RSA digital signature – documentation

Maciej Marcinkiewicz, Katarzyna Bielecka

20th January 2022

1 Description of the used algorithm

RSA (Rivest-Shamir-Adleman) is a public key cryptosystem, which was invented in 1977. It is not the newest one but it is still commonly used for securing data transmisson. It can be used both for encryption as well as for digital signatures. In general, it makes use of the integer factorization problem, which in number theory is the decompositio of a composite number into a product of smaller integers. In case of these integers being prime it is then called prime factorization.

2 Functional description of the application

The RSA cryptosystem application provides the following functionalities: public and private key generation, digital signature generation and signature verification. Below, algorithms needed for each of those functionalities are described.

2.1 Keys generation

1. Generate two random, large prime numbers **p** and **q**
2. Compute $n = p * q$ and $\phi = (p - 1) * (q - 1)$
3. Find **e**, such that $1 < e < \phi$ and $\gcd(e, \phi) = 1$
4. Using the extended Euclidean algorithm produce a unique **d**, such that $1 < d < \phi$ and $e * d \equiv 1 (\text{mod } \phi)$
5. (n,e) is the resulting public key and d is the private key

2.2 Signature generation

1. Compute $h = sha512(m)$ where **h** is the hash of the message **m**
2. Compute $s = h^d \pmod{n}$
3. **s** is the generated signature

2.3 Signature verification

1. Compute $\tilde{h} = s^e \pmod{n}$
2. Compute $h =$sha512 (m) where **h** is the hash of the message **m**
3. If $\tilde{h}$ is equal to h then the signature is valid

3 Description of designed code structure

3.1 Miller-Rabin primality test

3.1.1 Code

```python
def is_prime(n: int, k: int) -> bool:
"""
The function implements the Miller-Rabin primality test.

Input:
n - the number to be tested
k - number of tests to be performed

Output: a boolean value
"""

# trivial cases: 0-2 and even numbers
if n == 2:
    return True
elif n <= 1 or n % 2 == 0:
    return False

# writing n as 2^r * d + 1 with d odd
r = 0
d = n - 1
while d % 2 == 0:
    d //= 2
    r += 1

# perform k number of tests
for _ in range(k):
    a = randrange(2, n - 2)
    x = pow(a, d, n)

    if x == 1 or x == n - 1:
        continue

    for _ in range(r - 1):
        x = pow(x, 2, n)
        if x == n - 1:
            break
    else:
        return False

return True
```

3.1.2 Description

The first key component of the program (and of the most of cryptographic software) is primality checker. As RSA requires two distinct large prime numbers there is a need for an efficient algorithm. Simple iterating through every number up to the half of tested number and checking remainder of division operation would not be very effective.

Miller-Rabin primality test perfectly fits to this problem. It is a probabilistic algorithm, so for certain numbers it could give wrong results, however the algorithm is executed in several rounds. Each of them increases certainty of the test's result.

In the beginning function rejects trivial cases of numbers – the first three of natural numbers and even numbers. It is obvious that they are not prime thus in a lot of cases function can finish its job earlier. Tested numbers are randomly generated, it means that in c.a. half of given cases function will finish job quickly.

Then the tested number has to be transformed into $2^r \cdot d + 1$ form. After that step test is performed k-times.

3.2 Large prime number generation

3.2.1 Code

```
def generate_prime_number(prime_size:int) -> int:
"""
Generates large prime numbers.

Input: prime_size - size of prime number expressed in number of bits

Output: p - a large prime number
"""

p = 0

# choose randomly a large number until prime is obtained
while not is_prime(p, 180):
    p = getrandbits(prime_size)

return p
```

3.2.2 Description

Function is mostly based on primality test. It takes randomly generated number and pass it to the Miller-Rabin test function. Numbers are generated by function from the standard Python module – random. Size of those numbers is given in bits to easily control the size in bits of generated keys. If the test is passed, value true is returned. If it is not, the process is being repeated until prime number is obtained.

3.3 Extended Euclidean algorithm

3.3.1 Code

```
def extended_euclidean(a: int, b: int) -> tuple[int, int, int]:
"""
Computes greatest common divisor and the coefficients of Bézout's identity.

Input: a, b - non-negative integers satisfying a >= b

Output: gcd  - greatest common divisor
        x, y - the coefficients of Bézout's identity
"""

if b == 0:
    return (a, 1, 0)

x1, x2, y1, y2 = 0, 1, 1, 0
while b > 0:
```

```
        q, r = divmod(a, b)
        x = x2 - q * x1
        y = y2 - q * y1

        a, b, x2, x1, y2, y1 = b, r, x1, x, y1, y

    gcd, x, y = a, x2, y2
    return (gcd, x, y)
```

3.3.2 Description

The second algorithm that is needed to be implemented before the process of RSA keys generation is extended Euclidean algorithm. What makes this version of algorithm special is that it not only computes the greatest common divisor of two numbers. It also yields two coefficients of of Bézout's identity $ax + by = gcd(a, b)$. One of this coefficient is modular multiplicative inverse and it makes the extended Euclidean algorithm one of the easiest methods of obtaining this inverse.

