far-reaching implications for society as a whole. Curiosity serves as a catalyst for progress and advancement. Throughout history, great discoveries and inventions have been born out of a relentless pursuit of answers driven by curiosity. From the exploration of distant lands to scientific breakthroughs, every leap forward is the result of individuals and communities driven by an insatiable curiosity about the unknown. Moreover, curiosity fosters empathy and understanding. When we approach others with genuine curiosity, we open the doors to deeper connections and meaningful relationships. By seeking to understand different perspectives, cultures, and experiences, we break down barriers and bridge divides. Curiosity allows us to appreciate the richness and diversity of the human tapestry, promoting tolerance and acceptance in an increasingly interconnected world. However, despite its profound potential, curiosity can be stifled if not nurtured and encouraged. In a society that often values conformity and certainty, curiosity can be viewed as a disruption or a distraction. The pressure to conform and the fear of the unknown can dampen the flames of curiosity, leading to missed opportunities and limited growth. As a society, we must recognize the transformative power of curiosity and create environments that foster its growth in individuals of all ages. To cultivate curiosity, education plays a vital role. Schools should embrace inquiry-based learning, encouraging students to ask questions, explore their interests, and pursue independent research. Teachers can serve as guides, nurturing curiosity by creating a safe space for exploration and discovery. By incorporating real-world applications and promoting interdisciplinary approaches, education can unleash the full potential of curiosity, preparing students to become lifelong learners and active contributors to society. In addition to education, parents and caregivers also have a crucial role to play in cultivating curiosity. By encouraging exploration, providing opportunities for hands-on learning, and celebrating curiosity-driven achievements, parents can foster a lifelong love of learning in their children. They can instill a sense of wonder, inspire critical thinking, and nurture the natural inquisitiveness that lies within each child. In conclusion, curiosity is a powerful force that drives human progress, fuels creativity, and fosters understanding. It is the key to unlocking new knowledge, inspiring innovation, and forging connections. By nurturing and embracing curiosity, both as individuals and as a society, we unlock a world of possibilities, paving the way for a future built on continuous growth, discovery, and enlightenment. Let us celebrate curiosity and embark on a journey of endless exploration and learning, for it is through curiosity that we truly come alive.

```
==============================================================================
|                              Signing Message                               |
==============================================================================
<Importing private key for sign>
Original message hash             : b'\xa1\xfa\xcf\xfa\x98\xbc\x88\xb7\x9dQ\xa24\x0f?\xadVW7\xed/\xbc\x8cfI\xfe\xf9
\x98\xf1\x87\x93\x03\xba'
Signature of original message hash : b"0D\x02 %\x1d\x8b\xfcjh:\x1a\xd1\xf3y\x8b\x9b$c\x9f\xb1\xca_\x12\xa6\xe25}\x00
\xdcHV9\x7f`\xcf\x02 x\xd91\xb1?\xdc\xd7\x8dvTC\xdb\x01\xf6\x8b\xd8!vV\xae'\xe4\xec\xf1(\x80\x85\xab\xc7\xf5\xd9q"
Filename: C:\Users\summe\source\repos\Blowfish_Coding Phase 3_Simple DSA Program\Blowfish_Coding Phase 3_Simple DSA
Program\Blowfish_Coding_Phase_3_Simple_DSA_Program.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    54     44.4 MiB     44.4 MiB           1   @profile()
    55                                         def sign_message(private_key, message):
    56     44.4 MiB      0.0 MiB           1       print_box("Signing Message")
    57     44.4 MiB      0.0 MiB           1       message_hash = hashlib.sha256(message.encode('utf-8')).digest()
    58     44.4 MiB      0.0 MiB           1       start_time_hash = time.time()  # Start measuring time for hashing

    59     44.4 MiB      0.0 MiB           1       print("<Importing private key for sign>")
    60
    61     44.4 MiB      0.0 MiB           2       signature = private_key.sign(
    62     44.4 MiB      0.0 MiB           1           message_hash,
    63     44.4 MiB      0.0 MiB           1           utils.Prehashed(hashes.SHA256())
    64                                             ) # Sign the hashed message using the private key
    65
    66     44.4 MiB      0.0 MiB           1       end_time_hash = time.time()  # End measuring time for hashing
    67                                             # The execution time for hashing the message
    68     44.4 MiB      0.0 MiB           1       execution_time_hash = end_time_hash - start_time_hash
    69
    70     44.4 MiB      0.0 MiB           1       execution_time = execution_time_hash
    71     44.4 MiB      0.0 MiB           1       print("Original message hash              :", message_hash)
    72     44.4 MiB      0.0 MiB           1       print("Signature of original message hash :", signature)
    73
    74                                             # The hexadecimal representation of the signature
    75     44.4 MiB      0.0 MiB           1       signature_hex = signature.hex()
    76
    77                                             # Store the signature in a text file
    78     44.4 MiB      0.0 MiB           1       with open('signature.txt', 'w') as file:
    79     44.4 MiB      0.0 MiB           1           file.write(signature_hex)
    80
    81     44.4 MiB      0.0 MiB           1       return signature_hex, execution_time


Time taken for signing       : 0.009247779846191406 seconds
Message signing memory usage : 44.359375 KB
```

Figure 24: Output of Phase 3 by Machine A for signing the Message 3

```
Time taken for signing       : 0.009247779846191406 seconds
Message signing memory usage : 44.359375 KB
```

Figure 25: This time excluded the file execution, time writing in the txt is not counted

```
================================================================================
|                           Validating Signature                               |
================================================================================
<Importing public key for validating signature>
Filename: C:\Users\summe\source\repos\Blowfish_Coding Phase 3_Simple DSA Program\Blowfish_Coding Phase 3_Simple DSA
Program\Blowfish_Coding_Phase_3_Simple_DSA_Program.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    85     44.4 MiB     44.4 MiB           1   @profile()
    86                                         def verify_signature(public_key, message, signature_hex):
    87     44.4 MiB      0.0 MiB           1       print_box("Validating Signature")
    88     44.4 MiB      0.0 MiB           1       print("<Importing public key for validating signature>")
    89
    90                                             # Read the signature from the text file
    91     44.4 MiB      0.0 MiB           1       with open('signature.txt', 'r') as file:
    92     44.4 MiB      0.0 MiB           1           signature_hex = file.read().strip()
    93
    94                                             # Convert the signature from hexadecimal to bytes
    95     44.4 MiB      0.0 MiB           1       signature = bytes.fromhex(signature_hex)
    96
    97                                             # Hash the original message
    98     44.4 MiB      0.0 MiB           1       message_hash = hashlib.sha256(message.encode('utf-8')).digest()
    99     44.4 MiB      0.0 MiB           1       start_time = time.time()
   100
   101     44.4 MiB      0.0 MiB           1       try:
   102     44.4 MiB      0.0 MiB           2           public_key.verify(
   103     44.4 MiB      0.0 MiB           1               signature,
   104     44.4 MiB      0.0 MiB           1               message_hash,
   105     44.4 MiB      0.0 MiB           1               utils.Prehashed(hashes.SHA256())
   106                                                 ) # Verify the signature using the public key
   107     44.4 MiB      0.0 MiB           1           end_time = time.time()
   108     44.4 MiB      0.0 MiB           1           execution_time = end_time - start_time
   109     44.4 MiB      0.0 MiB           1           return True, execution_time
   110                                             except Exception:
   111                                                 return False, None


Time taken for verification          : 0.015234947204589844 seconds
Signature validation memory usage    : 44.3671875 KB

Condition        : The message is authentic.
Original Message : Curiosity is a remarkable human trait that drives us to explore, question, and seek knowledge abo
ut the world around us. It is the spark that ignites our imagination, propelling us on a lifelong journey of discove
ry. From our earliest years, curiosity is inherent in our nature, driving us to touch, taste, and explore our enviro
nment. As we grow, this innate curiosity blossoms, shaping our identities and influencing the paths we choose to fol
low. One of the most significant aspects of curiosity is its ability to fuel learning. When we approach new ideas, s
ubjects, or experiences with a curious mindset, we are open to the possibilities they hold. Curiosity compels us to
ask questions, seek answers, and engage in critical thinking. It challenges us to delve deeper, to unravel the compl
exities of the world and expand our knowledge. By nurturing curiosity, we embrace a lifelong pursuit of learning, co
```

```
ntinuously evolving and growing intellectually. Curiosity also plays a pivotal role in fostering creativity. When we
 allow our curiosity to roam freely, we invite inspiration and innovation into our lives. Curiosity compels us to ex
plore uncharted territories, pushing the boundaries of what is known and comfortable. It encourages us to challenge
assumptions, think outside the box, and connect seemingly unrelated ideas. By embracing our natural curiosity, we un
lock the potential to create, invent, and bring forth new ideas that can shape the world around us. Beyond its influ
ence on personal growth, curiosity has far-reaching implications for society as a whole. Curiosity serves as a catal
yst for progress and advancement. Throughout history, great discoveries and inventions have been born out of a relen
tless pursuit of answers driven by curiosity. From the exploration of distant lands to scientific breakthroughs, eve
ry leap forward is the result of individuals and communities driven by an insatiable curiosity about the unknown. Mo
reover, curiosity fosters empathy and understanding. When we approach others with genuine curiosity, we open the doo
rs to deeper connections and meaningful relationships. By seeking to understand different perspectives, cultures, an
d experiences, we break down barriers and bridge divides. Curiosity allows us to appreciate the richness and diversi
ty of the human tapestry, promoting tolerance and acceptance in an increasingly interconnected world. However, despi
te its profound potential, curiosity can be stifled if not nurtured and encouraged. In a society that often values c
onformity and certainty, curiosity can be viewed as a disruption or a distraction. The pressure to conform and the f
ear of the unknown can dampen the flames of curiosity, leading to missed opportunities and limited growth. As a soci
ety, we must recognize the transformative power of curiosity and create environments that foster its growth in indiv
iduals of all ages. To cultivate curiosity, education plays a vital role. Schools should embrace inquiry-based learn
ing, encouraging students to ask questions, explore their interests, and pursue independent research. Teachers can s
erve as guides, nurturing curiosity by creating a safe space for exploration and discovery. By incorporating real-wo
rld applications and promoting interdisciplinary approaches, education can unleash the full potential of curiosity,
preparing students to become lifelong learners and active contributors to society. In addition to education, parents
 and caregivers also have a crucial role to play in cultivating curiosity. By encouraging exploration, providing opp
ortunities for hands-on learning, and celebrating curiosity-driven achievements, parents can foster a lifelong love
of learning in their children. They can instill a sense of wonder, inspire critical thinking, and nurture the natura
l inquisitiveness that lies within each child. In conclusion, curiosity is a powerful force that drives human progre
ss, fuels creativity, and fosters understanding. It is the key to unlocking new knowledge, inspiring innovation, and
 forging connections. By nurturing and embracing curiosity, both as individuals and as a society, we unlock a world
of possibilities, paving the way for a future built on continuous growth, discovery, and enlightenment. Let us celeb
rate curiosity and embark on a journey of endless exploration and learning, for it is through curiosity that we trul
y come alive.
```

Figure 26: Output of Phase 3 by Machine A for signature verification on Message 3

```
Time taken for verification         : 0.015234947204589844 seconds
Signature validation memory usage   : 44.3671875 KB

Condition        : The message is authentic.
Original Message : Curiosity is a remarkable human trait that drives us to explore, question, and seek knowledge abo
ut the world around us. It is the spark that ignites our imagination, propelling us on a lifelong journey of discove
ry. From our earliest years, curiosity is inherent in our nature, driving us to touch, taste, and explore our enviro
nment. As we grow, this innate curiosity blossoms, shaping our identities and influencing the paths we choose to fol
low. One of the most significant aspects of curiosity is its ability to fuel learning. When we approach new ideas, s
ubjects, or experiences with a curious mindset, we are open to the possibilities they hold. Curiosity compels us to
ask questions, seek answers, and engage in critical thinking. It challenges us to delve deeper, to unravel the compl
exities of the world and expand our knowledge. By nurturing curiosity, we embrace a lifelong pursuit of learning, co
ntinuously evolving and growing intellectually. Curiosity also plays a pivotal role in fostering creativity. When we
 allow our curiosity to roam freely, we invite inspiration and innovation into our lives. Curiosity compels us to ex
plore uncharted territories, pushing the boundaries of what is known and comfortable. It encourages us to challenge
assumptions, think outside the box, and connect seemingly unrelated ideas. By embracing our natural curiosity, we un
```

Figure 27: This time excluded the file execution, time reading from the txt is not counted. The message verification shows that the message is authentic as the signature is valid.

