Warsaw University of Technology

Faculty of Electronics and Information Technology

Dimitrios Georgousis, K-7729

Cryptography

Project

Topic: V.5 (Rabin digital signature)

Original topic was V.11 (One time Rabin signature). The topic was changed to V.5 upon agreement with the instructor.

A large part of this project's documentation also exists in the `README.md` file of our code files and in comments in our code.

Introduction

This Python project implements the Rabin Signature algorithm, a method used in cryptography for signing and verifying messages.

Functions

```python
# Wikipedia algorithm for extended Euclidean algorithm
def _extended_gcd(a, b):
    old_r, r = a, b
    old_s, s = 1, 0
    old_t, t = 0, 1

    while r != 0:
        quotient = old_r // r
        old_r, r = r, old_r - quotient * r
        old_s, s = s, old_s - quotient * s
        old_t, t = t, old_t - quotient * t

    return old_r, old_s, old_t
```

```python
def _jacobi_symbol(a, n):
    if n < 0 or not n % 2:
        raise ValueError("n should be an odd positive integer")
    if a < 0 or a > n:
        a %= n
    # a is formatted as 0 <= a < n and n is an odd positive integer

    if not a:
        return int(n == 1)
    if n == 1 or a == 1:
        return 1

    # if a and n are not coprime then Jacobi(a, n) = 0
    if sympy.igcd(a, n) != 1:
        return 0

    # Initialize the Jacobi symbol
    j = 1
    # Follow the algorithm steps (wikipedia page for Jacobi symbol)
    while a != 0:
        while a % 2 == 0 and a > 0:
            a >>= 1
            if n % 8 in [3, 5]:
                j = -j
        a, n = n, a
        if a % 4 == n % 4 == 3:
            j = -j
        a %= n
    return j


def _generate_rabin_prime(bits):
    while True:
        p = sympy.randprime(2 ** (bits - 1), 2**bits)
        if p % 4 == 3:
            return p


def _hash_function(message):
    return int(hashlib.sha256(message.encode()).hexdigest(), 16)
```

```python
def _random_string(k):
    """
    Generate a random string of printable characters.
    Input: k - the number of bits in the generated string
    Output: a random string of printable characters
    """
    characters = k // 8
    return "".join(random.choices(string.printable, k=characters))
```

Perhaps, we may need to explain why we implemented the prime generator this way.

In this algorithm we use properties of quadratic residues (q.r.) a lot. It is known that a number c is a q.r. modulo p if there exists number x, such that $x^2 \equiv c \ mod \ p$.

If $p \ \equiv 3 \ mod \ 4$ it has been shown that $x \ \equiv c^{p+1/4} \ mod \ p$ is one such solution (Formula 1). We use such primes to do these types of calculations faster.

```python
def key_generation(bits):
    '''
    Generate a Rabin signature key pair.
    Input: bits - the number of bits in the primes p and q
    Output: n - the public key, (p, q) - the private key
    '''
    p = _generate_rabin_prime(bits)
    q = _generate_rabin_prime(bits)
    while p == q:
        q = _generate_rabin_prime(bits)
    n = p*q

    return n, (p, q)
```

The key generation function doesn't have any particularly outstanding qualities. We show it above.

```python
def sign(message, private_key, k=256):
    """
    Sign a message using the Rabin signature scheme.
    Input: message - the message to sign,
           private_key - the private key,
           k - the number of random bits to use
    Output: (x, u) - the signature
    """
    p, q = private_key
    n = p * q

    while True:
        u = _random_string(k)
        c = _hash_function(message + u) % n

        # check x^2 = c mod n, this will be true iff
        # c is a quadratic residue mod p and mod q.
        if _jacobi_symbol(c, p) != 1 or _jacobi_symbol(c, q) != 1:
            continue

        # find x_p^2 = c mod p and x_q^2 = c mod q
        # solve for x using the Chinese Remainder Theorem
        # x = x_p mod p, x = x_q mod q
        # y_p * p + y_q * q = 1
        # y_p = p^-1 mod q, y_q = q^-1 mod p
        # a = q * y_q, b = p * y_p
        # a = 0 mod q, a = 1 mod p
        # b = 0 mod p, b = 1 mod q
        # x = x_p * a + x_q * b mod n, so
        # x = x_p * q * y_q + x_q * p * y_p mod n
        x_p = pow(c, (p + 1) // 4, p)  # known formula
        x_q = pow(c, (q + 1) // 4, q)  # known formula

        _, y_p, y_q = _extended_gcd(p, q)
        x = (x_p * q * y_q + x_q * p * y_p) % n
        return x, u
```

The idea of the signature algorithm is that finding x such that $x^2 = c \ mod \ n$, when n's factorization is unknown, is equivalent to factorizing n, which is a hard problem. Thus, safety is ensured.

How does our algorithm find such a number though?

The signer knows the private key, which means that the signer knows the factors prime p, q such that n = p*q.

We use the following facts to solve the congruence $x^2 = c \ mod \ n$:

Solve congruencies: $x_p^2 = c \; mod \; p$ and $x_q^2 = c \; mod \; q$ separately and then combine the solutions to solve $x^2 = c \; mod \; n$. To solve the congruencies above we use the fact that p, q were chosen such that $p \equiv q \equiv 3 \; mod \; 4$. We can find x_p, x_q using (Formula 1) from earlier.

The jacobi symbol test we do is to ensure that c will be a q.r. modulo both p and q. The expected number of tries is 4 and each time we try for a different random suffix `u` to our message (in order to get a different result).

