Project report

jacob

December 9, 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].

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

```
~$ dir=jik-crypto
```

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

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

I have carefully documented the most important functions and classes in each module, 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)  # module help
>>> help(homework.euclid.euclid)  # function help
```

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.

1.1 Example documentation

Here's 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.
```

1.2 Key object listing

This is the list of key functions in the root module. For conveinence, I've included the basic parameter lists for some of the functions as well.[2]

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

[2]Refer to their documentation for full parameter lists, or for functions without parameters listed here.

        * 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

The root module also exports various submodules, which are described in the module's documentation.

1.3 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)
```

```
>>>
>>> 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)
>>>
>>>
```

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:

```
~/jik-crypto$ 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 1 shows the outputs of all three tools indicating no issues.

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]

[3]Note that I modified my code slightly after taking these screenshots; the field visible

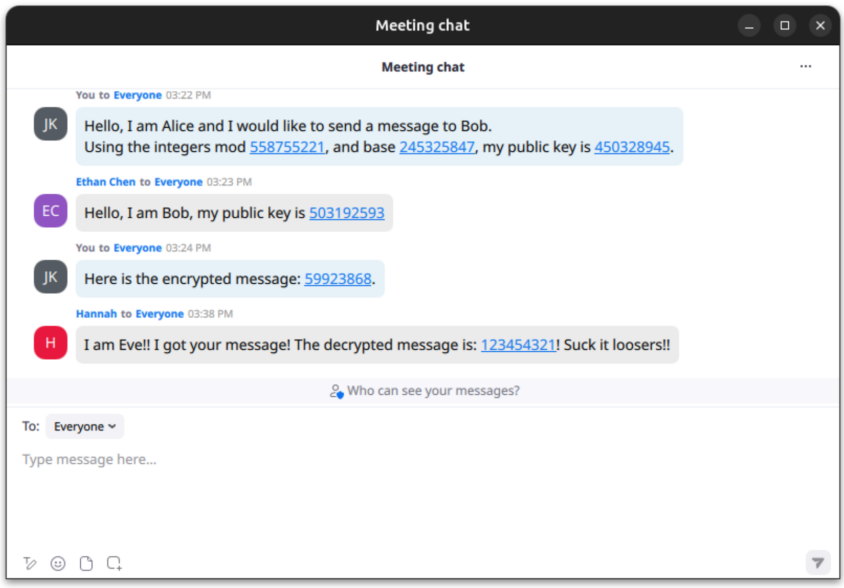

Figure 1: ElGamal Alice (sender): public channel

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):

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

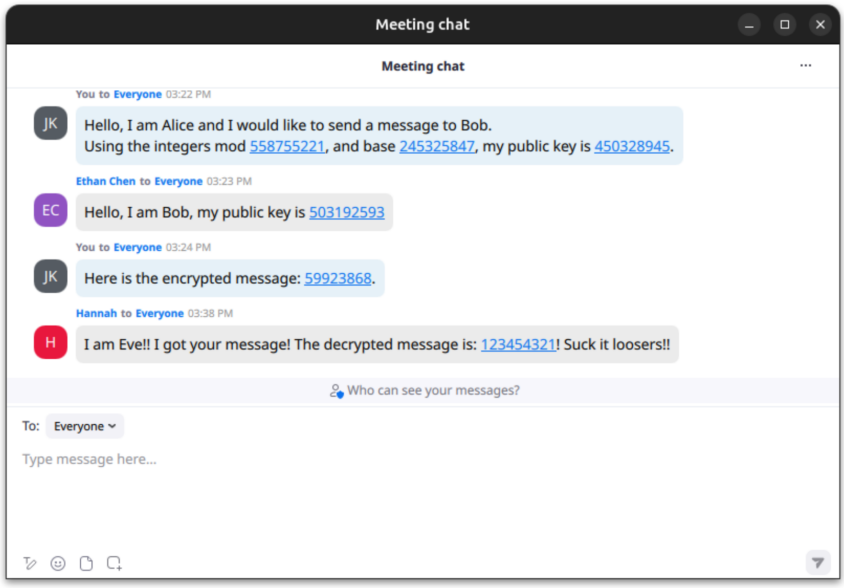

Figure 2: ElGamal Alice (sender): public channel

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.

[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.

