Digital Signature using RSA

RSA Digital Signature is a digital signature scheme based on the RSA (Rivest-Shamir-Adleman) cryptosystem. It provides a way to verify the authenticity and integrity of a message, ensuring that it comes from a known sender and hasn't been tampered with.

Key Concepts

1. **Public-Private Key Pair**: RSA uses asymmetric cryptography, where each user has a public key (widely shared) and a private key (kept secret).
2. **Signing Process**: The sender uses their private key to create a signature for a message.
3. **Verification Process**: Anyone with the sender's public key can verify the signature.

How RSA Digital Signature Works

Signing a Message

1. The sender computes a hash (e.g., SHA-256) of the message.
2. This hash is then encrypted using the sender's private key, creating the signature.
3. The original message and the signature are sent together.

Verifying a Signature

1. The receiver computes the hash of the received message.
2. The receiver decrypts the signature using the sender's public key.
3. If the decrypted signature matches the computed hash, the signature is valid.

Mathematical Foundation

RSA Digital Signature relies on the mathematical properties of modular exponentiation and the difficulty of factoring large numbers.

- Private Key: (d, n)
- Public Key: (e, n)

Where n is the product of two large prime numbers, e is the public exponent, and d is the private exponent.

Signing: $S = M^d \bmod n$

Verifying: $M = S^e \bmod n$

(M is the message hash, S is the signature)

Security Considerations

- The security of RSA Digital Signature depends on keeping the private key secret.
- The key size should be sufficiently large (e.g., 2048 bits or more) to resist factorization attacks.
- Proper padding schemes (e.g., PSS) should be used to enhance security.

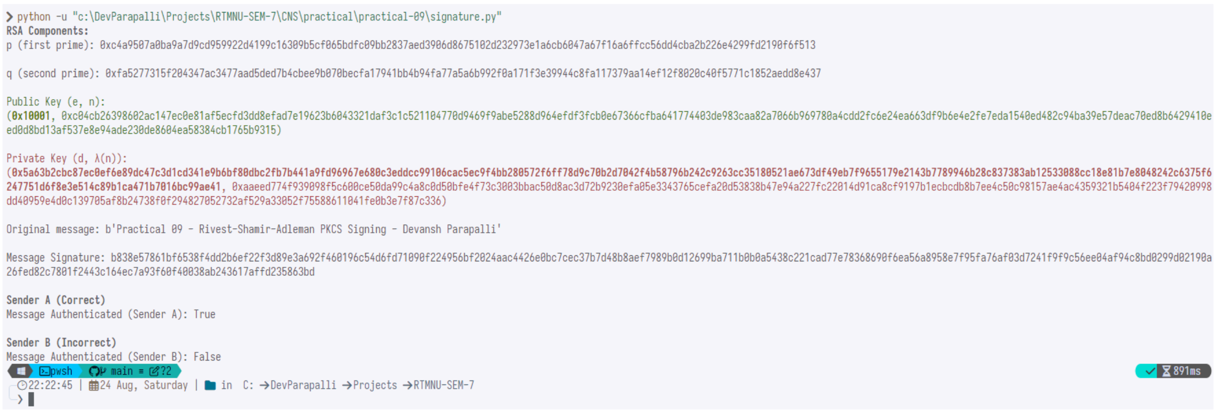
Applications

RSA Digital Signature is widely used in various applications, including:

- Secure email (PGP, S/MIME)
- Code signing
- SSL/TLS certificates
- Blockchain transactions

