A black and white logo

Description automatically generated

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

# 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

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)

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 .

If it has been shown that is one such solution (Formula 1). We use such primes to do these types of calculations faster.

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.

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 such that , 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 :

Solve congruencies: and separately and then combine the solutions to solve . To solve the congruencies above we use the fact that were chosen such that . We can find 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 such that and because   
 is such a solution to our original congruency.

Proof of this inference:

We have . Suppose . We have

Thus for some j. We can now write:

But, n = pg, so , thus .

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

The Extended Euclidean Algorithm gives us

We can easily verify that .

The CRT will return

We see that satisfy the conditions we described previously, thus the calculated is a solution to .

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:

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](https://en.wikipedia.org/wiki/Rabin_signature_algorithm)
2. [Rabin Publication](http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-212.pdf)