Python Paillier Documentation

Release 1.2.2

NICTA

June 02, 2016

Contents

A Python 3 library for **P**artially **H**omomorphic **E**ncryption using the Paillier crypto system.

The homomorphic properties of the Paillier crypto system are:

- Encrypted numbers can be multiplied by a non encrypted scalar.
- Encrypted numbers can be added together.
- Encrypted numbers can be added to non encrypted scalars.

CHAPTER 1

Installation

Note: This library requires a minimum Python version of at least 3.3.

1.1 Using pip

Using pip at the command line, to install the base library from PyPi:

```
$ pip install phe
```

To also install the *command line utility*, introduced at version 1.2:

```
pip install "phe[cli]>1.2"
```

1.2 Manual installation

To install from the source package, first install the optional dependencies (eg Crypto):

```
$ pip install -r requirements.txt
```

Then install as normal:

```
$ python setup.py install
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv phe
$ pip install -e ".[CLI]"
```

CHAPTER 2

Usage

There are two roles that use this library. In the first, you control the private keys. In the second, you don't. This guide shows you how to play either role.

In either case, you of course begin by importing the library:

```
from phe import paillier
```

2.1 Role #1

This party holds the private keys and typically will generate the keys and do the decryption.

2.1.1 Key generation

First, you're going to have to generate a public and private key pair:

```
>>> public_key, private_key = paillier.generate_paillier_keypair()
```

If you're going to have lots of private keys lying around, then perhaps you should invest in a keyring on which to store your `PaillierPrivateKey` instances:

```
>>> keyring = paillier.PaillierPrivateKeyring()
>>> keyring.add(private_key)
>>> public_key1, private_key1 = paillier.generate_paillier_keypair(keyring)
>>> public_key2, private_key2 = paillier.generate_paillier_keypair(keyring)
```

In any event, you can then start encrypting numbers:

```
>>> secret_number_list = [3.141592653, 300, -4.6e-12]
>>> encrypted_number_list = [public_key.encrypt(x) for x in secret_number_list]
```

Presumably, you would now share the ciphertext with whoever is playing Role 2 (see *Serialisation* and *Compatibility with other libraries*).

2.1.2 Decryption

To decrypt an `EncryptedNumber`, use the relevant `PaillierPrivateKey`:

```
>>> [private_key.decrypt(x) for x in encrypted_number_list]
[3.141592653, 300, -4.6e-12]
```

If you have multiple key pairs stored in a `PaillierPrivateKeyring`, then you don't need to manually find the relevant `PaillierPrivateKey`:

```
>>> [keyring.decrypt(x) for x in encrypted_number_list]
[3.141592653, 300, -4.6e-12]
```

2.2 Role #2

This party does not have access to the private keys, and typically performs operations on supplied encrypted data with their own, unencrypted data.

Once this party has received some `EncryptedNumber` instances (e.g. see *Serialisation*), it can perform basic mathematical operations supported by the Paillier encryption:

1. Addition of an `EncryptedNumber` to a scalar
2. Addition of two `EncryptedNumber` instances
3. Multiplication of an `EncryptedNumber` by a scalar

```
>>> a, b, c = encrypted_number_list
>>> a
<phe.paillier.EncryptedNumber at 0x7f60a28c90b8>

>>> a_plus_5 = a + 5
>>> a_plus_b = a + b
>>> a_times_3_5 = a * 3.5
```

as well as some simple extensions:

```
>>> a_minus_1_3 = a - 1             # = a + (-1)
>>> a_div_minus_3_1 = a / -3.1      # = a * (-1 / 3.1)
>>> a_minus_b = a - b               # = a + (b * -1)
```

Numpy operations that rely only on these operations are allowed:

```
>>> import numpy as np
>>> enc_mean = np.mean(encrypted_number_list)
>>> enc_dot = np.dot(encrypted_number_list, [2, -400.1, 5318008])
```

Operations that aren't supported by Paillier's *partially* homomorphic scheme raise an error:

```
>>> a * b
NotImplementedError: Good luck with that...

>>> 1 / a
TypeError: unsupported operand type(s) for /: 'int' and 'EncryptedNumber'
```

Once the necessary computations have been done, this party would send the resulting `EncryptedNumber` instances back to the holder of the private keys for decryption.

In some cases it might be possible to boost performance by reducing the precision of floating point numbers:

```
>>> a_times_3_5_lp = a * paillier.EncodedNumber.encode(a.public_key, 3.5, 1e-2)
```

CHAPTER 3

Serialisation

This library does not enforce any particular serialisation scheme.

Every *EncryptedNumber* instance has a *public_key* attribute, and serialising each *EncryptedNumber* independently would be heinously inefficient when sending a large list of instances. It is up to you to serialise in a way that is efficient for your use case.

3.1 Basic JSON Serialisation

This basic serialisation method is an example of serialising a vector of encrypted numbers. Note that if you are only using the python-paillier library **g** will always be **n + 1**, so these is no need to serialise it as part of the public key.