Listing 1: signature.py

```python
import random
import os

def miller_rabin(n, k=5):
    if n <= 1 or n == 4:
        return False
    if n <= 3:
        return True

    # Find r and s
    s = 0
    d = n - 1
    while d % 2 == 0:
        s += 1
        d //= 2

    # Witness loop
    for _ in range(k):
        a = random.randrange(2, n - 1)
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            continue
        for _ in range(s - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False
    return True

def generate_prime(bits=512):
    while True:
        n = random.getrandbits(bits)
        # Ensure n is odd
        n |= (1 << bits - 1) | 1
        if miller_rabin(n):
            return n

def greatest_common_divisor(a: int, b: int):
    if b == 0:
        return a
    else:
        return greatest_common_divisor(b, a % b)

def least_common_multiple(a: int, b: int):
    # LCM using the Euclidean Algorithm
    gcd = greatest_common_divisor(a, b)
    return abs(a * b) // gcd


class RSA:
    def __init__(self):
        self.p = generate_prime()
        self.q = generate_prime()

        # Compute n = pq
        self.n = self.p * self.q

        # Carmichael's totient function (lambda(n))
        self.lambda_n = least_common_multiple(self.p - 1, self.q - 1)
        # IMP: Secret

        # Choose e such that 1 < e < lambda(n) and
        # gcd(e, lambda(n)) == 1, that is e and lambda(n) are co-prime
        # for i in range(lambda_n, 1, -1):
        #     if greatest_common_divisor(i, lambda_n) == 1:
        #         self.e = i
        # Most common value for e is 65537 (2**16 + 1)
        self.e = 65537

        # MMI(of x in base p) => pow(x, -1, p)
        self.d = pow(self.e, -1, self.lambda_n)
        # IMP: Secret

        self.private_key = (self.d, self.lambda_n)
        self.public_key = (self.e, self.n)

    @staticmethod
    def pad_pkcs1(unpadded: bytes, n: int) -> bytes:
        """pads a RSA string using PKCS1 v1.5 padding scheme

        Args:
            unpadded (bytes): Unpadded Plaintext (M)
            n (int): the modulus, from the public key

        Returns:
            bytes: padded plaintext (m)
        """
        mod_len = (n.bit_length() + 7) // 8 # Round up to next byte
        msg_len = len(unpadded)

        # Padding String must be atleast 8 bytes
        if msg_len > mod_len - 11:
            raise ValueError("[PKCS1 v1.5] PaddingError: Message too long.")

        padding_length = mod_len - msg_len - 3
        padding_string = b""
        while len(padding_string) < padding_length:
            padding_byte = os.urandom(1)
            if padding_byte != b"\x00":
                padding_string += padding_byte

        #            00||02  ||       PS      ||    00  ||M
        padded = b"\x00\x02" + padding_string + b"\x00" + unpadded

        return padded

    @staticmethod
    def unpad_pkcs1(padded: bytes) -> bytes:
        if len(padded) < 11:
            raise ValueError("[PKCS1 v1.5] PaddingError: Padded message too short.")

        if padded[:2] != b"\x00\x02":
            raise ValueError("[PKCS1 v1.5] PaddingError: Incorrect padding format.")

        sep_idx = padded.find(b'\x00', 2)
        if sep_idx == -1:
            raise ValueError("[PKCS1 v1.5] PaddingError: Separator not found.")

        return padded[sep_idx+1:]

    def sign(self, message: bytes) -> bytes:
        """
        Sign a message using the private key.

        Args:
            message (bytes): The message to be signed.

        Returns:
            bytes: The signature.
        """
        # Hash the message
        # message = hashlib.sha256(message).digest()

        # Pad the hash (message)
        padded = self.pad_pkcs1(message, self.n)

        # Convert to integer
        m = int.from_bytes(padded, "big")

        # Sign (encrypt with private key)
        s = pow(m, self.d, self.n)

        return s.to_bytes((self.n.bit_length() + 7) // 8, "big")

    @staticmethod
