Johnny Can Encrypt Documentation

Release 0.14.1

Kushal Das

May 21, 2023

Contents:

This is a Python module providing encryption and decryption operations based on OpenPGP. It uses sequoia-pgp project for the actual operations. This does not depend on the existing *gpg* tooling.

CHAPTER 1

Installing for usage in a virtualenvironment

Building this module requires Rust's nightly toolchain. You can install it following the instructions from rustup.rs.

You will need *libnettle* and *libnettle-dev* & *clang*, *libpcsclite1*, *libpcsclite-dev* (on Debian/Ubuntu) and *nettle* & *nettle-dev pcsc-lite-devel* & *clang* packages in Fedora.

```
sudo apt install -y python3-dev libnettle8 nettle-dev libhogweed6 python3-pip python3-
↪venv clang libpcsclite-dev libpcsclite1 libclang-9-dev
```

```
sudo dnf install nettle clang clang-devel nettle-devel python3-devel pcsc-lite-devel
```

Then you can just use *pip* module to install in your virtualenvironment.

```
python -m pip install johnnycanencrypt
```

CHAPTER 2

Building Johnny Can Encrypt for development

After you have the dependencies mentioned above, you can follow the steps below to build a wheel.

```
python3 -m venv .venv
source .venv/bin/activate
python -m pip install -r requirements-dev.txt
python setup.py develop
```

Only to build and test locally, you should execute

```
python setup.py develop
```

To build a wheel use the following command.

```
python setup.py bdist_wheel
```

2.1 How to run the tests?

After you did the *python setup.py develop* as mentioned above, execute the following command.

```
python -m pytest -vvv
```

2.2 How to run the smartcard related tests?

Warning: The following test will reset any Yubikey or smartcard connected to the system. Use it carefully.

All of these tests are right now kept as a Python script, and requires Yubikey series 5 hardware to test.

```
python smartcardtests/smartcards.py
```

When asked, please make sure that only the test smartcard is conneccted to the system, and then type “Yes”, without quotes.

CHAPTER 3

Introduction to johnnycanencrypt

Johnnycanencrypt provides only a few selected operations from the OpenPGP spec. If you need more and better granular access to the operations, this may not be the module for you.

The module has 2 parts, one high level API, which can be directly accessed via importing the module. There is another internal module with the same name (which is native in Rust), and has all the low level functions and classes.

We will import the module as jce.

```
>>> import johnnycanencrypt as jce
```

3.1 The KeyStore

The module interacts over a *KeyStore* object, which points to a directory path on the system. Inside of the directory, it will create a *jce.db* sqlite3 database if missing. Below is an example where we are creating a keystore in an empty directory at "/var/lib/myapplication/keys", and then we will import a few keys in there. For our example, we will use the keys from our tests directory.

```
>>> ks = jce.KeyStore("/var/lib/myapplication/keys")
>>> ks.import_key("tests/files/store/secret.asc")
<Key fingerprint=F4F388BBB194925AE301F844C52B42177857DD79 type=SECRET>
>>> ks.import_key("tests/files/store/pgp_keys.asc")
<Key fingerprint=A85FF376759C994A8A1168D8D8219C8C43F6C5E1 type=PUBLIC>
>>> ks.import_key("tests/files/store/public.asc")
<Key fingerprint=F4F388BBB194925AE301F844C52B42177857DD79 type=PUBLIC>
```

Now, if we check the directory from the shell, we will find the keys imported there.

```
 ls -l /var/lib/myapplication/keys
.rw-rw-r--@ 9.5k user  6 Oct 18:48 jce.db
```

Note: This keystore directory is very much application specific. As a developer you should choose which directory on the system you will use as the key store. SecureDrop uses **/var/lib/securedrop/store** as their key storage (via gpg's python binding).

Warning: This module does not handle keys still using *sha1* or *md5* for hash algorithms. If you are using any such old key, please generate new key and use them along wtih the module. This function explains in some details why.

3.2 KeyStore path for the applicaitons which can run per user

If you are writing a desktop application or any other tool which can have per user configuration, you should look into the base dir spec. If your application name is **myapplication**, then the store path can be like: **$XDG_DATA_HOME/myapplication**.