To send a list of values encrypted against one public key, the following is one way to serialise:

```
>>> import json
>>> enc_with_one_pub_key = {}
>>> enc_with_one_pub_key['public_key'] = {'g': public_key.g,
...                                       'n': public_key.n}
>>> enc_with_one_pub_key['values'] = [
...     (str(x.ciphertext()), x.exponent) for x in encrypted_number_list
... ]
>>> serialised = json.dumps(enc_with_one_pub_key)
```

Deserialisation of the above scheme might look as follows:

```
>>> received_dict = json.loads(serialised)
>>> pk = received_dict['public_key']
>>> public_key_rec = paillier.PaillierPublicKey(g=int(pk['g']),
...                                             n=int(pk['n']))
>>> enc_nums_rec = [
...     paillier.EncryptedNumber(public_key_rec, int(x[0]), int(x[1]))
...     for x in received_dict['values']
... ]
```

If both parties already know *public_key*, then you might instead send a hash of the public key.

3.2 JWK Serialisation

This serialisation scheme is used by the *Command Line Utility*, and is based on the JSON Web Key (JWK) format. This serialisation scheme should be used to increase compatibility between libraries. All cryptographic integers are represented as Base64UrlEncoded numbers. Note the existence of *base64_to_int()* and *int_to_base64()*.

3.2.1 “kty” (Key Type) Parameter

We define the family for all Paillier keys as “DAJ” for Damgard Jurik.

3.2.2 “alg” (Algorithm) Parameter

We identify the algorithm for our Paillier keys as: “PAI-GN1”

3.2.3 “key_ops” (Key Operations) Parameter

Values will be “encrypt” and “decrypt” for public and private keys respectively. We decided not to add homomorphic properties to the key operations.

3.2.4 “kid” (Key Identifier)

The **kid** may be set to any ascii string. Useful for storing key names, generation tools, times etc.

3.2.5 Public Key

In addition to the “kty”, “kid”, “key_ops” and “alg” attributes, a public key will have:

- **n** The public key’s modulus - *Base64 url encoded*

Example of a 256 bit public key:

```
python -m phe.command_line genpkey --keysize 256 - | python -m phe.command_line extract - -
{
    "kty": "DAJ",
    "kid": "Example Paillier public key",
    "key_ops": [ "encrypt" ],
    "n": "m0lOEwDHVA_VieL2k3BKMjf_HIgagfhNIZy1YhgZF5M",
    "alg": "PAI-GN1"
}
```

3.2.6 Private Key

Note: The serialised private key includes the public key.

In addition to the “kty”, “kid”, “key_ops” and “alg” attributes, a private key will have:

- **mu** and **lambda** - The private key’s secrets. See Paillier’s paper for details.
- **pub** - The Public Key serialised as described above.

Example of a 256 bit private key:

```
python -m phe.command_line genpkey --keysize 256 -
{
    "mu": "Dzq1_tz2qDX_-S4shia9Rw34Z9ix9b-fhPi3In76NaI",
    "kty": "DAJ",
    "key_ops": [ "decrypt" ],
    "kid": "Paillier private key generated by pheutil on 2016-05-24 14:18:25",
    "lambda": "haFTvA70KcI5XXReJUlQWRQdYHxaUS8baGQGug9dewA",
    "pub": {
        "alg": "PAI-GN1",
        "n": "haFTvA70KcI5XXReJUlQWoZus12aSJJ5EXAvu93xR7k",
        "kty": "DAJ",
        "key_ops": [ "encrypt" ],
        "kid": "Paillier public key generated by pheutil on 2016-05-24 14:18:25"
    }
}
```

Warning: “kty” and “alg” values should be registered in the IANA “JSON Web Key Types” registry established by JWA. We have not registered **DAJ** or **PAI-GN1** - however we intend to begin that conversation.

CHAPTER 4

Security Caveats

4.1 Information leakage

The *`exponent`* of an *`EncryptedNumber`* is not encrypted. By default, for floating point numbers this leads to some information leakage about the magnitude of the encrypted value. This leakage can be patched up by deciding on a fixed value for all exponents as part of the protocol; then for each *`EncryptedNumber`*, *`decrease_exponent_to()`* can be called before sharing. In practice this exponent should be a lower bound for any exponent that would naturally arise.

4.2 Alternative Base for EncodedNumber

If you need to interact with a library using another base, create a simple subclass of `paillier.EncodedNumber` and ensure you include the *BASE* and *LOG2_BASE* attributes:

```
class AltEncodedNumber(paillier.EncodedNumber):
    BASE = 2
    LOG2_BASE = math.log(BASE, 2)
```

> **Warning:** As always, if you don't require a specific value for the unencrypted exponents after an operation, you might be leaking information about what happened - but with smaller bases this problem is exacerbated.

4.3 No audit

This code has neither been written nor vetted by any sort of crypto expert. The crypto parts are mercifully short, however.

CHAPTER 5

Command Line Utility

This cli interface allows a user to:

- generate and serialize key pairs (of different key sizes)
- encrypt and serialize given a public key and a plaintext number
- decrypt given a private key and the ciphertext
- add two encrypted numbers together
- add an encrypted number to a plaintext number
- multiply an encrypted number to a plaintext number

5.1 Installation

The command line utility is not installed by default. When installing with pip you must specify the optional extra eg:

```
pip install "phe[cli]" --upgrade
```

After *Installation*, the **pheutil** command line program will be installed on your path.

To use the command line client without installing *python-paillier*, run the `phe.command_line` module from the project root:

```
python -m phe.command_line
```

5.2 Usage Help

For commands, and examples call –*help*:

```
$ pheutil --help

Usage: pheutil [OPTIONS] COMMAND [ARGS]...

  CLI for interacting with python-paillier

Options:
  --version      Show the version and exit.
  -v, --verbose  Enables verbose mode.
  --help         Show this message and exit.
```

```
Commands:
  add       Add encrypted number to unencrypted number.
  addenc    Add two encrypted numbers together.
  decrypt   Decrypt ciphertext with private key.
  encrypt   Encrypt a number with public key.
  extract   Extract public key from private key.
  genpkey   Generate a paillier private key.
  multiply  Multiply encrypted num with unencrypted num.
```

Each command also includes more detail, e.g. for *genpkey*:

```
$ pheutil genpkey --help
Usage: pheutil genpkey [OPTIONS] OUTPUT

  Generate a paillier private key.

  Output as JWK to given output file. Use "-" to output the private key to
  stdout. See the extract command to extract the public component of the
  private key.

  Note:     The default ID text includes the current time.

Options:
  --keysize INTEGER  The keysize in bits. Defaults to 2048
  --id TEXT          Add an identifying comment to the key
```

5.3 Example Session

```
$ pheutil genpkey --keysize 1024 example_private_key.json
Generating a paillier keypair with keysize of 1024
Keys generated
Private key written to example_private_key.json
$ pheutil extract example_private_key.json example_public_key.json
Loading paillier keypair
Public key written to example_public_key.json
$ pheutil encrypt --output test.enc example_public_key.json 5000
Loading public key
Encrypting: +5000.0000000000000000
$ cat test.enc | python -m json.tool
{
    "e": -32,
    "v": "894546896128285225677822098923822217215045642580837395357822930177580320540956563722368800
}
$ pheutil add --output result.enc example_public_key.json test.enc 100
Loading public key
Loading encrypted number
Loading unencrypted number
Adding
Exponent is less than -32
$ pheutil decrypt example_private_key.json result.enc
Loading private key
Decrypting ciphertext
5100.0
```

5.4 Bash Completion

Bash completion can be enabled by adding the following to your *.bashrc* file:

```
eval "$(_PHEUTIL_COMPLETE=source pheutil)"
```

Further information on bash completion can be found in the click documentation.

CHAPTER 6

API Documentation

6.1 Paillier

Paillier encryption library for partially homomorphic encryption.

class `phe.paillier.`**`EncodedNumber`**(*public_key*, *encoding*, *exponent*)

Bases: `object`

Represents a float or int encoded for Paillier encryption.

For end users, this class is mainly useful for specifying precision when adding/multiplying an `EncryptedNumber` by a scalar.

If you want to manually encode a number for Paillier encryption, then use `encode()`, if de-serializing then use `__init__()`.

Note: If working with other Paillier libraries you will have to agree on a specific `BASE` and `LOG2_BASE` - inheriting from this class and overriding those two attributes will enable this.

Notes

Paillier encryption is only defined for non-negative integers less than `PaillierPublicKey.n`. Since we frequently want to use signed integers and/or floating point numbers (luxury!), values should be encoded as a valid integer before encryption.

The operations of addition and multiplication [1] must be preserved under this encoding. Namely:

1.Decode(Encode(a) + Encode(b)) = a + b

2.Decode(Encode(a) * Encode(b)) = a * b

for any real numbers a and b.

Representing signed integers is relatively easy: we exploit the modular arithmetic properties of the Paillier scheme. We choose to represent only integers between +/-`max_int`, where *max_int* is approximately `n`/3 (larger integers may be treated as floats). The range of values between *max_int* and *n* - *max_int* is reserved for detecting overflows. This encoding scheme supports properties #1 and #2 above.

[1] Technically, since Paillier encryption only supports multiplication by a scalar, it may be possible to define a secondary encoding scheme *Encode'* such that property #2 is relaxed to:

Decode(Encode(a) * Encode'(b)) = a * b

We don't do this.

Representing floating point numbers as integers is a harder task. Here we use a variant of fixed-precision arithmetic. In fixed precision, you encode by multiplying every float by a large number (e.g. 1e6) and rounding the resulting product. You decode by dividing by that number. However, this encoding scheme does not satisfy property #2 above: upon every multiplication, you must divide by the large number. In a Paillier scheme, this is not possible to do without decrypting. For some tasks, this is acceptable or can be worked around, but for other tasks this can't be worked around.

In our scheme, the "large number" is allowed to vary, and we keep track of it. It is:

> `BASE` ** `exponent`

One number has many possible encodings; this property can be used to mitigate the leak of information due to the fact that `exponent` is never encrypted.

For more details, see `encode()`.

Parameters

- **`public_key`** (`PaillierPublicKey`) – public key for which to encode (this is necessary because `max_int` varies)
- **`encoding`** (*int*) – The encoded number to store. Must be positive and less than `max_int`.
- **`exponent`** (*int*) – Together with `BASE`, determines the level of fixed-precision used in encoding the number.

`public_key`
PaillierPublicKey

public key for which to encode (this is necessary because `max_int` varies)

`encoding`
int

The encoded number to store. Must be positive and less than `max_int`.

`exponent`
int

Together with `BASE`, determines the level of fixed-precision used in encoding the number.

`BASE` = 16
Base to use when exponentiating. Larger *BASE* means that `exponent` leaks less information. If you vary this, you'll have to manually inform anyone decoding your numbers.

