2019

SOFTENG 364: Lab 8
Public Key Cryptography

Craig Sutherland

Contents

Overview

To review the material on public key cryptography with RSA.

Conventions

1. Lab tasks are marked with bullets, like this.

> **Notes** have a blue marker on the edge

Assessment

To receive credit for completing the worksheet, please complete the Lab 8 Quiz on Canvas. You'll receive feedback immediately afterwards, and may choose to redo the quiz if you wish.

Preparation

We will use the same environment as Lab 5. If you are switching to a new PC, please follow the Preparation steps in the Lab 5 worksheet.

1. Install sympy

In addition to the Python Standard Library, we'll need the SymPy (http://www.sympy.org/) library for computer algebra.

1. Open an Anaconda command prompt and list your existing Anaconda/Python environments:

```
> conda info --envs
```

2. Activate the environment that you'd like to use e.g. `softeng364python3` from Lab 5, or any other that you're happy to modify:

```
> # Examples only: You may prefer a different environment
> activate softeng364python3          # on Windows
> source activate softeng364python3   # on Mac/Linux
```

3. Install `sympy` in your current environment:

```
> conda install sympy
```

2. Configure Spyder

1. Launch `Spyder`; unless you are using the base (default) Anaconda/Python environment:
   - Copy the path to your selected environment (e.g. C:\Users\Jon\Anaconda3\envs\softeng346python3) from the output of conda info --envs (above).
   - In Anaconda, click Tools > Preferences > Python Interpreter; select Use the following Python interpreter and paste your environment's path into the field beneath.
2. Set Spyder's working directory to your preferred location using Spyder's File Explorer tab or address bar.
3. Create a new Python script called e.g. lab8.py.

As you proceed through the worksheet, you can append new code snippets and re/execute them by clicking Run file (F5) or by typing %run lab8.py at Spyder's ipython command prompt.

> A complete version of the lab code is available in Canvas, but starting from scratch may well be a better approach.

Part 1: Review of Slide 8-24

This part reviews Slides 8-24 and 8-25.

Slide 8-25 is not examinable, but studying it with the aim of feeling at ease with why RSA works may well be worthwhile.

Please run each of the following lines in sequence and print the value of each variable as you proceed.

```
import math  # for math.gcd

p = 5                # "prime p"
q = 7                # "prime q"
n = p*q              # modulus
z = (p - 1)*(q - 1)  # Euler totient
e = 5                # encryption/public exponent
assert math.gcd(e, z) == 1
d = 29               # decryption/private exponent
assert (e*d - 1) % z == 0

m = 12               # plaintext
print(bin(m))        # plaintext bitstring
c = pow(m, e, n)     # ciphertext
assert c is 17

mm = pow(c, d, n)    # plaintext?...
assert m is mm       # ... yes!
```

Discuss with your neighbour:

- What does `math.gcd` compute? What other names might be used for the GCD?
- What does `math.gcd(e, z) == 1` tell us about `e` and `z`?
- What does Python's operator `%` do? How might it be defined (in terms of other operators)? What is its priority level? See here for further information.
- How wide (how many bits) would be required to store `pow(m, e)` and `pow(c, d)`?
- What is the role of the third argument to `pow()`?
- Extra for Experts: The Euler totient function relates `p - 1` and `p`, `q - 1` and `q`, `n` and `z`: How is this function defined? How does it relate to Euler's (Totient) Theorem?

Part 2: A more realistic scenario

In the next section, we'll apply the RSA algorithm with more realistic data.

Part 2.1: Prime factors

On Slide 8-24, we've specified two **very small** prime factors for the RSA modulus to limit the number of digits in subsequent calculations.

This part relates to the first bullet point on Slide 8-22, which recommends that we *choose two **very large** prime numbers p, q. (e.g., 1024 bits each).*

- Why is it critically important that `n = p*q` be large? Discuss.