3.3 Encrypting and decrypting some bytes for a given fingerprint

```
>>> ks = jce.KeyStore("tests/files/store")
>>> key = ks.get_key("F4F388BBB194925AE301F844C52B42177857DD79")
>>> enc = ks.encrypt(key, "Sequoia is amazing.")
>>> print(enc[:27])
b'-----BEGIN PGP MESSAGE-----'
>>> text = ks.decrypt(key, enc, "redhat")
>>> print(text)
b'Sequoia is amazing.'
```

3.4 Verify Tor Browser download using the signature and public key

In this example we will download the Tor Browser 10.0, and the signature and the public key using **wget**, and then verify via our module.

```
curl -s -O https://dist.torproject.org/torbrowser/12.0.5/tor-browser-linux64-12.0.5_
↪ALL.tar.xz
curl -s -O https://dist.torproject.org/torbrowser/12.0.5/tor-browser-linux64-12.0.5_
↪ALL.tar.xz.asc
KEYURL=https://openpgpkey.torproject.org/.well-known/openpgpkey/torproject.org/hu/
↪kounek7zrdx745qydx6p59t9mqjpuhdf
curl -s -o kounek7zrdx745qydx6p59t9mqjpuhdf.pub $KEYURL
```

Now let us import the key and verify.

```
import tempfile
import johnnycanencrypt as jce


filename = "tor-browser-linux64-12.0.5_ALL.tar.xz"
signame = "tor-browser-linux64-12.0.5_ALL.tar.xz.asc"

with tempfile.TemporaryDirectory() as tmpdir:
```

(continues on next page)

(continued from previous page)

```
    ks = jce.KeyStore(tmpdir)
    torkey = ks.import_key("kounek7zrdx745qydx6p59t9mqjpuhdf.pub")
    if ks.verify_file_detached(torkey, filename, signame):
        print("Verified.")
    else:
        print("Verification failed.")
```

CHAPTER 4

API Documentation

For the rest of the documentation we assume that you imported the module as following.

```
>>> import johnnycanencrypt as jce
```

class KeyStore(*path: str*) → None:

Returns a KeyStore object. This is the primary class of the module, and all high level usage is available via methods of this class. It takes a path to the directory where it stores/reads the keys. Please make sure that only the **user** has read/write capability to this path.

The keys are represented inside the directory in the **jce.db** sqlite3 database. Every time there is any change in the DB schema, we automatically create a temporary database called **jce_upgrade.db** in the same path, and then reimport all the keys, and rename the file and continue with the steps. This is one time operation when we do a new release.

You can check for existance of any fingerprint (str) or *Key* object in the via *in* opertor.

```
>>> ks = jce.KeyStore("/var/lib/myamazingapp")
>>> "HEXFINGERPRINT" in ks
```

add_userid(*key: Key*, *userid: str*, *password: str*) → Key:

Returns the updated key with a new userid. If you need to upload the key to the https://keys.openpgp.org, then remember to have to an email address in the user id.

certify_key(*key: Union[Key, str], otherkey: Union[Key, str], uids: List[str], sig_type: SignatureType = SignatureType.GenericCertification, password: str = "", oncard=False*) → Key:

This method signs the given list of userid(s) in *otherkey* using the primary key of the *key*, by default it signs as *SignatureType.GenericCertification*, but you can do other types too. If the primary key is on a smartcard, then pass *oncard=True*, default value is *False*.

create_key(*password: str, uids: Optional[Union[List[str], str]] = [], ciphersuite: Cipher = Cipher.RSA4k, creation: Optional[datetime.datetime] = None, expiration: Optional[datetime.datetime] = None, subkeys_expiration= False, whichkeys = 7, can_primary_sign: bool = False, can_primary_expire=False*) → Key:

Returns the public part of the newly created *Key* in the store directory. You can mention ciphersuite `Cipher` as *Cipher.RSA2k* or *Cipher.RSA4k*, or *Cipher.Cv25519*, while *Cipher.RSA4k* is the default. You can also provide *datetime.datetime* objects for creation time and expiration time. By default it will use the current time as creation time, and keys don't expire. You can provide a string for uid, or multiple strings using a List for multiple uids. It can also create a key without any uids.

If you want the primary key to have signing capability, then pass *can_primary_sign=True*.

