Experiment No : 9

**Aim:** To Study and Implement a program for RSA Algorithm to encrypt and decrypt the message.

Introduction :

The RSA (Rivest-Shamir-Adleman) algorithm is a widely used public-key cryptosystem that was invented by Ron Rivest, Adi Shamir, and Leonard Adleman in 1977. RSA is renowned for its ability to securely encrypt and digitally sign data, making it a fundamental cornerstone of modern information security. The algorithm relies on the mathematical properties of large prime numbers and modular arithmetic to provide secure communication and data protection. RSA has applications in secure email, e-commerce, digital signatures, and more.

In RSA, each user has a pair of keys: a public key, which is known to everyone, and a private key, which is kept secret. The public key is used for encryption, while the private key is used for decryption. The security of RSA is grounded in the difficulty of factoring the product of two large prime numbers, which is an essential component of the algorithm. As prime factorization is a computationally intensive task, RSA encryption remains secure as long as sufficiently large keys are used.

One of the key advantages of RSA is its versatility in providing both confidentiality and authenticity in digital communication. It ensures that only the intended recipient can decrypt the message with their private key, while the sender can digitally sign their messages with their private key to prove their identity. RSA's enduring popularity, based on the principles of asymmetric cryptography, has made it a vital tool in the protection of sensitive information and the establishment of secure online transactions in our increasingly digital world.

**Program (Source Code): (hillcipher)**

import random

# Helper function to check if a number is prime

def **is\_prime**(num):

    if num <= 1:

        return False

    if num <= 3:

        return True

    if num % 2 == 0 or num % 3 == 0:

        return False

    i = 5

    while i \* i <= num:

        if num % i == 0 or num % (i + 2) == 0:

            return False

        i += 6

    return True

# Helper function to compute the greatest common divisor (GCD)

def **gcd**(a, b):

    while b:

        a, b = b, a % b

    return a

# Helper function to find the modular multiplicative inverse

def **mod\_inverse**(a, m):

    m0, x0, x1 = m, 0, 1

    while a > 1:

        q = a // m

        m, a = a % m, m

        x0, x1 = x1 - q \* x0, x0

    return x1 + m0 if x1 < 0 else x1

def **list\_prime**(num):

    prime\_list = []

    for i in range(2, num):

        if **is\_prime**(i):

            prime\_list.**append**(i)

    return prime\_list

# Generate two large prime numbers

def **generate\_prime**(bits):

    num = random.getrandbits(bits)

    return prime\_number\_list[num % **len**(prime\_number\_list)]

# Generate public and private keys

def **generate\_keys**(bits):

    p = **generate\_prime**(bits)

**print**(f"p={p}")

    q = **generate\_prime**(bits)

**print**(f"q={q}")

    n = p \* q

    phi = (p - 1) \* (q - 1)

    e = 65537  # A common choice for the public exponent

    d = **mod\_inverse**(e, phi)

    return (e, n), (d, n)

# Encrypt a message

def **encrypt**(public\_key, message):

    e, n = public\_key

    encrypted = [**pow**(**ord**(char), e, n) for char in message]

    return encrypted

# Decrypt a message

def **decrypt**(private\_key, encrypted):

    d, n = private\_key

    decrypted = [**chr**(**pow**(char, d, n)) for char in encrypted]

    return ''.**join**(decrypted)

if \_\_name\_\_ == '\_\_main\_\_':

    prime\_number\_list = **list\_prime**(10000)

    public\_key, private\_key = **generate\_keys**(1024)

    message = "Kem Cho, Majama"