To follow the recommendation, let's select two (random) prime numbers with 1024 bits:

- Use the shift operator << to compute the values of the smallest- and largest integer with 1024 bits.

```
num_bits = 1024
lower = 1 << (num_bits - 1)
upper = (1 << num_bits) - 1

>>> lower.bit_length()
1024
>>> upper.bit_length()
1024
```

- Discuss these formula with your neighbour and verify that the values computed really are, respectively, as large and small as possible i.e.

```
assert (lower - 1).bit_length() == num_bits - 1
assert (upper + 1).bit_length() == num_bits + 1
```

Since Python's standard library doesn't provide support (aside from `math.gcd`) for computing with prime numbers, we'll use the `sympy.ntheory` (*number theory*) modules.

- Execute the following lines to generate two prime factors:

```
from sympy.ntheory.generate import randprime
p = randprime(lower, upper)
q = randprime(lower, upper)

from sympy.ntheory.primetest import isprime
assert isprime(p)
assert isprime(q)
assert p.bit_length() == num_bits
assert q.bit_length() == num_bits
```

- Have a very quick look at the documentation of `sympy.ntheory.generate.randprime()` and note the mention of *Bertrand's postulate*: Were we guaranteed to find at least one random number in the range [`lower`, `upper`)? The following output might help.

```
>>> 2*lower - upper
1
```

- Were/Are we guaranteed that `p` and `q` will be distinct values?
- Perhaps to help put the magnitudes of `p`, `q`, and `n` into perspective, use `math.log10` to calculate the number of decimal digits in each e.g.

```
>>> int(math.log10(321) + 1)
3
>>> int(math.log10(n) + 1)
617
```

Part 2.2: RSA encryption/public factor

Here, we continue with the remaining points on Slide 8-22.

In the preceding section, we selected realistic (large!) prime factors `p` and `q`. The RSA modulus `n` and its Euler totient `z` are easily calculated as follows:

```
n = p*q
z = (p - 1)*(q - 1)
#from sympy.ntheory import totient
#assert totient(n) == z  # too expensive to run!
```

Let's now generate the RSA exponents.

- Scan-read the Wikipedia entry on the number 65,537, noting the statement *65537 is commonly used as a public exponent in the RSA cryptosystem*. Discuss with your neighbour:
  - Is it OK that this particular value is "commonly used" for the exponent `e`? Wouldn't this be insecure?
  - For what reasons (according to the Wikipedia article) is 65,537 chosen?
  - Point 3 on Slide 8-22 specifies that `e` should be chosen such that `e` and `z` are **relatively prime**. How do we know that 65,537 and `z` are relatively prime (without using `math.gcd`)?

Let's employ this choice of `e`. In fact, it appears that some implementations of RSA actually require it!

```
e = 65537
assert math.gcd(e, z) == 1  # Using z as before
```

Part 2.3: RSA decryption/private factor

Step 4 on Slide 8-22 says that we choose `d` such that `(e*d) % z == 1`, which is to say that `d` is the modular multiplicative inverse of `e` with respect to `z`.

Happily, modular multiplicative inverses are efficiently computable via the extended Euclidean algorithm. The is the Euclidean algorithm, familiar from SOFTENG 250, "extended" to keep track of intermediate values. Although the algorithm is nice and short (around 12 lines of pseudocode), we'll make use of an existing implementation in `sympy`:

```
from sympy.core.numbers import mod_inverse
d = mod_inverse(e, z)  # Using e, z as before
assert (e*d - 1) % z == 0
print(d)
```

Part 2.4: Extracting an integer from a bit pattern

This part relates to Slide 8-21 ("RSA: getting ready"), which reminds us that the bit pattern underlying any digital data has an associated numerical value: Here, we'll see use `int.from_bytes()` to extract the integer underlying an arbitrary sequence of `bytes`.