Message 1: Hello, World! (2 words)

Performance Report

Algorithm	Time Complexity
Public and Private Key Pair Generation	0.924609 seconds
Encryption : Message Signing	0.983675 seconds
Decryption : Signature Validation	0.986379 seconds
Signature Execution	0.001016 seconds
Overall Process	2.894662 seconds

Algorithm	Memory Usage
Public and Private Key Pair Generation	44.152344 KB
Encryption : Message Signing	44.199219 KB
Decryption : Signature Validation	44.222656 KB
Signature Execution	44.199219 KB
Overall Process	132.574219 KB

Message 2: Lorem ipsum dolor sit amet,…(85 words)

Performance Report

Algorithm	Time Complexity
Public and Private Key Pair Generation	0.924609 seconds
Encryption : Message Signing	0.974469 seconds
Decryption : Signature Validation	1.030369 seconds
Signature Execution	0.000504 seconds
Overall Process	2.929447 seconds

Algorithm	Memory Usage
Public and Private Key Pair Generation	44.152344 KB
Encryption : Message Signing	44.230469 KB
Decryption : Signature Validation	44.265625 KB
Signature Execution	44.230469 KB
Overall Process	132.648438 KB

Message 3: Curiosity is a remarkable human trait… (654 words)

Performance Report	
Algorithm	**Time Complexity**
Public and Private Key Pair Generation	0.924609 seconds
Encryption : Message Signing	0.981928 seconds
Decryption : Signature Validation	0.975550 seconds
Signature Execution	0.001086 seconds
Overall Process	2.882087 seconds

Algorithm	Memory Usage
Public and Private Key Pair Generation	44.152344 KB
Encryption : Message Signing	44.359375 KB
Decryption : Signature Validation	44.367188 KB
Signature Execution	44.359375 KB
Overall Process	132.878906 KB

The time complexity table shows different time from the output on the above section. It is because the time complexities at here include the whole process and take in the time of file execution like reading and writing from the file. While the time from the section above did not include file execution, which only counted the process of generating, hashing and verification only.

In the case of our program at Phase 3 and Phase 4, the overall time complexity is obtained by adding the time taken for key pair generation, message signing, and signature validation. The formula to calculate the overall time complexity is as follows:

overall_time_complexity = key_pair_time + sign_time + validate_time

In the given performance report, the overall time complexity is calculated by summing the time taken for key pair generation, message signing, and signature validation. However, the time taken for signature execution is not included in the overall time complexity. The reason for excluding the time taken for signature execution from the overall time complexity is that it is a separate step that occurs after the key pair generation, message signing, and signature validation processes. The signature execution step involves creating a signature by applying a cryptographic algorithm to a message using a private key. It is not directly related to the generation of key pairs or the signing and validation of messages. Including the time taken for signature execution in the overall time complexity calculation would give a misleading representation of the performance of the cryptographic process as a whole. Since the signature execution step has a significantly lower time complexity compared to the other steps (0.001236 seconds in the provided output), it would have a negligible impact on the overall time complexity. Therefore, it is reasonable to exclude the time taken for signature execution from the overall time complexity calculation to focus on the main steps of the cryptographic process.

In the case of "Encryption: Message Signing" and "Signature Execution" having the same memory usage, it could be due to them using similar data structures or memory allocation patterns. The reason for not including the "Signature Execution" step in the calculation of overall memory usage is because it is not a separate step that requires additional memory. "Signature Execution" refers to the process of executing or applying the signature. However, this step does not involve any additional memory usage beyond what has already been accounted for in the previous steps: key pair generation, message signing, and signature validation.

Comparison Between Different Lengths of Message

Based on these results, we can observe that the time complexity for each step remains relatively consistent regardless of the message length. The key pair generation consistently takes around 0.925 seconds. The encryption (message signing) and decryption (signature validation) steps have slightly varying times, but they are still within a similar range.

Comparing the three different message lengths, we can conclude that the time complexity does not significantly vary based on the length of the message being signed and verified. This observation suggests that the time complexity is primarily dependent on the cryptographic operations and key generation rather than the size of the input message.

Therefore, we can infer that the time complexity of the signature generation and validation process is relatively efficient and not directly proportional to the length of the message being processed. This characteristic makes the cryptographic algorithm suitable for handling messages of various sizes efficiently.

The time complexity for signature execution remains relatively constant and negligible across all message lengths. This indicates that the actual signing or verification of the signature is efficient and does not significantly impact the overall performance of the DSA algorithm. In the performance reports provided, the time taken for signature execution is consistently low, with values around 0.001 seconds.

Then, the overall time complexity of the DSA algorithm increases with shorter messages, in which Message 1 with shortest length takes longer time than Message 3. However, in the case of Message 2, despite being medium in length, it is possible that the message content results in a more complicated hashing process, leading to an increase time complexity causing it to become the slowest among all 3 messages. The time required to hash a message is generally proportional to the length of the message, such as Message 3 longest and fastest. However, this relationship does not necessarily hold true for all messages. The nature of the message content and its unique characteristics can influence the hashing process.

From another perspective, the memory usage for key pair generation remains constant across all message lengths. Similar to time complexity, key pair generation primarily depends on the algorithm and key size, rather than the message length. In the provided performance reports, the memory usage for key pair generation using the DSA algorithm is approximately 44.152344 KB.

The memory usage for message signing and signature validation slightly increases as the message length increases. This is expected, as larger message data and additional intermediate computations associated with longer messages require more memory resources.

The memory usage for signature execution remains relatively constant and negligible. This suggests that the memory requirements for the actual signing or verification process in the DSA algorithm are minimal and do not significantly impact the overall memory usage. The provided performance reports consistently show low and stable memory usage for signature execution.

The overall memory usage of the DSA algorithm increases slightly with longer messages, reflecting the increased memory demands during message signing and signature validation.

2.4.2 Phase 4 - Modified DSA Program

Machine A

```
================================================================================
|                              Generating Key Pair                             |
================================================================================
Private key object                 : <cryptography.hazmat.backends.openssl.rsa._RSAPrivateKey object at 0x00000228AD
293280>
Public key object                  : <cryptography.hazmat.backends.openssl.rsa._RSAPublicKey object at 0x00000228AD2
93550>
Filename: C:\Users\summe\source\repos\Group Blowfish_Phase 4_Modified DSA Program\Group Blowfish_Phase 4_Modified DS
A Program\Group_Blowfish_Phase_4_Modified_DSA_Program.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    29     44.1 MiB     44.1 MiB           1   @profile
    30                                         def generate_keys():
    31     44.1 MiB      0.0 MiB           1       print_box("Generating Key Pair")
    32                                             # Use RSA algorithm
    33     44.1 MiB      0.0 MiB           2       private_key = rsa.generate_private_key(
    34     44.1 MiB      0.0 MiB           1           public_exponent=65537,
    35     44.1 MiB      0.0 MiB           1           key_size=2048
    36                                             )
    37     44.1 MiB      0.0 MiB           1       public_key = private_key.public_key()
    38
    39                                             # Serialize the private key to PEM format
    40     44.1 MiB      0.0 MiB           2       private_key_pem = private_key.private_bytes(
    41     44.1 MiB      0.0 MiB           1           encoding=serialization.Encoding.PEM,
    42     44.1 MiB      0.0 MiB           1           format=serialization.PrivateFormat.PKCS8,
    43     44.1 MiB      0.0 MiB           1           encryption_algorithm=serialization.NoEncryption()
    44                                             )
    45
    46                                             # Write the serialized private key to a text file
    47     44.1 MiB      0.0 MiB           1       with open('private_key.txt', 'wb') as file:
    48     44.1 MiB      0.0 MiB           1           file.write(private_key_pem)
    49
    50                                             # Serialize the public key to PEM format
    51     44.1 MiB      0.0 MiB           2       public_key_pem = public_key.public_bytes(
    52     44.1 MiB      0.0 MiB           1           encoding=serialization.Encoding.PEM,
    53     44.1 MiB      0.0 MiB           1           format=serialization.PublicFormat.SubjectPublicKeyInfo
    54                                             )
    55
    56                                             # Write the serialized public key to a text file
    57     44.1 MiB      0.0 MiB           1       with open('public_key.txt', 'wb') as file:
    58     44.1 MiB      0.0 MiB           1           file.write(public_key_pem)
    59
    60     44.1 MiB      0.0 MiB           1       print("Private key object                 :", private_key)
    61     44.1 MiB      0.0 MiB           1       print("Public key object                  :", public_key)
    62     44.1 MiB      0.0 MiB           1       return private_key, public_key


Time taken for key generation      : 0.41829872131347656 seconds
Key pair generation memory usage   : 44.14453125 KB
```

Figure 28: Output of Phase 4 by Machine A for key pair generation

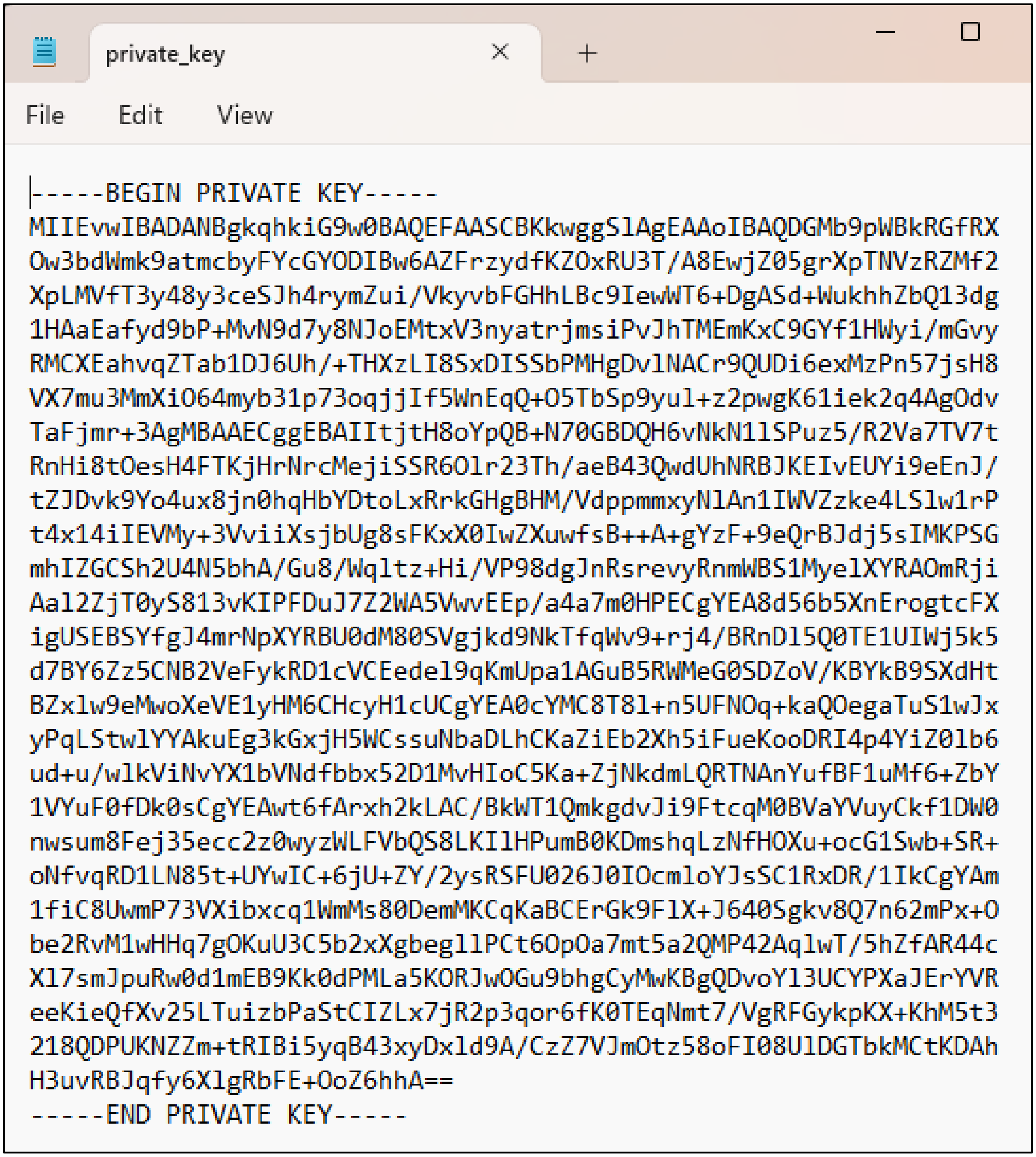
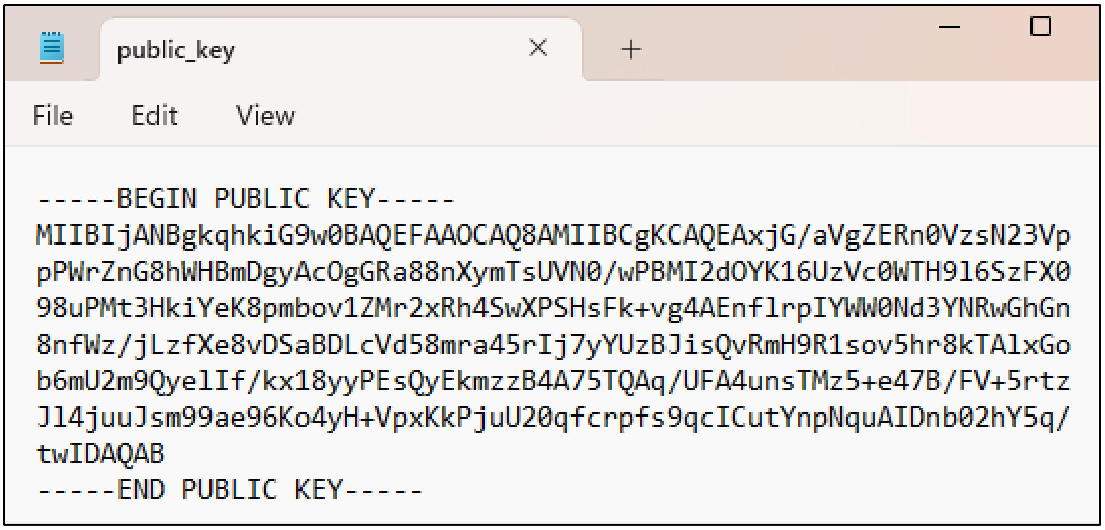
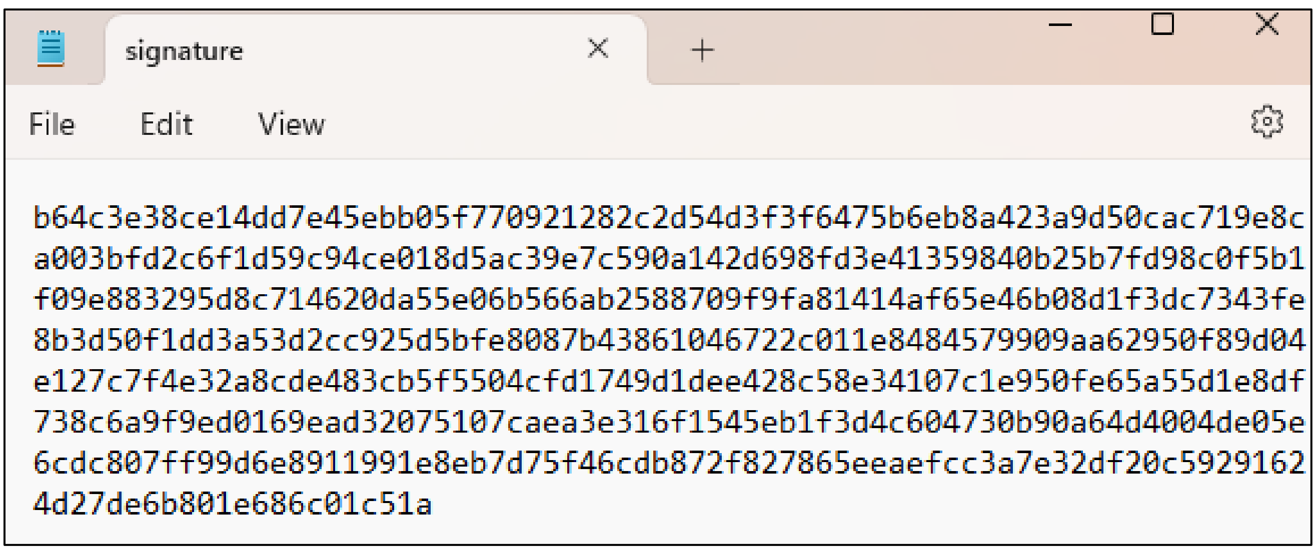