You can pass *whichkeys* = *1* to generate only the encryption subkey, 2 for signing, 4 for authentication. By default it will create all three subkeys (7).

```
>>> ks = jce.KeyStore("/var/lib/myamazingapp")
>>> newkey = ks.create_key("supersecretpassphrasefromdiceware", "test key1
↪<email@example.com>", jce.KeyType.RSA4k)
```

encrypt (*keys*, *data*, *outputfile=""*, *armor=True*) → bytes:
Encrypts the given data (either as str or bytes) via the list of keys or fingerprints. You can also just pass one single key or fingerprint. If you provide *outputfile* argument with a path, the encrypted output will be written to that path. By default the encrypted output is armored, but by passing *armor=False* you can get raw bytes returned.

```
>>> ks = jce.KeyStore("tests/files/store")
>>> key1 = ks.get_key("6AC6957E2589CB8B5221F6508ADA07F0A0F7BA99")
>>> key2 = ks.get_key("BB2D3F20233286371C3123D5209940B9669ED621")
>>> encrypted = ks.encrypt([key1, key2], "Encrypted this string")
>>> assert encrypted.startswith(b"-----BEGIN PGP MESSAGE-----\n")
```

encrypt_file (*keys*, *inputfilepath: Union[str,bytes,BinaryIO]*, *outputfilepath: Union[str, bytes]*, *armor=True*) → bool:
Returns *True* after encrypting the given *inputfilepath* to the *outputfilepath*. The *inputfilepath* could be *str*, or *bytes*, or a opened file handler for bytes.

```
>>> ks = jce.KeyStore("tests/files/store")
>>> key1 = ks.get_key("6AC6957E2589CB8B5221F6508ADA07F0A0F7BA99")
>>> key2 = ks.get_key("BB2D3F20233286371C3123D5209940B9669ED621")
>>> assert ks.encrypt_file([key1, key2], "/tmp/data.txt", "/tmp/data.txt.asc")
```

decrypt (*key*, *data*, *password=""*) → bytes:
Returns the decrypted bytes from the given data and the secret key. You can either pass fingerprint or a secret *Key* object as the *key* argument.

```
>>> plain_bytes = ks.decrypt(secret_key2, encrypted_bytes, password=password)
```

decrypt_file(key, encrypted_path: Union[str,bytes,BinaryIO], outputfile, password=""):
Decryptes the given *encrypted_path* and wrties the output to the *outputfile* path (both given as str or bytes). In the *encrypted_path* can be an opened file handler to read binary data.

```
>>> ks.decrypt_file(secret_key1, "/tmp/data.txt.asc", "/tmp/plain.txt",␣
↪password=password)
>>> with open("/tmp/hello.gpg", "rb") as fobj:
...     ks.decrypt_file(secret_key1, fobj, "/tmp/plain.txt",␣
↪password=password)
```

delete_key (*key: Union[str, Key]*) → None:
Deletes the key based on the fingerprint or the Key object from the KeyStore.

```
>>> ks.delete_key("BB2D3F20233286371C3123D5209940B9669ED621")
```

Note: Raises *KeyNotFoundError* if the key is not available in the KeyStore.

details () → Tuple[int, int]:
Returns a tuple containing the total number of public and secret keys available in the KeyStore.

fetch_key_by_email (*email: str*) → Key:
Searches and retrives a key at https://keys.openpgp.org based on the given email address. Current limit is 1 email address can be searched only once per minute. The key is also stored in the local keystore. Raises *KeyNotFoundError* if the key is not found.

fetch_key_by_fingerprint (*fingerprint: str*) → Key:
Searches and retrives a key at https://keys.openpgp.org based on the given fingerprint, one can search 6 times per minute. Raises *KeyNotFoundError* if the key is not found.

get_all_keys () → List[Key]:
Returns a list of all the keys in the KeyStore.

get_key (*fingerprint: str = ""*) → Key:
Returns a key from the keystore based on the fingerprint. Raises **KeyNotFoundError** if no such key available in the keystore.

get_keys (*qvalue="", qtype="email"*) → List[Key]:
Returns a list of keys based on either email or name or value of the UIDs or URIs in the key (searchs on one of the terms first come basis). qtype can be one of the *email*, *value*, *name*, *uri*.

```
>>> keys_via_names = ks.get_keys(qvalue="test key", qtype="value")
>>> keys_via_email = ks.get_keys(qvalue="email@example.com")
```

get_keys_by_keyid (*keyid: str*) → List[Key]:
Returns a list of keys matching with the keyids, first directly checks the master keys and then subkeys. Raises **KeyNotFoundError** in case no such keyid is found on the store.

