todo_project_name

Jakub Kaczor, Wojciech Michalczuk, Łukasz Pawlak

Jan 30, 2023

CONTENTS:

CHAPTER ONE

THE GENERAL IDEA

The purpose of this program is to allow the user to sign, and check RSA signatures of files.

The standard use of this program is as follows: Alice and Bob generate RSA keys. They exchange public keys with each other, and keep private keys for themselves. Once Alice has Bob's public key, she can verify Bob's signatures. If Bob wants to confirm that send message indeed came from him, he can sign it with his private key, and send signature alongside original file. Then Alice can verify if signature matches acquired file, and Bob's public key.

It is extremely unlikely that another public key would match given file and signature. It is hard to generate fake signature to match fixed message and public key as well. Thus, such signature can be a solid proof of identity of sender, as long as his private key stays confidential.

The RSA is based around idea of "factorization of large numbers is hard" and modular arithmetic. Private key is pair (d, n), where d is co-prime with $\phi(n)$, and n is product of 2 large prime numbers p, q. Public key is pair (e, n), where $e \equiv d^{-1} \bmod \phi(n)$. To compute d knowing only e and $n = pq$, one would have to compute $\varphi(pq) = (p-1)(q-1)$, for which factorization of n is needed. This is where security of RSA comes from.

To encrypt message m (by message we mean natural number smaller than n; regular data could be divided into blocks of appropriate size and interpreted as numbers to fit this criteria) we compute $m' = m^e \bmod n$. To encrypt m', we compute $(m')^d \bmod n = m$. This works thanks to identity $a^{ed} \equiv a \bmod n$.

To sign file, we reverse this procedure in a way. First, Bob computes hash of the message to sign $h(m)$. Hashed message should in general be shorter, so that applying RSA is easier and less computationally expensive. Then, Bob computes $h' = h(m)^d \bmod n$. To check signature, Alice computes $(h')^e \bmod n$ using public key, and compares it to $h(m)$. This method allows anyone with access to file and public key to check Bob's identity as the sender.

In our program, we use 2 hashing algorithms: MD4 and MD5. They are both similar in structure. More about them can be found in papers [2] and [1]

BIBLIOGRAPHY

[1] Ronald Rivest. *The MD5 message-digest algorithm*. Tech. rep. 1992.

[2] Ronald L Rivest. *MD4 message digest algorithm*. Tech. rep. 1990.

CHAPTER

TWO

HOW TO USE THE PROGRAM

The program offers 4 core functionalities:

1. Hashing text files and other files, using MD4 or MD5 hash functions;
2. Generating public-private RSA key pairs;
3. Signing text message or file with RSA protocol;
4. Veryfing RSA signature of text message or file.

2.1 Hashing

To hash text message or other file do the following:

1. Choose action *Generate checksum* from drop menu at the top of the window.
2. Click *Change message path...* button and select file you want to hash.
3. Choose hashing function (drop menu next to *algorithm* label).
4. Choose checksum file destination and name by clicking *Change checksum path...* button.
5. Click *Proceed* button.

Hash of selected file should appear under path specified in step 4. This is text file containing message digest encoded as hex string.

2.2 Generating keys

To generate RSA key pair do the following:

1. Choose action *Generate key pair* from drop menu at the top of the window.
2. Click *Change keypair save location...* and pick folder in which keys will be saved. Make sure it is private location, that other users can't access.
3. Fill text field labelled *Base name of generated keys*. In the end, 2 files will be created in folder specified in step 2, with this base name.
4. Fill text field labelled *Key pair id*. This field should contain description of whose key it is, or other identifier.
5. Click *Proceed* button.

Two files should appear in folder specified in step 2. They should have the same base name. One should have extension `.private`. This is the private key, which should not be shared. The other should have extension `.public`. This is the public key, and it should be shared to make verification possible.

2.3 Signing

To sign text message or other file do the following:

1. Choose action *Sign* from drop menu at the top of the window.
2. Click *Change message path...* button and select file you want to sign.
3. Click *Change private key...* button and select private key you want to sign with.
4. Choose signature file destination and name by clicking *Change signature path...* button.
5. Click *Proceed* button.

Signature of selected file should appear under path specified in step 4. This is text file containing number representation of signature.

2.4 Veryfying signature

To verify signature do the following:

1. Choose action *Verify* from drop menu at the top of the window.
2. Click *Change public key...* button and select public key that is supposed to match given signature.
3. Click *Change message path...* button and select file whose signature you are checking.
4. Choose signature file path by clicking *Change signature path...* button.
5. Click *Proceed* button.