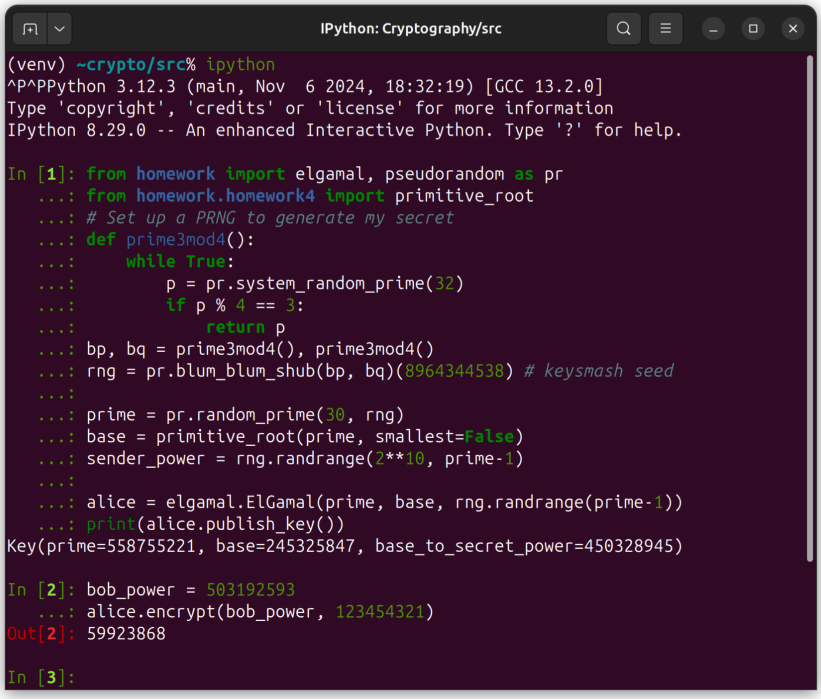

Figure 3: ElGamal Alice (sender): private computation

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

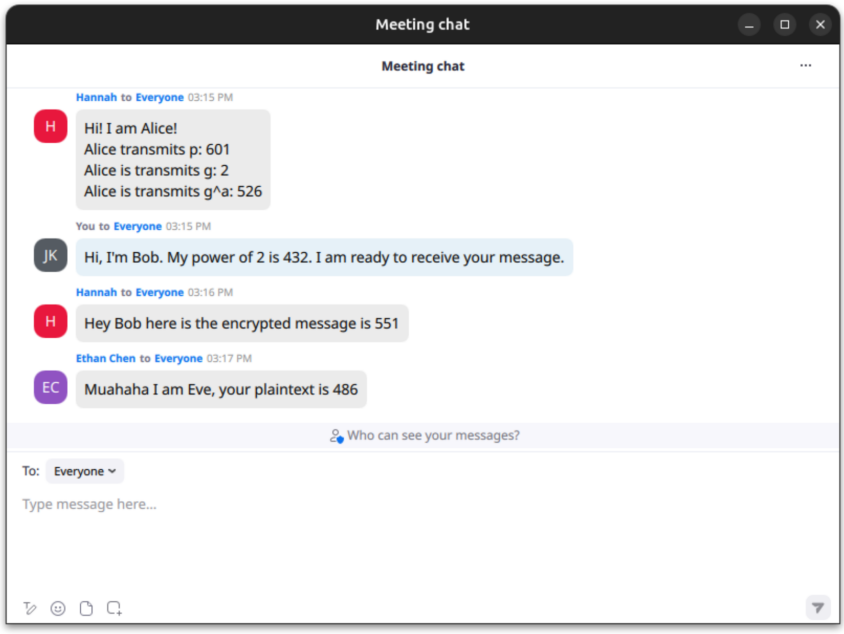

Figure 4: ElGamal Bob (recipient): public channel

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

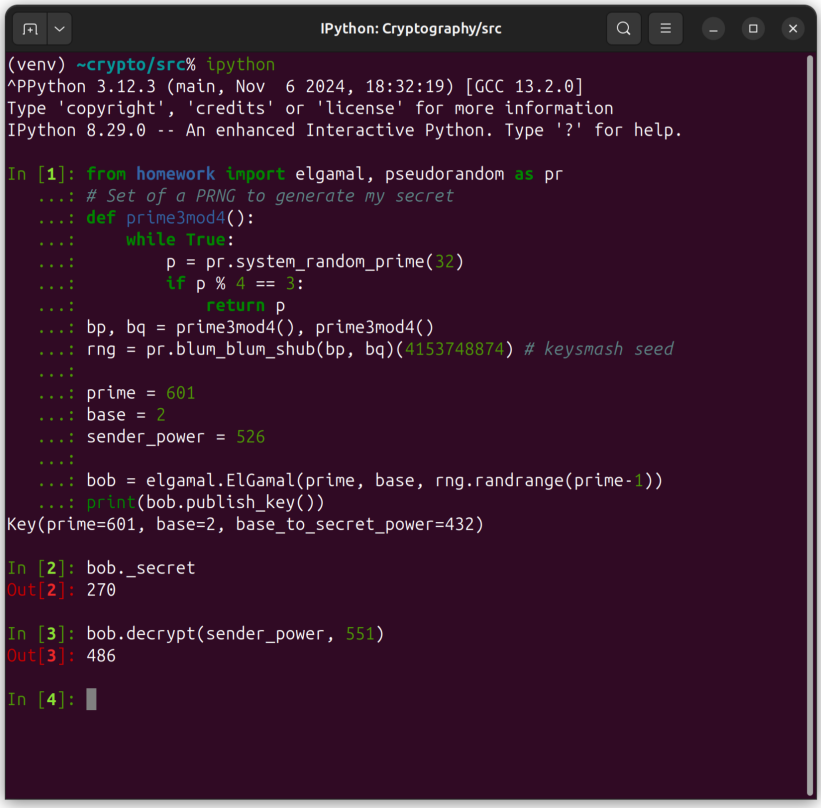

Figure 5: ElGamal Bob (recipient): private computation