Figure 29: Private key stored in the txt



-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAxjG/aVgZERn0VzsN23Vp
pPWrZnG8hWHBmDgyAcOgGRa88nXymTsUVN0/wPBMI2dOYK16UzVc0WTH916SzFX0
98uPMt3HkiYeK8pmbov1ZMr2xRh4SwXPSHsFk+vg4AEnflrpIYWW0Nd3YNRwGhGn
8nfWz/jLzfXe8vDSaBDLcVd58mra45rIj7yYUzBJisQvRmH9R1sov5hr8kTAlxGo
b6mU2m9QyelIf/kx18yyPEsQyEkmzzB4A75TQAq/UFA4unsTMz5+e47B/FV+5rtz
Jl4juuJsm99ae96Ko4yH+VpxKkPjuU20qfcrpfs9qcICutYnpNquAIDnb02hY5q/
twIDAQAB
-----END PUBLIC KEY-----

Figure 30: Public key stored in the txt



b64c3e38ce14dd7e45ebb05f770921282c2d54d3f3f6475b6eb8a423a9d50cac719e8c
a003bfd2c6f1d59c94ce018d5ac39e7c590a142d698fd3e41359840b25b7fd98c0f5b1
f09e883295d8c714620da55e06b566ab2588709f9fa81414af65e46b08d1f3dc7343fe
8b3d50f1dd3a53d2cc925d5bfe8087b43861046722c011e8484579909aa62950f89d04
e127c7f4e32a8cde483cb5f5504cfd1749d1dee428c58e34107c1e950fe65a55d1e8df
738c6a9f9ed0169ead32075107caea3e316f1545eb1f3d4c604730b90a64d4004de05e
6cdc807ff99d6e8911991e8eb7d75f46cdb872f827865eeaefcc3a7e32df20c5929162
4d27de6b801e686c01c51a

Figure 31: Signature stored in the txt

```
Message 1: Hello, World!
```

```
==============================================================================
|                               Signing Message                              |
==============================================================================
<Importing private key for sign>
Original message hash             : b"8\xe0\\3\xd7\xb0g\x12\x7f!}\x8c\x85nUO\xcf\xf0\x9c\x93 \xb8\xa5\x97\x9c\xe2\x
ff]\x95\xdd'\xba5\xd1\xfb\xa5\x0cV-\xfd\x1dl\xc4\x8b\xc9\xc5\xba\xa49\x08\x94A\x8c\xc9B\xd9h\xf9{\xcbe\x94\x19\xed"
Signature of original message hash : b"0\xa9s\x97\xaeh\x1e\x9bw{\xd1\xc7SR\xdd(2\xe5I\x10j\xdb\xc6n\x10u\x1cy1{\xa1\
x04\xe5\x88-\xc9\x84\xa3\xb3\x81EbJ:\xf05$\x96\x046\xca\xb8\xae\xed\x941<,D\xe1\xfe\xc9\xebW\xfc\xdfT'B\xd2\x0f\x07P
\xba\x8e\xa5\xcc'3m;w\xf3^\x1f\x08\xf5\x93l\xd1\x16\x1d\xf98\x8dt:\nY\x8e\xf0=\x90\xe6\xd9~p #x6\xa0\xd0\x7f>\xc1\xb
8\x0c \x9a\xd8\x02\xf1\xad\xef\x15\xd8M+&d\x97u\x93\xde\x9d\x999v\x1b\t\x01\x7f4Zo\xefvm\x9d\xd2\xd7\x0f;\xba\xb1\x1
1\x95\x0c\xe6G\xca\xae\x15\x00\x8cLxn;\xfd'\xb7\xeb\xd1\xd9\x94%s`[\xdf\xdd\x80\xfdK\x8f\x15'Z\x8d\xf0,~\x82z\xf0t\x
eeY\xa9\xc1\x84\x0c\x1av(\t\xb5\xab\x95\xf9\xca\xc2o{\x84?\xe3\xab\x16\xe3\xd8zHe\x1b<+9\x85\x1c\xbb\xaa\x19\xfc\x9b
\x85\xa6\xc08\x8b\x82\xaaUBwh\x12\xe5\x1a_\xeb\x1cq\xd5"
Filename: C:\Users\summe\source\repos\Group Blowfish_Phase 4_Modified DSA Program\Group Blowfish_Phase 4_Modified DS
A Program\Group_Blowfish_Phase_4_Modified_DSA_Program.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    65     44.2 MiB     44.2 MiB           1   @profile
    66                                         def sign_message(private_key, message):
    67     44.2 MiB      0.0 MiB           1       print_box("Signing Message")
    68                                             # Use SHA3-512 algorithm
    69     44.2 MiB      0.0 MiB           1       message_hash = hashlib.sha3_512(message.encode('utf-8')).digest()

    70     44.2 MiB      0.0 MiB           1       start_time_hash = time.time()  # Start measuring time for hashing

    71     44.2 MiB      0.0 MiB           1       print("<Importing private key for sign>")
    72
    73     44.2 MiB      0.0 MiB           2       signature = private_key.sign(
    74     44.2 MiB      0.0 MiB           1           message.encode('utf-8'),
    75     44.2 MiB      0.0 MiB           1           padding.PKCS1v15(),
    76     44.2 MiB      0.0 MiB           1           hashes.SHA3_512()
    77                                             )
    78
    79     44.2 MiB      0.0 MiB           1       end_time_hash = time.time()  # End measuring time for hashing
    80     44.2 MiB      0.0 MiB           1       execution_time_hash = end_time_hash - start_time_hash
    81                                             # Calculate the time excluding file execution
    82     44.2 MiB      0.0 MiB           1       execution_time = execution_time_hash
    83     44.2 MiB      0.0 MiB           1       print("Original message hash              :", message_hash)
    84     44.2 MiB      0.0 MiB           1       print("Signature of original message hash :", signature)
    85
    86                                             # Convert the signature to hexadecimal format
    87     44.2 MiB      0.0 MiB           1       signature_hex = signature.hex()
    88
    89                                             # Store the signature in a text file
    90     44.2 MiB      0.0 MiB           1       with open('signature.txt', 'w') as file:
    91     44.2 MiB      0.0 MiB           1           file.write(signature_hex)
    92                                             # The function returns the signature and the execution time for t
he hashing process.
    93     44.2 MiB      0.0 MiB           1       return signature_hex, execution_time
```

Figure 32: Output of Phase 4 by Machine A for signing Message 1

```
<Results Exclude File Execution>
Time taken for signing        : 0.008131742477416992 seconds
Message signing memory usage : 44.23046875 KB
```

Figure 33: This time excluded the file execution, time writing in the txt is not counted

```
===============================================================================
|                           Validating Signature                             |
===============================================================================
<Importing public key for validating signature>
Filename: C:\Users\summe\source\repos\Group Blowfish_Phase 4_Modified DSA Program\Group Blowfish_Phase 4_Modified DS
A Program\Group_Blowfish_Phase_4_Modified_DSA_Program.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    96     44.4 MiB     44.4 MiB           1   @profile
    97                                         def verify_signature(public_key, message, signature_hex):
    98     44.4 MiB      0.0 MiB           1       print_box("Validating Signature")
    99     44.4 MiB      0.0 MiB           1       print("<Importing public key for validating signature>")
   100
   101                                             # Read the signature from the text file
   102     44.4 MiB      0.0 MiB           1       with open('signature.txt', 'r') as file:
   103     44.4 MiB      0.0 MiB           1           signature_hex = file.read().strip()
   104
   105                                             # Convert the signature from hexadecimal to bytes
   106     44.4 MiB      0.0 MiB           1       signature = bytes.fromhex(signature_hex)
   107
   108     44.4 MiB      0.0 MiB           1       message_hash = hashlib.sha3_512(message.encode('utf-8')).digest()
   109     44.4 MiB      0.0 MiB           1       start_time = time.time()
   110
   111                                             # If the signature is valid, the function returns True along
   112                                             # with the execution time for the verification process.
   113                                             # Otherwise, it returns False and None.
   114     44.4 MiB      0.0 MiB           1       try:
   115     44.4 MiB      0.0 MiB           2           public_key.verify(
   116     44.4 MiB      0.0 MiB           1               signature,
   117     44.4 MiB      0.0 MiB           1               message.encode('utf-8'),
   118     44.4 MiB      0.0 MiB           1               padding.PKCS1v15(),
   119     44.4 MiB      0.0 MiB           1               hashes.SHA3_512()
   120                                                 )
   121     44.4 MiB      0.0 MiB           1           end_time = time.time()
   122     44.4 MiB      0.0 MiB           1           execution_time = end_time - start_time
   123                                                 # Calculate time without file execution
   124     44.4 MiB      0.0 MiB           1           return True, execution_time
   125                                             except Exception:
   126                                                 return False, None

<Results Exclude File Execution>
Time taken for verification         : 0.019484996795654297 seconds
Signature validation memory usage   : 44.390625 KB

Condition        : The message is authentic.
Original Message : Hello, World!
```

Figure 34: Output of Phase 4 by Machine A for signature verification using Message 1

```
<Results Exclude File Execution>
Time taken for verification         : 0.019484996795654297 seconds
Signature validation memory usage   : 44.390625 KB

Condition        : The message is authentic.
Original Message : Hello, World!
```