We, now, look for a, b such that $\begin{cases} a = 1 \; mod \; p \\ a = 0 \; mod \; q \end{cases}$ and $\begin{cases} b = 0 \; mod \; p \\ b = 1 \; mod \; q \end{cases}$ because $x = ax_p + bx_q \; mod \; n$ is such a solution to our original congruency.

Proof of this inference:

We have $x^2 \equiv c \; mod \; p, x^2 \equiv c \; mod \; q$. Suppose $c = ep + c' = gq + c'', where \; c' = c \; mod \; p, c'' = c \; mod \; q$. We have

$$x^2 = fp + c' = hq + c'' = hq + ep + c' - gp = (h - g)q + ep + c'$$

Thus $(f - e)p = (h - g)q = jpq$ for some j. We can now write:

$$x^2 = fp + c' = (f - e)p + ep + c' = jpg + c$$

But, n = pg, so $x^2 = jn + c$, thus $x^2 \equiv c \; mod \; n$.

To find a, b the Chinese Remainder Theorem (CRT) is used:

The Extended Euclidean Algorithm gives us $y_p, y_q \; such \; that \; y_p p + y_q q = 1$

We can easily verify that $y_p = p^{-1} \; mod \; q, y_q = q^{-1} \; mod \; p$.

The CRT will return $x = ax_p + bx_q \; mod \; n, where \; a = y_q q, b = y_p p$

We see that a, b satisfy the conditions we described previously, thus the calculated x is a solution to $x^2 = c \; mod \; n$.

```python
def verify(message, signature, public_key):
    '''
    Verify a message using the Rabin signature scheme.
    Input: message - the message to verify,
           signature - the signature,
           public_key - the public key
    Output: True if the signature is valid, False otherwise
    '''
    n = public_key
    x, u = signature
    c = _hash_function(message + u) % n
    return pow(x, 2, n) == c
```

Finally, the verification function is shown above. It doesn't have anything noteworthy.

Environment

```
$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.4 LTS
Release:        22.04
Codename:       jammy
$ python3 --version
Python 3.10.12
$ pip show sympy
Name: sympy
Version: 1.12
Summary: Computer algebra system (CAS) in Python
Home-page: https://sympy.org
Author: SymPy development team
Author-email: sympy@googlegroups.com
License: BSD
Location: /home/dimjimitris/.local/lib/python3.10/site-packages
Requires: mpmath
Required-by:
```

As required by the project description our program was tested and works in Python 3.10 environments.

Usage

To use this implementation, follow these steps:

1. Clone the repository
2. Install the required dependencies
3. Run `test.py` to observe results of some testing we did on our algorithms
4. The `rabin_signature.py` file contains the functions used for key generation, signing and verifying. By running this file you can use a simple program which utilizes all our functions with a fixed seed so that results are reproducible.

We show an example:

```
$ python3 rabin_signature.py
Enter a message: hello world
Public key:
6357382886902003328248888279462972226537252741435595111411540660487002133323
3754508499971708426764315102540691267177843227432502862111706660472833604093
8414586317574511699343409919026008174450836492542781124525063542152388909552
4905541607923864493720155938409240533879534101367381670240224079444095738101
1133
Private key:
(74273349260495781781519213155035663499504688072430118917653458248793998040
3280759180412983876431732646305217034785640376905264884469486972739642487986
1391,
8559440162857100680747467757185182334528479657284870922407995852830651814775
1948572024719072052109513660056871381166861016042458962082506700759590711285
63)
Signature:
(6106294445346649558609684551920451553475717469648351676303101099295486302641
7470550878365662189586190013890233989613948616503076637550341163115900108
```

```
1835101208439854659369856005224511885810307088212838848997727829584684805095
49858490785344747733058768725960779169600188115063856533761460920872799686
928512464, 'E,\tXDjpf')
Verification: True
$ python3 rabin_signature.py
Enter a message: hello world
Public key:
635738288690200332824888827946297222653725274143559511141154066048700213332
3375450849997170842676431510254069126717784322743250286211170666047283360409
384145863175745116993434099190260081744508364925427811245250635421523889095
5249055416079238644937201559384092405338795341013673816702402240794440957381011133
Private key:
(7427334926049578178151921315503566349950468807243011891765345824879399804032
807591804129838764317326463052170347856403769052648844694869727396424879861391,
855944016285710068074746775718518233452847965728487092240799585283065181477
5194857202471907205210951366005687138116686101604245896208250670075959071128563)
Signature:
(610629444534664955860968455192045155347571746964835167630310109929548630264
174705508783656621895861900138902339896139486165030766375503411631159001081
835101208439854659369856005224511885810307088212838848997727829584684805095
49858490785344747733058768725960779169600188115063856533761460920872799686
928512464, 'E,\tXDjpf')
Verification: True
```

Rabin Signature API

The `rabin_signature.py` module provides the following functions for key generation, signing, and verifying:

- `key_generation(bits)`: Generates a public-private key pair for Rabin digital signature, the primes used are of length `bits`.
- `sign(message, private_key, k)`: Signs a message using the private key and returns the signature. `k` is a parameter used for specifying the length of a random string appended at the end of the message.
- `verify(message, signature, public_key)`: Verifies the signature of a message using the public key.

To use these functions, import the `rabin_signature` module and call the respective functions.

References

1. Wikipedia
2. Rabin Publication