RSA Digital Signature

Project report

Author : Wiktor Łazarski

Index number : 281875

Field of study : Computer Science

Faculty : Electronics and Information Technology, Warsaw University of Technology

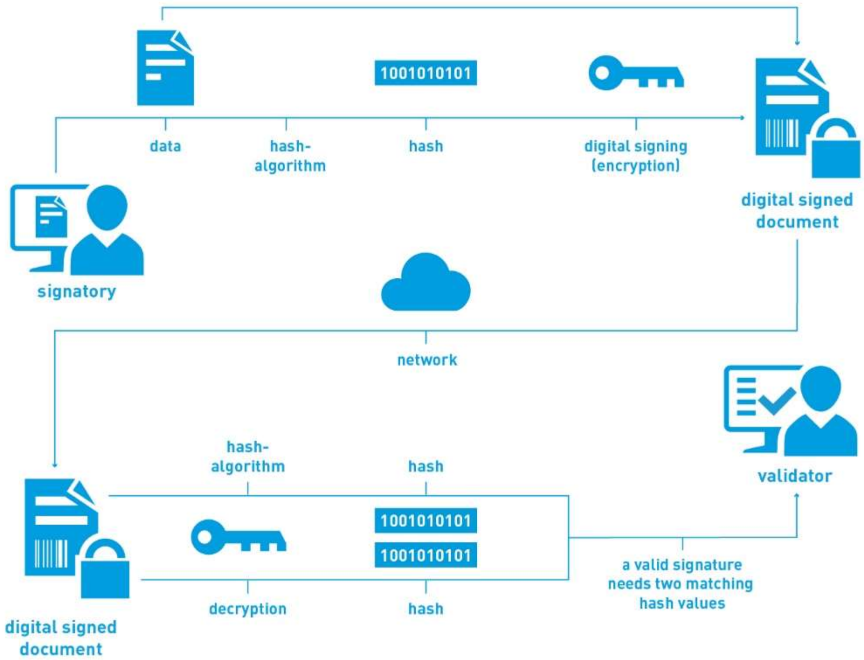
Table of content

1.1 Digital signature.

Digital signatures, like handwritten signatures, are unique to each signer. Digital signature solution providers, follow a specific protocol, called PKI. PKI requires the provider to use a mathematical algorithm to generate two long numbers, called keys. One key is public, and one key is private.

When a signer electronically signs a document, the signature is created using the signer's private key, which is always securely kept by the signer. The mathematical algorithm acts like a cipher, creating data matching the signed document, called a hash, and encrypting that data. The resulting encrypted data is the digital signature. The signature is also marked with the time that the document was signed. If the document changes after signing, the digital signature is invalidated.

As an example, Jane signs an agreement to sell a timeshare using her private key. The buyer receives the document. The buyer who receives the document also receives a copy of Jane's public key. If the public key can't decrypt the signature (via the cipher from which the keys were created), it means the signature isn't Jane's, or has been changed since it was signed. The signature is then considered invalid.



https://easy-software.com/en/newsroom/the-digital-signature-your-electronic-signature/

1.2 RSA cipher.

RSA (Rivest–Shamir–Adleman) is an algorithm used by modern computers to encrypt and decrypt messages. It is an asymmetric cryptographic algorithm. Asymmetric means that there are two different keys. This is also called public key cryptography, because one of the keys can be given to anyone. The other key must be kept private. The algorithm is based on the fact that finding the factors of a large composite number is difficult: when the factors are prime numbers, the problem is called prime factorization. It is also a key pair (public and private key) generator.

Generating keys

RSA involves a public key and private key. The public key can be known to everyone; it is used to encrypt messages. Messages encrypted using the public key can only be decrypted with the private key. The keys for the RSA algorithm are generated the following way:

1. Choose two different large random prime numbers p and q.
2. Calculate $n = pq$. (n – modulus for the public and the private keys)
3. Calculate the totient $\phi(n) = (p - 1)(q - 1)$.
4. Choose an integer e such that $1 < e < \phi(n)$ and e is a co-prime to $\phi(n)$. (e – public key)
5. Compute d to satisfy the congruence relation $de \equiv 1 \ (mod \ \phi(n))$. (d – private key)

Encrypting message

Alice gives her public key (n & e) to Bob and keeps her private key secret. Bob wants to send message m to Alice. First he turns m into a number smaller than n by using some padding scheme. He then computes the cryptogram c using formula :

$$c = m^e \ (mod \ n)$$

Bob then sends c to Alice.

Decrypting message

Alice can recover m from c by using her private key d in the following procedure:

$$m = c^d \ (mod \ n)$$

1.3 RSA digital signature.

RSA digital signature is an algorithm of generating digital signature using RSA cipher approach. Idea behind digital signature remains the same and to encrypt and decrypt hash we use RSA cipher.

<u>RSA signature generation and verification</u>

Alice signs a message m belonging to M. Bob can verify Alice's signature and recover the message m from the signature.