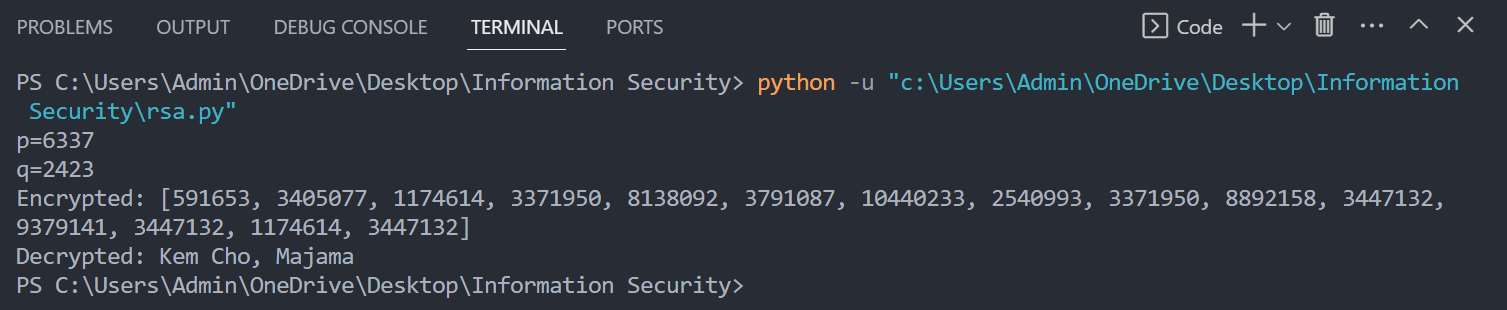
    encrypted\_message = **encrypt**(public\_key, message)

**print**("Encrypted:", encrypted\_message)

    decrypted\_message = **decrypt**(private\_key, encrypted\_message)

**print**("Decrypted:", decrypted\_message)

**OUTPUT(PROGRAM):**



**OUTPUT(CRYPTOOL):**

**CRYPTANALYSIS:**

Cryptanalysis of the RSA algorithm involves attempting to break its security by exploiting weaknesses in the underlying mathematics or implementation. While RSA is considered a robust encryption method, there are several cryptanalysis techniques and potential vulnerabilities to consider:

**Integer Factorization:** RSA's security relies on the difficulty of factoring the product of two large prime numbers. Cryptanalysts continually work on more efficient factoring algorithms. The development of powerful quantum computers poses a potential threat to RSA, as they may be able to factor large numbers much faster than classical computers.

**Timing Attacks:** Cryptanalysts can exploit information leakage through the timing of encryption or decryption operations. By analyzing the time taken to execute these operations, attackers might gain insights into the private key.

**Chosen Ciphertext Attacks (CCA)**: In CCA attacks, an attacker can interact with an oracle to decrypt chosen ciphertexts. While modern RSA implementations incorporate padding schemes like OAEP or PKCS#1 v1.5 to mitigate these attacks, vulnerabilities may still arise if these schemes are not correctly implemented.

**Side-Channel Attacks:** RSA implementations can be susceptible to side-channel attacks based on the electromagnetic emissions, power consumption, or other observable aspects of the hardware or software during cryptographic operations. Attackers can use these observable signals to deduce private key information.

**Weak Key Generation:** Weak key generation can lead to vulnerabilities in RSA. If users generate keys with insufficient randomness or use predictable prime numbers, it becomes easier for cryptanalysts to crack the encryption. Secure key generation practices are crucial to RSA's strength.

**APPLICATIONS:**

The RSA (Rivest-Shamir-Adleman) algorithm is a versatile encryption and digital signature scheme with various applications in information security. Here are six prominent applications of the RSA algorithm:

**Secure Communication:** RSA is widely used in securing communication over the internet, particularly for encrypting email and instant messaging. It ensures that data exchanged between parties remains confidential and cannot be intercepted or deciphered by unauthorized individuals.

**Digital Signatures:** RSA is employed for creating digital signatures to verify the authenticity and integrity of digital documents, software, or messages. By signing data with a private key, a sender can prove that the content has not been tampered with and is genuinely from them.

