# RSA and the cat in the middle

Jeffrey Goldberg
1Password

21 October, 2021
Revised: April 25, 2023

# Contents

# Introduction

*1Password Security Design*, Appendix E describes what I call a Cat in the Middle Attack (CitM) as it involves the nefarious Mr. Talk. Historically, these have been called “Man in the Middle” (MitM), and more recent parlance uses the term “Adversary in the Middle” (AitM).[1]

Today we go into a deeper dive of the math and algorithms involved.

Chapter 1 reiterates that story with more detail, including bits of a Python session showing the steps. Chapter 2 details the algorithms used to perform the calculations in chapter 1. If chapter 3 existed, it would explain why the algorithms in chapter 2 produce the results we see in chapter 1.

I have entirely failed to pitch this at a consistent level of experience with reading code and comfort with mathematical abstractions. This leaves it up to you to decide what to skip over. The bits of Python code are included to help clarify the text, but that will only work for some people. There are certainly parts of the mathematical discussion that can be glossed over. depending on your interests.

In general, this document should be useful to you if you are willing to accept that there are parts that won’t make sense.

There are a few places where code details are not presented in this document and you are referred to the source. The source lives at `https://gitlab.1password.io/security/sec-team-training/-/tree/main/2021-10-29`.

[1]In addition to the obvious improvement in terminology, this avoids confusion with a “Meet in the Middle” attack, which is something else entirely.

# Chapter 1

# A cat in the middle

## Code preliminaries

We use constructs here that were introduced in Python 3.9, so we need to run a Python version check before continuing, along with some other cruft that I will skip here but you can see by inspecting the source files.

We will be making use of the `rsa.py` module that is detailed in chapter 2.

```
import rsa
```

## 1.1 Critters and their keys

This is a story of the dogs Patty and Molly and the neighbor's evil cat, Mr. Talk. Each of these have a name and an RSA key pair. `Critter`s get set up by the `__init()__` method shown in listing 1.1.

**Listing 1.1** Critters are initialized with their names and key pairs

```
class Critter:
    """A Critter has a name and a key pair"""
    def __init__(self, name: str, key: rsa.KeyPair) -> None:
        self.name = name
        self._keypair: rsa.KeyPair = key
        self.pub_key = self._keypair.pub_key()
```

Encrypting something to a Critter is just using the public key to encrypt. Decrypting requires the Critter's key pair, which includes the secret part of the key, The methods `encrypt_to()` and `decrypt()` are shown in code listing 1.2.

**Listing 1.2** Methods for a Critter to encrypt and decrypt

```
    def encrypt_to(self, ptext: int) -> int:
        return self.pub_key.encrypt(ptext)

    def decrypt(self, ctext: int) -> int:
        return self._keypair.decrypt(ctext)
```

### 1.1.1 Patty

We can use `rsa.KeyPair()` to create an RSA key pair from two primes. We (or Patty) can create a key pair with the primes 107 and 151 with something like `rsa.KeyPair(107, 151)`.[1]

```
patty = Critter("Patty", rsa.KeyPair(107, 151))
print(f"Patty's public key: {patty.pub_key}")
```

```
Patty's public key: exponent: 17, modulus: 16157
```

Let's suppose we want to encrypt the message 1234 to Patty.

```
m1 = 1234
c = patty.encrypt_to(m1)
print(f"Encrypting the message {m1} to Patty"\
      f" gives us the ciphertext {c}.")
```

```
Encrypting the message 1234 to Patty gives us the ciphertext 8900.
```

Patty, using her private key, can decrypt the message.

```
m2 = patty.decrypt(c)
print(f"Patty decrypts {c} as {m2}")
```

```
Patty decrypts 8900 as 1234
```

## 1.2 Sharing a vault

We will start with Patty sharing a vault key with Molly. Getting the encrypted vault key from Patty to Molly will involve a server as will Patty getting Molly's public key.

### 1.2.1 The server

Our server is going to keep track of users and their public keys. It will also pass encrypted messages from one user to another. Quite crucially it will never have the ability to get at any of the users' private keys.

[1]In real life the primes would be much larger and the would be generated in ways that give us confidence that they are prime and that the have the right properties with respect to each other and to the public exponent. The public exponent is introduced in chapter 2.