import_key (*keypath: str*) → Key:
Imports a pgp key file from a path on the system. The method returns the newly import *Key* object to the caller.

```
>>> key = ks.import_key("tests/files/store/public.asc")
>>> print(key)
```

revoke_userid (*key: Key*, *userid: str*, *pass: str*) → Key:
Revokes the given user id from the given secret key and returns the updated key.

update_expiry_in_subkeys (*key: Key, subkeys: List[str], expiration: datetime, password: str*) → Key:
Updates the expiry time for the given subkeys (as a list of fingerprints) for the given secret key.

sign_detached (*key*, *data*, *password*) → str:
Signs the given *data* (can be either str or bytes) using the secret key. Returns the armored signature string.

sign_file_detached (*key*, *filepath*, *password*, *write=False*) → str:
Returns the armored signature of the *filepath* argument using the secret key (either fingerprint or secret *Key* object). If you pass *write=True*, it will also write the armored signature to a file named as *filepath.asc*

verify (*key, data: Union[str, bytes], signature:Optional[str]*) → bool:
Verifies the given *data* using the public key, and signature string if given, returns **True** or **False** as result.

verify_file_detached (*key: Union[str, Key], filepath: Union[str, bytes], signature_path*) → bool:
Verifies the given *filepath* using the public key, and signature string, returns **True** or **False** as result.

`verify_file`(*key*, *filepath*) → bool:
Verifies the given signed *filepath* using the public key, returns **True** or **False** as result.

`verify_and_extract_bytes`(*key: Union[str, Key], data: Union[str, bytes]*) → bytes:
Verifies the given signed *data* using the public key, returns the actual data as bytes.

`verify_and_extract_file`(*self, key: Union[str, Key], filepath: Union[str, bytes], output: Union[str, bytes]*) → bool::
Verifies the given signed *filepath* and writes the actual data in *output*.

`class Cipher` → Cipher:
This is the enum class to metion the type of ciphersuite to be used while creating a new key. Possible values are **Cipher.RSA4k**, **Cipher.RSA2k**, **Cipher.Cv25519**.

`class Key`(*keyvalue: bytes*, *fingerprint: str*, *uids: Dict[str, str] = {}*, *keytype: KeyType=KeyType.PUBLIC*, *expirationtime=None*, *creationtime=None*, *othervalues={}*, *oncard: str = ""*, *can_primary_sign: bool = False*, *primary_on_card: str = ""*) → Key:
Returns a Key object and fingerprint. The keytype enum *KeyType*.

You can compare two key object with == operator.

For most of the use cases you don't have to create one manually, but you can retrive one from the *KeyStore*.

`keyvalue`
keyvalue holds the actual key as bytes.

`fingerprint`
The string representation of the fingerprint

`uids`
A dictionary holding all uids from the key, also stores related **certification** of the given uids.

`creationtime`
The datetime.datetime object mentioning when the key was created.

`expirationtime`
The datetime.datetime object mentioning when the key will expire or *None* otherwise.

`get_pub_key`() → str:
Returns the armored version of the public key as string.

`keyid`
The keyid of the master key

`primary_on_card`
A string containing the smartcard ID, this will be populated only after *sync_smartcard* call in the *KeyStore*.

`oncard`
A string containing the smartcard ID if the card contains any of the subkeys, this will be populated only after *sync_smartcard* call in the *KeyStore*.

`othervalues`
A dictionary containing subkeys's keyids and fingerprints.

`can_primary_sign`
A boolean value telling if the primary key has signing capability or not.

`available_subkeys`() → Tuple[bool, bool, bool]:
Returns a tuple with 3 boolean values as (got_enc, got_sign, got_auth) to tell us which all subkeys are available. The subkeys will not be expired keys (based on the date of the system) and also not revoked.

`class KeyType` → KeyType:
Enum class to mark if a key is public or private. Possible values are **KeyType.PUBLIC** and **KeyType.SECRET**.

`class SignatureType` → SignatureType:

Enum class to mark the kind of certification one can do on another key. Possible values are **SignatureType.GenericCertification**, **SignatureType.PersonaCertification**, **SignatureType.CasualCertification**, **SignatureType.PositiveCertification**.

`get_card_touch_policies``()` → List[TouchMode]

Returns a list of Enum values from TouchMode. To be used to determine the touch capabilities of the smartcard. Remember to verify this list before calling *`set_keyslot_touch_policy()`*.

CHAPTER 5

The internal johnnycanencrypt module written in Rust

You can access the low level functions or *Johnny* class by the following way:

```
>>> from johnnycanencrypt import johnnycanencrypt as jce
```

In most cases you don't have to use these, but if you have a reason, feel free to use them.

encrypt_bytes_to_file (*publickeys*, *data*, *output*, *armor=False*)
This function takes a list of public key file paths, and encrypts the given data in bytes to an output file. You can also pass boolen flag armor for armored output in the file.

```
>>> jce.encrypt_bytes_to_file(["tests/files/public.asc", "tests/files/hellopublic.
↪asc"], b"Hello clear text", b"/tmp/encrypted_text.asc", armor=True)
```

Note: Use this function if you have to encrypt for multiple recipents.

get_ssh_pubkey (*certdata, comment: Optional[str]*) → str:
This function takes a public key and optional comment and then provides a string representing the authentication subkey to be used inside of SSH.

enable_otp_usb () → bool
This function enables OTP application in the Yubikey.

disable_otp_usb () → bool
This function disables OTP application in the Yubikey.

get_key_cipher_details (*certdata: bytes*) → List[tuple[str, str, int]]
This function takes the key data as bytes, and returns a list of tuples containing (fingerprint, public key algorithm, bits size).

```
>>> rjce.get_key_cipher_details(key.keyvalue)
[('F4F388BBB194925AE301F844C52B42177857DD79', 'EdDSA', 256), (
↪'102EBD23BD5D2D340FBBDE0ADFD1C55926648D2F', 'EdDSA', 256), (
↪'85B67F139D835FA56BA703DB5A7A1560D46ED4F6', 'ECDH', 256)]
```

class Johnny(*filepath*)

It creates an object of type *Johnny*, you can provide path to the either public key, or the private key based on the operation you want to do.

encrypt_bytes(*data: bytes*, *armor=False*)

This method encrypts the given bytes and returns the encrypted bytes. If you pass *armor=True* to the method, then the returned value will be ascii armored bytes.

```
>>> j = jce.Johnny("tests/files/public.asc")
>>> enc = j.encrypt_bytes(b"mysecret", armor=True)
```

encrypt_file(*inputfile: bytes*, *output: bytes*, *armor=False*)

This method encrypts the given inputfile and writes the raw encrypted bytes to the output path. If you pass *armor=True* to the method, then the output file will be written as ascii armored.

```
>>> j = jce.Johnny("tests/files/public.asc")
>>> enc = j.encrypt_file(b"blueleaks.tar.gz", b"notblueleaks.tar.gz.pgp",␣
↪armor=True)
```

decrypt_bytes(*data: bytes*, *password: str*)

Decrypts the given bytes based on the secret key and given password. If you try to decrypt while just using the public key, then it will raise *AttributeError*.

```
>>> jp = jce.Johnny("tests/files/secret.asc")
>>> result = jp.decrypt_bytes(enc, "redhat")
```

decrypt_file(*inputfile: bytes*, *output: bytes*, *password: str*)

Decrypts the inputfile path (in bytes) and wrties the decrypted data to the *output* file. Both the filepaths to be given as bytes.

```
>>> jp = jce.Johnny("tests/files/secret.asc")
>>> result = jp.decrypt_file(b"notblueleaks.tar.gz.pgp", "blueleaks.tar.gz",
↪"redhat")
```

sign_bytes_detached(*data: bytes*, *pasword: str*)

Signs the given bytes and returns the detached ascii armored signature as bytes.

```
>>> j = jce.Johnny("tests/files/secret.asc")
>>> signature = j.sign_bytes_detached(b"mysecret", "redhat")
```

Note: Remember to save the signature somewhere on disk.

verify_bytes(*data: bytes*)

Verifies if the signature is correct for the given signed data (as bytes). Returns *True* or *False*.

```
>>> j = jce.Johnny("tests/files/public.asc")
>>> j.verify_bytes(encrypted_bytes)
```

verify_and_extract_bytes(*data: bytes*)

Verifies if the signature is correct for the given signed data (as bytes). Returns the actual message in Bytes.

```
>>> j = jce.Johnny("tests/files/public.asc")
>>> j.verify_and_extract_bytes(encrypted_bytes)
```

verify_bytes_detached(*data: bytes*, *signature: bytes*)

Verifies if the signature is correct for the given data (as bytes). Returns *True* or *False*.

```
>>> j = jce.Johnny("tests/files/public.asc")
>>> j.verify_bytes(encrypted_bytes, signature)
```

verify_file (*filepath: bytes*)
Verifies if the signature is correct for the given signed file (path as bytes). Returns *True* or *False*.

```
>>> j = jce.Johnny("tests/files/public.asc")
>>> j.verify_file(encrypted_bytes, signature)
```

verify_and_extract_file (*filepath: bytes*, *output: bytes*)
Verifies and extracts the message from the signed file, return *True* in case of a success.

verify_file_detached (*filepath: bytes*, *signature: bytes*)
Verifies if the signature is correct for the given signed file (path as bytes). Returns *True* or *False*.

```
>>> j = jce.Johnny("tests/files/public.asc")
>>> j.verify_file_detached(encrypted_bytes, signature)
```

CHAPTER 6

Smartcard access

Johnnycanencrypt provides limilted smardcard access for OpenPGP operations. This is built on top of the 3.4.1 spec.

We only tested the code against Yubikey 5 and Yubikey 4 series.

Note: Remember the *Cv25519* keys will only work on Yubikey 5 series.

The part of the code is written in Rust, so you will have to import the internal *johnnycanencrypt* module.

```
import johnnycanencrypt.johnnycanencrypt as rjce
```

6.1 Smartcard API

`class KeySlot`

These are the available KeySlots in a card.

`Signature`

`Encryption`

`Authentication`

`Attestation`

`class TouchMode`

The different touch mode for a key.

`Off`

`On`

`Fixed`

`Cached`

`CachedFixed`

`set_keyslot_touch_policy`(*adminpin: bytes*, *slot: KeySlot*, *mode: TouchMode*) → bool:
Sets the given *TouchMode* to the slot. Returns False if it is already set as Fixed.

Important: Remember to verify the available touch modes via `get_card_touch_policies()` first.

`get_keyslot_touch_policy`(*slot: KeySlot*) → TouchMode:
Returns the available *TouchMode* of the given slot in the smartcard.

`get_card_version`() → tuple[int, int, int]:
Returns a tuple containing the Yubikey firmware version. Example: (5,2,7) or (4,3,1).

`reset_yubikey`() → bool:
Returns *True* after successfully resetting a Yubikey.

Warning: This is a dangerous function as it will destroy all the data in the card. Use it carefully.

`get_card_details`() → Dict[str, Any]:
Returns a dictionary containing various card information in a dictionary.

Available keys:

- *serial_number*, the serial number of the card
- *url*, for the public key url.
- *name*, the card holder's name, surname<<<firstname
- *PW1*, number of user pin retries left
- *RC*, number of reset pin retries left
- *PW2*, number of admin pin retries left
- *signatures*, total number signatures made by the card
- *sig_f* Signature key fingerprint
- *enc_f* encryption key fingerprint
- *auth_f* authentication key fingerprint

`change_user_pin`(*adminpin: bytes*, *newpin: bytes*) → bool:
Changes the user pin to the given pin. The pin must be 6 chars or more. Requires current admin pin of the card.

`change_admin_pin`(*adminpin: bytes*, *newadminpin: bytes*) → bool:
Changes the admin pin to the given pin. The pin must be 8 chars or more. Requires current admin pin of the card.

`decrypt_bytes_on_card(certdata: bytes, data: bytes, pin:bytes): -> bytes`
Decryptes the given encrypted bytes using the smartcard. You will have to pass the public key as the *certdata* argument.

`decrypt_file_on_card(certdata: bytes, filepath: bytes, output: bytes, pin:bytes): -> None`
Decryptes the given *filepath* and writes the output to the given *output* path using the smartcard. You will have to pass the public key as the *certdata* argument.

`decrypt_filehandler_on_card(certdata: bytes, fh: typing.IO, output: bytes, pin:bytes): ->`
Decryptes the given opened *fh* and writes the output to the given *output* path using the smartcard. You will have to pass the public key as the *certdata* argument.

Note: This function first reads the whole file and then decrypts it. So, try to use the *decrypt_file_on_card* function instead.

is_smartcard_connected() → bool:
Returns *True* if it can find a Yubikey attached to the system, or else returns *False*.

set_name(*name: bytes*, *pin: bytes*) → bool:
Sets the name of the card holder (in bytes) in *surname<<firstname* format. The length must be less than 39 in size. Requires admin pin in bytes.

set_url(*url: bytes*, *pin: bytes*) → bool:
Sets the public key URL on the card. Requires the admin pin in bytes.

sign_bytes_detached_on_card(*certdata: bytes*, *data: bytes*, *pin: bytes*) → str:
Signs the given bytes on the card, and returns the detached signature as base64 encoded string. Also requires the public key in *certdata* argument.

sign_bytes_on_card(*certdata: bytes*, *data: bytes*, *pin: bytes*) → bytes:
Signs the given bytes on the card, and returns the signed bytes. Also requires the public key in *certdata* argument.

sign_file_detached_on_card(*certdata: bytes*, *filepath: bytes*, *pin: bytes*) → str:
Signs the given filepath and returns the detached signature as base64 encoded string. Also requires the the public in *certdata* argument.

sign_file_on_card(*certdata: bytes*, *filepath: bytes*, *output: bytes*, *pin: bytes*, *cleartext: bool*) → bool:
Signs the given filepath and writes to output. Also requires the the public in *certdata* argument. For things like email, you would want to sign them in clear text.

upload_to_smartcard(*certdata: bytes*, *pin: bytes*, *password: str*, *whichkeys: int*) → bool:
Uploads the marked (via whichkeys argument) subkeys to the smartcard. Takes the whole certdata (from *Key.keyvalue*) in bytes, and the admin pin of the card, the password (as string) of the key. You can choose which subkeys to be uploaded via the following values of *whichkeys* argument:

- *1* for encryption
- *2* for signing
- *4* for authentication

And then you can add them up for the required combination. For example *7* means you want to upload all 3 kinds of subkeys, but *3* means only encryption and signing subkeys will be loaded into the smartcard.

- *3* for both encryption and signing
- *5* for encryption and authentication
- *6* for signing and authentication
- *7* for all 3 different subkeys

```
import johnnycanencrypt as jce
import johnnycanencrypt.johnnycanencrypt as rjce

ks = jce.KeyStore("/tmp/demo")
# By default it creates all 3 subkeys
key = ks.create_key("redhat", ["First Last <fl@example.com>",], jce.Cipher.
↪Cv25519)
print(key.fingerprint)
# We want to upload only the encryption and signing subkeys to the smartcard
result = rjce.upload_to_smartcard(key.keyvalue, b"12345678", "redhat", 3)
print(result)
```

upload_primary_to_smartcard(*certdata: bytes*, *pin: bytes*, *password: str*, *whichslot: int*) → bool:

Uploads the primary key to the smartcard in the given slot. Takes the whole certdata (from *Key.keyvalue*) in bytes, and the admin pin of the card, the password (as string) of the key. You can choose which subkeys to be uploaded via the following values of *whichslot* argument:

- *2* for signing slot
- *4* for authentication slot

```
import johnnycanencrypt as jce
import johnnycanencrypt.johnnycanencrypt as rjce

ks = jce.KeyStore("/tmp/demo")
# Create a primary key with signing capability & an encryption subkey
key = ks.create_key("redhat", ["First Last <fl@example.com>",], jce.Cipher.
↪Cv25519, whichkeys=1, can_primary_sign=True)
print(key.fingerprint)
# We want to upload first the primary key to the signing slot of the card
result = rjce.upload_primary_to_smartcard(key.keyvalue, b"12345678", "redhat",␣
↪whichslot=2)
# We want to upload only the encryption subkey to the smartcard
result = rjce.upload_to_smartcard(key.keyvalue, b"12345678", "redhat", 1)
print(result)
```

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index