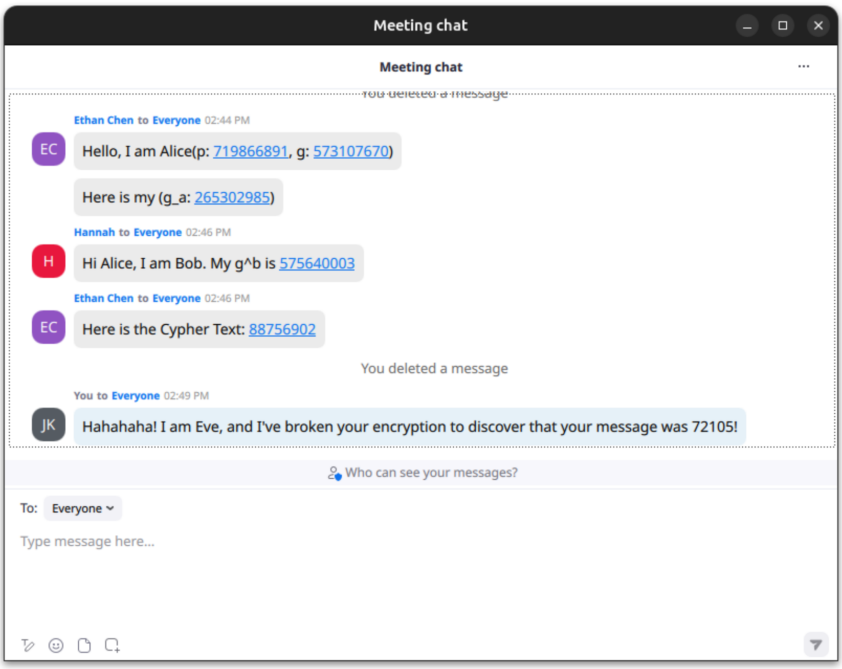

Figure 6: ElGamal Eve: public channel

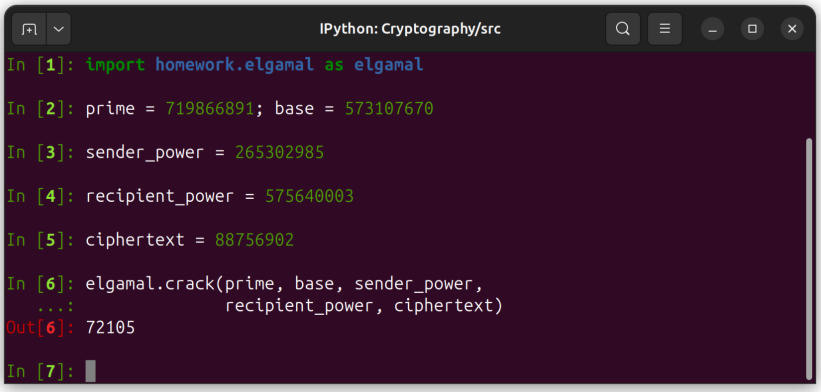

Figure 7: ElGamal Eve: private computation

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

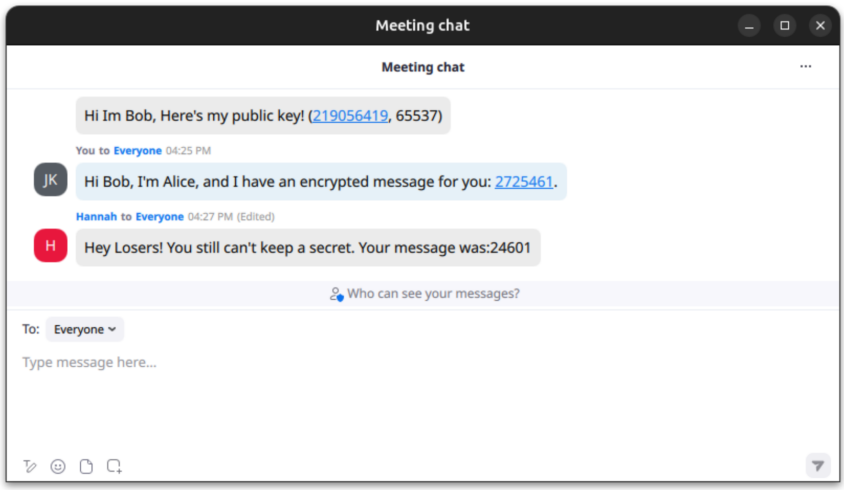

Figure 8: RSA Alice (recipient): public channel

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).