**Secure Web Browsing:** RSA is a fundamental component of secure web browsing. It is used to establish SSL/TLS (Secure Sockets Layer/Transport Layer Security) connections, encrypting data exchanged between web browsers and servers. This ensures the privacy of sensitive information like credit card details during online transactions.

**Authentication and Access Control:** RSA is employed for user authentication in various systems, including VPNs, remote access, and secure login processes. Users have their private keys to prove their identity, which enhances security in access control.

**REFERENCES:**

**Stallings, W. (2017). *Cryptography and Network Security: Principles and Practice.*** Pearson. <https://www.pearson.com/en-us/subject-catalog/p/cryptography-and-network-security-principles-and-practice/P200000003477>

Experiment No : 10

**Aim:** To Study and Implement a program for Python code to use RSA for generation and verification of digital signature on file. It should include models for sender's authentication only and for message confidentiality and sender's authentication. Also, include a user interface and a menu.

Introduction :

RSA, the renowned encryption and digital signature algorithm developed by Ron Rivest, Adi Shamir, and Leonard Adleman in 1977, plays a pivotal role in securing digital communications. This cryptographic powerhouse serves as a cornerstone in the realm of digital signatures, enabling the generation and verification of digital signatures on files. A digital signature is a cryptographic mechanism that ensures the authenticity and integrity of digital documents. In this context, RSA provides a robust solution to achieve sender authentication while also facilitating message confidentiality and sender authentication.This multifaceted functionality is crucial for a wide array of applications, from secure email communications to legal documents and financial transactions.

In the realm of digital signatures, RSA operates in two principal models: one for sender authentication only and another for both message confidentiality and sender authentication. In the sender authentication model, RSA is used to create a digital signature that can be appended to a file or document. The sender computes the signature using their private key, and recipients can verify the signature using the sender's public key. This process ensures that the document indeed originated from the claimed sender, providing a robust mechanism for sender authentication.

In the more comprehensive model, RSA not only offers sender authentication but also provides message confidentiality. In this case, the file is not only digitally signed but also encrypted using the recipient's public key. As a result, not only is the sender's authenticity guaranteed, but the contents of the message remain confidential, as only the recipient with the corresponding private key can decrypt it.

To implement RSA-based digital signatures, a user-friendly interface is essential. The user interface typically consists of a menu where users can select various options, such as "Sign File" to generate digital signatures and "Verify Signature" to confirm the authenticity and integrity of received files. This menu-driven approach simplifies the process, making it accessible to users with varying levels of technical expertise.

In essence, RSA's application in the generation and verification of digital signatures on files serves as a robust mechanism for enhancing trust and security in the digital realm. It empowers individuals and organizations to ensure that documents are tamper-proof and indeed come from the claimed sender. Moreover, when combined with encryption, it extends its utility to preserving message confidentiality. Through a user-friendly interface and menu, RSA's capabilities are harnessed effectively, making it a fundamental tool in securing modern digital communications and transactions.

**Program (Source Code): (hillcipher)**

import os

import hashlib

from cryptography.hazmat.primitives import serialization

from cryptography.hazmat.primitives.asymmetric import rsa

from cryptography.hazmat.primitives.asymmetric import padding

from cryptography.hazmat.primitives import hashes

def **generate\_key\_pair**():

    private\_key = rsa.**generate\_private\_key**(

        public\_exponent=65537,

        key\_size=2048

    )

    private\_pem = private\_key.**private\_bytes**(

        encoding=serialization.Encoding.PEM,

        format=serialization.PrivateFormat.TraditionalOpenSSL,

        encryption\_algorithm=serialization.NoEncryption()

    )

    public\_key = private\_key.**public\_key**().**public\_bytes**(

        encoding=serialization.Encoding.PEM,

        format=serialization.PublicFormat.SubjectPublicKeyInfo

    )

    return private\_pem, public\_key, private\_key

def **sign\_message**(private\_key, message):

    private\_key = serialization.**load\_pem\_private\_key**(private\_key, password=None)

    signature = private\_key.**sign**(

        message.encode('utf-8'),

        padding.PSS(

            mgf=padding.MGF1(hashes.SHA256()),

            salt\_length=padding.PSS.MAX\_LENGTH

        ),

        hashes.SHA256()

