CSC110 Tutorial 7: The RSA Cryptosytem, Proofs and in Practice

Print this handout

Exercise 1: Completing the proof of RSA correctness

This week in lecture, you learned about the RSA cryptosystem. (*Now is a great time to pause to review the steps of the RSA algorithm!!*)

We proved that RSA encryption and decryption work correctly when the (integer) message m is coprime to the modulus n. In this exercise, you'll extend this proof to cases where $\gcd(m, n) > 1$, showing that even for these numbers that RSA encryption and decryption work correctly.

Before starting, please review the two statements below, the first of which is the familiar Fermat's Little Theorem, and the latter is a statement that you proved in last week's tutorial.

(*Fermat's Little Theorem*) Let $p, a \in \mathbb{Z}$ and assume p is prime and that $p \nmid a$. Then $a^{p-1} \equiv 1 \pmod{p}$.
(*Statement from Tutorial 6*) For all integers a, b, and x, if $\gcd(a, b) = 1$ and $a \mid x$ and $b \mid x$, then $ab \mid x$.

Your task is to prove each of the following statements. As with last week's tutorial, for each proof you can use the statements in previous parts as "external facts" in your proof, as long as you are explicit about where you use them. You can also use the above two "framed" statements in your proofs, and general arithmetic properties of modular equivalence (like the fact that you can add/subtract/multiply to both sides of an equivalence).

1. Prove that for all $a, b \in \mathbb{Z}^+$ and $x, y \in \mathbb{Z}$, if $\gcd(a, b) = 1$ and $x \equiv y \pmod{a}$ and $x \equiv y \pmod{b}$, then $x \equiv y \pmod{ab}$. (This is a good warm-up because it's mainly an exercise in expanding the definition of modular equivalence!)

2. Prove that for all $a, b \in \mathbb{Z}^+$ and $x, y \in \mathbb{Z}$, if $x \equiv y \pmod{ab}$ then $x \equiv y \pmod{a}$. (It is also true that $x \equiv y \pmod{b}$ by switching the roles of a and b, but you do not need to prove this.)

3. Let $p, q, n, e, d, m, c, m' \in \mathbb{Z}$ be the variables in the RSA cryptosystem. Prove that $m' \equiv m \pmod{p}$ and $m' \equiv m \pmod{q}$.

   Notes/hints:

   - You should *not* assume that $\gcd(m, n) = 1$.
   - You should *not* assume that $m' \equiv m \pmod{n}$.
   - You *should* use the fact that $ed \equiv 1 \pmod{\varphi(n)}$ and that $\varphi(n) = (p-1)(q-1)$, and Fermat's Little Theorem.
   - You *should* use cases depending on whether $p \mid m$ and $q \mid m$ (it's up to you to determine the exact cases to use).
   - A good first step in your proof is to work towards writing $m' \equiv m^k \pmod{n}$ for some $k \in \mathbb{Z}^+$, using what you know about RSA encryption/decryption.

4. Let $p, q, n, e, d, m, c, m' \in \mathbb{Z}$ be the variables in the RSA cryptosystem. Prove that $m' \equiv m \pmod{n}$ (without assuming $\gcd(m, n) = 1$).

   (Even though this is the "main" proof for this exercise, this proof should be quite short by using the other statements you proved earlier! It's more of an exercise in *chaining* together a few statements in a proof.)

Exercise 2: Real-world RSA public keys

Now we're going to look at how RSA public and private keys are stored in practice. To start, here is a sample RSA public key that's been generated by the popular software tool `ssh-keygen`.