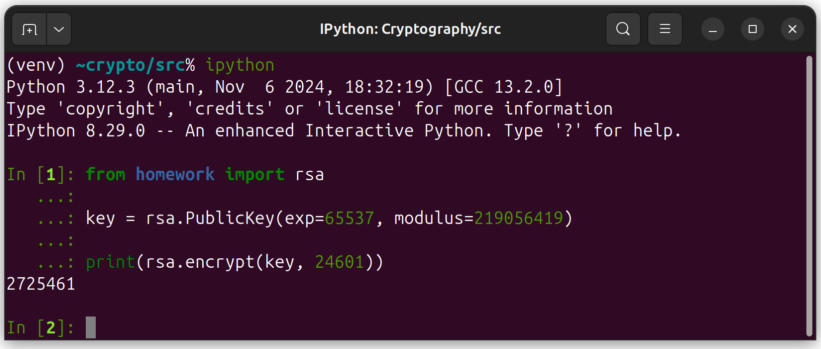

Figure 9: RSA Alice (recipient): private computation

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

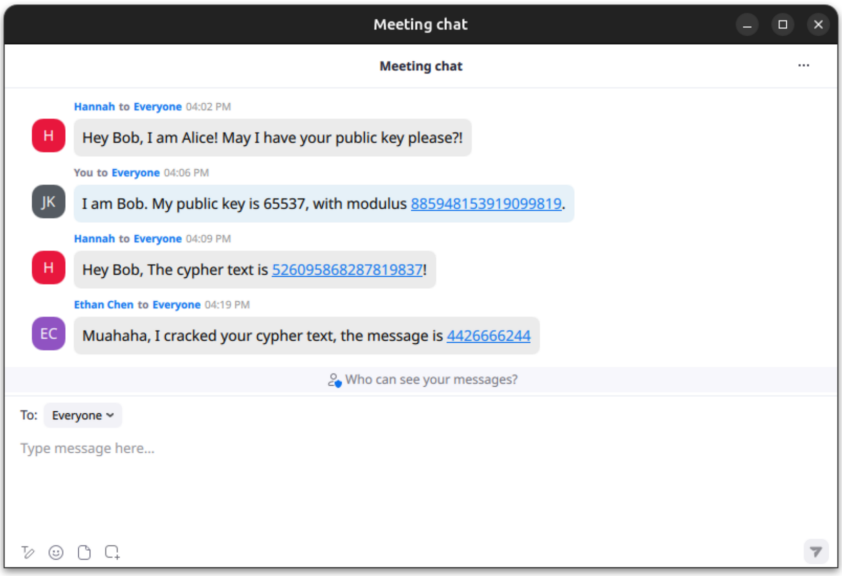

Figure 10: RSA Bob (sender): public channel