    )

    return signature

def **verify\_signature**(public\_key, message, signature):

    public\_key = serialization.**load\_pem\_public\_key**(public\_key)

    try:

        public\_key.**verify**(

            signature,

            message.encode('utf-8'),

            padding.PSS(

                mgf=padding.MGF1(hashes.SHA256()),

                salt\_length=padding.PSS.MAX\_LENGTH

            ),

            hashes.SHA256()

        )

        return True

    except Exception:

        return False

def **encrypt\_file**(public\_key, input\_file, output\_file):

    public\_key = serialization.**load\_pem\_public\_key**(public\_key)

    with **open**(input\_file, 'rb') as f:

        file\_data = f.**read**()

    encrypted\_data = public\_key.**encrypt**(

        file\_data,

        padding.OAEP(

            mgf=padding.MGF1(algorithm=hashes.SHA256()),

            algorithm=hashes.SHA256(),

            label=None

        )

    )

    with **open**(output\_file, 'wb') as f:

        f.**write**(encrypted\_data)

def **decrypt\_file**(private\_key, input\_file, output\_file):

    private\_key = serialization.**load\_pem\_private\_key**(private\_key, password=None)

    with **open**(input\_file, 'rb') as f:

        encrypted\_data = f.**read**()

    decrypted\_data = private\_key.**decrypt**(

        encrypted\_data,

        padding.OAEP(

            mgf=padding.MGF1(algorithm=hashes.SHA256()),

            algorithm=hashes.SHA256(),

            label=None

        )

    )

    with **open**(output\_file, 'wb') as f:

        f.**write**(decrypted\_data)

def **main**():

    private\_key = None

    public\_key = None

    while True:

**print**("Menu:")

**print**("1. Generate Key Pair")

**print**("2. Sign a File")

**print**("3. Verify Signature")

**print**("4. Encrypt File")

**print**("5. Decrypt File")

**print**("6. Quit")

        choice = **input**("Enter your choice: ")

        if choice == "1":

            private\_key, public\_key, key\_pair = **generate\_key\_pair**()

**print**("Key pair generated.")

**print**("Private Key:\n", key\_pair.**private\_bytes**(

                encoding=serialization.Encoding.PEM,

                format=serialization.PrivateFormat.TraditionalOpenSSL,

                encryption\_algorithm=serialization.NoEncryption()).**decode**('utf-8'))

**print**("Public Key:\n", public\_key.**decode**('utf-8'))

        elif choice == "2":

            if private\_key is None:

**print**("Please generate a key pair first.")

            else:

                input\_file = **input**("Enter the name of the file to sign: ")

                if os.path.**exists**(input\_file):

                    with **open**(input\_file, 'r') as f:

                        message = f.**read**()

                    signature = **sign\_message**(private\_key, message)

                    with **open**("signature.bin", "wb") as signature\_file:

                        signature\_file.**write**(signature)

**print**("Signature generated and saved as signature.bin")

                else:

**print**("File not found.")

        elif choice == "3":

            if public\_key is None:

**print**("Please generate a key pair first.")

            else:

                input\_file = **input**("Enter the name of the file to verify: ")

                if os.path.**exists**(input\_file):

                    with **open**(input\_file, 'r') as f:

                        message = f.**read**()

                    with **open**("signature.bin", "rb") as signature\_file:

                        signature = signature\_file.**read**()

                    if **verify\_signature**(public\_key, message, signature):

**print**("Signature is valid.")

                    else:

**print**("Signature is not valid.")

                else:

**print**("File not found.")

        elif choice == "4":

            if public\_key is None:

**print**("Please generate a key pair first.")

            else:

                input\_file = **input**("Enter the name of the file to encrypt: ")

                output\_file = **input**("Enter the name of the encrypted file: ")

                if os.path.**exists**(input\_file):

**encrypt\_file**(public\_key, input\_file, output\_file)

**print**("File encrypted and saved.")

                else:

**print**("File not found.")

        elif choice == "5":

            if private\_key is None:

**print**("Please generate a key pair first.")

            else:

                input\_file = **input**("Enter the name of the file to decrypt: ")

                output\_file = **input**("Enter the name of the decrypted file: ")

                if os.path.**exists**(input\_file):

**decrypt\_file**(private\_key, input\_file, output\_file)

**print**("File decrypted and saved.")

                else:

**print**("File not found.")

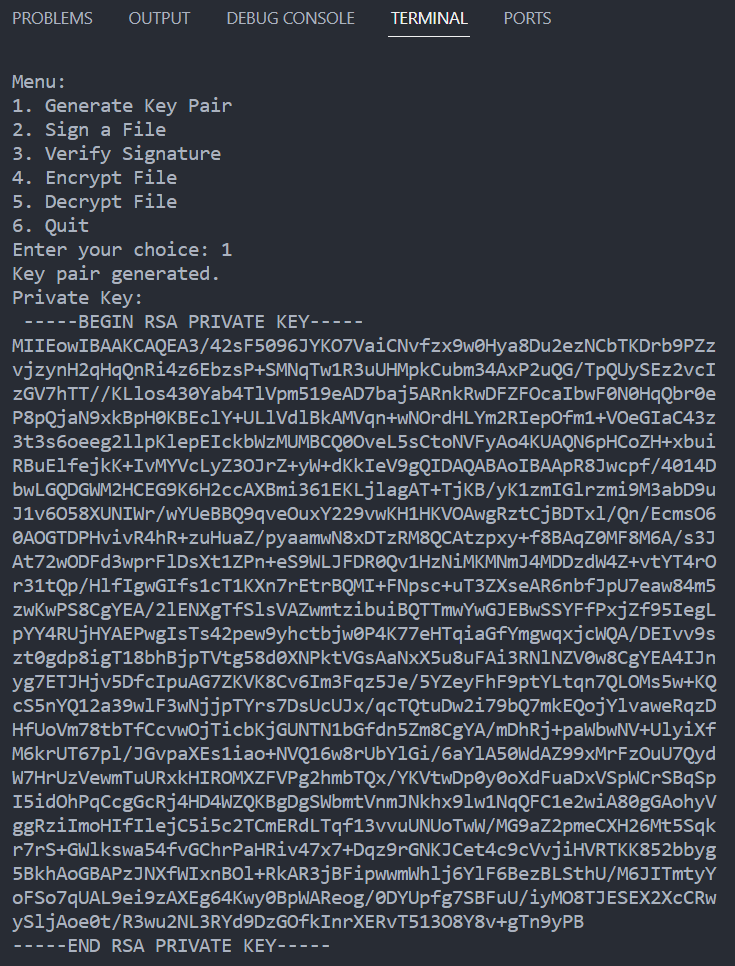
        elif choice == "6":