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDKh75lsX5zJxxfzo1AYSnM+BiaDaXCZ1iGhOJlDaKDNVgnnTQx
QMoQU1qYujfQzt3RmdH4dvMX0pK7R7y65hlYbMbBn8f82+Wo+7MLv6/vgBA8vWUa8NYi0Nsfz1Dh43ATYm5vi8I/
6PpXkkaq54Ba3kUtrvwVbCyfDY+u8G9+sA+GlZ9pkb7+B3sPox8RnAn4TqSksNSXX+kz8OwZRSkYtbz9PNy2SIPU
AuK7S2GpYwPq2y+HEsoMyS1bODPhJZ58xtqGyijoJObSH8TEk7kwVm1azl/1SPiHU6gaiENdgVgy7Jb3UKe/xkGl
GhifyrHvoW0AMSt/Hno8Wwiybr0V david@my-computer
```

Somehow, this file stores a public key generated by the RSA key generation algorithm, just like the ones we saw in class. Our goal for the remainder of this tutorial is to understand how this happens.

To start, please download the starter files `tutorial7.py` and `sample_public_key.txt` and save them into this week's tutorial folder.

1. Binary representation of numbers

Our first step is to understand a bit more formally how a computer represents numbers. When you read a number such as "324" in decimal, you see a sequence of decimal digits, $d_{k-1}d_{k-2}\dots d_1d_0$, where each digit d_i is in $\{0, 1, 2, \dots, 9\}$. The number that corresponds to this sequence of digits is $\sum_{i=0}^{k-1} d_i \times 10^i$. In words, the right-most digit is multiplied by 10^0, the next digit to the left is multiplied by 10^1, and so on. Each digit to the left has a multiplier that is 10 times the multiplier of the previous digit. In our example "324", we have $d_2 = 3$, $d_1 = 2$, and $d_0 = 4$, and so the value is $3 \times 10^2 + 2 \times 10^1 + 4 \times 10^0$.

As we discussed all the way back in Week 1, computers store information in *bits*, which can have one of two states: 0 or 1. The **binary (base 2) representation** of a number uses the binary digits $\{0, 1\}$ instead of the ten decimal digits $\{0, 1, 2, \dots, 9\}$. We write numbers in binary in the same sort of way that we write numbers in our traditional base 10 system. Again we represent a number by a sequence of binary digits, $d_{k-1}d_{k-2}\dots d_1d_0$, but now each digit d_i is in $\{0, 1\}$. The value of the number corresponding to this sequence is: $\sum_{i=0}^{k-1} d_i \times 2^i$. Note that the change in the expression is the change from powers of 10 to powers of 2. When discussing the binary representation of a number, the digits d_i are often called *bits*. The number represented in its decimal form as 139 would represented in binary as: $1 \times 2^7 + 1 \times 2^3 + 1 \times 2^1 + 1 \times 2^0 = (10001011)_2$.[1]

Here is a table showing the binary representation of the first twenty positive integers.

Decimal	Binary
1	$(1)_2$
2	$(10)_2$
3	$(11)_2$
4	$(100)_2$
5	$(101)_2$
6	$(110)_2$
7	$(111)_2$
8	$(1000)_2$
9	$(1001)_2$
10	$(1010)_2$
11	$(1011)_2$
12	$(1100)_2$
13	$(1101)_2$
14	$(1110)_2$
15	$(1111)_2$
16	$(10000)_2$
17	$(10001)_2$
18	$(10010)_2$
19	$(10011)_2$
20	$(10100)_2$

1. What is the decimal representation of the number with binary representation $(101011)_2$?

2. In Python, we can represent a binary representation as a list of numbers, where each number is either 0 or 1. For example, we could represent the binary representation $(101011)_2$ as the list `[1, 0, 1, 0, 1, 1]`. Inside `tutorial7.py`, implement the function `binary_to_int`, which takes a Python list that stores a binary representation, and returns the `int` value that it represents.

2. From bits to bytes

Even though computers store data in bits, working with individual bits is quite cumbersome, and so computers typically work instead by dividing bits into groups of 8, where each group of 8 is called a **byte**.

As a sequence of 8 bits, a single byte can store numbers ranging from $(00000000)_2 = 0$ to $(11111111)_2 = 255$. Just as a sequence of bits forms a *binary* representation of a number, a sequence of bytes forms a *base-256* representation of a number: $d_{k-1}d_{k-2}\dots d_1d_0$, but now each digit d_i is in $\{0, 1, \dots, 255\}$. The value of the number corresponding to this sequence is: $\sum_{i=0}^{k-1} d_i \times 256^i$.

For example, the base-256 number $(105)_{256} = 1 \cdot 256^2 + 0 \cdot 256^1 + 5 \cdot 256^0 = 65541$.

1. In `tutorial7.py`, implement the function `bytes_to_int`, which its analogous to `binary_to_int` except it uses base-256 instead of base-2.

3. The RSA public key format and Base64 encoding

Okay, now let's turn our attention to the sample RSA public key from the start of this section.

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDKh75lsX5zJxxfzo1AYSnM+BiaDaXCZ1iGhOJlDaKDNVgnnTQx
QMoQU1qYujfQzt3RmdH4dvMX0pK7R7y65hlYbMbBn8f82+Wo+7MLv6/vgBA8vWUa8NYi0Nsfz1Dh43ATYm5vi8I/
6PpXkkaq54Ba3kUtrvwVbCyfDY+u8G9+sA+GlZ9pkb7+B3sPox8RnAn4TqSksNSXX+kz8OwZRSkYtbz9PNy2SIPU
AuK7S2GpYwPq2y+HEsoMyS1bODPhJZ58xtqGyijoJObSH8TEk7kwVm1azl/1SPiHU6gaiENdgVgy7Jb3UKe/xkGl
GhifyrHvoW0AMSt/Hno8Wwiybr0V david@my-computer
```

This string consists of three words separated by spaces. The first word, `'ssh-rsa'`, is a label for the cryptosystem used to generate this key.[2] The third word, `david@my-computer`, is a label for the name of the user associated with this key.

The middle word is the most interesting. This sequence of seemingly-random characters is yet another form of number representation known as **Base64 encoding**. As you might guess from its name, this encoding is based on a base-64 number representation, except it uses ASCII characters to represent the individual digits. For example, in this encoding, `'A'` corresponds to 0, `'B'` corresponds to 1, `'a'` corresponds to 26, and `'/'` corresponds to 63.[3]

We can use the `base64` Python module to convert between a Base64-encoded string and a sequence of bytes. This is actually returned as a new built-in data type called `bytes`, but we can convert that into a familiar list by calling `list` on the result.

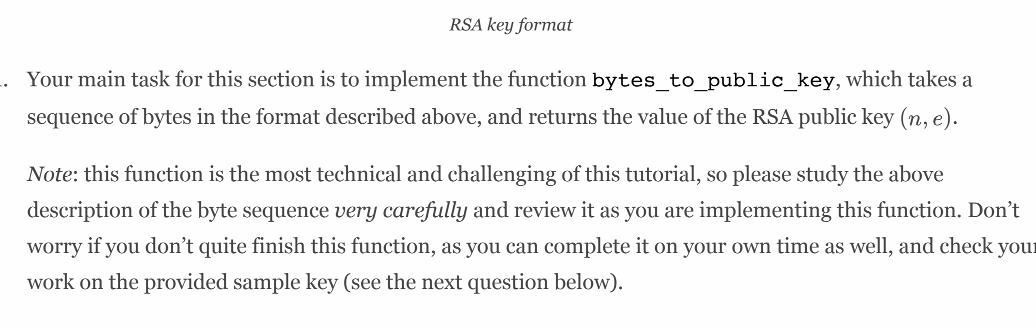
```
>>> import base64
>>> base64.b64decode("RGF2aWQgc2F5cyBoaQ==")
b'David says hi'
>>> list(base64.b64decode("RGF2aWQgc2F5cyBoaQ=="))
[68, 97, 118, 105, 100, 32, 115, 97, 121, 115, 32, 104, 105]
```

1. In `tutorial7.py`, implement the function `public_key_to_bytes`, which takes the name of a public key file and returns a list of bytes corresponding to the Base64-encoded string. You can assume the format of the file is the one discussed above.

4. Extracting n and e

Okay, so now we have a list of bytes from the public key file. These bytes represent the RSA public key, but how exactly? Recall that the public key consists of two numbers (n, e); how do we know which bytes correspond to n, and which bytes correspond to e? This RSA key format uses a standard technique to solve the problem of storing compound data in a sequence: prefixing each part of the data with four bytes that represent the size of that data. This pattern repeats three times in the RSA key format:

- The first four bytes store the size s_1 of a label for the key type.
- The next s_1 bytes store the label. For our purposes, this label should be `[115, 115, 104, 45, 114, 115, 97]`, but could be more complex in general.[4]
- The next four bytes store the size s_2 of the exponent e.
- The next s_2 bytes store the actual value of e. (Finally, part of the public key!)
- The next four bytes store the size s_3 of the modulus n.
- The next s_3 bytes store the actual value of n.



RSA key format

1. Your main task for this section is to implement the function `bytes_to_public_key`, which takes a sequence of bytes in the format described above, and returns the value of the RSA public key (n, e).

   Note: this function is the most technical and challenging of this tutorial, so please study the above description of the byte sequence *very carefully* and review it as you are implementing this function. Don't worry if you don't quite finish this function, as you can complete it on your own time as well, and check your work on the provided sample key (see the next question below).

2. Finally, put everything together to implement the function `extract_public_key`, which takes a file containing an RSA public key in the format described above and returns the public key contained in that file. After you've implemented the function, you should be able to extract the following public key from our given file `sample_public_key.txt`:

   ```
   >>> extract_public_key('sample_public_key.txt')
   (255670753357552824327817797638606564965881813485659205661792576013855597932219001172
   ```

   That's right—the tool `ssh-keygen` generated a modulus `n` that consists of 617 digits, which your computer represents using 2048 bits! Try running any factoring algorithm you wish on this `n`, and you'll find yourself waiting a very, very long time to compute the private `p` and `q` that generated this `n`. :)

Further reading

If you'd like to learn more about generating your own public/private key pairs, you can check out https://www.ssh.com/ssh/putty/windows/puttygen (for Windows) or https://www.techrepublic.com/article/how-to-generate-ssh-keys-on-macos-mojave/ (for macOS).

The popular code hosting platform **GitHub** allows you to link your account to a public key, so that you can upload code to the platform directly from your computer without typing in your password each time. For more on that, check out this GitHub guide.

(If you're interested in learning more about GitHub, we recommend checking out this Quickstart guide first, though!)

---

1. We typically surround binary representations with parentheses and a subscript 2 to avoid confusion with decimal representation.↩

2. We've only studied RSA in this course, but there are many others that follow this standard format.↩

3. You can find a complex reference of the characters used in a Base64 encoding at https://en.wikipedia.org/wiki/Base64. Somewhat confusingly, the choice of letters does *not* correspond to their ASCII values returned by `ord`.↩

4. This list correspond to the `ord` values of the string `'ssh-rsa'`.↩