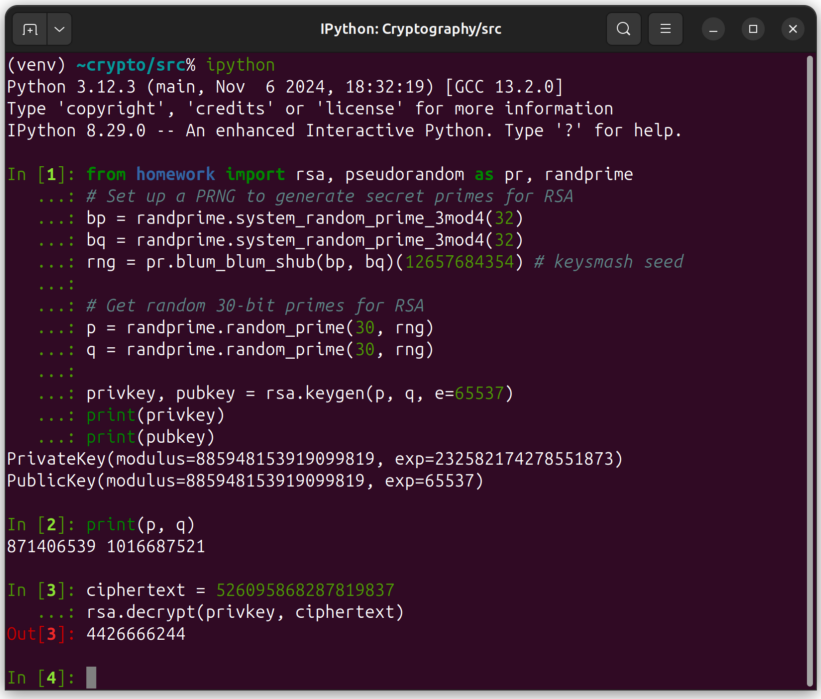

Figure 11: RSA Bob (sender): private computation

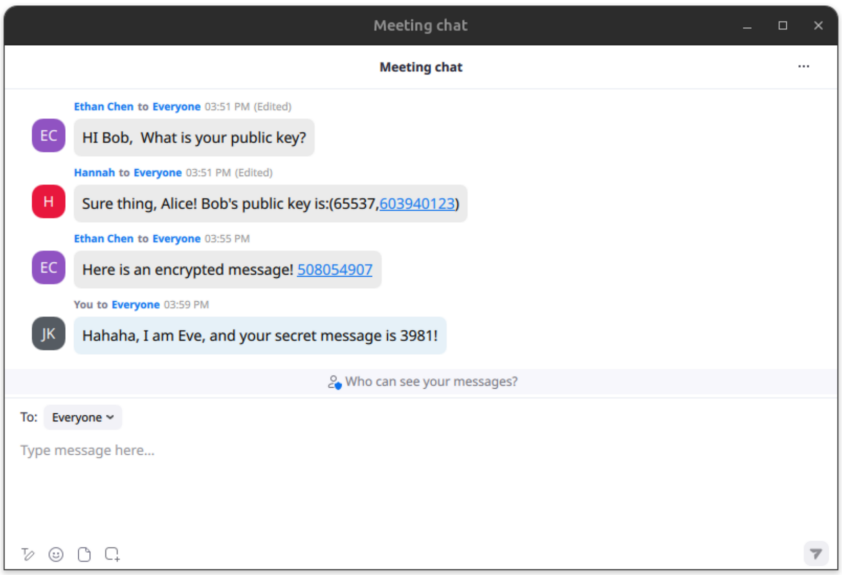

Figure 12: RSA Eve: public channel

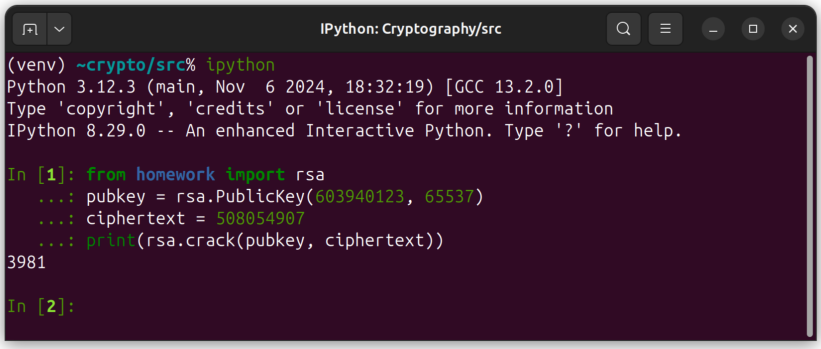

Figure 13: RSA Eve: private computation