Figure 35: This time excluded the file execution, time reading from the txt is not counted. The message verification shows that the message is authentic as the signature is valid.

```
Message 2: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer pretium, lacus a consequat suscipit, mag
na sem sagittis erat, vitae posuere ex ex sed justo. Curabitur commodo ex lectus, in consequat nisi pellentesque a.
Suspendisse id malesuada lectus, nec elementum neque. Nullam venenatis eros sit amet odio pharetra, id varius tellus
 auctor. Curabitur sit amet ex eget leo laoreet pulvinar. Nulla facilisi. Fusce ac mauris ut mauris semper aliquam i
n et est. Donec condimentum neque a interdum consectetur. Nulla facilisi. Donec venenatis dapibus mauris at iaculis.
```

```
================================================================================
|                               Signing Message                                |
================================================================================
<Importing private key for sign>
Original message hash              : b"\x15f\xd0\x95\x821\xec\xc7_\xf8{\xfa\xec1\x10?0-\xb1\xa3I\xbd\x02\xd4\xb6\x13
\xbb\xa9\xfckj\xef\xf0\xde\xad\xa5\xa8\xfd_\x1a\xc0S4S'g\x12X\x1f\xa8\xafy\xe2\x13\xf2\x98*\xcc+\x80\x88q\x11\xd1"
Signature of original message hash : b'F\xd4\x13\xb87\xf5D\x18\xf7\x11f\xc4\xca\xd9\n\x9c+<G\xd8\xff\x0b8\xee(PZ{\xa
7\xab\xde\x0eg\x8eQ\xe6\x8c\xce\x9f3d"xY\x1f\x17\xd9\t\xf2\x96\xb5\xda\xcej\xe2\xb2\xd5/\xb8\x0cg\xd3\x88\xd5\x89{4\
xe14\x19QB6\x1bW\xc5\xc6\x8b\t\x92[\x8a\xcc\xa9\r`%H\x80:\x16\xfej\xfa\x97\x8e\xdd0\x02\xb6\xe2\xd362o\xfb\xe4\xda\x
deNA^\xe3\x15S]G+\xcd\x02\x997\xec\xf3rH\xd0\xd6I\x9dP\xfb4b\xce9 \xe2\xfezy\xee\xd5\xf0L\xf2\xcb\xf1\xb7\ra\x00\xdb
\n\xdd\xd9\xaa\t+^BR"]\xe8<>\xc2\x85\x0f\xdb\x05`)M\x8bC&\x81\xc5\x9c\x85R\xbaG<\xcb\xa7`&o\xaf\xaf\xb9.\xea\x1ff\xa
d\xb2\x0e\x11E\x9a\xa3\x9c+\x87\xc1\xa8\xdf8l\xb1zH\xda\xe99\x0c;}6\xc2[\x16\x978F\x1a2\xba\xbe}\x02\x99\xf9\xd4\x8f
\xa8s\x01\x82^\xc7\x99Y\x9d\xda\xd3\xe2\x07\xa0\xcb\x8c\x92'
Filename: C:\Users\summe\source\repos\Group Blowfish_Phase 4_Modified DSA Program\Group Blowfish_Phase 4_Modified DS
A Program\Group_Blowfish_Phase_4_Modified_DSA_Program.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    65     44.4 MiB     44.4 MiB           1   @profile
    66                                         def sign_message(private_key, message):
    67     44.4 MiB      0.0 MiB           1       print_box("Signing Message")
    68                                             # Use SHA3-512 algorithm
    69     44.4 MiB      0.0 MiB           1       message_hash = hashlib.sha3_512(message.encode('utf-8')).digest()
    70     44.4 MiB      0.0 MiB           1       start_time_hash = time.time()  # Start measuring time for hashing
    71     44.4 MiB      0.0 MiB           1       print("<Importing private key for sign>")
    72
    73     44.4 MiB      0.0 MiB           2       signature = private_key.sign(
    74     44.4 MiB      0.0 MiB           1           message.encode('utf-8'),
    75     44.4 MiB      0.0 MiB           1           padding.PKCS1v15(),
    76     44.4 MiB      0.0 MiB           1           hashes.SHA3_512()
    77                                             )
    78
    79     44.4 MiB      0.0 MiB           1       end_time_hash = time.time()  # End measuring time for hashing
    80     44.4 MiB      0.0 MiB           1       execution_time_hash = end_time_hash - start_time_hash
    81                                             # Calculate the time excluding file execution
    82     44.4 MiB      0.0 MiB           1       execution_time = execution_time_hash
    83     44.4 MiB      0.0 MiB           1       print("Original message hash              :", message_hash)
    84     44.4 MiB      0.0 MiB           1       print("Signature of original message hash :", signature)
    85
    86                                             # Convert the signature to hexadecimal format
    87     44.4 MiB      0.0 MiB           1       signature_hex = signature.hex()
    88
    89                                             # Store the signature in a text file
    90     44.4 MiB      0.0 MiB           1       with open('signature.txt', 'w') as file:
    91     44.4 MiB      0.0 MiB           1           file.write(signature_hex)
    92                                             # The function returns the signature and the execution time for t
he hashing process.
    93     44.4 MiB      0.0 MiB           1       return signature_hex, execution_time
```

Figure 36: Output of Phase 4 by Machine A for signing Message 2

```
<Results Exclude File Execution>
Time taken for signing        : 0.007998466491699219 seconds
Message signing memory usage : 44.41796875 KB
```

Figure 37: This time excluded the file execution, time writing in the txt is not counted

```
===============================================================================
|                            Validating Signature                             |
===============================================================================
<Importing public key for validating signature>
Filename: C:\Users\summe\source\repos\Group Blowfish_Phase 4_Modified DSA Program\Group Blowfish_Phase 4_Modified DS
A Program\Group_Blowfish_Phase_4_Modified_DSA_Program.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    96     44.4 MiB     44.4 MiB           1   @profile
    97                                         def verify_signature(public_key, message, signature_hex):
    98     44.4 MiB      0.0 MiB           1       print_box("Validating Signature")
    99     44.4 MiB      0.0 MiB           1       print("<Importing public key for validating signature>")
   100
   101                                             # Read the signature from the text file
   102     44.4 MiB      0.0 MiB           1       with open('signature.txt', 'r') as file:
   103     44.4 MiB      0.0 MiB           1           signature_hex = file.read().strip()
   104
   105                                             # Convert the signature from hexadecimal to bytes
   106     44.4 MiB      0.0 MiB           1       signature = bytes.fromhex(signature_hex)
   107
   108     44.4 MiB      0.0 MiB           1       message_hash = hashlib.sha3_512(message.encode('utf-8')).digest()
   109     44.4 MiB      0.0 MiB           1       start_time = time.time()
   110
   111                                             # If the signature is valid, the function returns True along
   112                                             # with the execution time for the verification process.
   113                                             # Otherwise, it returns False and None.
   114     44.4 MiB      0.0 MiB           1       try:
   115     44.4 MiB      0.0 MiB           2           public_key.verify(
   116     44.4 MiB      0.0 MiB           1               signature,
   117     44.4 MiB      0.0 MiB           1               message.encode('utf-8'),
   118     44.4 MiB      0.0 MiB           1               padding.PKCS1v15(),
   119     44.4 MiB      0.0 MiB           1               hashes.SHA3_512()
   120                                                 )
   121     44.4 MiB      0.0 MiB           1           end_time = time.time()
   122     44.4 MiB      0.0 MiB           1           execution_time = end_time - start_time
   123                                                 # Calculate time without file execution
   124     44.4 MiB      0.0 MiB           1           return True, execution_time
   125                                             except Exception:
   126                                                 return False, None
```

Figure 38: Output of Phase 4 by Machine A for signature verification using Message 2

```
<Results Exclude File Execution>
Time taken for verification         : 0.015199661254882812 seconds
Signature validation memory usage   : 44.44140625 KB

Condition        : The message is authentic.
Original Message : Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer pretium, lacus a consequat susci
pit, magna sem sagittis erat, vitae posuere ex ex sed justo. Curabitur commodo ex lectus, in consequat nisi pellente
sque a. Suspendisse id malesuada lectus, nec elementum neque. Nullam venenatis eros sit amet odio pharetra, id variu
s tellus auctor. Curabitur sit amet ex eget leo laoreet pulvinar. Nulla facilisi. Fusce ac mauris ut mauris semper a
liquam in et est. Donec condimentum neque a interdum consectetur. Nulla facilisi. Donec venenatis dapibus mauris at
iaculis.
```

Figure 39: This time excluded the file execution, time reading from the txt is not counted. The message verification shows that the message is authentic as the signature is valid.

Message 3: Curiosity is a remarkable human trait that drives us to explore, question, and seek knowledge about the world around us. It is the spark that ignites our imagination, propelling us on a lifelong journey of discovery. From our earliest years, curiosity is inherent in our nature, driving us to touch, taste, and explore our environment. As we grow, this innate curiosity blossoms, shaping our identities and influencing the paths we choose to follow. One of the most significant aspects of curiosity is its ability to fuel learning. When we approach new ideas, subjects, or experiences with a curious mindset, we are open to the possibilities they hold. Curiosity compels us to ask questions, seek answers, and engage in critical thinking. It challenges us to delve deeper, to unravel the complexities of the world and expand our knowledge. By nurturing curiosity, we embrace a lifelong pursuit of learning, continuously evolving and growing intellectually. Curiosity also plays a pivotal role in fostering creativity. When we allow our curiosity to roam freely, we invite inspiration and innovation into our lives. Curiosity compels us to explore uncharted territories, pushing the boundaries of what is known and comfortable. It encourages us to challenge assumptions, think outside the box, and connect seemingly unrelated ideas. By embracing our natural curiosity, we unlock the potential to create, invent, and bring forth new ideas that can shape the world around us. Beyond its influence on personal growth, curiosity has far-reaching implications for society as a whole. Curiosity serves as a catalyst for progress and advancement. Throughout history, great discoveries and inventions have been born out of a relentless pursuit of answers driven by curiosity. From the exploration of distant lands to scientific breakthroughs, every leap forward is the result of individuals and communities driven by an insatiable curiosity about the unknown. Moreover, curiosity fosters empathy and understanding. When we approach others with genuine curiosity, we open the doors to deeper connections and meaningful relationships. By seeking to understand different perspectives, cultures, and experiences, we break down barriers and bridge divides. Curiosity allows us to appreciate the richness and diversity of the human tapestry, promoting tolerance and acceptance in an increasingly interconnected world. However, despite its profound potential, curiosity can be stifled if not nurtured and encouraged. In a society that often values conformity and certainty, curiosity can be viewed as a disruption or a distraction. The pressure to conform and the fear of the unknown can dampen the flames of curiosity, leading to missed opportunities and limited growth. As a society, we must recognize the transformative power of curiosity and create environments that foster its growth in individuals of all ages. To cultivate curiosity, education plays a vital role. Schools should embrace inquiry-based learning, encouraging students to ask questions, explore their interests, and pursue independent research. Teachers can serve as guides, nurturing curiosity by creating a safe space for exploration and discovery. By incorporating real-world applications and promoting interdisciplinary approaches, education can unleash the full potential of curiosity, preparing students to become lifelong learners and active contributors to society. In addition to education, parents and caregivers also have a crucial role to play in cultivating curiosity. By encouraging exploration, providing opportunities for hands-on learning, and celebrating curiosity-driven achievements, parents can foster a lifelong love of learning in their children. They can instill a sense of wonder, inspire critical thinking, and nurture the natural inquisitiveness that lies within each child. In conclusion, curiosity is a powerful force that drives human progress, fuels creativity, and fosters understanding. It is the key to unlocking new knowledge, inspiring innovation, and forging connections. By nurturing and embracing curiosity, both as individuals and as a society, we unlock a world of possibilities, paving the way for a future built on continuous growth, discovery, and enlightenment. Let us celebrate curiosity and embark on a journey of endless exploration and learning, for it is through curiosity that we truly come alive.

```
================================================================================
|                               Signing Message                                |
================================================================================
<Importing private key for sign>
Original message hash              : b'/\x14\xc2)&#\xffU\xdd\x9f\xcf\xfe\x8bGV\xbc\xa79D\xd4r\xdf\xc7\xe0\x19\xf3\xf
6^\xc6Z\x7f>h\xbb\xb7\x99\xee\xe4\xae\xb9=\xf8s\xb3\x9a)\xaa\xe57\xd7\\\xfe\xc5\xf5\xeb\x1bw\xea`v\xbc\xf0\xe6='
Signature of original message hash : b'\xb6L>8\xce\x14\xdd~E\xeb\xb0_w\t!(,-T\xd3\xf3\xf6G[n\xb8\xa4#\xa9\xd5\x0c\xa
cq\x9e\x8c\xa0\x03\xbf\xd2\xc6\xf1\xd5\x9c\x94\xce\x01\x8dZ\xc3\x9e|Y\n\x14-i\x8f\xd3\xe4\x13Y\x84\x0b%\xb7\xfd\x98\
xc0\xf5\xb1\xf0\x9e\x882\x95\xd8\xc7\x14b\r\xa5^\x06\xb5f\xab%\x88p\x9f\x9f\xa8\x14\x14\xafe\xe4k\x08\xd1\xf3\xdcsC\
xfe\x8b=P\xf1\xdd:S\xd2\xcc\x92][\xfe\x80\x87\xb48a\x04g"\xc0\x11\xe8HEy\x90\x9a\xa6)P\xf8\x9d\x04\xe1\'\xc7\xf4\xe3
*\x8c\xdeH<\xb5\xf5PL\xfd\x17I\xd1\xde\xe4(\xc5\x8e4\x10|\x1e\x95\x0f\xe6ZU\xd1\xe8\xdfs\x8cj\x9f\x9e\xd0\x16\x9e\xa
d2\x07Q\x07\xca\xea>1o\x15E\xeb\x1f=L`G0\xb9\nd\xd4\x00M\xe0^l\xdc\x80\x7f\xf9\x9dn\x89\x11\x99\x1e\x8e\xb7\xd7_F\xc
d\xb8r\xf8\'\x86^\xea\xef\xcc:~2\xdf \xc5\x92\x91bM\'\xdek\x80\x1ehl\x01\xc5\x1a'
Filename: C:\Users\summe\source\repos\Group Blowfish_Phase 4_Modified DSA Program\Group Blowfish_Phase 4_Modified DS
A Program\Group_Blowfish_Phase_4_Modified_DSA_Program.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    65     44.5 MiB     44.5 MiB           1   @profile
    66                                         def sign_message(private_key, message):
    67     44.5 MiB      0.0 MiB           1       print_box("Signing Message")
    68                                             # Use SHA3-512 algorithm
    69     44.5 MiB      0.0 MiB           1       message_hash = hashlib.sha3_512(message.encode('utf-8')).digest()
    70     44.5 MiB      0.0 MiB           1       start_time_hash = time.time()  # Start measuring time for hashing
    71     44.5 MiB      0.0 MiB           1       print("<Importing private key for sign>")
    72
    73     44.5 MiB      0.0 MiB           2       signature = private_key.sign(
    74     44.5 MiB      0.0 MiB           1           message.encode('utf-8'),
    75     44.5 MiB      0.0 MiB           1           padding.PKCS1v15(),
    76     44.5 MiB      0.0 MiB           1           hashes.SHA3_512()
    77                                             )
    78
    79     44.5 MiB      0.0 MiB           1       end_time_hash = time.time()  # End measuring time for hashing
    80     44.5 MiB      0.0 MiB           1       execution_time_hash = end_time_hash - start_time_hash
    81                                             # Calculate the time excluding file execution
    82     44.5 MiB      0.0 MiB           1       execution_time = execution_time_hash
    83     44.5 MiB      0.0 MiB           1       print("Original message hash              :", message_hash)
    84     44.5 MiB      0.0 MiB           1       print("Signature of original message hash :", signature)
    85
    86                                             # Convert the signature to hexadecimal format
    87     44.5 MiB      0.0 MiB           1       signature_hex = signature.hex()
    88
    89                                             # Store the signature in a text file
    90     44.5 MiB      0.0 MiB           1       with open('signature.txt', 'w') as file:
    91     44.5 MiB      0.0 MiB           1           file.write(signature_hex)
    92                                             # The function returns the signature and the execution time for t
he hashing process.
    93     44.5 MiB      0.0 MiB           1       return signature_hex, execution_time
```