1. *Signature generation*. Alice should do the following :
   - *(a)* Compute $\dot{m} = R(m)$, an integer in the range $[0, n - 1]$. (R – is a chosen hash function)
   - *(b)* Compute $s = \dot{m}^d \ (mod\ n)$.
   - *(c)* Alice's signature for m is s.

2. *Verification*. To verify Alice's signature and recover the message m, Bob should:
   - *(a)* Obtain Alice's authentic public key (n, e).
   - *(b)* Compute $\dot{m} = s^e \ (mod\ n)$.
   - *(c)* Verify that $\dot{m}$ belongs to M_R; if not reject the signature.
   - *(d)* Recover $m = R^{-1}(\dot{m})$.

2.1 *RSAKeys* class.

RSAKeys class was implemented using Python 3.8 for Windows. *RSAKeys* is a class that can be used both to generate and manipulate prime numbers which can be used to perform *RSA* encryption and decryption.

Member types

Member type	Definition
`self.e`	Public key
`self.d`	Private key
`self.n`	Quotient of two primes number (*p, q*) used to generate public and private key.

Member functions

Static member function :

Key generator
`@staticmethod` `def generate(bits=5)`
Parameters : bits – specify bit length of a keys
Returns : RSAKeys object with specified *e, d, n*
Notes : Generates public and private key as well as n = p * q value used in cipher.

List primes
`@staticmethod` `def list_primes(lo, hi)`
Parameters : lo – lower boundary of search space hi – higher boundary of search space
Returns : List object containing prime numbers between *[lo, hi)*
Notes : Returns a list of prime numbers in a given range *[lo, hi)*.

Initialize method:

2.1 *RSAKeys* class.

__init__
`def __init__(self, e, d, n)`
Parameters : e – public key d – private key n – quotient of two primes number (*p, q*) used to generate keys
Returns : *RSAKeys* object

2.2 *RSASignature class*

RSASignature class was implemented using Python 3.8 for Windows. *RSASignature* is a class that can be used both to generate RSA digital signature both from a plain text message and already created hash. Class also provide method to decrypt hash.

Member types

Member type	Definition
`self.keys`	*RSAKeys* or any other class that provides similar functionality.
`self.hash_fun`	Hash algorithm that can be used both to encrypt and decrypt signature.

Member functions

Initialize method:

__init__
`def __init__(self, **params)`
Parameters : params[‘rsa_keys’] – *RSAKeys* or any other class similar one params[‘hash_fun’] – hash algorithm
Returns : *RSASignature* object

Encryption methods

Encrypt message
`def encrypt_message(self, message)`
Parameters : message – plain text message that will be converted into digital signature
Returns : encrypted hash / RSA digital signature
Notes : Produces signature from a given message. Raises an exception if hash is greater than *n*. **Beware** : method calls *update* function of hash function from *hashlib* hence it is highly recommended to change *hash_fun* after every call to avoid unproper results.

Encrypt hash
`def encrypt_hash(self, hash)`
Parameters : hash – already hashed plain text message that will be converted into signature
Returns : encrypted hash / RSA digital signature
Notes : Produces signature from a given hash. Raises an exception if hash is greater that *n*.

Decryption method

Decrypt
`def decrypt(self, cryptogram)`
Parameters : cryptogram – cryptogram / RSA digital signature that will be converted into hash
Returns : decrypted hash
Notes : Performes decryption of a cryptogram (RSA digital signature)

Beware that ones used *encrypt_message* cannot be repeated due to the fact that

3. Testing approach

All tests are implemented and performed in *test_rsa_keys.py* and *test_rsa_signature.py* files. Framework used for testing is standard Python *unittest* library which allows to write efficient unit test.

The following test cases where performed :

1) Checking if *encrypt_message* raise an exception if hash is bigger than *n*.
2) Checking if *encrypt_hash* raise an exception if hash is bigger than *n*.
3) Checking if the same hash was returned after encryption – decryption process for different stings and string lengths.
4) Checking if *RSAKeys* returns valid RSA keys by implementing simple RSA ciphering.

4. References

"*Handbook of Applied Cryptography*" by A.Menez, P. van Oorschot, Scott A. Vanstone

https://www.docusign.com/how-it-works/electronic-signature/digital-signature/digital-signature-faq

https://simple.wikipedia.org/wiki/RSA_algorithm

https://easy-software.com/en/newsroom/the-digital-signature-your-electronic-signature/

5. Source code – RSAKeys class

```python
import random
from random import randint
from math import gcd

class RSAKeys:
    '''Class produces and represents keys for RSA algorithm.'''

    @staticmethod
    def generate(bits=5):
        '''Generates public and private key as well as n = p * q value used in cipher.

            Parameters:
            bits (int): Specifies the number of bits used to create e, d, n

            Returns:
            int: e - public key
            int: d - private key
            int: n - quotient of two prime numbers used to produce keys
        '''
        primes = RSAKeys.list_primes(2 ** (bits - 1) + 1, 2 ** bits)

        p, q = random.sample(primes, 2)
        n = p * q
        phi_n = (p - 1) * (q - 1)

        primes.remove(p)
        primes.remove(q)
        primes = [p for p in primes if gcd(p, phi_n) == 1]

        e = primes[randint(0, len(primes) - 1)]
        d = e
        while (e * d) % phi_n != 1: d = d + 1

        return RSAKeys(e, d, n)

    @staticmethod
    def list_primes(lo, hi):
        '''Returns a list of prime numbers in a given range [lo, hi).

            Parameters:
            lo (int): lower boundary of search space
            hi (int): higher boundary of search space

            Returns:
            []: list of prime numbers between [lo, hi)
        '''
        primes = []
        for num in range(lo, hi, 2):
            for i in range(2, num):
                if num % i == 0:
                    break
            else:
                primes.append(num)

        return primes

    def __init__(self, e, d, n):
        self.e, self.d, self.n = e, d, n

    @property
    def e(self):
        return self.__e

    @e.setter
    def e(self, e):
        self.__e = e

    @property
    def d(self):
        return self.__d

    @d.setter
    def d(self, d):
        self.__d = d

    @property
    def n(self):
        return self.__n

    @n.setter
    def n(self, n):
        self.__n = n
```

5. Source code – RSAKeys tests

```python
import unittest
from rsa_keys import RSAKeys

class TestRSAKeys(unittest.TestCase):

    characters = ['W', 'i', 'K', 't', 'O', 'r']

    def test_keys_generation(self):
        # Test generate method by performing simple RSA encryption/decryption process
        keys = RSAKeys.generate(bits=8)

        for char in self.characters:
            char = ord(char)
            c = pow(char, keys.e, keys.n)
            m = pow(c, keys.d, keys.n)

            self.assertEqual(char, m)
```

5. Source code – RSASignature class

```python
from hashlib import sha256
from Crypto.PublicKey import RSA
from rsa_keys import RSAKeys

class RSASignature:
    '''Class produce and represents RSA digital signature.'''

    def __init__(self, **params):
       self.keys = params['rsa_keys']
       self.hash_fun = params['hash_fun']

    def encrypt_message(self, message):
        '''Produces signature from a given message.

            Parameters:
            message (str): plain text message that will be converted into signature.

            Returns:
            int: encrypted hash / RSA digital signature
        '''
        self.hash_fun.update(message.encode())
        hash = int.from_bytes(self.hash_fun.digest(), byteorder='big')

        if hash >= self.keys.n:
            raise ValueError("RSASignature.encrypt_message : hash greater or equal N")

        signature = pow(hash, self.keys.d, self.keys.n)
        return signature

    def encrypt_hash(self, hash):
        '''Produces signature from a given hash.

            Parameters:
            hash (int): already hashed plain text message that will be converted into signature.

            Returns:
            int: encrypted hash / RSA digital signature
        '''
        if hash >= self.keys.n:
            raise ValueError("RSASignature.encrypt_hash : hash greater or equal N")

        signature = pow(hash, self.keys.d, self.keys.n)
        return signature

    def decrypt(self, cryptogram):
        '''Performes decryption of a cryptogram(RSA digital signature).

            Parameters:
            cryptogram (str): cryptogram / RSA digital signature that will be converted into hash.

            Returns:
            int: decrypted hash
        '''
        hash = pow(cryptogram, self.keys.e, self.keys.n)
        return hash

    @property
    def hash_fun(self):
        return self.__hash_fun

    @hash_fun.setter
    def hash_fun(self, hash_fun):
        self.__hash_fun = hash_fun
```

5. Source code – RSASignature tests

```python
import unittest
from hashlib import sha256
from rsa_signature import RSASignature
from rsa_keys import RSAKeys
from Crypto.PublicKey import RSA

class TestRSASignature(unittest.TestCase):

    messages = ['ECRYP', 'lorem ipsum', 'wiktor', 'cryptography']
    hashes = [2 ** 5, 2 ** 6, 2 ** 10, 2 ** 14]

    def test_encrypt_msg(self):
        # Test hashes equals after ecryption-decryption process
        signature = RSASignature(rsa_keys=RSA.generate(1024), hash_fun=sha256())

        for msg in self.messages:
            hash_fun = sha256()
            signature.hash_fun = hash_fun.copy()

            hash_fun.update(msg.encode())
            before = int.from_bytes(hash_fun.digest(), byteorder='big')

            after = signature.decrypt(signature.encrypt_message(msg))

            self.assertEqual(before, after)

        # Test if encrypt_message method raise exception when hash >= n
        signature = RSASignature(rsa_keys=RSAKeys.generate(bits=8), hash_fun=sha256())

        self.assertRaises(ValueError, signature.encrypt_message, self.messages[0])

    def test_encrypt_hash(self):
        # Test hashes equals after ecryption-decryption process
        signature = RSASignature(rsa_keys=RSAKeys.generate(bits=8), hash_fun=sha256())

        for hash in self.hashes:
            after = signature.decrypt(signature.encrypt_hash(hash))

            self.assertEqual(hash, after)

        # Test if encrypt_message method raise exception when hash >= n
        signature = RSASignature(rsa_keys=RSAKeys.generate(bits=8), hash_fun=sha256())

        self.assertRaises(ValueError, signature.encrypt_hash, 2 ** 20)
```