Information, about whether the file signature is correct or not should appear in info box. Be careful to pick the right key and signature file each time.

CHAPTER

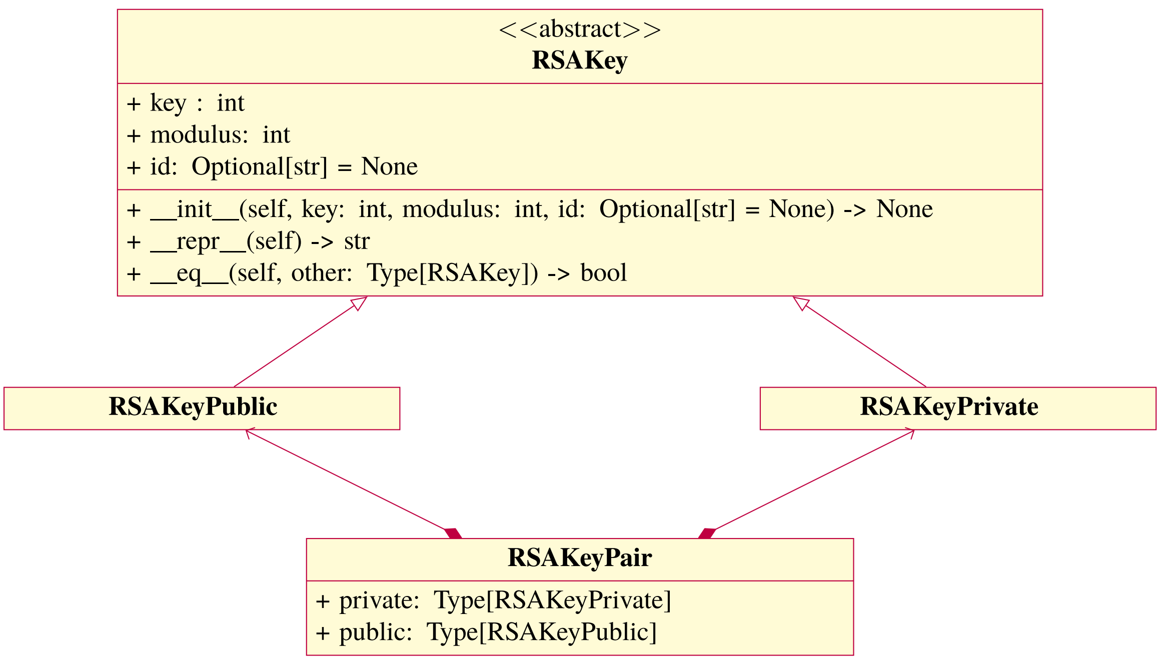
THREE

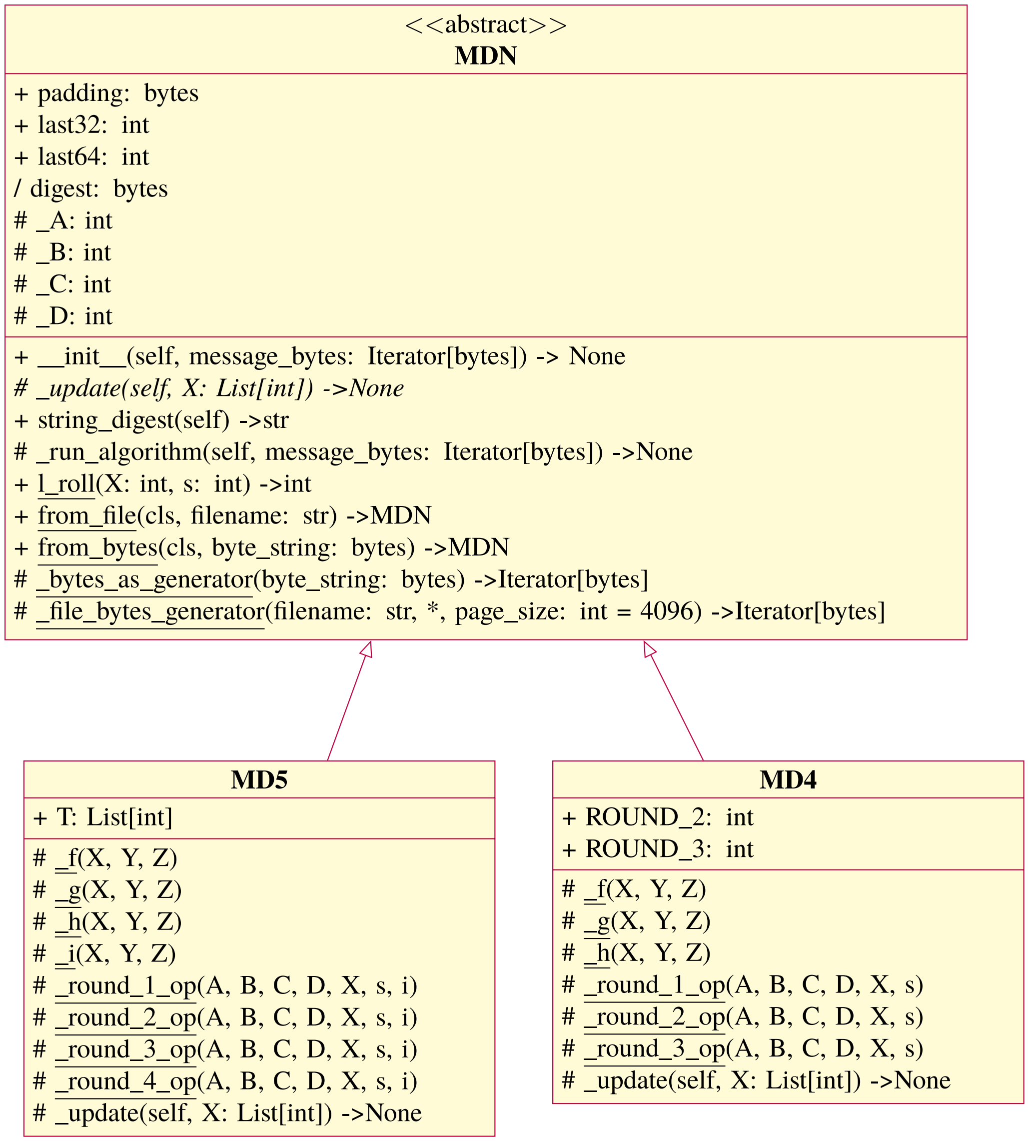
CODE STRUCTURE

3.1 Diagrams

Huge part of this program uses object oriented programming as its paradigm. In such, a class diagrams are standard tools of visualization. Below you can see diagrams for classes RSAKey, its descendants and class RSAKeyPair.

On the next page you can find diagrams for classes MD*, which are responsible for hashing algorithms.





3.2 External packages

In this project, we used external `secrets` library alongside well known `random`. The first one we use for generating pseudo-random numbers in order to find large primes (key generation). The second is used in testing only. See secrets in python documentation.

CHAPTER

FOUR

TODO_PROJECT_NAME PACKAGE

4.1 todo_project_name.core module

`todo_project_name.core.`**`md4_string`**`(`*message: str*`)` → str

Returns md4 digest of given string encoded as UTF-8 byte strings.

4.1.1 Parameters

message : string whose hash is to be computed.

`todo_project_name.core.`**`md5_string`**`(`*message: str*`)` → str

Returns md5 digest of given string encoded as UTF-8 byte strings.

4.1.2 Parameters

message : string whose hash is to be computed.

4.2 todo_project_name.find_prime module

`todo_project_name.find_prime.`**`find_prime`**`(`*n: int*`)` → int

Return *n*-bit probable prime.