Figure 40: Output of Phase 4 by Machine A for signing Message 3

```
<Results Exclude File Execution>
Time taken for signing        : 0.010300159454345703 seconds
Message signing memory usage : 44.50390625 KB
```

Figure 41: This time excluded the file execution, time writing in the txt is not counted

```
================================================================================
|                              Validating Signature                            |
================================================================================
<Importing public key for validating signature>
Filename: C:\Users\summe\source\repos\Group Blowfish_Phase 4_Modified DSA Program\Group Blowfish_Phase 4_Modified DS
A Program\Group_Blowfish_Phase_4_Modified_DSA_Program.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    96     44.6 MiB     44.6 MiB           1   @profile
    97                                         def verify_signature(public_key, message, signature_hex):
    98     44.6 MiB      0.0 MiB           1       print_box("Validating Signature")
    99     44.6 MiB      0.0 MiB           1       print("<Importing public key for validating signature>")
   100
   101                                             # Read the signature from the text file
   102     44.6 MiB      0.0 MiB           1       with open('signature.txt', 'r') as file:
   103     44.6 MiB      0.0 MiB           1           signature_hex = file.read().strip()
   104
   105                                             # Convert the signature from hexadecimal to bytes
   106     44.6 MiB      0.0 MiB           1       signature = bytes.fromhex(signature_hex)
   107
   108     44.6 MiB      0.0 MiB           1       message_hash = hashlib.sha3_512(message.encode('utf-8')).digest()
   109     44.6 MiB      0.0 MiB           1       start_time = time.time()
   110
   111                                             # If the signature is valid, the function returns True along
   112                                             # with the execution time for the verification process.
   113                                             # Otherwise, it returns False and None.
   114     44.6 MiB      0.0 MiB           1       try:
   115     44.6 MiB      0.0 MiB           2           public_key.verify(
   116     44.6 MiB      0.0 MiB           1               signature,
   117     44.6 MiB      0.0 MiB           1               message.encode('utf-8'),
   118     44.6 MiB      0.0 MiB           1               padding.PKCS1v15(),
   119     44.6 MiB      0.0 MiB           1               hashes.SHA3_512()
   120                                                 )
   121     44.6 MiB      0.0 MiB           1           end_time = time.time()
   122     44.6 MiB      0.0 MiB           1           execution_time = end_time - start_time
   123                                                 # Calculate time without file execution
   124     44.6 MiB      0.0 MiB           1           return True, execution_time
   125                                             except Exception:
   126                                                 return False, None


<Results Exclude File Execution>
Time taken for verification         : 0.022504091262817383 seconds
Signature validation memory usage   : 44.56640625 KB
```

Figure 42: Output of Phase 4 by Machine A for signature verification using Message 3

```
<Results Exclude File Execution>
Time taken for verification          : 0.022504091262817383 seconds
Signature validation memory usage    : 44.56640625 KB
```

```
Condition        : The message is authentic.
Original Message : Curiosity is a remarkable human trait that drives us to explore, question, and seek knowledge abo
ut the world around us. It is the spark that ignites our imagination, propelling us on a lifelong journey of discove
ry. From our earliest years, curiosity is inherent in our nature, driving us to touch, taste, and explore our enviro
nment. As we grow, this innate curiosity blossoms, shaping our identities and influencing the paths we choose to fol
low. One of the most significant aspects of curiosity is its ability to fuel learning. When we approach new ideas, s
ubjects, or experiences with a curious mindset, we are open to the possibilities they hold. Curiosity compels us to
ask questions, seek answers, and engage in critical thinking. It challenges us to delve deeper, to unravel the compl
exities of the world and expand our knowledge. By nurturing curiosity, we embrace a lifelong pursuit of learning, co
ntinuously evolving and growing intellectually. Curiosity also plays a pivotal role in fostering creativity. When we
 allow our curiosity to roam freely, we invite inspiration and innovation into our lives. Curiosity compels us to ex
plore uncharted territories, pushing the boundaries of what is known and comfortable. It encourages us to challenge
assumptions, think outside the box, and connect seemingly unrelated ideas. By embracing our natural curiosity, we un
lock the potential to create, invent, and bring forth new ideas that can shape the world around us. Beyond its influ
ence on personal growth, curiosity has far-reaching implications for society as a whole. Curiosity serves as a catal
yst for progress and advancement. Throughout history, great discoveries and inventions have been born out of a relen
tless pursuit of answers driven by curiosity. From the exploration of distant lands to scientific breakthroughs, eve
ry leap forward is the result of individuals and communities driven by an insatiable curiosity about the unknown. Mo
reover, curiosity fosters empathy and understanding. When we approach others with genuine curiosity, we open the doo
rs to deeper connections and meaningful relationships. By seeking to understand different perspectives, cultures, an
d experiences, we break down barriers and bridge divides. Curiosity allows us to appreciate the richness and diversi
ty of the human tapestry, promoting tolerance and acceptance in an increasingly interconnected world. However, despi
te its profound potential, curiosity can be stifled if not nurtured and encouraged. In a society that often values c
onformity and certainty, curiosity can be viewed as a disruption or a distraction. The pressure to conform and the f
ear of the unknown can dampen the flames of curiosity, leading to missed opportunities and limited growth. As a soci
ety, we must recognize the transformative power of curiosity and create environments that foster its growth in indiv
iduals of all ages. To cultivate curiosity, education plays a vital role. Schools should embrace inquiry-based learn
ing, encouraging students to ask questions, explore their interests, and pursue independent research. Teachers can s
erve as guides, nurturing curiosity by creating a safe space for exploration and discovery. By incorporating real-wo
rld applications and promoting interdisciplinary approaches, education can unleash the full potential of curiosity,
preparing students to become lifelong learners and active contributors to society. In addition to education, parents
 and caregivers also have a crucial role to play in cultivating curiosity. By encouraging exploration, providing opp
ortunities for hands-on learning, and celebrating curiosity-driven achievements, parents can foster a lifelong love
of learning in their children. They can instill a sense of wonder, inspire critical thinking, and nurture the natura
l inquisitiveness that lies within each child. In conclusion, curiosity is a powerful force that drives human progre
ss, fuels creativity, and fosters understanding. It is the key to unlocking new knowledge, inspiring innovation, and
 forging connections. By nurturing and embracing curiosity, both as individuals and as a society, we unlock a world
of possibilities, paving the way for a future built on continuous growth, discovery, and enlightenment. Let us celeb
rate curiosity and embark on a journey of endless exploration and learning, for it is through curiosity that we trul
y come alive.
```

Figure 43: This time excluded the file execution, time reading from the txt is not counted. The message verification shows that the message is authentic as the signature is valid.

Machine A

Message 1: Hello, World!

Performance Report

Algorithm	Time Complexity
Public and Private Key Pair Generation	0.576640 seconds
Encryption : Message Signing	1.011022 seconds
Decryption : Signature Validation	1.032025 seconds
Signature Execution	0.002031 seconds
Overall Process	2.619687 seconds

Algorithm	Memory Usage
Public and Private Key Pair Generation	44.144531 KB
Encryption : Message Signing	44.230469 KB
Decryption : Signature Validation	44.390625 KB
Signature Execution	44.230469 KB
Overall Process	132.765625 KB

Message 2: Lorem ipsum dolor sit amet,…(85 words)

Performance Report

Algorithm	Time Complexity
Public and Private Key Pair Generation	0.576640 seconds
Encryption : Message Signing	0.969634 seconds
Decryption : Signature Validation	0.988760 seconds
Signature Execution	0.000999 seconds
Overall Process	2.535034 seconds

Algorithm	Memory Usage
Public and Private Key Pair Generation	44.144531 KB
Encryption : Message Signing	44.417969 KB
Decryption : Signature Validation	44.441406 KB
Signature Execution	44.417969 KB
Overall Process	133.003906 KB

Message 3: Curiosity is a remarkable human trait… (654 words)

Performance Report

Algorithm	Time Complexity
Public and Private Key Pair Generation	0.576640 seconds
Encryption : Message Signing	1.011166 seconds
Decryption : Signature Validation	0.967355 seconds
Signature Execution	0.001205 seconds
Overall Process	2.555160 seconds

Algorithm	Memory Usage
Public and Private Key Pair Generation	44.144531 KB
Encryption : Message Signing	44.503906 KB
Decryption : Signature Validation	44.566406 KB
Signature Execution	44.503906 KB
Overall Process	133.214844 KB

Figure 44: Performance report 1, 2, 3 display the time complexities that counted whole process including file execution and memory usage of 3 different length messages.

The performance reports provide valuable insights into the time complexity and memory usage of a digital signature algorithm (DSA) system for three different message lengths. By examining the results, we can analyze how these factors vary based on the length of the message being processed.

The time complexity refers to the computational resources required by an algorithm and is often measured in seconds. In the given performance reports, the time taken for key pair generation, message signing encryption, signature validation decryption, and the overall process remained relatively consistent across all three messages, except for the encryption and decryption stages.

"Hello, World!" It remains at around 1 second for each algorithm. The message of "Lorem ipsum dolor sit amet,…" encryption time for this longer message is slightly shorter (0.97 seconds) compared to "Hello, World!" (1.01 seconds). This decrease is expected since the algorithm used is more suitable to encrypt a longer message, leading to slightly lower time complexity. The decryption time for this longer message " Curiosity is a remarkable human trait..." is the lowest (0.98 seconds) compared to "Hello, World!" (1.03 seconds) and “Lorem ipsum dolor sit amet,…” (0.99 seconds).

Despite the longer length of Message 2 ("Lorem ipsum dolor sit amet,...") compared to Message 1, the encryption time is actually shorter. This can be attributed to the factor that is based on the input data characteristics. Although Message 2 is longer in terms of characters, it is possible that it contains repeating patterns or other structures that allow for more efficient processing. On the other hand, Message 1 may have unique or complex patterns that require additional computational operations, leading to a slightly longer encryption time.

Another possible reason for the longer encryption time for Message 1 compared to Message 2 could be the need for padding. For Message 3, which is a relatively long message, it is possible that the message length is not an exact multiple of the block size. As a result, padding needs to be added to fill the remaining space in the last block. Generating longer padding for Message 3 could require additional computational operations, leading to a slightly longer encryption time compared to Message 2. Message 3, being a longer message than Message 2, might have a length that is closer to an exact multiple of the block size. Consequently, the need for padding might be reduced or eliminated, resulting in a shorter encryption time.

Across the different messages, the memory usage for most of the algorithms remained consistent, indicating that the length of the message had a minimal impact on memory requirements. However, the encryption process showed a slight increase in memory usage for the longer messages compared to the shorter "Hello, World!" message. This increase can be attributed to the larger input size of the messages being encrypted. As the length of the message increases, more memory is required to store the additional data during the encryption process.

For instance, "Hello, World!" remains at around 44 KB for each algorithm. Message 2 encryption process consumes slightly more memory (44.42 KB) compared to "Hello, World!" (44.23 KB). While the encryption process of Message 3 consumes the largest memory (44.50 KB) among these 3 messages.

Machine B

Message 1: Hello, World! (2 words)

Performance Report

Algorithm	Time Complexity
Public and Private Key Pair Generation	0.662479 seconds
Encryption : Message Signing	1.125717 seconds
Decryption : Signature Validation	1.034765 seconds
Signature Execution	0.008004 seconds
Overall Process	2.822961 seconds

Algorithm	Memory Usage
Public and Private Key Pair Generation	35.449219 KB
Encryption : Message Signing	35.828125 KB
Decryption : Signature Validation	35.851562 KB
Signature Execution	35.828125 KB
Overall Process	107.128906 KB

Message 2: Lorem ipsum dolor sit amet,…(85 words)

Performance Report

Algorithm	Time Complexity
Public and Private Key Pair Generation	0.662479 seconds
Encryption : Message Signing	1.021825 seconds
Decryption : Signature Validation	0.992967 seconds
Signature Execution	0.000000 seconds
Overall Process	2.677272 seconds

Algorithm	Memory Usage
Public and Private Key Pair Generation	35.449219 KB
Encryption : Message Signing	35.871094 KB
Decryption : Signature Validation	35.878906 KB
Signature Execution	35.871094 KB
Overall Process	107.199219 KB

Message 3: Curiosity is a remarkable human trait… (654 words)

```
--------------------------------------------------------------------------
|                            Performance Report                          |
|------------------------------------------------------------------------|
|                  Algorithm                  |      Time Complexity      |
|------------------------------------------------------------------------|
| Public and Private Key Pair Generation     |         0.662479 seconds |
| Encryption : Message Signing               |         1.031748 seconds |
| Decryption : Signature Validation          |         1.037492 seconds |
| Signature Execution                        |         0.001082 seconds |
| Overall Process                            |         2.731719 seconds |
|------------------------------------------------------------------------|


|------------------------------------------------------------------------|
|                  Algorithm                  |        Memory Usage       |
|------------------------------------------------------------------------|
| Public and Private Key Pair Generation     |             35.449219 KB |
| Encryption : Message Signing               |             35.894531 KB |
| Decryption : Signature Validation          |             35.914062 KB |
| Signature Execution                        |             35.894531 KB |
| Overall Process                            |            107.257812 KB |
|------------------------------------------------------------------------|
```

Figure 45: Performance report 1, 2, 3 display the time complexities that counted whole process including file execution and memory usage of 3 different length messages by Machine B.

The differences in time complexity and memory usage between Machine A and Machine B could be attributed to several factors, including the specifications of the machines and potential variations in their performance.

Machine A and Machine B have different hardware specifications, including the CPU, GPU, and memory (RAM) capacity. Machine A has an AMD Ryzen 7 4800H CPU and NVIDIA GeForce GTX 1650 GPU, while Machine B has an AMD Ryzen 5 4500U CPU and AMD Radeon Graphics. The higher performance and capabilities of the CPU and GPU in Machine A may contribute to faster execution and potentially more efficient memory usage.

Machine A and Machine B both have 8 GB of DDR4 RAM, but the specific memory speed and configurations might differ. The available RAM capacity and memory speed can affect the execution speed and memory utilization of the algorithms. Machine A has faster or more optimized RAM, it could result in better performance in terms of both time complexity and memory usage. In summary, our group will use the output of Machine A in the following section (Section 3).

2.5 Data Used Specification

2.5.1 Phase 3 - Original DSA Program

The program does not rely on any external data for its execution. However, it generates a DSA key pair consisting of a private key and a corresponding public key. The private and public keys are serialized to PEM format and stored in text files (private_key.txt and public_key.txt). The key generation process utilizes the DSA algorithm with a key size of 2048 bits.

In addition to the key pair, the program operates on a 3 different message, which is same in Phase 4 (mentioned in 2.5.2). The message is used for signing and verifying the signature using the generated private-public key pair."

2.5.2 Phase 4 - Modified DSA Program

In the given program, the public and private keys are generated using the RSA algorithm. The “generate_keys()” function uses the “rsa.generate_private_key()” method from the “cryptography.hazmat.primitives.asymmetric.rsa” module to generate a private key. The private key is then used to derive the corresponding public key.

The key generation process in this program utilizes a specific set of parameters. The value 65537 is used as the public exponent, which is a commonly used value for RSA. The key size is set to 2048 bits. These parameters are considered to be secure and widely accepted in practice. The “rsa.generate_private_key()” method uses a secure random number generator internally to generate the keys, ensuring the randomness and security of the generated keys.

Other than that, the data used in this program consists of messages of different sizes. The messages are same with the original DSA Program, they are mentioned as follows:

1. (2 words) "Hello, World!"
2. (85 words) "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer pretium, lacus a consequat suscipit, magna sem sagittis erat, vitae posuere ex ex sed justo. Curabitur commodo ex lectus, in consequat nisi pellentesque a. Suspendisse id malesuada lectus, nec elementum neque. Nullam venenatis eros sit amet odio pharetra, id varius tellus auctor. Curabitur sit amet ex eget leo laoreet pulvinar. Nulla facilisi. Fusce ac mauris ut mauris semper aliquam in et est. Donec

condimentum neque a interdum consectetur. Nulla facilisi. Donec venenatis dapibus mauris at iaculis."

3. (654 words) "Curiosity is a remarkable human trait that drives us to explore, question, and seek knowledge about the world around us. It is the spark that ignites our imagination, propelling us on a lifelong journey of discovery. From our earliest years, curiosity is inherent in our nature, driving us to touch, taste, and explore our environment. As we grow, this innate curiosity blossoms, shaping our identities and influencing the paths we choose to follow. One of the most significant aspects of curiosity is its ability to fuel learning. When we approach new ideas, subjects, or experiences with a curious mindset, we are open to the possibilities they hold. Curiosity compels us to ask questions, seek answers, and engage in critical thinking. It challenges us to delve deeper, to unravel the complexities of the world and expand our knowledge. By nurturing curiosity, we embrace a lifelong pursuit of learning, continuously evolving and growing intellectually. Curiosity also plays a pivotal role in fostering creativity. When we allow our curiosity to roam freely, we invite inspiration and innovation into our lives. Curiosity compels us to explore uncharted territories, pushing the boundaries of what is known and comfortable. It encourages us to challenge assumptions, think outside the box, and connect seemingly unrelated ideas. By embracing our natural curiosity, we unlock the potential to create, invent, and bring forth new ideas that can shape the world around us. Beyond its influence on personal growth, curiosity has far-reaching implications for society as a whole. Curiosity serves as a catalyst for progress and advancement. Throughout history, great discoveries and inventions have been born out of a relentless pursuit of answers driven by curiosity. From the exploration of distant lands to scientific breakthroughs, every leap forward is the result of individuals and communities driven by an insatiable curiosity about the unknown. Moreover, curiosity fosters empathy and understanding. When we approach others with genuine curiosity, we open the doors to deeper connections and meaningful relationships. By seeking to understand different perspectives, cultures, and experiences, we break down barriers and bridge divides. Curiosity allows us to appreciate the richness and diversity of the human tapestry, promoting tolerance and acceptance in an increasingly interconnected world. However, despite its profound potential, curiosity can be stifled if not nurtured and encouraged. In a

> society that often values conformity and certainty, curiosity can be viewed as a disruption or a distraction. The pressure to conform and the fear of the unknown can dampen the flames of curiosity, leading to missed opportunities and limited growth. As a society, we must recognize the transformative power of curiosity and create environments that foster its growth in individuals of all ages. To cultivate curiosity, education plays a vital role. Schools should embrace inquiry-based learning, encouraging students to ask questions, explore their interests, and pursue independent research. Teachers can serve as guides, nurturing curiosity by creating a safe space for exploration and discovery. By incorporating real-world applications and promoting interdisciplinary approaches, education can unleash the full potential of curiosity, preparing students to become lifelong learners and active contributors to society. In addition to education, parents and caregivers also have a crucial role to play in cultivating curiosity. By encouraging exploration, providing opportunities for hands-on learning, and celebrating curiosity-driven achievements, parents can foster a lifelong love of learning in their children. They can instill a sense of wonder, inspire critical thinking, and nurture the natural inquisitiveness that lies within each child. In conclusion, curiosity is a powerful force that drives human progress, fuels creativity, and fosters understanding. It is the key to unlocking new knowledge, inspiring innovation, and forging connections. By nurturing and embracing curiosity, both as individuals and as a society, we unlock a world of possibilities, paving the way for a future built on continuous growth, discovery, and enlightenment. Let us celebrate curiosity and embark on a journey of endless exploration and learning, for it is through curiosity that we truly come alive."

These messages are used for signing and verifying signatures using a private-public key pair. The program generates the key pair and performs the signing and verification process for each message in the list.

SECTION 3: RESULTS

This section provides a detailed comparison between the two systems based on their structure, time, memory usage, and performance. The DSA systems under comparison are implemented in Python and involve cryptographic operations such as key generation, signing, and verification.

3.1 Comparison of Structure

Table 2: Compare on Original and Modified DSA systems with the algorithms used

Aspect	Original DSA Program	Modified DSA Program
Cryptographic Algorithm	DSA	RSA
Hash Function	SHA256	SHA3-512
Key Pair Generation	DSA key pair with 2048-bit size	RSA key pair with 2048-bit size, public exponent 65537
Serialization Format	Private key: PEM Public key: SubjectPublicKeyInfo	
Signature Padding	Prehashed SHA256	PKCS1v15
Signature Storage	Hexadecimal string in a text file	
Memory Profiling	Uses the memory_profiler library	

The original system utilizes the DSA (Digital Signature Algorithm) for key generation, signing, and verification, while the modified system employs the RSA (Rivest-Shamir-Adleman) algorithm for the same purposes. DSA provides efficient signing and verification capabilities, while RSA is known for its strong encryption and key exchange abilities.

For hashing the message, the original system employs the SHA256 hash function, whereas the modified system uses the SHA3-512 hash function. The choice of hash function can impact the security and efficiency of cryptographic operations. SHA256 is a widely used cryptographic hash function known for its collision resistance and efficiency, while SHA3-512 is a member of the SHA3 family, designed to provide secure and efficient hashing with a larger output size.

In terms of key pair generation, the original system generates a DSA key pair with a key size of 2048 bits, whereas the modified system generates an RSA key pair with the same key size and a public exponent of 65537. The key size is an important parameter that determines the security level of the cryptographic keys. Both DSA and RSA are asymmetric encryption algorithms that rely on the difficulty of certain mathematical problems for their security.

Both systems serialize the private key in PEM format and the public key in SubjectPublicKeyInfo format. Serialization allows the keys to be easily stored and transmitted in a standardized format. The choice of serialization format can depend on the requirements of the cryptographic system or the infrastructure in which it will be used.

For signing and verification, the original system utilizes the utils.Prehashed (hashes.SHA256()) padding for DSA, while the modified system uses the padding.PKCS1v15() padding for RSA. The padding scheme ensures that the signature operation produces a unique and verifiable signature. The choice of padding scheme depends on the cryptographic algorithm and the specific requirements of the system.

Both systems store the signature as a hexadecimal string in a text file. Storing the signature allows for later verification of the message's authenticity. The hexadecimal representation is commonly used to store binary data in a human-readable format.

Both systems utilize the memory_profiler library for memory profiling, which allows for monitoring and analyzing the memory usage of the functions. Memory profiling is useful for optimizing memory consumption and identifying potential memory leaks or inefficiencies in the code.

In conclusion, the two systems demonstrate different cryptographic algorithms, hash functions, key pair generation techniques, and signature padding schemes. However, they are the same in utilizing serialization formats and signature storage methods.

3.2 Comparison on Time Complexity

In this section, we will analyze the output of the Modified DSA Program, with a specific focus on the results generated by Machine A in Section 2. Furthermore, we will concentrate on the output related 3 messages since both systems utilized it, as the messages serve as a standardized reference point for comparison.

Message 1: Hello, World! (2 words)

Table 3: Compare on Original and Modified DSA systems with the time complexities generated by the Message "Hello, World!"

Algorithm	Original DSA Program *(seconds)*	Modified DSA Program *(seconds)*
Public and Private Key Pair Generation	0.924609	0.576640
Encryption : Message Signing	0.983675	1.011022
Decryption : Signature Validation	0.986379	1.032025
Signature Execution	0.001016	0.002031
Overall Process	2.894662	2.619687

In the original DSA program, the key pair generation, message signing, and signature validation operations take 0.924609, 0.983675, and 0.986379 seconds, respectively. This discrepancy can be attributed to the DSA algorithm's inherent complexity in performing modular operations and the generation of larger key sizes, such as the 2048-bit size in this case. The signature execution operation completes remarkably fast, taking only 0.001016 seconds. Overall, the original DSA program processes the "Hello, World!" message in approximately 2.894662 seconds.

Comparatively, the modified DSA program exhibits slightly lower and better time complexities. The key pair generation, message signing, and signature validation operations take 0.576640, 1.011022, and 1.032025 seconds, respectively. The signature execution operation in the modified DSA program is also marginally slower, requiring 0.002031 seconds. Consequently, the overall processing time of the modified DSA program for the "Hello, World!" message is approximately 2.619687 seconds lesser than the original DSA Program.

Message 2: Lorem ipsum dolor sit amet, consectetur adipiscing elit… (85 words)

Table 4: Compare on Original and Modified DSA systems with the time complexities generated by the Message "Lorem ipsum dolor sit amet, consectetur adipiscing elit…"

Algorithm	Original DSA Program ***(seconds)***	Modified DSA Program ***(seconds)***
Public and Private Key Pair Generation	0.924609	0.576640
Encryption : Message Signing	0.974469	0.969634
Decryption : Signature Validation	1.030369	0.988760
Signature Execution	0.000504	0.000999
Overall Process	2.929447	2.535034

When comparing the key pair generation process, the original DSA program demonstrates slightly higher time complexity (0.924609 seconds) compared to the modified DSA program (0.576640 seconds). The modified DSA program exhibits lower time complexities for both message signing and signature validation operations.

In message signing, the modified DSA program performs significantly better (0.969634 seconds) compared to the original DSA program (0.974469 seconds). Similarly, in signature validation, the modified DSA program shows better performance (0.988760 seconds) compared to the original DSA program (1.030369 seconds). These improvements are a result of the RSA algorithm's efficient signature generation and verification processes.

Overall, when considering the time complexity of the entire process, the modified DSA program surpasses the original DSA program. The modified DSA program completes the overall process in 2.535034 seconds, while the original DSA program takes 2.929447 seconds.

Message 3: Curiosity is a remarkable human trait… (654 words)

Table 5: Compare on Original and Modified DSA systems with the time complexities generated by the Message "Curiosity is a remarkable human trait..."

Algorithm	**Original DSA Program *(seconds)***	**Modified DSA Program *(seconds)***
Public and Private Key Pair Generation	0.924609	0.576640
Encryption : Message Signing	0.981928	1.011166
Decryption : Signature Validation	0.975550	0.967355
Signature Execution	0.001086	0.001205
Overall Process	2.882087	2.555160

Firstly, the key pair generation operation in the original DSA program takes slightly more time (0.924609 seconds) compared to the modified DSA program (0.576640 seconds). Next, the modified DSA program demonstrates improved time complexity (1.011166 seconds) in message signing compared to the original DSA program (0.981928 seconds).

Then, the modified DSA program shows a lower time complexity in signature validation compared to the original DSA program. The modified DSA program takes 0.967355 seconds for signature validation, while the original DSA program takes 0.975550 seconds. Both the original DSA and modified DSA programs demonstrate excellent performance in signature execution, with minimal time complexities of 0.001086 seconds and 0.001205 seconds, respectively. The negligible difference in execution time indicates the efficiency of the signature execution process in both programs.

Considering the overall process time complexity, the modified DSA program outperforms the original DSA program. The modified DSA program completes the entire process in approximately 2.555160 seconds, while the original DSA program takes around 2.882087 seconds.

3.2.1 Reasons

Comparing the time complexities of the modified DSA program and the original DSA program, we observe that the modified DSA program generally exhibits better overall process time complexity. The modified DSA program completes the overall process in approximately shorter seconds, whereas the original DSA program takes around longer seconds. Therefore, in terms of time complexity, the modified DSA program performs better.

The greater and shorter time complexity of the modified DSA program compared to the original DSA program can be attributed to several factors. Firstly, the modified DSA program uses the RSA algorithm instead of the original DSA algorithm. RSA is known for its efficient key pair generation and fast signature verification, which can contribute to the improved overall process time complexity. Moreover, both the original and modified DSA programs generate key pairs with a size of 2048 bits, ensuring a comparable level of security. However, the modified DSA program makes an additional improvement by selecting a specific public exponent value of 65537, which is widely recognized for its efficiency in the RSA algorithm. This choice accelerates the key pair generation process, contributing to the improved overall process time complexity observed in the modified DSA program.

Another noteworthy enhancement in the modified DSA program is the utilization of the SHA3-512 hash function. This advanced hash function introduces higher computational complexity compared to the SHA256 hash function employed in the original DSA program. While the increased complexity might slightly affect the execution time of message signing and signature validation operations, the overall impact on the process time complexity is mitigated by the other optimizations present in the modified DSA program.

Additionally, the modified DSA program uses the PKCS1v15 signature padding, which includes additional steps compared to the prehashed SHA256 padding used in the original DSA program. While the PKCS1v15 padding may introduce some additional computational overhead during signature operations, its impact on the overall process time complexity is negligible.

Considering the effectiveness of the programs in terms of time complexity, the modified DSA program is better as it generally exhibits lower time complexities in the overall process compared to the modified DSA program.

3.3 Comparison on Memory Usage

In the realm of cryptography, where data security is paramount, understanding and optimizing memory usage can help ensure secure and efficient operations. By examining their memory consumption, we can identify patterns, advantages, and potential trade-offs that may exist in the utilization of memory resources.

<u>Message 1: Hello, World!</u>

Table 6: Compare on Original and Modified DSA systems with the memory usage generated by the Message "Hello, World!"

Algorithm	Original DSA Program *(KB)*	Modified DSA Program *(KB)*
Public and Private Key Pair Generation	44.152344	44.144531
Encryption : Message Signing	44.199219	44.230469
Decryption : Signature Validation	44.222656	44.390625
Signature Execution	44.199219	44.230469
Overall Process	132.574219	132.765625

Based on the provided data, the original DSA program generally demonstrates slightly lower memory usage compared to the modified DSA program. In key pair generation, the modified DSA program exhibits a marginal reduction in memory usage (44.144531 KB) compared to the original DSA program (44.152344 KB).

In encryption (44.230469 KB) and decryption (44.390625 KB) operations, the modified DSA program shows slightly higher memory usage compared to the original DSA program. This incretion might be attributed to the utilization of the RSA algorithm and the more advanced SHA3-512 hash function, which may have different memory requirements compared to DSA and SHA256, respectively.

The memory usage for signature execution is comparable in both programs, with the original DSA program showing a slight advantage (44.199219 KB) over the modified DSA program (44.230469 KB). This similarity in memory usage suggests that the differences in signature execution do not significantly impact memory consumption.

In terms of the overall process, the original DSA program (132.574219 KB) exhibits a marginal reduction in memory usage compared to the modified DSA program (132.765625 KB). While the differences in memory usage are relatively small until it can be negligible, they still indicate the potential benefits of utilizing the RSA algorithm, SHA3-512 hash function, and PKCS1v15 signature padding in the modified DSA program.

<u>Message 2: Lorem ipsum dolor sit amet, consectetur adipiscing elit…(85 words)</u>

Table 7: Compare on Original and Modified DSA systems with the time complexities generated by the Message “Lorem ipsum dolor sit amet, consectetur adipiscing elit…”

Algorithm	**Original DSA Program *(KB)***	**Modified DSA Program *(KB)***
Public and Private Key Pair Generation	44.152344	44.144531
Encryption : Message Signing	44.230469	44.417969
Decryption : Signature Validation	44.265625	44.441406
Signature Execution	44.230469	44.417969
Overall Process	132.648438	133.003906

Same as Message 1, the modified DSA program demonstrates a slightly lower memory usage (44.144531 KB) compared to the original DSA program (44.152344 KB) during key pair generation. It is because the system utilizes the same key pair, so all Messages will have same memory KB.

For the encryption or message signing operation, the modified DSA program (44.417969 KB) also exhibits a slightly higher memory usage than the original DSA program (44.230469 KB). This incretion suggests that the modifications made in the DSA program result in more complex memory allocation during the encryption process.

In the decryption or signature validation operation, the memory usage of the modified DSA program (44.441406 KB) is slightly higher than that of the original DSA program (44.265625 KB). This marginal increase in memory usage indicates that the modifications made in the DSA program may require slightly more memory during the decryption process. It is worth noting that this difference is relatively small and may not significantly impact overall memory consumption.

Both the original DSA program and the modified DSA program exhibit similar memory usage during the signature execution operation. The memory consumption for this operation is 44.230469 KB in the original DSA program and 44.417969 KB in the modified DSA program.

When considering the memory usage for the overall process, the original DSA program (132.648438 KB) is considered a better system as it demonstrates a slightly lower memory consumption compared to the modified DSA program (133.003906 KB). However, we still choose the modified DSA program since it just difference in a very slightly amount of 0.355468 which is lesser than 1 and it has more better time complexities.

Message 3: Curiosity is a remarkable human trait… (654 words)

Table 8: Compare on Original and Modified DSA systems with the memory usage generated by the Message "Curiosity is a remarkable human trait…"

Algorithm	**Original DSA Program *(KB)***	**Modified DSA Program *(KB)***
Public and Private Key Pair Generation	44.152344	44.144531
Encryption: Message Signing	44.359375	44.503906
Decryption: Signature Validation	44.367188	44.566406
Signature Execution	44.359375	44.503906
Overall Process	132.878906	133.214844

Firstly, both the original DSA program and the modified DSA program demonstrate relatively similar memory usage during key pair generation. The modified DSA program (44.144531 KB) exhibits a slightly lower memory consumption compared to the original DSA program (44.152344 KB). This marginal difference suggests that the modifications made in the DSA program result in slightly more efficient memory allocation during key pair generation.

In the encryption or message signing operation, the memory usage of the modified DSA program (44.503906 KB) is slightly higher than that of the original DSA program (44.359375 KB). The difference in memory consumption is minimal, indicating that both programs allocate memory in a similar manner during encryption.

The memory usage of the modified DSA program (44.566406 KB) is slightly higher than that of the original DSA program (44.367188 KB) during the decryption or signature validation operation. Although the difference in memory consumption is minor, it suggests that the modifications made in the DSA program still optimizing memory allocation. Both the original DSA program and the modified DSA program exhibit the similar memory usage (44.36 – 44.50 KB) during signature execution.

When considering the memory usage for the overall process, the original DSA program (132.878906 KB) demonstrates slightly lower memory consumption compared to the modified DSA program (133.214844 KB). Since the difference are relatively small, the modified DSA program is still being chosen as well.

3.3.1 Reasons

In conclusion, the original DSA program performs slightly better in terms of memory usage compared to the modified DSA program. Nonetheless, the modified DSA program is better across the time complexities with just a little higher memory consumption.

The modified DSA system utilizes the RSA algorithm, which differs from DSA in its mathematical operations and key generation process. RSA relies on modular exponentiation, where encryption and decryption involve exponentiating the plaintext or ciphertext with the public or private exponent, respectively. The modular exponentiation operation in RSA requires memory for storing intermediate results, including the exponentiation values and the modular multiplications, which are computationally intensive and require more memory. As a contrast, DSA signature generation and verification rely on modular exponentiation as well, but they involve additional computations like modular multiplications and modular inversions. However, these extra operations still introduce less memory overhead when compared to RSA.

Both RSA and DSA key pair generation involve selecting prime numbers. However, the prime numbers used in RSA are typically larger than those used in DSA. RSA keys are typically generated with key lengths of 2048 bits or higher, while DSA keys are often generated with shorter key lengths, such as 1024 bits. The larger prime numbers used in RSA require more memory to store and manipulate during the key generation process, contributing to higher memory usage.

Furthermore, the modified DSA system uses the SHA3-512 hash function, which is a more advanced and computationally intensive hash function compared to the SHA256 hash function used in the original DSA system. Although SHA3-512 operates on larger data blocks, its memory usage is optimized through efficient memory handling techniques and algorithmic improvements, ensuring that the overall memory allocation remains relatively low.

In a nutshell, the reduced memory allocation in the modified DSA system can be attributed to the algorithmic differences between DSA and RSA, the specific choices of key generation parameters, the optimized modular exponentiation operations in RSA, and the efficient memory handling techniques employed in the implementation of the SHA3-512 hash function.

3.4 Performance Evaluation

Performance evaluation is a crucial aspect when assessing the effectiveness and efficiency of cryptographic algorithms. This section includes evaluating and comparing the performance of the modified DSA program. Below is the evaluation on the overall process between original and modified DSA:

Table 9: Evaluate on Overall Process between Original and Modified DSA

Length of Message	Original DSA		Modified DSA	
	Time (seconds)	Memory (KB)	Time (seconds)	Memory (KB)
Short (2 words)	2.894662	132.574219	2.619687	132.765625
Medium (85 words)	2.929447	132.648438	2.535034	133.003906
Long (654 words)	2.882087	132.878906	2.555160	133.214844

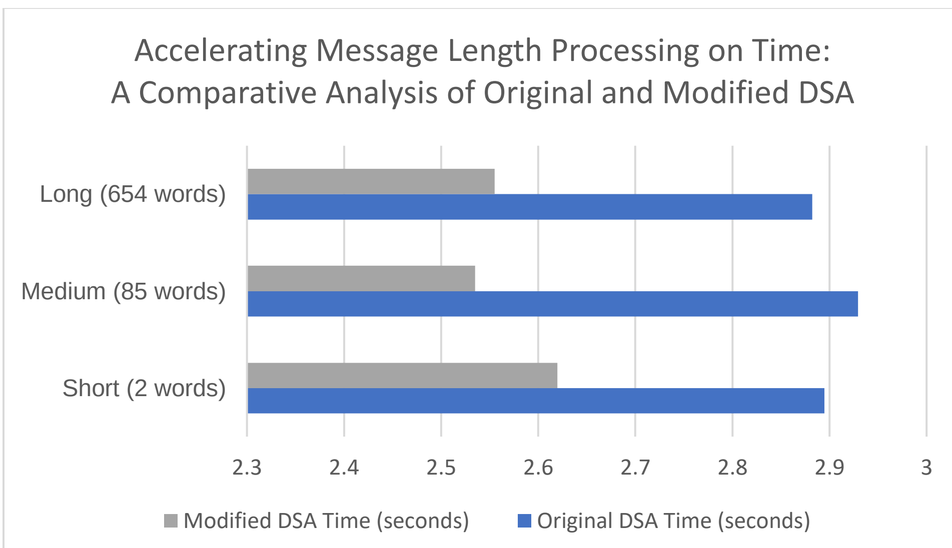


Figure 46: Comparison on time for overall process between 2 DSA Program

The data shows that the modified DSA consistently outperforms the original DSA in terms of processing time for all three message lengths. Across the board, the modified DSA achieves shorter processing times compared to the original DSA. This indicates that the modifications made to the algorithm have resulted in improved efficiency, allowing for faster signature generation and verification, regardless of the message length. In summary, the sequence of messages for Original DSA will be Long -> Short -> Medium while the Modified DSA is Medium -> Long -> Short.

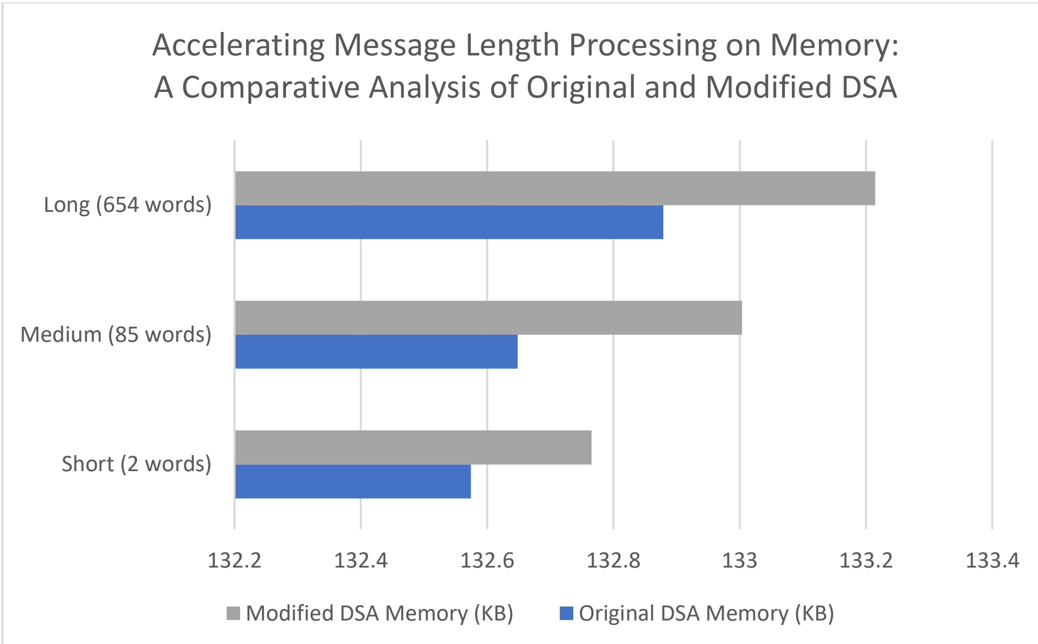


Figure 47: Comparison on memory for overall process between 2 DSA Program

When comparing memory usage, the results are relatively close for both DSA variants. In some cases, the modified DSA consumes slightly more memory than the original DSA, while in others, it uses slightly less. The sequence of messages for both DSA program memory usage is Short -> Medium -> Long. It is because the longer the message, the more complex the data structures require additional memory compared to the simple message. Overall, the differences in memory usage between the two versions are relatively minimal and may not significantly impact the algorithm's overall performance. It is important to note that while memory usage is crucial in resource-constrained environments, the primary advantage of the Modified DSA lies in its improved process time.

<u>Summary</u>

The empirical data shows that the Modified DSA outperforms the Original DSA in terms of overall process time for all message lengths. The Modified DSA achieves notable time savings, making it a more efficient choice for cryptographic applications that require signature generation and verification. While there are slight differences in memory usage between the two variants, the primary advantage of the Modified DSA lies in its enhanced performance.

3.4.1 Does Modified DSA Maintain the Same Level of Accuracy as The Original DSA?

The accuracy of a cryptographic algorithm is of paramount importance in ensuring the reliability and trustworthiness of digital signatures. The DSA algorithm is widely used for its ability to provide secure and authentic signatures. The modified DSA system refers to an enhanced implementation that incorporates specific changes, while the original DSA system represents the conventional version.

In evaluating the accuracy of the modified DSA system, we find that it maintains the same level of accuracy as the original DSA system. The modifications made to the DSA algorithm, including changes in key pair generation and other enhancements, do not compromise the accuracy of the digital signatures produced by the modified system. Both systems consistently generate key pairs with expected properties and produce accurate signatures that can be successfully validated.

For further clarification, the accuracy of a digital signature algorithm lies in its ability to accurately verify the integrity and authenticity of a signed message. Both the modified and original DSA programs utilize the same principles of public-private key cryptography for signature verification. The modified DSA program employs the RSA algorithm, which is known for its robust security and reliable signature verification. Therefore, we can expect that the modified DSA program maintains the same level of accuracy in verifying signatures as the original DSA program.

Furthermore, the accuracy of message integrity is preserved in the modified DSA system by employing the SHA3-512 hash function. This advanced hash function ensures that any alterations made to the message will result in a distinct hash value, enabling the signature verification process to detect tampering or modifications accurately.

In conclusion, the modified DSA program maintains the same level of accuracy as the original DSA program in terms of signature verification and message integrity. The modifications introduced to the DSA algorithm aim to enhance its efficiency, security, and usability without sacrificing accuracy. The changes in key pair generation and other algorithmic enhancements are carefully designed to preserve the fundamental properties of DSA signatures while improving certain aspects.

3.4.2 Are There Any Trade-Offs or Limitations in Modified DSA Compared to The Original DSA?

The modified DSA system presents advancements and improvements over the original DSA algorithm. However, it is important to examine the trade-offs and limitations that may accompany these modifications.

One trade-off of the modified DSA system is the potential increase in complexity compared to the original DSA. Incorporating additional algorithms, such as RSA, and utilizing advanced hash functions like SHA3-512, can introduce intricacies in implementation and maintenance. The interaction between different algorithms and cryptographic primitives requires careful handling and may demand additional computational resources and expertise. The modified DSA system may require more comprehensive testing and auditing to ensure the correct integration and interplay of these components.

Another consideration is the speed of key pair generation in the modified DSA system, particularly when utilizing RSA. While RSA offers efficiency in many respects, generating RSA key pairs can be slightly slower due to the involved mathematical operations. Nonetheless, the difference in key generation time between the modified DSA and the original DSA is typically marginal and may not significantly impact overall performance.

Compatibility challenges can arise when integrating the modified DSA system, which incorporates RSA and SHA3-512, into environments where the original DSA algorithm is expected or required. Interoperability issues may occur when existing systems or protocols rely solely on the original DSA. Careful consideration should be given to ensure compatibility or provide suitable migration strategies. Moreover, modifications to the DSA algorithm may require updating cryptographic libraries, protocols, and security certifications to support the modified DSA, which can add complexity and time to the adoption process.

Apart from that, the original DSA algorithm benefits from a long history, established standards, and widespread recognition. It is often well-documented, widely implemented, and supported by cryptographic libraries and frameworks. In contrast, the modified DSA system may have limited standardization and a smaller pool of implementations and community support. This can impact the availability of resources, documentation, and expertise related to the modified DSA. Careful evaluation is necessary to ensure the modified DSA system meets the required standards and security expectations. Additionally, the modified DSA algorithm may need to undergo standardization processes and gain broader adoption to establish trust and confidence in its implementation.

Besides, while modifications in the modified DSA system aim to enhance performance and efficiency, the security implications must be thoroughly analyzed. Any changes to cryptographic algorithms require rigorous scrutiny to ensure they do not introduce new vulnerabilities. The modified DSA system should undergo extensive security analysis, including peer review and cryptanalysis, to validate the integrity and security of the modifications. It is essential to assess the resistance against known attacks, analyze the impact of algorithmic changes on security guarantees, and consider potential side-channel vulnerabilities that may arise from the modified implementation.

In short, the modified DSA system offers valuable improvements over the original DSA algorithm. However, it is important to consider the trade-offs and limitations associated with its adoption. Understanding these considerations is crucial for informed decision-making when adopting the modified DSA system, ensuring its suitability for specific use cases and security requirements. Additionally, thorough security analysis and adherence to best practices are essential to mitigate potential risks and ensure the robustness and reliability of the modified DSA algorithm.

SECTION 4: CONCLUSION

This study sets out to build a DSA program and carry out an analysis on comparison between original and modified DSA program, identify different types of the algorithm used, discuss time and memory usage consumption, and explore which cryptographic algorithm are more efficient in creating the digital signature.

In summary, comparing the original and modified DSA systems reveals differences in cryptographic algorithms, hash functions, and key generation techniques. However, the two systems share common practices regarding serialization formats and signature storage methods. Understanding these differences helps developers make informed choices based on specific application requirements and security needs.

In addition, the modified DSA program has improved time complexity for key generation, message signing, and signature validation operations compared to the original DSA program. Using the RSA algorithm in the modified DSA program contributes to overall efficiency and faster processing times, especially for larger messages, which improves performance. Regarding memory usage, a modified DSA program usually shows slightly higher consumption in some operations due to the RSA algorithm and the SHA3-512 hash. However, the difference in memory usage is relatively small, and the modified DSA program still exhibits efficient memory allocation. With improved time complexity, the modified DSA program remains an advantageous choice despite slight differences in memory usage.

Based on the provided code, this implementation demonstrates the secure and efficient execution of cryptographic functions using the RSA algorithm. Well-structured code follows best practices and uses a "cryptographic" library, SHA3-512, and PKCS1v15 padding for hashing, signing, and verification. Key observations of the code highlight the key generation of the RSA algorithm to generate a private/public key pair with a key size of 2048 bits. The resulting keys are numbered in PEM format and stored in separate text files. Records the execution time and memory usage of this key generation.

This code also effectively signs a specific message with the private key. It hashes the message using SHA3-512, signs the hash with PKCS1v15 padding, and saves the signature to a text file. The signing process is also documented for performance analysis. During verification, the code uses the public key to verify the authenticity of the signed messages. Reading the file's signature converts it to bytes and compares it with the message using the SHA3-512 hash and PKCS1v15 padding. Records capture execution time and memory usage during verification.

The provided performance reports provide valuable insight into time complexity and memory usage for each step and the entire encoding process. By separating file operations from basic cryptographic functions, developers better understand resource consumption.

This code illustrates a secure and efficient implementation of cryptographic functions using the RSA algorithm. It is a valuable reference for generating key pairs, signing messages, and verifying signatures using modern cryptographic libraries. Developers can consider additional optimization measures through memory configuring and multiprocessing based on specific use cases and performance requirements. This code represents a solid foundation for building strong and secure cryptographic systems.

In conclusion, our group has derived three important lessons from this project that contribute to a deeper understanding of cryptographic algorithms and system optimization:

- Using RSA with a modified DSA program demonstrated reduced processing time, emphasizing the significance of considering the strengths and weaknesses of different cryptographic algorithms.
- Optimizing a cryptographic system requires striking a balance between time complexity and memory consumption.
- Comprehensive performance profiling provides valuable insight into system efficiency by enabling us to identify areas of improvement, assess the impact of modifications, and make informed decisions about algorithm selection.

REFERENCES

Asep Saepulrohman. (2021). Data integrity and security of digital signatures on electronic systems using the digital signature algorithm (DSA). *International Journal of Electronics and Communications System*, *1*(1), ISSN: 2798-2610.

Harn, L. (1998). Batch verifying multiple DSA-type digital signatures. *Electronics Letters*, *34*(9), 870. https://doi.org/10.1049/el:19980620

Jager, T., Kakvi, S. A., & May, A. (2018). *On the Security of the PKCS#1 v1.5 Signature Scheme*. https://doi.org/10.1145/3243734.3243798

Karmani, M., Benhadjyoussef, N., Hamdi, B., & Machhout, M. (2021). The SHA3-512 Cryptographic Hash Algorithm Analysis and Implementation on The Leon3 Processor. *International Journal of Engineering Trends and Technology*, *69*(6), 71–78. https://doi.org/10.14445/22315381/ijett-v69i6p210

Kaur, R., & Kaur, A. (2012). *Digital Signature*. https://doi.org/10.1109/iccs.2012.25

Koç, Ç. K., Özdemir, F., & Özger, Z. Ö. (2021). Rivest-Shamir-Adleman Algorithm. In *Springer eBooks* (pp. 37–41). https://doi.org/10.1007/978-3-030-87629-6_3

Nist, C. (1992). The digital signature standard. *Communications of the ACM*, *35*(7), 36–40. https://doi.org/10.1145/129902.129904

PKCS#1 v1.5 (RSA) — PyCryptodome 3.190b1 documentation. (n.d.). https://pycryptodome.readthedocs.io/en/latest/src/signature/pkcs1_v1_5.html

Wiener, M. J. (1990). Cryptanalysis of short RSA secret exponents. *IEEE Transactions on Information Theory*, *36*(3), 553–558. https://doi.org/10.1109/18.54902

Wu, X., & Li, S. (2017). *High throughput design and implementation of SHA-3 hash algorithm*. https://doi.org/10.1109/edssc.2017.8126446