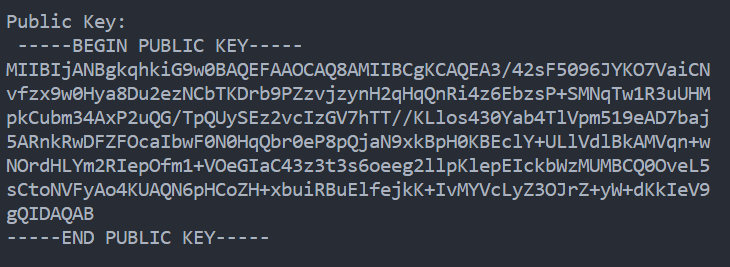
            break

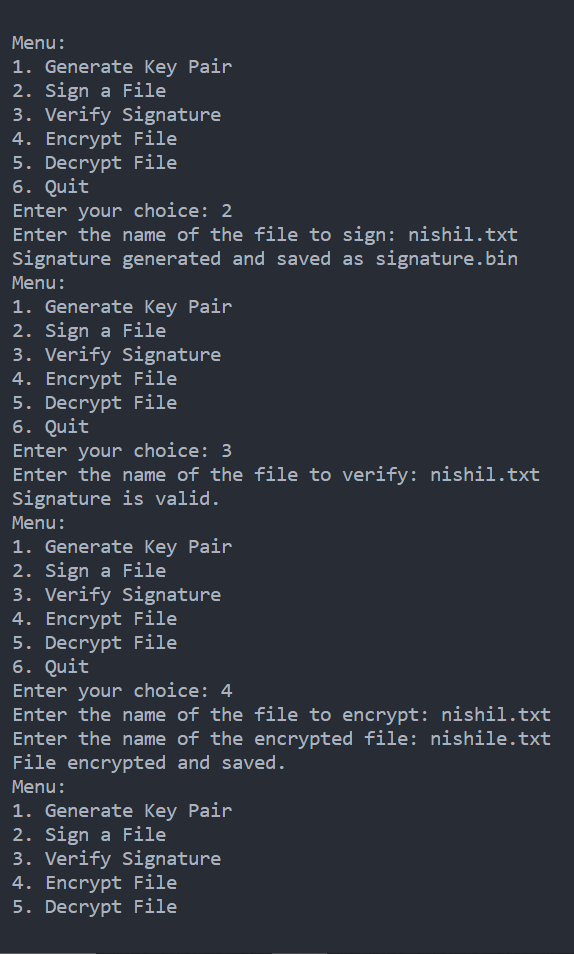
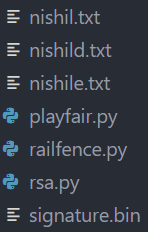
if \_\_name\_\_ == "\_\_main\_\_":

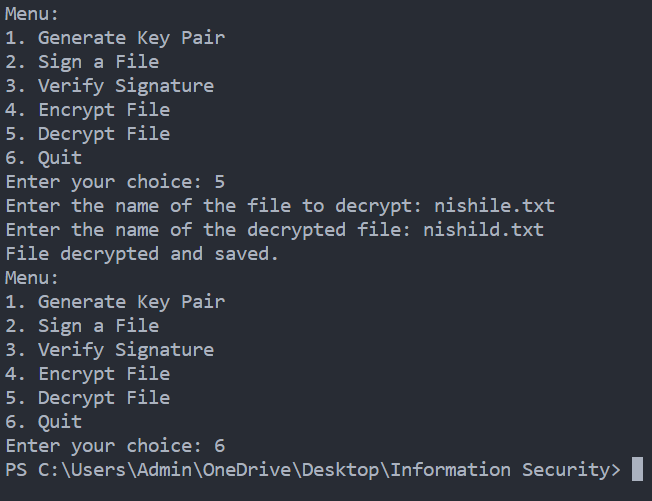
**main**()

**OUTPUT(PROGRAM):**







**OUTPUT(CRYPTOOL):**

**CRYPTANALYSIS:**

Cryptanalysis for RSA, particularly in the context of digital signature generation and verification, involves examining potential vulnerabilities and attacks against the system. RSA is widely regarded as a robust algorithm, but it's essential to understand the potential weaknesses and how to mitigate them.

Cryptanalysis of RSA for Digital Signature Generation and Verification

**Brute Force Attack:** One common cryptanalysis technique is to perform a brute force attack by systematically trying all possible private key combinations to forge a valid digital signature. This attack becomes increasingly difficult as the key size grows, making longer key lengths more resistant to brute force attacks.

