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. Run `tests\_with\_output.py` to see some simple messages and the outputs of our algorithm.
5. 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. You input a text message and a signature is produced and verified for it.

We show an example run of all our executable files:

$ python3 test.py

..

----------------------------------------------------------------------

Ran 2 tests in 1.553s

OK

$ python3 tests\_with\_output.py

Message: hello world

Public key: 63573828869020033282488882794629722265372527414355951114115406604870021333233754508499971708426764315102540691267177843227432502862111706660472833604093841458631757451169934340991902600817445083649254278112452506354215238890955249055416079238644937201559384092405338795341013673816702402240794440957381011133

Private key: (7427334926049578178151921315503566349950468807243011891765345824879399804032807591804129838764317326463052170347856403769052648844694869727396424879861391, 8559440162857100680747467757185182334528479657284870922407995852830651814775194857202471907205210951366005687138116686101604245896208250670075959071128563)

Square root: 61062944453466495586096845519204515534757174696483516763031010992954863026417470550878365662189586190013890233989613948616503076637550341163115900108183510120843985465936985600522451188581030708821283884899772782958468480509549858490785344747733058768725960779169600188115063856533761460920872799686928512464

Random string: E, XDjpf

Verification: True

--------------------

Message: this is a test

Public key: 63573828869020033282488882794629722265372527414355951114115406604870021333233754508499971708426764315102540691267177843227432502862111706660472833604093841458631757451169934340991902600817445083649254278112452506354215238890955249055416079238644937201559384092405338795341013673816702402240794440957381011133

Private key: (7427334926049578178151921315503566349950468807243011891765345824879399804032807591804129838764317326463052170347856403769052648844694869727396424879861391, 8559440162857100680747467757185182334528479657284870922407995852830651814775194857202471907205210951366005687138116686101604245896208250670075959071128563)

Square root: 47018992453649762197107493128989630545029746050794461520995744665223841472219500337068920681882409510808382123154728598468259873887305392702182078065255094147548846011466159738580916020811563542144878585316435631702639207994359394183556843384306671880251595316428442497216650930367098926287508897685958179250

Random string: wV}TZA

.

Verification: True

--------------------

Message: another test

Public key: 63573828869020033282488882794629722265372527414355951114115406604870021333233754508499971708426764315102540691267177843227432502862111706660472833604093841458631757451169934340991902600817445083649254278112452506354215238890955249055416079238644937201559384092405338795341013673816702402240794440957381011133

Private key: (7427334926049578178151921315503566349950468807243011891765345824879399804032807591804129838764317326463052170347856403769052648844694869727396424879861391, 8559440162857100680747467757185182334528479657284870922407995852830651814775194857202471907205210951366005687138116686101604245896208250670075959071128563)

Square root: 38610218700606805874220054475430334722975069714604703562517037347792895572228222072344653120490846949302513250418838970424413614895981241420979838946568206841132848080198820307374143129238622625375569145230820758249746880538045021976992032099786675524795053498574446900639309735230215413015573946240055869609

Random string: DMTPI

.m

Verification: True

--------------------

Message: 1

Public key: 63573828869020033282488882794629722265372527414355951114115406604870021333233754508499971708426764315102540691267177843227432502862111706660472833604093841458631757451169934340991902600817445083649254278112452506354215238890955249055416079238644937201559384092405338795341013673816702402240794440957381011133

Private key: (7427334926049578178151921315503566349950468807243011891765345824879399804032807591804129838764317326463052170347856403769052648844694869727396424879861391, 8559440162857100680747467757185182334528479657284870922407995852830651814775194857202471907205210951366005687138116686101604245896208250670075959071128563)

Square root: 29758435130139242108668165454725192534165802861287734479830651463153556595410656315366382281534715730262860555116719084923553366950604213345929501951299905107807270612918596079743529405173758405806721923208497253821938747721675068576611921782637530889641985968370055510409927797045459281173629523986628333859

Random string: E, XDjpf

Verification: True

--------------------

Message: what about this message

Public key: 63573828869020033282488882794629722265372527414355951114115406604870021333233754508499971708426764315102540691267177843227432502862111706660472833604093841458631757451169934340991902600817445083649254278112452506354215238890955249055416079238644937201559384092405338795341013673816702402240794440957381011133

Private key: (7427334926049578178151921315503566349950468807243011891765345824879399804032807591804129838764317326463052170347856403769052648844694869727396424879861391, 8559440162857100680747467757185182334528479657284870922407995852830651814775194857202471907205210951366005687138116686101604245896208250670075959071128563)

Square root: 55368687330487664878995246806791937784726070908817748923716109754932591571386466621379042778844691583000204286394459612927139946404718597295681160744186346695709530694088032112729526376689991719226065773979593500299607747920856085353358168546922747423579371299744066543590752207070290387571831684851297646710

Random string: E, XDjpf

Verification: True

--------------------

Message: lorem ipsum

Public key: 63573828869020033282488882794629722265372527414355951114115406604870021333233754508499971708426764315102540691267177843227432502862111706660472833604093841458631757451169934340991902600817445083649254278112452506354215238890955249055416079238644937201559384092405338795341013673816702402240794440957381011133

Private key: (7427334926049578178151921315503566349950468807243011891765345824879399804032807591804129838764317326463052170347856403769052648844694869727396424879861391, 8559440162857100680747467757185182334528479657284870922407995852830651814775194857202471907205210951366005687138116686101604245896208250670075959071128563)

Square root: 12516658576225686023627562711605968409764753551369034338408024196143744971596618444651849907844890893838902703975787815051885418559063154328506566930494730098345812620174964029712320246529480927093990951469900990199816036092566784384532607797811752416428754727647741296213930589785314916629122381804027014515

Random string: bl[R=}>4

Verification: True

--------------------

Message: answer to universe and everything

Public key: 63573828869020033282488882794629722265372527414355951114115406604870021333233754508499971708426764315102540691267177843227432502862111706660472833604093841458631757451169934340991902600817445083649254278112452506354215238890955249055416079238644937201559384092405338795341013673816702402240794440957381011133

Private key: (7427334926049578178151921315503566349950468807243011891765345824879399804032807591804129838764317326463052170347856403769052648844694869727396424879861391, 8559440162857100680747467757185182334528479657284870922407995852830651814775194857202471907205210951366005687138116686101604245896208250670075959071128563)

Square root: 17797069185431468878812290995107499555664023913530425101270249859320153089731766656299268934085763265047472078402040161689090000063807343853738441475785913398041749559806196894542631632251336519489351588749295424021322243817293812863589116828479756959290586403070013505004025858625584771848516731113286505770

Random string: }9:@$E78

Verification: True

--------------------

$ python3 rabin\_signature.py

Enter a message: hello world

Public key: 63573828869020033282488882794629722265372527414355951114115406604870021333233754508499971708426764315102540691267177843227432502862111706660472833604093841458631757451169934340991902600817445083649254278112452506354215238890955249055416079238644937201559384092405338795341013673816702402240794440957381011133

Private key: (7427334926049578178151921315503566349950468807243011891765345824879399804032807591804129838764317326463052170347856403769052648844694869727396424879861391, 8559440162857100680747467757185182334528479657284870922407995852830651814775194857202471907205210951366005687138116686101604245896208250670075959071128563)

Square root: 61062944453466495586096845519204515534757174696483516763031010992954863026417470550878365662189586190013890233989613948616503076637550341163115900108183510120843985465936985600522451188581030708821283884899772782958468480509549858490785344747733058768725960779169600188115063856533761460920872799686928512464

Random string: E, XDjpf

Verification: True

$ python3 rabin\_signature.py

Enter a message: hello world

Public key: 63573828869020033282488882794629722265372527414355951114115406604870021333233754508499971708426764315102540691267177843227432502862111706660472833604093841458631757451169934340991902600817445083649254278112452506354215238890955249055416079238644937201559384092405338795341013673816702402240794440957381011133

Private key: (7427334926049578178151921315503566349950468807243011891765345824879399804032807591804129838764317326463052170347856403769052648844694869727396424879861391, 8559440162857100680747467757185182334528479657284870922407995852830651814775194857202471907205210951366005687138116686101604245896208250670075959071128563)

Square root: 61062944453466495586096845519204515534757174696483516763031010992954863026417470550878365662189586190013890233989613948616503076637550341163115900108183510120843985465936985600522451188581030708821283884899772782958468480509549858490785344747733058768725960779169600188115063856533761460920872799686928512464

Random string: E, XDjpf

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. Simply running the `rabin\_signature.py` file allows you to produce a signature, sign and verify a text message taken as input from the console and observe the results of these steps in the standard output.

Libraries

We make use of `sympy` and `hashlib`. `hashlib` is simply used to get a hash function for our messages (which is another project topic thus was not implemented specifically for this project) and `sympy` is used to generate random prime numbers in the key generation part of our algorithm. There are many ways to implement such a generator and some of them have varying complexity, which seems outside the scope of this project. We make use of `sympy`’s `igcd()` function which calculates the gcd of two integers, but we have already demonstrated the gcd algorithm in the `\_extended\_gcd()` function, thus did not reimplement it. We focus only on the Rabin Digital Signature Scheme.

Testing

The Rabin Digital Signature Scheme depends on the random string `u` appended to the `message` and the hash function used. I could not find any test vectors for this thus testing happens in the following way: a key is generated -> message is signed using the key -> message is verified.

Our algorithm can be tested as follows:

- `test.py` file contains a lot of messages (more than 7). We generate a private and public key and test all messages using these keys. We repeat this process 10 times (`test\_sign\_and\_verify()`).

- `tests\_with\_output.py` tests some messages and output the keys, signatures and verification results in the console.

- `rabin\_signature.py` when executed simply takes a message as input from the console. Then creates a private/public key, sings the message and verifies. All this behaviour is tracked on the console output.

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)