Let's study the following snippet and execute each line at the Python command prompt:

```
import sys
m_str = 'Tøp Secrèt!'  # Note the two non-ASCII characters
m_bytes = m_str.encode()
m = int.from_bytes(m_bytes, byteorder=sys.byteorder, signed=False)

print('string: {}'.format(m_str))
print(' bytes: {} ({} bits)'.format(m_bytes, 8*len(m_bytes)))
print('   int: {}'.format(m))
print('  bits: {} ({} bits)'.format(bin(m), m.bit_length()))

# Convert from int back to str
```

```
assert m.bit_length() <= 8*len(m_bytes)
m_bytes_new = m.to_bytes(len(m_bytes), byteorder=sys.byteorder, signed=False)
m_str_new = m_bytes_new.decode()

assert m_bytes_new == m_bytes
assert m_str_new == m_str
```

Discuss with your neighbour:

- What is the difference between `m_str` and `m_bytes`? (What is the role of `str.encode()`?)
- What are the roles of the keyword arguments `byteorder` and `signed`?
- Why is `m.bit_length() <= 8*len(m_bytes)` (i.e. <= as opposed to ==)?
- Why does the call to `int.to_bytes(len(..), ..)` require a length?

Part 2.5: RSA encryption & decryption

This part relates to Slide 8-23.

Finally, let's see our (long!) public/encryption and private/decryption keys in action:

```
import sys
m_str = 'Tøp Secrèt!'
m_bytes = m_str.encode()
m = int.from_bytes(m_bytes, byteorder=sys.byteorder, signed=False)
assert m < n, 'Modulus is too small for message'

# Using e, n as before
c = pow(m, e, n)  # ciphertext
envelope = (c, len(m_bytes))  # "pack"
#
# Transmit ciphertext over insecure channel...
#
c_received, length = envelope  # "unpack"
m_new = pow(c_received, d, n)
m_bytes_new = m_new.to_bytes(length, byteorder=sys.byteorder, signed=False)
m_str_new = m_bytes_new.decode()
print(m_str_new)
assert m_str_new == m_str
```

- Briefly discuss why `envelope` contains the ciphertext and a `length`.

Part 3: RSA in the Java Security APIs

This part is intended to indicate how an application developer might view the RSA algorithm. Please don't spend more than a few minutes on this part.

The Java platform provides extensive support for encryption and authentication. Hopefully, our discussion of Chapter 8 will make the API documentation seem reasonably familiar.

- Visit the documentation for java.security and discuss with your neighbour:
  - Which classes inherit from `java.security.Key`? Which of these relate to RSA?

- Visit the documentation for java.security.spec.RSAKeyGenParameterSpec and discuss with your neighbour:
    - What parameters are required to specify an RSA key pair from the perspective of the application developer?
    - Which two values appear to be recommended for the public key exponent? Recall our earlier discussion.
    - Which data type is used to represent the RSA parameters?
- Visit the documentation for java.security.spec.RSAMultiPrimePrivateCrtKeySpec and discuss with your neighbour:
    - Which parameters to `RSAMultiPrimePrivateCrtKeySpec` are familiar from our discussion? Which are new?

At some later stage, you may be interested in exploring other classes that will be familiar from Chapter 8 e.g. javax.crypto.Mac that represents Message Authentication Codes.

End of lab worksheet: Please don't forget to complete the lab quiz.

Extra for Experts

None of the quiz questions relate to this part: Please don't actually spend any time on the programming here unless you're taking a break from other things - perhaps after the exams.

Challenge 1: Brute force attacks with `sympy`

We can presume that someone (Trudy) aiming to break RSA encryption has access to the public key (n, e): To compute d, she needs e and z (as inputs to e.g. sympy.core.numbers.mod_inverse()), but z = (p - 1)*(q - 1) depends on p and q, which are (unknown, prime) factors of (known) n.

It is the extreme difficulty of determining p and q from (large) n that makes RSA secure.

To get a taste of how the scale of the problem Trudy might face, compute the prime factors of n of various lengths using Sympy's primefactors function: The algorithm employed is presumably much more efficient than a direct search.

Challenge 2: Modular multiplicative inverses

Study the pseudocode function `inverse()` on Wikipedia (https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm#Computing_multiplicative_inverses_in_modular_structures); this specialized variant of the extended Euclidean algorithm computes modular multiplicative inverses. Write your own implementation in Python or Java.

Challenge 3: Euler's totient function

Implement Euler's totient function using math.gcd:

- Compare its performance with that of sympy.ntheory.totient.
- Use matplotlib of an alternative visualization module to reproduce the figure shown below: