CS789 Project Report

Jacob Komissar

December 10, 2024

Contents

1 Usage

To install my program, clone the git repository and install Python 3.12 or newer (previous versions of Python will not work[1].

[1] I used new language features for type hints, so 3.11 will get syntax errors

To use it, run a python interpreter in the `src` directory, and import from the `homework` module.[2]

```
~$ dir=cryptograpy
~$ git clone https://github.com/jirassimok/roll-your-own-crypto.git "$dir"
~$ cd "$dir/src"
~/cryptograpy/src$ python3.12
>>> # These are examples of ways you can import functions from the modules.
>>> import homework
>>> from homework import discrete_log, primitive_root
>>> from homework import *
```

I have carefully documented the most important functions and classes in each module, lists of which can be viewed at the top of each module's file or in the `DESCRIPTION` section of the module's help in Python REPL.

```
>>> import homework
>>> help(homework)
>>> help(homework.euclid)  # submodule help
>>> help(homework.euclid.euclid)  # function help
```

The individual documentation for the functions and classes can be found in the same way in the REPL, or in the source code as a string immediately after their declarations. See the appendices for a list of the key functions in the root module and an example of a function's documentation.

If you want to try any of the functions in particular, please consult the examples of usage below to understand the basic calling conventions, and the help strings for the particulars of each function.

If you install the development tools (see 2.2), you can use the `ipython` interpreter, which provides nicer tab completion and also lets you view the documentation using the special `?` operator.

```
(venv) ~/cryptograpy/src$ ipython
In [1]: import homework
In [2]: homework.primitive_root?
```

[2]Using the project root directory will also work; you'll just have to import from `src.homework` instead.

1.1 Extended usage example

Here is an extended example of the usage of my code, covering a few of the basic

```
>>> import homework
>>> from homework import *
>>> # For a list of the imports from that *, consult one of these:
>>> help(homework)  # Module list and extended docs for all contents.
>>> homework.__all__  # List of attributes imported by *
>>>
>>> discrete_log(101, base=26, modulus=137) \
... # Also available as homework4.bsgs_log
22
>>> pow(26, 22, 137)  # also available as fastexp.fastexp
101
>>> # Documented keyword parameters must be given by name, as below:
>>> fastexp.fastexp(26, 22, 137, verbose=True)
101
>>> rng = BlumBlumShub(62423, 49523, seed=1411122231)
>>> next(rng)
1
>>> rng.next_bit()
0
>>> rng.next_int(12)
3267
>>> p = random_prime(16, rng)
>>> q = random_prime(16, rng)
>>> privkey, pubkey = rsa.keygen(p, q, 65537)
>>> ciphertext = rsa.encrypt(pubkey, 1234567890)
>>> rsa.decrypt(privkey, ciphertext)
1234567890
>>>
>>> p = random_prime(32, rng)
>>> base = primitive_root(p, smallest=False) # get a random root
>>> sender = ElGamal(p, base, 123341151)
>>> sender_key = sender.publish_key()
>>>
>>> recipient = ElGamal(sender_key.prime, sender_key.base, 55152557)
>>> recipient_key = recipient.publish_key()
>>>
```

```
>>> ciphertext = sender.encrypt(recipient_key.power, 987654321)
>>> recipient.decrypt(sender_key.power, ciphertext)
987654321
>>>
>>> # Some functions return iterators or generators; you can make those
>>> # into lists like this:
>>> list(factors(2*5*13*29))
[2, 5, 13, 29]
```