3.4 RSA keys generation

3.4.1 Code

```
def generate_keys(key_size:int = 2048, return_primes: bool = False)
     -> Union[tuple[int, int, int], tuple[int, int, int, int, int]]:
    """
    Generates RSA public and private keys.

    Input: key_size - size of the key expressed in number of bits (2048 bits by default)
           return_prime - if set, function returns additionaly tuple of prime numbers
           p and q used in key generation

    Output: (n, e) - public key
                 d - private key
            (p, q) - primes used in generation (optional)
    """

    # generate two large random primes
    p = generate_prime_number(key_size // 2)
    q = generate_prime_number(key_size // 2)

    n = p * q
    fi = (p - 1) * (q - 1)

    # find such e, that e and fi are coprimes
    e = 0
    while gcd(e, fi) != 1:
        e = randrange(1 + 1, fi - 1)

    # find modular multiplicative inverse
    _, x, _ = extended_euclidean(e, fi)
    d = x + fi

    # return public and private keys (optionally primes p and q)
    if return_primes:
        return ((n, e), d, (p, q))
    else:
        return ((n, e), d)
```

3.4.2 Description

This function generates public and private keys, optionally returns also a tuple of prime numbers that have been used for generation process.

It starts with generating those two primes. Size of the key is specified thus primes have to be half of the key's size because they are multiplied one by another in the next step.

After n and ϕ are calculated the procesess of finding e starts. According to the theory e just has to be a coprime with ϕ. However in real life cases usually a Fermat primes are being used. Early implementations of RSA were vulnerable to small e exponents and one of the most common choice for that exponent is 65537 – the largest known Fermat prime.

The last step is to find modular multiplicative inverse of e and ϕ which is finally a d private exponent. Note that ϕ has to be added to the Bézout coefficient in order to get proper inverse. Non-negative number is preferred and both x and x + ϕ has the same remainder after division operation, so it is allowed to apply such addition.

3.5 Signature generation

3.5.1 Code

```
def generate_signature(m, n: int, e: int, d: int) -> int:
"""
The function generates an RSA digital signature.

Input:
m - message
n, e - public key of sender
d - private key of sender

Output:
a digital signature s

"""
# compute hash of message
h = int(sha512(m.encode("utf-8")).hexdigest(), 16) % 10 ** 8

# compute signature and return it
s = pow(h, d, n)
return s
```

3.5.2 Description

In order to generate the signature for a given message, function just have to compute remainder. Before it, hash of the message has to be generated (SHA-512 in this case). Hash in form of an integer is raised to the power of private exponent d. Such power is then divided by n and remainder of this operation is the signature.

It is very theoretical implementation of the signing, in practise signatures are padded to make them less vulnerable to existential forgery attacks. Commonly used formats is PKCS #1 (part of standards defined in Public-Key Cryptography Standards). Such method takes the hash of the message and an identifier of the hash function, converts to the ASN.1 form, codes them with BER rules. Then it is formatted in a different way and the octets of that data are converted into one integer. As this process is not the purpose of the project the way data formatting will not be described in this document.

3.6 Signature verification

3.6.1 Code

```
def verify(m, n: int, e: int, s: int) -> bool:
    """
    The function generates verifies the received signature using public key.

    Input:
    (n,e) - public key of sender
    s - digital signature

    Output:
    a boolean value

    """
    # decrypt the message
    h_ = pow(s, e, n)

    # calculate message hash
    h = int(sha512(m.encode("utf-8")).hexdigest(), 16) % 10 ** 8

    # compare and return
    return h_ == h
```

3.6.2 Description

Process of verification is similar to the signature generation. Verification is passed when decrypted message's hash is the same as newly generated. In real-life example it would be also implemented for instance in PKCS #1 way.

4 Tests

5 Bibliography

1. Menezes, Alfred; van Oorschot, Paul C.; Vanstone, Scott A. (October 1996). Handbook of Applied Cryptography, Chapter 8

2. https://en.wikipedia.org/wiki/RSA_(cryptosystem)