`FLOAT_MANTISSA_BITS` = 53

`LOG2_BASE` = 4.0

`decode`()
Decode plaintext and return the result.

Returns the decoded number. N.B. if the number returned is an integer, it will not be of type float.

Return type an int or float

Raises `OverflowError` – if overflow is detected in the decrypted number.

`decrease_exponent_to`(*new_exp*)
Return an *EncodedNumber* with same value but lower exponent.

If we multiply the encoded value by `BASE` and decrement `exponent`, then the decoded value does not change. Thus we can almost arbitrarily ratchet down the exponent of an `EncodedNumber` - we only run into trouble when the encoded integer overflows. There may not be a warning if this happens.

This is necessary when adding `EncodedNumber` instances, and can also be useful to hide information about the precision of numbers - e.g. a protocol can fix the exponent of all transmitted `EncodedNumber` to some lower bound(s).

Parameters **`new_exp`** (`int`) – the desired exponent.

Returns Instance with the same value and desired exponent.

Return type *EncodedNumber*

Raises `ValueError` – You tried to increase the exponent, which can't be done without decryption.

classmethod **`encode`**(*public_key*, *scalar*, *precision=None*, *max_exponent=None*)

Return an encoding of an int or float.

This encoding is carefully chosen so that it supports the same operations as the Paillier cryptosystem.

If *scalar* is a float, first approximate it as an int, *int_rep*:

scalar = int_rep * (`BASE` ** `exponent`),

for some (typically negative) integer exponent, which can be tuned using *precision* and *max_exponent*. Specifically, `exponent` is chosen to be equal to or less than *max_exponent*, and such that the number *precision* is not rounded to zero.

Having found an integer representation for the float (or having been given an int *scalar*), we then represent this integer as a non-negative integer < `n`.

Paillier homomorphic arithemetic works modulo `n`. We take the convention that a number x < n/3 is positive, and that a number x > 2n/3 is negative. The range n/3 < x < 2n/3 allows for overflow detection.

Parameters

- **`public_key`** (`PaillierPublicKey`) – public key for which to encode (this is necessary because `n` varies).
- **`scalar`** – an int or float to be encrypted. If int, it must satisfy abs(*value*) < `n`/3. If float, it must satisfy abs(*value / precision*) << `n`/3 (i.e. if a float is near the limit then detectable overflow may still occur)
- **`precision`** (`float`) – Choose exponent (i.e. fix the precision) so that this number is distinguishable from zero. If *scalar* is a float, then this is set so that minimal precision is lost. Lower precision leads to smaller encodings, which might yield faster computation.
- **`max_exponent`** (`int`) – Ensure that the exponent of the returned *EncryptedNumber* is at most this.

Returns Encoded form of *scalar*, ready for encryption against *public_key*.

Return type *EncodedNumber*

class `phe.paillier.`**`EncryptedNumber`**(*public_key*, *ciphertext*, *exponent=0*)

Bases: `object`

Represents the Paillier encryption of a float or int.

Typically, an *EncryptedNumber* is created by `PaillierPublicKey.encrypt()`. You would only instantiate an *EncryptedNumber* manually if you are de-serializing a number someone else encrypted.

Paillier encryption is only defined for non-negative integers less than `PaillierPublicKey.n`. `EncodedNumber` provides an encoding scheme for floating point and signed integers that is compatible with the partially homomorphic properties of the Paillier cryptosystem:

1. D(E(a) * E(b)) = a + b

2. D(E(a)**b) = a * b

where *a* and *b* are ints or floats, *E* represents encoding then encryption, and *D* represents decryption then decoding.

Parameters

- **public_key** (`PaillierPublicKey`) – the `PaillierPublicKey` against which the number was encrypted.
- **ciphertext** (*int*) – encrypted representation of the encoded number.
- **exponent** (*int*) – used by `EncodedNumber` to keep track of fixed precision. Usually negative.

public_key
PaillierPublicKey

the `PaillierPublicKey` against which the number was encrypted.

exponent
int

used by `EncodedNumber` to keep track of fixed precision. Usually negative.

Raises `TypeError` – if *ciphertext* is not an int, or if *public_key* is not a `PaillierPublicKey`.

__add__(*other*)
Add an int, float, *EncryptedNumber* or *EncodedNumber*.

__mul__(*other*)
Multiply by an int, float, or EncodedNumber.

__radd__(*other*)
Called when Python evaluates *34 + <EncryptedNumber>* Required for builtin *sum* to work.

_add_encoded(*encoded*)
Returns E(a + b), given self=E(a) and b.

Parameters **encoded** (`EncodedNumber`) – an `EncodedNumber` to be added to *self*.

Returns E(a + b), calculated by encrypting b and taking the product of E(a) and E(b) modulo `n` ** 2.

Return type *EncryptedNumber*

Raises `ValueError` – if scalar is out of range or precision.

_add_encrypted(*other*)
Returns E(a + b) given E(a) and E(b).

Parameters **other** (`EncryptedNumber`) – an *EncryptedNumber* to add to self.

Returns E(a + b), calculated by taking the product of E(a) and E(b) modulo `n` ** 2.

Return type *EncryptedNumber*

Raises `ValueError` – if numbers were encrypted against different keys.

_add_scalar(*scalar*)
Returns E(a + b), given self=E(a) and b.

Parameters **scalar** – an int or float b, to be added to *self*.

Returns E(a + b), calculated by encrypting b and taking the product of E(a) and E(b) modulo `n` ** 2.

Return type *EncryptedNumber*

Raises `ValueError` – if scalar is out of range or precision.

_raw_add(*e_a*, *e_b*)
Returns the integer E(a + b) given ints E(a) and E(b).

N.B. this returns an int, not an *EncryptedNumber*, and ignores `ciphertext`

Parameters

- **e_a** (*int*) – E(a), first term
- **e_b** (*int*) – E(b), second term

Returns E(a + b), calculated by taking the product of E(a) and E(b) modulo `n` ** 2.

Return type int

_raw_mul(*plaintext*)
Returns the integer E(a * plaintext), where E(a) = ciphertext

Parameters **plaintext** (*int*) – number by which to multiply the *EncryptedNumber*. *plaintext* is typically an encoding. 0 <= *plaintext* < `n`

Returns Encryption of the product of *self* and the scalar encoded in *plaintext*.

Return type int

Raises

- `TypeError` – if *plaintext* is not an int.
- `ValueError` – if *plaintext* is not between 0 and `PaillierPublicKey.n`.

ciphertext(*be_secure=True*)
Return the ciphertext of the EncryptedNumber.

Choosing a random number is slow. Therefore, methods like `__add__()` and `__mul__()` take a short-cut and do not follow Paillier encryption fully - every encrypted sum or product should be multiplied by r ** `n` for random r < n (i.e., the result is obfuscated). Not obfuscating provides a big speed up in, e.g., an encrypted dot product: each of the product terms need not be obfuscated, since only the final sum is shared with others - only this final sum needs to be obfuscated.

Not obfuscating is OK for internal use, where you are happy for your own computer to know the scalars you've been adding and multiplying to the original ciphertext. But this is *not* OK if you're going to be sharing the new ciphertext with anyone else.

So, by default, this method returns an obfuscated ciphertext - obfuscating it if necessary. If instead you set *be_secure=False* then the ciphertext will be returned, regardless of whether it has already been obfuscated. We thought that this approach, while a little awkward, yields a safe default while preserving the option for high performance.

Parameters **be_secure** (*bool*) – If any untrusted parties will see the returned ciphertext, then this should be True.

Returns an int, the ciphertext. If *be_secure=False* then it might be possible for attackers to deduce numbers involved in calculating this ciphertext.

decrease_exponent_to(*new_exp*)

Return an EncryptedNumber with same value but lower exponent.

If we multiply the encoded value by *EncodedNumber.BASE* and decrement *exponent*, then the decoded value does not change. Thus we can almost arbitrarily ratchet down the exponent of an *EncryptedNumber* - we only run into trouble when the encoded integer overflows. There may not be a warning if this happens.

When adding *EncryptedNumber* instances, their exponents must match.

This method is also useful for hiding information about the precision of numbers - e.g. a protocol can fix the exponent of all transmitted *EncryptedNumber* instances to some lower bound(s).

Parameters **new_exp** (*int*) – the desired exponent.

Returns Instance with the same plaintext and desired exponent.

Return type *EncryptedNumber*

Raises ValueError – You tried to increase the exponent.

obfuscate()

Disguise ciphertext by multiplying by r ** n with random r.

This operation must be performed for every *EncryptedNumber* that is sent to an untrusted party, otherwise eavesdroppers might deduce relationships between this and an antecedent *EncryptedNumber*.

For example:

```
enc = public_key.encrypt(1337)
send_to_nsa(enc)       # NSA can't decrypt (we hope!)
product = enc * 3.14
send_to_nsa(product)   # NSA can deduce 3.14 by bruteforce attack
product2 = enc * 2.718
product2.obfuscate()
send_to_nsa(product)   # NSA can't deduce 2.718 by bruteforce attack
```

class phe.paillier.**PaillierPrivateKey**(*public_key*, *Lambda*, *mu*)

Bases: object

Contains a private key and associated decryption method.

Parameters

- **public_key** (*PaillierPublicKey*) – The corresponding public key.
- **Lambda** (*int*) – private secret - see Paillier's paper.
- **mu** (*int*) – private secret - see Paillier's paper.

public_key

PaillierPublicKey

The corresponding public key.

Lambda

int

private secret - see Paillier's paper.

mu

int

private secret - see Paillier's paper.

decrypt(*encrypted_number*)

Return the decrypted & decoded plaintext of *encrypted_number*.

Uses the default *EncodedNumber*, if using an alternative encoding scheme, use *decrypt_encoded()* or *raw_decrypt()* instead.

Parameters **encrypted_number** (EncryptedNumber) – an *EncryptedNumber* with a public key that matches this private key.

Returns the int or float that *EncryptedNumber* was holding. N.B. if the number returned is an integer, it will not be of type float.

Raises

- `TypeError` – If *encrypted_number* is not an *EncryptedNumber*.
- `ValueError` – If *encrypted_number* was encrypted against a different key.

decrypt_encoded(*encrypted_number*, *Encoding=None*)

Return the *EncodedNumber* decrypted from *encrypted_number*.

Parameters

- **encrypted_number** (EncryptedNumber) – an *EncryptedNumber* with a public key that matches this private key.
- **Encoding** (*class*) – A class to use instead of *EncodedNumber*, the encoding used for the *encrypted_number* - used to support alternative encodings.

Returns The decrypted plaintext.

Return type *EncodedNumber*

Raises

- `TypeError` – If *encrypted_number* is not an *EncryptedNumber*.
- `ValueError` – If *encrypted_number* was encrypted against a different key.

raw_decrypt(*ciphertext*)

Decrypt raw ciphertext and return raw plaintext.

Parameters **ciphertext** (*int*) – (usually from *EncryptedNumber.ciphertext()*) that is to be Paillier decrypted.

Returns Paillier decryption of ciphertext. This is a positive integer < `public_key.n`.

Return type int

Raises `TypeError` – if ciphertext is not an int.

class phe.paillier.**PaillierPrivateKeyring**(*private_keys=None*)

Bases: `collections.abc.Mapping`

Holds several private keys and can decrypt using any of them.

Acts like a dict, supports `del()`, and indexing with **[]**, but adding keys is done using *add()*.

Parameters **private_keys** (*list of PaillierPrivateKey*) – an optional starting list of *PaillierPrivateKey* instances.

add(*private_key*)

Add a key to the keyring.

Parameters **private_key** (PaillierPrivateKey) – a key to add to this keyring.

decrypt (*encrypted_number*)

Return the decrypted & decoded plaintext of *encrypted_number*.

Parameters **encrypted_number** (EncryptedNumber) – encrypted against a known public key, i.e., one for which the private key is on this keyring.

Returns the int or float that *encrypted_number* was holding. N.B. if the number returned is an integer, it will not be of type float.

Raises KeyError – If the keyring does not hold the private key that decrypts *encrypted_number*.

class phe.paillier.**PaillierPublicKey** (*g*, *n*)

Bases: object

Contains a public key and associated encryption methods.

Parameters

- **g** (*int*) – part of the public key - see Paillier's paper.
- **n** (*int*) – part of the public key - see Paillier's paper.

g

int

part of the public key - see Paillier's paper.

n

int

part of the public key - see Paillier's paper.

nsquare

int

n ** 2, stored for frequent use.

max_int

int

Maximum int that may safely be stored. This can be increased, if you are happy to redefine "safely" and lower the chance of detecting an integer overflow.

encrypt (*value*, *precision=None*, *r_value=None*)

Encode and Paillier encrypt a real number *value*.

Parameters

- **value** – an int or float to be encrypted. If int, it must satisfy abs(*value*) < *n*/3. If float, it must satisfy abs(*value* / *precision*) << *n*/3 (i.e. if a float is near the limit then detectable overflow may still occur)
- **precision** (*float*) – Passed to *EncodedNumber.encode()*. If *value* is a float then *precision* is the maximum **absolute** error allowed when encoding *value*. Defaults to encoding *value* exactly.
- **r_value** (*int*) – obfuscator for the ciphertext; by default (i.e. if *r_value* is None), a random value is used.

Returns An encryption of *value*.

Return type *EncryptedNumber*

Raises ValueError – if *value* is out of range or *precision* is so high that *value* is rounded to zero.

encrypt_encoded(*encoding*, *r_value*)

Paillier encrypt an encoded value.

Parameters

- **encoding** – The EncodedNumber instance.
- **r_value** (*int*) – obfuscator for the ciphertext; by default (i.e. if *r_value* is None), a random value is used.

Returns An encryption of *value*.

Return type *EncryptedNumber*

get_random_lt_n()

Return a cryptographically random number less than `n`

raw_encrypt(*plaintext*, *r_value=None*)

Paillier encryption of a positive integer plaintext < `n`.

You probably should be using `encrypt()` instead, because it handles positive and negative ints and floats.

Parameters

- **plaintext** (*int*) – a positive integer < `n` to be Paillier encrypted. Typically this is an encoding of the actual number you want to encrypt.
- **r_value** (*int*) – obfuscator for the ciphertext; by default (i.e. r_value is None), a random value is used.

Returns Paillier encryption of plaintext.

Return type int

Raises `TypeError` – if plaintext is not an int.

phe.paillier.**generate_paillier_keypair**(*private_keyring=None*, *n_length=2048*)

Return a new `PaillierPublicKey` and `PaillierPrivateKey`.

Add the private key to *private_keyring* if given.

Parameters

- **private_keyring** (`PaillierPrivateKeyring`) – a `PaillierPrivateKeyring` on which to store the private key.
- **n_length** – key size in bits.

Returns The generated `PaillierPublicKey` and `PaillierPrivateKey`

Return type tuple

6.2 Utilities

phe.util.**base64_to_int**(*source*)

phe.util.**base64url_decode**(*payload*)

phe.util.**base64url_encode**(*payload*)

phe.util.**getprimeover**(*N*)

Return a random N-bit prime number using the System's best Cryptographic random source.

Use GMP if available, otherwise fallback to PyCrypto phe.util.**int_to_base64**(*source*)

phe.util.**invert**(*a*, *b*)

The multiplicitive inverse of a in the integers modulo b.

Return int x, where a * x == 1 mod b

phe.util.**powmod**(*a*, *b*, *c*)

Uses GMP, if available, to do a^b mod c where a, b, c are integers.

Return int (a ** b) % c

6.3 CLI

phe.command_line.**load_encrypted_number**(*enc_number_file*, *pub*)

phe.command_line.**load_public_key**(*public_key_data*)

phe.command_line.**log**(*m*, *color='red'*)

phe.command_line.**serialise_encrypted**(*enc*)

CHAPTER 7

Compatibility with other libraries

This library may, with *care*, be used with other Paillier implementations.

- Javallier - library for Java/Scala also maintained by NICTA. Somewhat different Encoding scheme. Base of 2 is fixed (see *Alternative Base for EncodedNumber*)
- paillier.js - Early prototype library for Javascript/Typescript

7.1 Alternative Libraries

These are brief notes on the libraries that we looked at before embarking on writing our own.

7.1.1 Python Libraries

charm-crypto

> Charm is a framework for rapidly prototyping advanced cryptosystems. Based on > the Python language, it was designed from the ground up to minimize development > time and code complexity while promoting the reuse of components. > > Charm uses a hybrid design: performance intensive mathematical operations are > implemented in native C modules, while cryptosystems themselves are written in > a readable, high-level language. Charm additionally provides a number of new > components to facilitate the rapid development of new schemes and protocols.

http://charm-crypto.com/Main.html

Paillier Code, Pascal Paillier (Public-Key)

https://github.com/JHUISI/charm/blob/master/charm/schemes/pkenc/pkenc_paillier99.py

Worth looking at their object hierarchy, e.g., http://jhuisi.github.io/charm/toolbox/PKEnc.html They use a Ciphertext class which has the __*add*__ and __*mul*__ methods overridden.

Example:

```
>>> from charm.toolbox.integergroup import RSAGroup
>>> group = RSAGroup()
>>> pai = Pai99(group)
>>> (public_key, secret_key) = pai.keygen()
>>> msg_1=12345678987654321
>>> msg_2=12345761234123409
>>> msg_3 = msg_1 + msg_2
>>> msg_1 = pai.encode(public_key['n'], msg_1)
>>> msg_2 = pai.encode(public_key['n'], msg_2)
>>> msg_3 = pai.encode(public_key['n'], msg_3)
>>> cipher_1 = pai.encrypt(public_key, msg_1)
>>> cipher_2 = pai.encrypt(public_key, msg_2)
>>> cipher_3 = cipher_1 + cipher_2
>>> decrypted_msg_3 = pai.decrypt(public_key, secret_key, cipher_3)
>>> decrypted_msg_3 == msg_3
True
```

They have even got it going on Android: http://jhuisi.github.io/charm/mobile.html

mikeivanov/paillier

> Pure Python Paillier Homomorphic Cryptosystem

Very simple easy to understand code. Doesn't use a Paillier object. No external dependencies. Based on the java library: https://code.google.com/p/thep/

https://github.com/mikeivanov/paillier

Example Usage:

```
>>> from paillier import *
>>> priv, pub = generate_keypair(128)
>>> x = encrypt(pub, 2)
>>> y = encrypt(pub, 3)
>>> x,y
(72109737005643982735171545918..., 9615446835366886883470187...)
>>> z = e_add(pub, x, y)
>>> z
71624230283745591274688669...
>>> decrypt(priv, pub, z)
5
```

Tests:

Could easily be reused.

https://github.com/mikeivanov/paillier/blob/master/tests/test_paillier.py

encrypted-bigquery-client

License: **Apache 2.0**

> Paillier encryption to perform homomorphic addition on encrypted data

The ebq client is an experimental client which encrypts data in the specified fields before loading to Bigquery service. Currently there are various limitations including support for only a subset of query types on encrypted data.

Paillier specific code:

http://pydoc.net/Python/encrypted_bigquery/1.0/paillier/

Uses openssl via *ctypes*.

Features a **Paillier** object with the following methods:

- *__init__(seed=None, g=None, n=None, Lambda=None, mu=None)*
- *Encrypt(plaintext, r_value=None)*
- *Decrypt(ciphertext)*
- *Add(ciphertext1, ciphertext2)* - returns E(m1 + m2) given E(m1) and E(m2)
- *Affine(self, ciphertext, a=1, b=0)* - Returns E(a*m + b) given E(m), a and b
- *EncryptInt64/DecryptInt64* - twos complement to allow negative addition
- *EncryptFloat/DecryptFloat* - IEEE754 binary64bit where exponent <= 389

Code is well documented python2. Most arguments are *long* or *int* types. There is also a comprehensive unit test at http://pydoc.net/Python/encrypted_bigquery/1.0/paillier_test/

Even if we don't reuse any of their code the tests would be great.

#### Floating point notes in code:

Paillier homomorphic addition only directly adds positive binary values, however, we would like to add both positive and negative float values of different magnitudes. To achieve this, we will:

- represent the mantissa and exponent as one long binary value. This means that with 1024 bit n in paillier, the maximum exponent value is 389 bits.
- represent negative values with twos complement representation.
- Nan, +inf, -inf are each indicated by values in there own 32 bit region, so that when one of them is added, the appropriate region would be incremented and we would know this in the final aggregated value, assuming less than 2^32 values were aggregated.
- We limit the number of numbers that can be added to be less than 2^32 otherwise we would not be able to detect overflows properly, etc.
- Also, in order to detect overflow after adding multiple values, the 64 sign bit is extended (or replicated) for an additional 64 bits. This allows us to detect if an overflow happened and knowing whether the most significant 32 bits out of 64 is zeroes or ones, we would know if the result should be a +inf or -inf.

Project Home: https://code.google.com/p/encrypted-bigquery-client/

7.1.2 C/C++

Encounter

> Encounter is a software library aimed at providing a production-grade > implementation of cryptographic counters

To date, Encounter implements a cryptocounter based on the Paillier public-key cryptographic scheme

https://github.com/secYOUre/Encounter

FNP privacy-preserving set intersection protocol

A toolchain and library for privacy-preserving set intersection

It comes with rudimentary command-line interface: client, server, and key-generation tool. Extension and reuse is possible through C++ interfaces. The implementation is fully thread-aware and multi-core ready, thus computation time can be shortened by modern many-core machines. We have verified significant performance gains with quad-core Xeons and Opterons, through the use of bucket allocation in the algorithm.

For homomorphic encryption and decryption, both modified ElGamal cryptosystem and **Paillier cryptosystem** have been implemented on top of gmp. And yes, the source of randomness is always a headache for cryptosystem implementers; we have keyboard, file and network packet as the sources of entropy.

It requires OpenSSL, gmp, gmpxx, boost, pthread, and pcap to build. It currently runs on Linux.

http://fnp.sourceforge.net/

libpaillier

Library written in C and uses GMP. The privss toolkit for private stream searching is built on libpaillier.

http://hms.isi.jhu.edu/acsc/libpaillier/

### HElib

> HElib is a software library that implements homomorphic encryption (HE). > Currently available is an implementation of the Brakerski-Gentry-Vaikuntanathan > (BGV) scheme, along with many optimizations to make homomorphic evaluation runs > faster, focusing mostly on effective use of the Smart-Vercauteren ciphertext > packing techniques and the Gentry-Halevi-Smart optimizations. > > At its present state, this library is mostly meant for researchers working on > HE and its uses. Also currently it is fairly low-level, and is best thought of > as "assembly language for HE". That is, it provides low-level routines (set, add, > multiply, shift, etc.), with as much access to optimizations as we can give. > Hopefully in time we will be able to provide higher-level routines.

https://github.com/shaih/HElib

Must read: http://tommd.github.io/posts/HELib-Intro.html

rinon/Simple-Homomorphic-Encryption

Another C++ fully homomorphic encryption implementation.

https://github.com/rinon/Simple-Homomorphic-Encryption

7.1.3 Javascript

Javascript Cryptography Considered Harmful - http://www.matasano.com/articles/javascript-cryptography/

mhe/jspaillier

Adds the methods to the Public and Private keys.

Dependencies: jsbn Demo Site: http://mhe.github.io/jspaillier/

p2p-paillier

> allows a peer to add two numbers over a peer-to-peer network. Peers add > these two numbers without even knowing what they are. It uses Firebase > (which is centralized) in order to push commands to the peers.

Demo: http://9ac345a5509a.github.io/p2p-paillier/ Code: https://github.com/9ac345a5509a/p2p-paillier

7.1.4 Haskell

There is a decent-looking haskell paillier library: https://github.com/onemouth/HsPaillier

BSD license

There's just one test, which encrypts 37, decrypts it, and checks that it's still 37.

7.1.5 Java

There are a bunch of paillier libraries for java.

Are there any tests?

UT Dallas

This one has documentation and two implementations:

https://www.utdallas.edu/~mxk093120/paillier/javadoc/paillierp/package-summary.html

> Provides the structures and methods to encrypt data with the Paillier encryption scheme with thresholding. This package a simplified implementation of what is specified in the paper A Generalization of Paillier's Public-Key System with Applications to Electronic Voting by Damgård et al. Within this paper, the authors generalize the Paillier encryption scheme to permit computations modulo ns+1, allowing block length for encryption to be chosen freely. In addition to this undertaking, Damgård et al. also constructed a threshold variant of the scheme.
>
> **This package provides the following features of the paper**
>
> - The degree of n is fixed to 1.
> - A fully functional simple Paillier encryption scheme with separate key classes for easy keysharing.
> - Proper Thresholding for an arbitrary number of decryption servers and threshold needed to decrypt.
> - Non-interactive zero knowledge proofs to ensure proper encryption and decryption.
>
> Of particular note, this implementation is simple as s is fixed to be 1. This allows for simplicity at this stage of the design. Further, we hope to have added methods which would make the actual use of this package to be easy and flexible.
>
> Future features would include support for encrypting arbitrary length strings/byte arrays to avoid padding issues.

BGU Crypto course

This one is also documented but is for a crypto course so I'm not sure how complete/practical it is intended to be. For example, it does its own keygen using *java.util.Random*. https://code.google.com/p/paillier-cryptosystem/

UMBC

This one is mercifully short but doesn't implement add, multiply as functions or methods. Also it uses *java.util.Random*.

http://www.csee.umbc.edu/~kunliu1/research/Paillier.html

CHAPTER 8

Example

```
>>> from phe import paillier
>>> public_key, private_key = paillier.generate_paillier_keypair()
>>> secret_number_list = [3.141592653, 300, -4.6e-12]
>>> encrypted_number_list = [public_key.encrypt(x) for x in secret_number_list]
>>> [private_key.decrypt(x) for x in encrypted_number_list]
[3.141592653, 300, -4.6e-12]
```

See *Usage* for more extensive examples taking advantage of the homomorphic properties of the *paillier* cryptosystem.

Documentation generated

June 02, 2016

Python Module Index

Index