**Padding Oracle Attack:** In digital signature schemes, attackers may exploit padding oracles to gain information about the private key. Padding schemes like PKCS#1 v1.5 are known to be vulnerable to this type of attack, which can reveal the private key if improperly implemented.

**Timing Attacks:** Cryptanalysts may leverage timing information to extract the private key. By measuring the time it takes to perform signature operations, an attacker can gain insights into the private key, especially when dealing with vulnerable software or hardware implementations.

**Adaptive Chosen-Message Attack:** Attackers can use adaptive chosen-message attacks to manipulate the signing oracle. In this scenario, the attacker submits messages for signing and observes the corresponding signatures. Over time, this could lead to the extraction of the private key.

**Random Number Generator (RNG) Vulnerabilities:** Weaknesses in the random number generator used for key pair generation can compromise the entire RSA system. If the generated prime numbers are not sufficiently random, an attacker might be able to predict them, which can lead to private key recovery.

**Small Private Exponent Attack:** If the private exponent is too small, it may be vulnerable to attacks like Wiener's attack. Cryptanalysts can use continued fraction expansions to recover the private key when the private exponent is small relative to the modulus.

**Fault Injection Attacks:** Attackers can manipulate the RSA operations by introducing faults in the computation process. By injecting errors, they may gain insights into the private key. This type of attack often requires physical access to the hardware.

**Quantum Computing Threat:** As quantum computers advance, they pose a potential threat to RSA. Shor's algorithm, when executed on a powerful quantum computer, may factor large RSA modulus efficiently, rendering RSA insecure. This emphasizes the need for post-quantum cryptographic solutions.

**APPLICATIONS:**

RSA digital signatures are widely used for securing data and verifying the authenticity of digital files. Here are the applications for RSA in the generation and verification of digital signatures on files, considering both sender's authentication and message confidentiality:

**Email Communication:**

Sender's Authentication Only: In email communication, RSA digital signatures ensure that emails are genuinely sent by the claimed sender, adding a layer of trust to the message. The recipient can verify the sender's identity based on the digital signature.

Message Confidentiality and Sender's Authentication: Combining RSA digital signatures with encryption (such as PGP or S/MIME) ensures both the sender's authenticity and message confidentiality. The recipient can verify the sender and decrypt the message securely.

**Software Distribution:**

Sender's Authentication Only: Software developers use RSA digital signatures to sign software packages. Users can verify that the software has not been tampered with and comes from the authentic source.

Message Confidentiality and Sender's Authentication: In cases where software also contains sensitive information, RSA digital signatures can be paired with encryption to ensure both the sender's authenticity and the confidentiality of the software's content.

**Legal Documents:**

Sender's Authentication Only: Law firms and notaries use RSA digital signatures to sign legal documents, making them legally binding. Recipients can verify the authenticity of documents in a court of law.

Message Confidentiality and Sender's Authentication: For confidential legal documents, combining digital signatures with encryption is essential to safeguard the content and the sender's identity.

**Financial Transactions:**

Sender's Authentication Only: In online banking and financial transactions, RSA digital signatures authenticate the sender and the integrity of transaction details. This builds trust in online banking services.

Message Confidentiality and Sender's Authentication: When financial data is sensitive, combining digital signatures with encryption ensures both the sender's authenticity and the confidentiality of the financial information.

**Secure File Sharing and Collaboration:**

Sender's Authentication Only: Secure file sharing platforms use RSA digital signatures to verify that shared files are unaltered and sent by the claimed user. This enhances the security of shared documents in collaborative environments.

Message Confidentiality and Sender's Authentication: For private or confidential documents, combining digital signatures with encryption provides comprehensive protection, ensuring the sender's authenticity and message confidentiality.

**REFERENCES:**

**Stallings, W. (2017). *Cryptography and Network Security: Principles and Practice.*** Pearson. <https://www.pearson.com/en-us/subject-catalog/p/cryptography-and-network-security-principles-and-practice/P200000003477>