4.2.1 Parameters

n : number of bits, must be greater than 1,

because otherwise such a prime doesn't exist.

`todo_project_name.find_prime.`**`is_probable_prime`**`(`*candidate: int*`)` → bool

Check if *candidate* is a probable prime.

4.2.2 Notes

This function uses Rabin-Miller test under the hood.

4.3 todo_project_name.md4 module

class todo_project_name.md4.**MD4**(*message_bytes: Iterator[bytes]*)

Bases: *MDN*

Class computing MD4 message digest. Works for little-endian architecture.

It is recommended to use methods *MD4.from_bytes* or *MD4.from_file* to create new objects.

To get message digest as *str* use *string_digest* method. To get message digest as *bytes* read *digest* property.

ROUND_2 = 1518500249

ROUND_3 = 1859775393

4.4 todo_project_name.md5 module

class todo_project_name.md5.**MD5**(*message_bytes: Iterator[bytes]*)

Bases: *MDN*

Class computing MD5 message digest. Works for little-endian architecture.

It is recommended to use methods *MD5.from_bytes* or *MD5.from_file* to create new objects.

To get message digest as *str* use *string_digest* method. To get message digest as *bytes* read *digest* property.

4.5 todo_project_name.mdn module

class todo_project_name.mdn.**MDN**(*message_bytes: Iterator[bytes]*)

Bases: ABC

Superclass of MD4 and MD5. Works for little-endian architecture.

property digest

The message digest as bytes.

classmethod from_bytes(*byte_string: bytes*) → *MDN*

This function serves as constructor, which allows to compute hash of *bytes*.