    def verify(message: bytes, signature: bytes, public_key: tuple[int, int]) -> bool:
        """
        Verify a signature using the public key.

        Args:
            message (bytes): The original message.
            signature (bytes): The signature to verify.
            public_key (tuple[int, int]): The public key (e, n).

        Returns:
            bool: True if the signature is valid, False otherwise.
        """
        e, n = public_key

        # Convert signature to integer
        s = int.from_bytes(signature, "big")

        # Verify (decrypt with public key)
        m = pow(s, e, n)
        padded = m.to_bytes((n.bit_length() + 7) // 8, "big")

        try:
            _message = RSA.unpad_pkcs1(padded)
        except ValueError:
            return False

        # Compare with the hash of the original message
        return _message == message


class ANSI:
    RED = "\033[0;31m"
    GREEN = "\033[0;32m"
    BLUE = "\033[0;34m"

    BOLD = "\033[1m"

    END = "\033[0m"


if __name__ == "__main__":
    RSA_A = RSA()

    print(f"{ANSI.BOLD}RSA Components:{ANSI.END}")
    print(f"p (first prime): {hex(RSA_A.p)}")
    print(f"\nq (second prime): {hex(RSA_A.q)}")

    print(f"\n{ANSI.GREEN}Public Key (e, n):")
    print(f"({ANSI.BOLD}{hex(RSA_A.e)}{ANSI.END}{ANSI.GREEN}, {hex(RSA_A.n)}){ANSI.END}")
    print(f"\n{ANSI.RED}Private Key (d, λ(n)):")
    print(f"({ANSI.BOLD}{hex(RSA_A.d)}{ANSI.END}{ANSI.RED}, {hex(RSA_A.lambda_n)}){ANSI.END}")

    message = b"Practical 09 - Rivest-Shamir-Adleman PKCS Signing - Devansh Parapalli"
    print(f"\nOriginal message: {message}")

    signature = RSA_A.sign(message)
    print(f"\nMessage Signature: {signature.hex()}")
print(f"\n{ANSI.BOLD}Sender A (Correct){ANSI.END}")
verification = RSA.verify(message, signature, RSA_A.public_key)
print(f"Message Authenticated (Sender A): {verification}")

RSA_B = RSA()

print(f"\n{ANSI.BOLD}Sender B (Incorrect){ANSI.END}")

verify_b = RSA.verify(message, signature, RSA_B.public_key)
print(f"Message Authenticated (Sender B): {verify_b}")
```



```
> python -u "c:\DevParapalli\Projects\RTMNU-SEM-7\CNS\practical\practical-09\signature.py"
RSA Components:
p (first prime): 0xc4a9507a0ba9a7d9cd959922d4199c16309b5cf065bdfc09bb2837aed3906d8675102d232973e1a6cb6047a67f16a6ffcc56dd4cba2b226e4299fd2190f6f513

q (second prime): 0xfa5277315f204347ac3477aad5ded7b4cbee9b070becfa17941bb4b94fa77a5a6b992f0a171f3e39944c8fa117379aa14ef12f8020c40f5771c1852aedd8e437

Public Key (e, n):
(0x10001, 0xc04cb26398602ac147ec0e81af5ecfd3dd8efad7e19623b6043321daf3c1c521104770d9469f9abe5288d964efdf3fcb0e67366cfba641774403de983caa82a7066b969780a4cdd2fc6e24ea663df9b6e4e2fe7eda1540ed482c94ba39e57deac70ed8b6429410e
ed0d8bd13af537e8e94ade230de8604ea58384cb1765b9315)

Private Key (d, λ(n)):
(0x5a63b2cbc87ec0ef6e89dc47c3d1cd341e9b6bf80dbc2fb7b441a9fd96967e680c3eddcc99106cac5ec9f4bb280572f6ff78d9c70b2d7042f4b58796b242c9263cc35180521ae673df49eb7f9655179e2143b7789946b28c837383ab12533088cc18e81b7e8048242c6375f6
247751d6f8e3e514c89b1ca471b7016bc99ae41, 0xaaeed774f939098f5c600ce50da99c4a8c0d50bfe4f73c3003bbac50d8ac3d72b9230efa05e3343765cefa20d53838b47e94a227fc22014d91ca8cf9197b1ecbcdb8b7ee4c50c98157ae4ac4359321b5404f223f79420998
dd40959e4d0c139705af8b24738f0f294827052732af529a33052f75588611041fe0b3e7f87c336)

Original message: b'Practical 09 - Rivest-Shamir-Adleman PKCS Signing - Devansh Parapalli'

Message Signature: b838e57861bf6538f4dd2b6ef22f3d89e3a692f460196c54d6fd71090f224956bf2024aac4426e0bc7cec37b7d48b8aef7989b0d12699ba711b0b0a5438c221cad77e78368690f6ea56a8958e7f95fa76af03d7241f9f9c56ee04af94c8bd0299d02190a
26fed82c7801f2443c164ec7a93f60f40038ab243617affd235863bd

Sender A (Correct)
Message Authenticated (Sender A): True

Sender B (Incorrect)
Message Authenticated (Sender B): False
 pwsh  main ≡ ?2                                                                                                    ✓ 891ms
 22:22:45 | 24 Aug, Saturday |  in  C: →DevParapalli →Projects →RTMNU-SEM-7
 ❯ ▌
```

Figure 1: Output