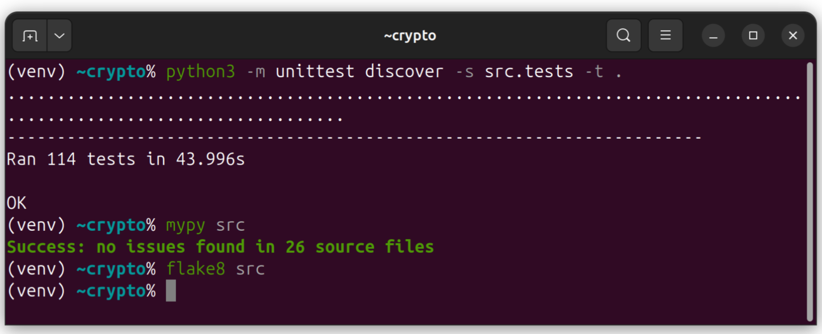
2 Testing

I wrote extensive tests for my algorithms using Python's `unittest` library. These tests are in the `src/tests`

To run the unit tests, use this command:

```
~/cryptograpy$ python3.12 -m unittest discover -s src.tests -t .
```

I also used the flake8 tool to keep my code conforming to the canonical Python style guide, and mypy to statically check types to help ensure I always used functions correctly. Figure 2 shows the outputs of all three tools indicating no issues[fn:newtests:There are actually a few more tests than the 114 named shown in the image, both because I added some after taking that screenshot, and because `unittest` doesn't count sub-tests separately.



2.1 Coverage

I also used the popular Python coverage tool coverage.py to monitor my test coverage. Here's the summary from my final coverage report:

File	statements	missing	excluded	coverage
src/homework/__init__.py	10	0	0	100%
src/homework/bit_class.py	99	34	7	66%
src/homework/bititer.py	96	22	1	77%
src/homework/cache_util.py	25	2	0	92%
src/homework/elgamal.py	33	0	3	100%
src/homework/euclid.py	212	0	0	100%
src/homework/factor.py	102	10	2	90%
src/homework/fastexp.py	170	0	0	100%
src/homework/homework4.py	99	0	2	100%
src/homework/pseudoprime.py	73	1	0	99%
src/homework/pseudorandom.py	196	33	13	83%
src/homework/randprime.py	16	10	0	38%
src/homework/rsa.py	41	5	0	88%
src/homework/sieve.py	119	0	0	100%
src/homework/util.py	88	17	29	81%
Total	1379	134	57	90%

The majority of the un-covered code falls into two categories:

- Code I implemented as part of an API that I did not use or test (like large parts of `bititer.py` and `bit_class.py`).
- Places where I added extra handling for bad inputs (like negative numbers where primes are expected), but didn't test those bad inputs (as in `rsa.py`).

If present, the `htmlcov` directory also contains a full test coverage report; see `htmlcov/index.html`.

2.2 Tools

To generate a new a coverage report, you have to first install the coverage tool then run these commands:

```
~/cryptograpy$ coverage run -m unittest discover -s src.tests -t .
~/cryptograpy$ coverage html
```

This will generate a directory called `htmlcov` containing the coverage report.

To install coverage.py, as well as mypy, flake8, and the other libraries I used while developing this code, you can install the libraries listed in `requirements.txt`.

```
~/cryptograpy$ python3.12 -m pip install -r requirements.txt
```

If the above command fails with "no module named pip," try creating and using a virtual environment:

```
~/cryptograpy$ venvdir=./venv
~/cryptograpy$ python3.12 -m venv "$venvdir"
~/cryptograpy$ . "$venvdir"/bin/activate
(venv) ~/cryptograpy/src$ pip install -r requirements.txt
```

3 Example exchanges

Note that in the ElGamal exchanges, I included a function `prime3mod4`, based on `pseudorandom.random_prime`. After the ElGamal exchanges, I moved `random_prime` to the `randprime` module along with the function to generate primes that are 3 mod 4.

For each part of each exchange, I include two images: one of the public transmission medium (a Zoom chat window), and one of the work I did to play my role in code.[3]

In each exchange where I needed a random number, I used the system's random number generation to generate two (32-bit) primes that I used to set up a Blum-Blum-Shub PRNG that I then seeded with a random number generated by mashing my numpad.[4]

I then used the Blum-Blum-Shub PRNG to generate the numbers used in the exchanges.

3.1 ElGamal

3.1.1 Alice

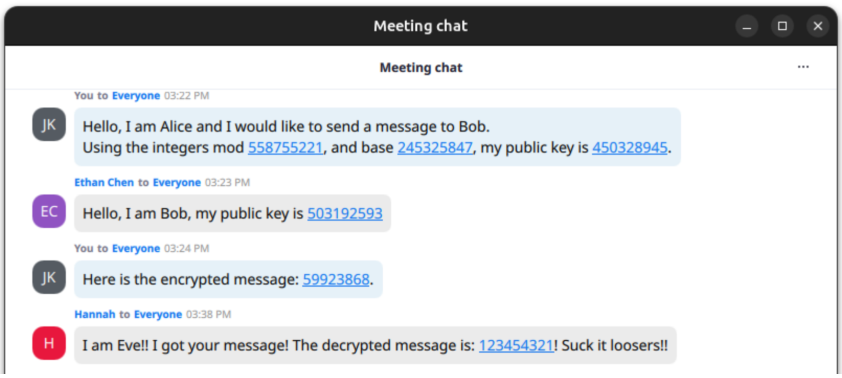
As Alice using ElGamal, I generated the shared prime and primitive root (and my own keys), recieved a public key from Bob, and used those numbers to encrypt a message for Bob.

These are the numbers I used (the prime is 30 bits):

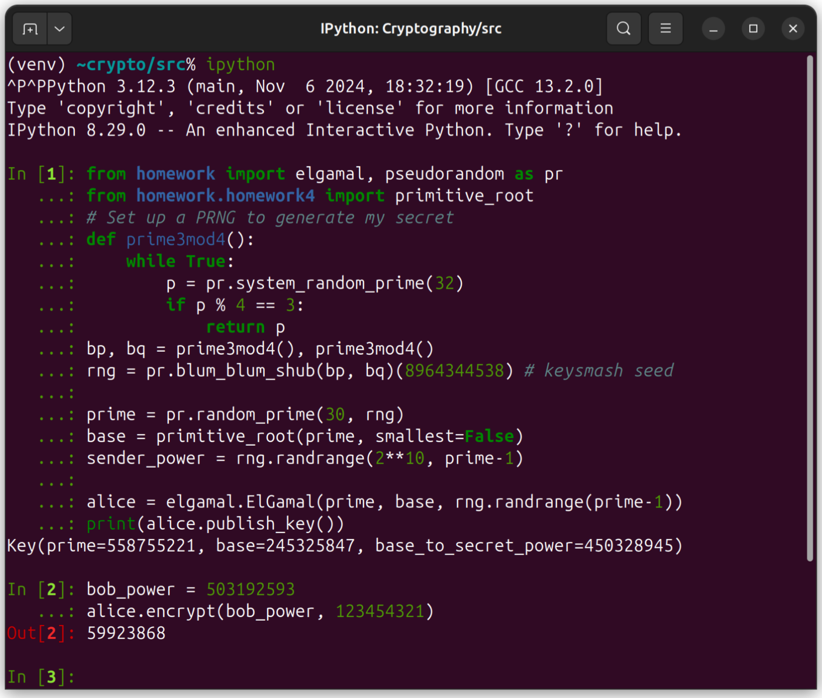
[3]Note that I modified my code slightly after taking these screenshots; the field visible as `base_to_secret_power` is now named `power`.

[4]I also added the functions I used to generate those initial primes in the `randprime` module, rather than the `pseudoprime` module I imported them from in the screenshots.

Prime	558755221
Primitive root	245325847
Alice's secret exponent	396825982[5]
Alice's public power	450328945
Bob's public power	503192593
Message	123454321
Encrypted message	59923868



[5] I didn't actually know what my secret exponent was during the exchange because I used a random value that I didn't print; to find it for this table, I had to take the discrete log of my public key.

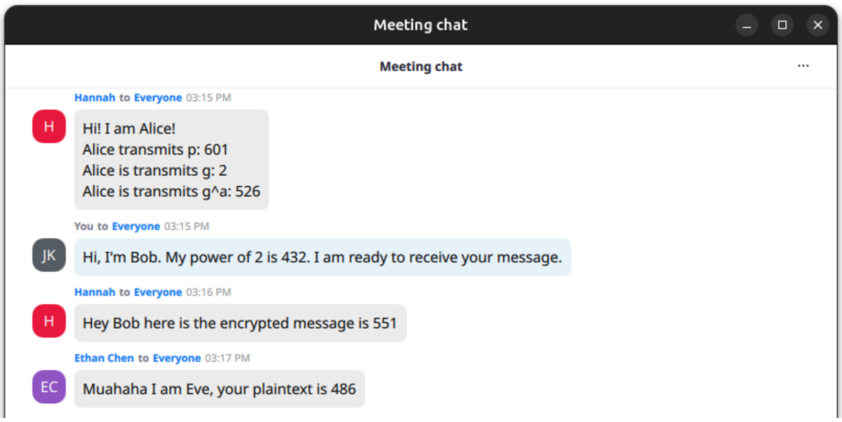


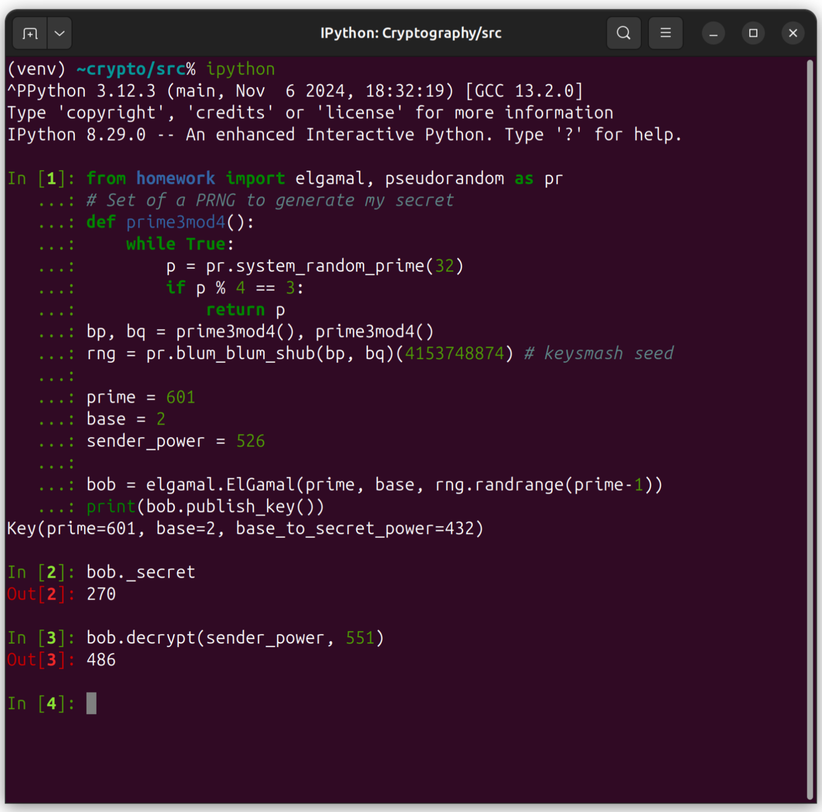
```
IPython: Cryptography/src
(venv) ~crypto/src% ipython
^P^PPython 3.12.3 (main, Nov  6 2024, 18:32:19) [GCC 13.2.0]
Type 'copyright', 'credits' or 'license' for more information
IPython 8.29.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from homework import elgamal, pseudorandom as pr
   ...: from homework.homework4 import primitive_root
   ...: # Set up a PRNG to generate my secret
   ...: def prime3mod4():
   ...:     while True:
   ...:         p = pr.system_random_prime(32)
   ...:         if p % 4 == 3:
   ...:             return p
   ...: bp, bq = prime3mod4(), prime3mod4()
   ...: rng = pr.blum_blum_shub(bp, bq)(8964344538) # keysmash seed
   ...:
   ...: prime = pr.random_prime(30, rng)
   ...: base = primitive_root(prime, smallest=False)
   ...: sender_power = rng.randrange(2**10, prime-1)
   ...:
   ...: alice = elgamal.ElGamal(prime, base, rng.randrange(prime-1))
   ...: print(alice.publish_key())
Key(prime=558755221, base=245325847, base_to_secret_power=450328945)

In [2]: bob_power = 503192593
   ...: alice.encrypt(bob_power, 123454321)
Out[2]: 59923868

In [3]:
```

3.1.2 Bob

As Bob using ElGamal, I received the shared prime, primitive root, and public key from Alice, generated my own keys, and sent my public key to Alice. Then, I recieved a ciphertext from Alice that I decypted using my private key.

Prime	601
Primitive root	2
Alice's public power	526
Bob's secret exponent	270
Bob's public power	432
Ciphertext	551
Decrypted ciphertext	586

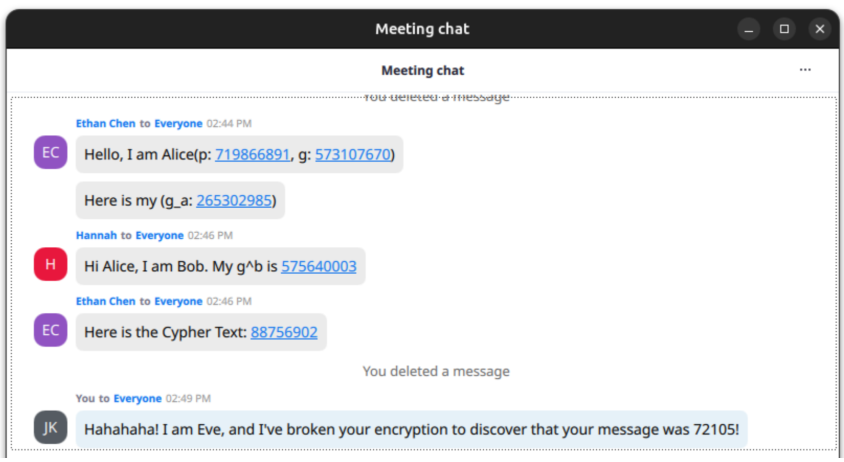

Meeting chat
Meeting chat
Hannah to Everyone 03:15 PM
Hi! I am Alice!
Alice transmits p: 601
Alice is transmits g: 2
Alice is transmits g^a: 526
You to Everyone 03:15 PM
Hi, I'm Bob. My power of 2 is 432. I am ready to receive your message.
Hannah to Everyone 03:16 PM
Hey Bob here is the encrypted message is 551
Ethan Chen to Everyone 03:17 PM
Muahaha I am Eve, your plaintext is 486

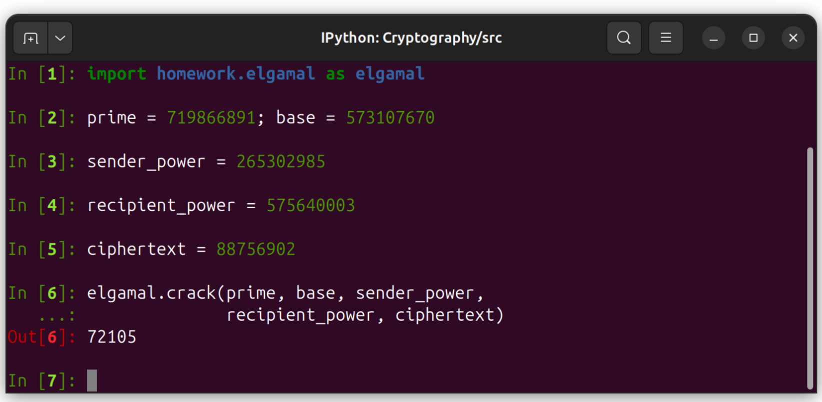

IPython: Cryptography/src
(venv) ~crypto/src% ipython
^PPython 3.12.3 (main, Nov 6 2024, 18:32:19) [GCC 13.2.0]
Type 'copyright', 'credits' or 'license' for more information
IPython 8.29.0 -- An enhanced Interactive Python. Type '?' for help.
In [1]: from homework import elgamal, pseudorandom as pr
   ...: # Set of a PRNG to generate my secret
   ...: def prime3mod4():
   ...:   while True:
   ...:   p = pr.system_random_prime(32)
   ...:   if p % 4 == 3:
   ...:   return p
   ...: bp, bq = prime3mod4(), prime3mod4()
   ...: rng = pr.blum_blum_shub(bp, bq)(4153748874) # keysmash seed
   ...:
   ...: prime = 601
   ...: base = 2
   ...: sender_power = 526
   ...:
   ...: bob = elgamal.ElGamal(prime, base, rng.randrange(prime-1))
   ...: print(bob.publish_key())
Key(prime=601, base=2, base_to_secret_power=432)
In [2]: bob._secret
Out[2]: 270
In [3]: bob.decrypt(sender_power, 551)
Out[3]: 486
In [4]:

3.1.3 Eve

As Eve attacking ElGamal, I observed Alice and Bob's prime, primitive root, public keys, and ciphertext in the public channel, and used them to decrypt the hidden message.

Prime	719866891
Primitive root	573107670
Alice's public power	265302985
Bob's public power	575640003
Ciphertext	88756902
Decrypted ciphertext	72105



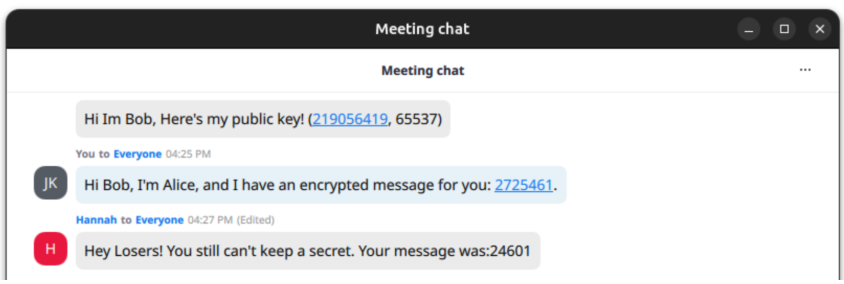


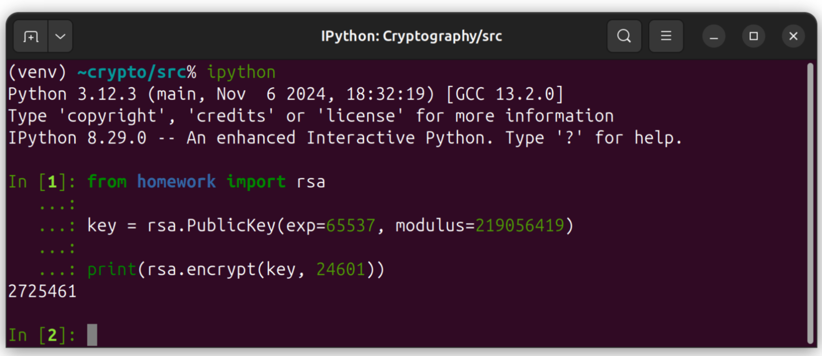
3.2 RSA

3.2.1 Alice

As Alice using RSA, I received Bob's public key (a large product of primes and encryption exponent), used it to encrypt a message, and sent the ciphertext to Bob.

Public modulus (n)	219056419
Public encryption exponent (e)	65537
Message	24601
Encrypted message	2725461





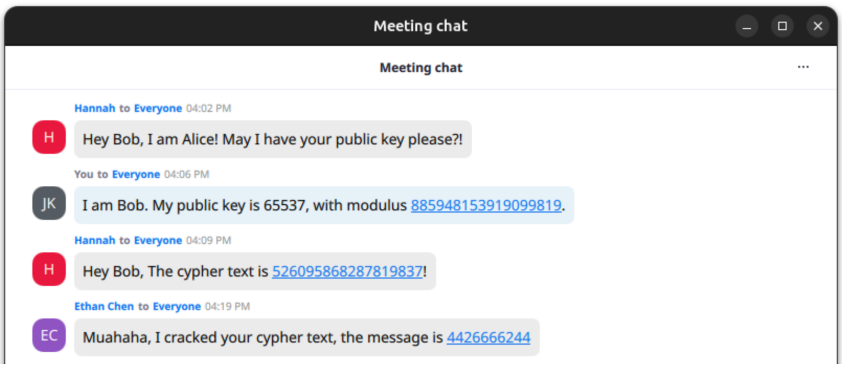
3.2.2 Bob

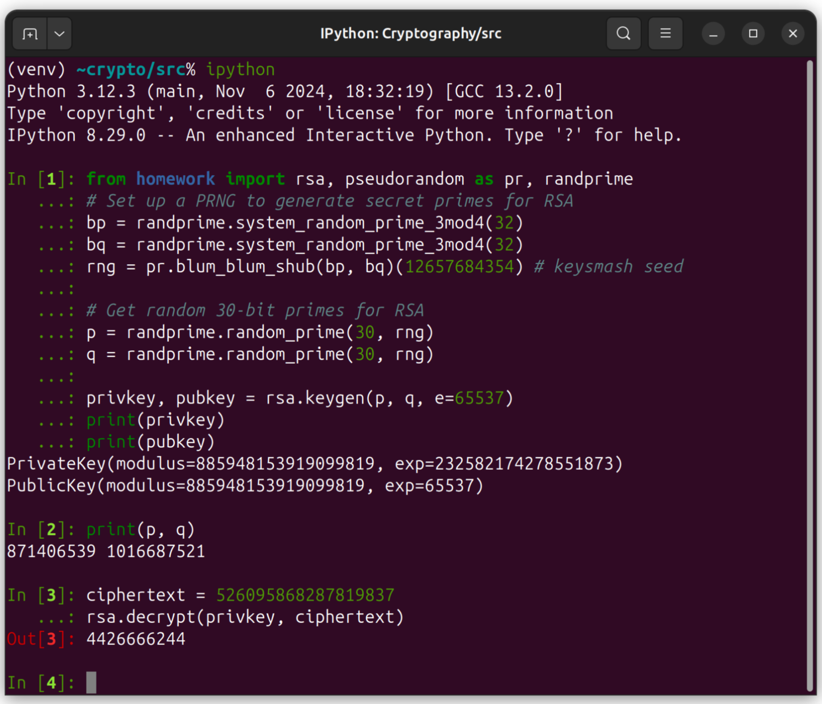
As Bob using RSA, I generated a large prime, chose a public key, and generated a secret key, recieved a ciphertext from Alice, and decrypted it.

This is the one case where I did not generate all of my parameters randomly, instead choosing the standard value of 65537 for my public key (as my entire group did).

I chose random 30-bit primes for p and q (and got a 60-bit n and 58-bit d).

p	871406539
q	1016687521
Public modulus (n)	885948153919099819
Public encryption exponent (e)	65537
Private decryption exponent (d)	232582174278551873
Ciphertext	526095868287819837
Decrypted ciphertext	4426666244

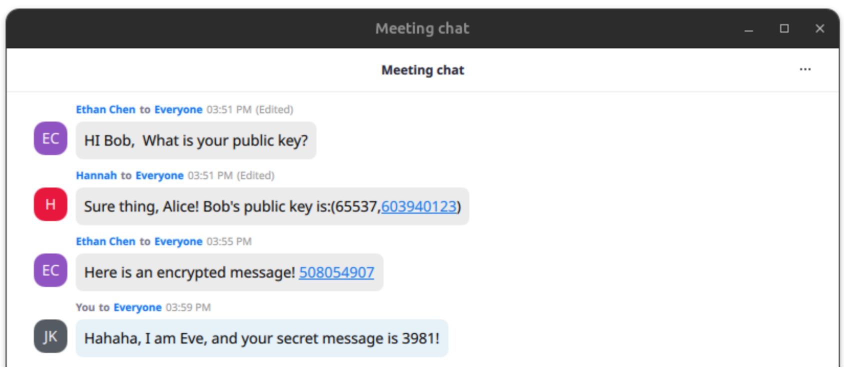


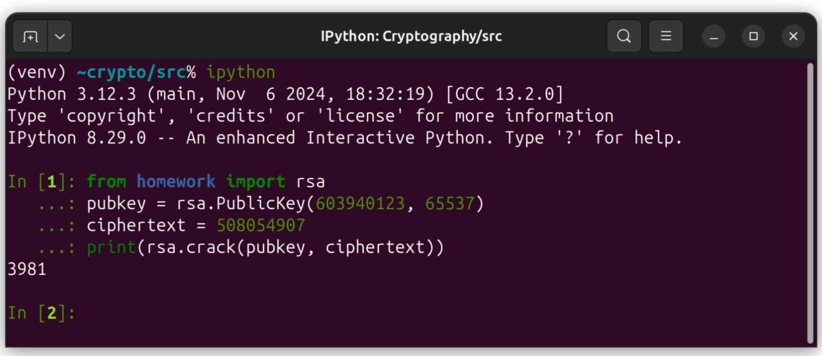


3.2.3 Eve

As Eve attacking RSA, I observed Alice's modulus and encryption exponent, as well as the encrypted message from Bob, and used Pollard's rho algorithm to factor n, allowing me to recreate Alice's decryption key and decrypt the message.

Public modulus (n)	603940123
Public encryption exponent (e)	65537
Ciphertext	508054907
Decrypted ciphertext	3981





4 Appendix: Key function listing

This is the list of key functions, classes and modules exported by the root module. For convenience, I've included the basic parameter lists for some of the functions as well. Refer to their documentation for full parameter lists, or for functions without parameters listed here.

Please refer to the documentation for the root module (found in `src/homework/__init__.py`) for the remaining exports, including the various submodules that actually define the functions.

- Encryption systems
  - `rsa` (actually a module)
    * `rsa.keygen(p, q, e)`
    * `rsa.encrypt(key, m)`
    * `rsa.decrypt(key, c)`
    * `rsa.crack`
  - `ElGamal(prime, base, secret)` (class constructor)
    * `ElGamal.publishkey` (instance method)
    * `ElGamal.encrypt` (instance method)
    * `ElGamal.decrypt` (instance method)
  - `crack_elgamal`
- General algorithms
  - `gcd`
  - `ext_euclid`
  - `pow(base, exp, mod)`
  - `primitive_root`
  - `is_primitive_root`
  - `discrete_log(power, base, mod)`
  - `strong_prime_test`
  - `is_prime`
  - Factorization
    * `find_factor_rho`
    * `find_factor_pm1`
    * `factors`

    * `unique_factors`

- PRNGs
  - `blum_blum_shub`
  - `BlumBlumShub` (class)
  - `naor_reingold`
  - `NaorReingold` (class)
- Additional utilities
  - `random_prime`
  - `random_prime_3mod4`
  - `system_random_prime`
  - `system_random_prime_3mod4`

5 Appendix: Example documentation

Here is the documentation for my primary implementation of the extended Euclidean algorithm, as an example:

```
Find GCD and coefficients using the Extended Euclidean algorithm.

Given m and n, returns g, s, and t, such that g is the greatest common
divisor of m and n, and m*s + n*t == g.

Parameters
----------
(m) : int
(n) : int

Keyword parameters
------------------
verbose : bool, optional
    If false, print nothing. If true, or if not given and util.VERBOSE
    is true, print the steps of the algorithm in a table-like format.
```