4.5.1 Parameters

byte_string : message whose digest is to be computed.

classmethod from_file(*filename: str*) → *MDN*

This function serves as constructor, which allows to compute hash of file under given path.

4.5.2 Parameters

filename : path to existing file whose digest is to be computed.

static l_roll(*X: int*, *s: int*) → int

Roll (rotate) bits of 32-bit unsigned integer *s* positions to the left.

4.5.3 Parameters

X: integer to be rolled. Its binary representation cannot exceed 32 bits.

s: number of digits to roll. Must be integer in [0, 32].

last32 = 4294967295

last64 = 18446744073709551615

string_digest() → str

Returns string representation of message digest.

4.6 todo_project_name.rsa module

class todo_project_name.rsa.**RSAKey**(*key: int*, *modulus: int*, *id: Optional[str] = None*)

Bases: `ABC`

class todo_project_name.rsa.**RSAKeyPair**(*public:* todo_project_name.rsa.RSAKeyPublic, *private:* todo_project_name.rsa.RSAKeyPrivate)

Bases: `object`

private: *RSAKeyPrivate*

public: *RSAKeyPublic*

class todo_project_name.rsa.**RSAKeyPrivate**(*key: int*, *modulus: int*, *id: Optional[str] = None*)

Bases: *RSAKey*

class todo_project_name.rsa.**RSAKeyPublic**(*key: int*, *modulus: int*, *id: Optional[str] = None*)

Bases: *RSAKey*

todo_project_name.rsa.**read_key**(*path: Path*, *key_type: Type[RSAKeyVar]*) → RSAKeyVar

Read RSA key from the file.

todo_project_name.rsa.**rsa_key_gen**(*N: int*) → *RSAKeyPair*

Generate RSA key pair.

Takes number *N* and returns RSAKeyPair with (2 * N)-bit modulus.

4.6.1 Parameters

N : determines the strength of the protocol.

todo_project_name.rsa.**rsa_sign**(*message: str*, *key: ~todo_project_name.rsa.RSAKeyPrivate*, *algorithm: ~typing.Type[~typing.Union[~todo_project_name.md4.MD4*, *~todo_project_name.md5.MD5]] = <class 'todo_project_name.md4.MD4'>*) → str

Function returns a digital signature based on the RSA protocol.

4.6.2 Parameters

message : string message to be singed

key : RSA private key

algorithm : hash method. Default: MD4. Available algorithms: MD4, MD5.

todo_project_name.rsa.**rsa_sign_file**(*filename: str*, *key: ~todo_project_name.rsa.RSAKeyPrivate*, *algorithm: ~typing.Type[~typing.Union[~todo_project_name.md4.MD4*, *~todo_project_name.md5.MD5]] = <class 'todo_project_name.md4.MD4'>*) → str

Function returns a digital signature based on the RSA protocol.

4.6.3 Parameters

filename : path to existing file to sign

key : RSA private key

algorithm : hash method. Default: MD4. Available algorithms: MD4, MD5.

todo_project_name.rsa.**rsa_verify**(*message: str*, *signature: str*, *key: ~todo_project_name.rsa.RSAKeyPublic*, *algorithm: ~typing.Type[~typing.Union[~todo_project_name.md4.MD4*, *~todo_project_name.md5.MD5]] = <class 'todo_project_name.md4.MD4'>*)

Function verifies digital singnature of a message basing on the RSA protocol. It compares decoded signature with hashed message and returns True if they are the same, otherwise False.

4.6.4 Parameters

message : string message

signature : signature for verification

key : RSA public key

algorithm : hash algorithm. Default: MD4. Available algorithms: MD4, MD5.

todo_project_name.rsa.**rsa_verify_file**(*filename: str*, *signature: str*, *key: ~todo_project_name.rsa.RSAKeyPublic*, *algorithm: ~typing.Type[~typing.Union[~todo_project_name.md4.MD4*, *~todo_project_name.md5.MD5]] = <class 'todo_project_name.md4.MD4'>*)

Function verifies digital singnature of a message basing on the RSA protocol. It compares decoded signature with hashed message and returns True if they are the same, otherwise False.

4.6.5 Parameters

filename : path to file against which signature is being checked

signature : signature for verification

key : RSA public key

algorithm : hash algorithm. Default: MD4. Available algorithms: MD4, MD5.

`todo_project_name.rsa.save_key`(*key: RSAKey*, *path: Path*) → Path

Save RSA key to the file.