**Listing 1.3** A Service maintains a list of users and message

```
class Service:
    def __init__(self) -> None:
        self._users: dict[str, rsa.PubKey] = {}
        self._messages: dict[str, int] = {}
```

Team members will be able to post and get messages and they will be able to post their public keys and get others' public keys. And so we will create four methods talking to a Service: `post_msg()`, `get_msg()`, `post_key()`, and `get_key()`. They are not shown here because so what we can focus on the story.

A real service would make sure that the critter posting a key or retrieving a message is who they say they are. Pretend that that is the case here. It isn't, but let's pretend.

We will set up a service for Molly and Patty to talk to. Let's call it '`b5`'.

```
b5 = Service()
```

Patty can now upload her public key to the server using the `post_key()` method.

```
b5.post_key(patty.name, patty.pub_key)
```

We can then look up Patty's public key from the server using the `get_key()` method and print it.

```
print(f"Patty's public key: {b5.get_key('Patty')}")
```

```
Patty's public key: exponent: 17, modulus: 16157
```

### 1.2.2 Molly

Molly is a member of Patty's family. And when Molly sets up her account, she creates her identity and adds her public key to the server.

```
molly = Critter("Molly", rsa.KeyPair(97, 43))
b5.post_key(molly.name, molly.pub_key)
```

The prime numbers, 97 and 43, that Molly used to create her key pair get multiplied together to create her public modulus $N$. They are also used to compute (as described in gory detail is chapter 2) a secret decryption exponent called $d$.

```
print(f"Molly's public modulus: {molly.pub_key.N}")
print(f"Molly's decryption exponent: {molly._keypair.d}")
```

```
Molly's public modulus: 4171
Molly's decryption exponent: 593
```

Suppose that Patty has a vault key, 1313, which she would like to share with Molly. She will get Molly's public key from the server and encrypt something to it.

```
vault_key = 1313
c = b5.get_key('Molly').encrypt(vault_key)

print(f"The encrypted vault key is {c}.")
```

```
The encrypted vault key is 530.
```

Patty gives the encrypted data to the server to pass to Molly.

```
b5.post_msg('Molly', c)
```

Molly signs in and checks to see if `b5` has anything for her. She gets it and decrypts it.

```
ctext_from_patty = b5.get_msg('Molly')
vk_from_patty = molly.decrypt(ctext_from_patty)

print(f"Ctext ({ctext_from_patty}) decrypts to {vk_from_patty}.")
```

```
Ctext (530) decrypts to 1313.
```

Throughout all of this the B5 server has neither Molly's nor Patty's private keys, and the vault keys that pass through the server are encrypted with someone's public key.

## 1.3 The Cat in the Middle

Suppose that Mr. Talk has gained control of the server. We represent this unfortunate state of affairs by allowing Mr. Talk to directly manipulate private[2] server objects such as `_users` and `_messages`.

First, Mr. Talk replaces Molly's public key in the server with one of his own. He does keep a copy of Molly's real public key,

[2]Here we use "private" to refer to the programming concept of access to methods and fields within an object. In the sense of keys, our server only ever deals with public keys, but it does have provide fields in which it keeps those public keys.

Python does not have or enforce any formal mechanisms to establish private methods and fields, but there is a convention of giving names beginning with a single underscore to those that should be considered private.

```
mr_talk = Critter("Mr. Talk", rsa.KeyPair(47, 89))
molly_real_pub_key = b5.get_key('Molly')
b5._users['Molly'] = mr_talk.pub_key
```

Now if someone asks for Molly's public key from the server they get 4183 instead of 4171.

Once again, Patty wishes to share a vault key with Molly. This vault key is 1729. She fetches what she believes to be Molly's public key from the server and encrypts this vault key with it. She then sends to encrypted vault key to the server.

```
vault_key = 1729
c = b5.get_key('Molly').encrypt(vault_key)
b5.post_msg('Molly', c)

print(f"The encrypted vault key is {c}.")
print(f"But it was encrypted with {b5.get_key('Molly').N}.")
```

```
The encrypted vault key is 2016.
But it was encrypted with 4183.
```

Mr. Talk is able to decrypt the ciphertext because it was really encrypted using the public key modulus 4183, for which he has the corresponding secret parts. In this case, the decryption exponent corresponding to that public key is 1905.

```
vk = mr_talk.decrypt(c)
print(f"Mr. Talk learns that the vault key is {vk}")
```

```
Mr. Talk learns that the vault key is 1729
```

Mr. Talk then encrypts the vault key with Molly's real public key, and has the server hold that.

```
c = molly_real_pub_key.encrypt(vk)
b5._messages['Molly'] = c
```

Now the ciphertext waiting for Molly (2826) is ecrypted with Molly's real key (4171). and so when Molly retrieves it as before nothing is amiss. She can decrypt it and get the vault key that was shared with her.

```
ctext_from_patty = b5.get_msg('Molly')
vk_from_patty = molly.decrypt(ctext_from_patty)

print(f"Ctext ({ctext_from_patty}) decrypts to {vk_from_patty}.")
```

```
Ctext (2826) decrypts to 1729.
```

Mr. Talk never had the ability to learn anyone's private keys other than his own. From Patty's and Molly's points of views, this vault key sharing went perfectly smoothly. Yet a malicious actor, with substantial control of the server, was able to intercept and decrypt the vault key.

## 1.4 Not just a public key problem

The problem illustrated by that story is real, serious, and the subject of much attention when considering public key encryption. But it is a good kind of problem to have when considering the alternative. Symmetric key encryption has the same problem but also has so many other problems with key distribution that this one of authenticating keys gets lost.

In the story above, Patty would need to know that the key she gets really comes from Molly; but however that is done, it does not matter if anyone eavesdrops on that. Furthermore, Molly does not need to know that she is talking to Patty when it comes to publishing her key,

Symmetric key distribution requires that *mutual* authentication. If Patty and Molly were setting themselves up to use a symmetric key and performing symmetric encryption, *both* Patty and Molly would need to prove to the other who they are before they could exchange keys. After that, the would still use public key encryption to make sure that their exchange of a symmetric key is safe against eavesdroppers.

Public key encryptions solves so many problems that we can get to the point where we we can concerned about the Cat in the Middle problem. So while the Cat in the Middle problem is a real problem for public key systems, it is also a problem from symmetric key encryption.

# Chapter 2

# Toy RSA in Python

In the previous chapter we ran through some examples of public key encryption. Now we turn to the computations needed to make it happen using the RSA algorithm. This section is not fully self-explanatory, There are going to be parts that you are not going to understand unless you know most of this already. If you are uncomfortable with that stop here, but if you continue take it easy.

This is not for anything that might ever be used or copied to something that might ever be used for real data. It is for illustrating basics of the RSA algorithm and GCD algorithms. Much more is needed to actually use RSA safely.

## Code preliminaries

The file that I am editing as I write this is the source for both the Python code and the PDF you are reading, so we have to get some stuff into the Python file we are creating. We will also be creating custom `__str__()` methods, but those aren't made visible in this document.

We need to be able to handle Tuples, as we will be passing around pairs of numbers. And for reasons[1] discussed later we will want some matrix manipulation tools.

**Listing 2.1** Imports

```python
from typing import Tuple
import numpy as np
```

[1]Not all reasons are good reasons.

## 2.1 Math

Learning math is like learning to play music, you can't really become comfortable with it without practice.[2] I can, and will, give a brief introduction to the math needed for what follows in this chapter. It really is quite accessible, using nothing more than the arithmetic taught in grade school, but it puts a twist on it. It is up to you do decide how much practice you want to do, which will influence how much you get out of what you are reading. You should be able to get some real value from this chapter if you don't practice or gain some fluency in modular arithmetic, but I don't know the extent to which things will come together.

### 2.1.1 Numbers

God created the whole numbers. All the rest are the work of man.

Leopold Kronecker

For technical reasons, we will sometimes need to talk about integers[3] instead of whole numbers. Integers are the whole numbers, zero, and the negatives of whole numbers. For historical reasons the set of integers is written '$\mathbb{Z}$'. (That notation may look fancy, but it is just from the German word for number.) And we can slightly more formally say

$$\mathbb{Z} = \{\dots, -3, -2 - 1, 0, 1, 2, 3, \dots\} \tag{2.1}$$

Indeed, in much of what follows when I use the word "numbers" I am just talking about integers. There may be exceptions, but those should be clear from the context.

### 2.1.2 Remainders of the day

Suppose it is Friday, and someone asks you what day of the week it will be in 7 days. You should be able to quickly determine that that is also Friday without having to count out the days. Seven days from your Friday is exactly one week in the future, and is also a Friday. Once you realize this, you should easily be able to answer if asked about 14 days or 28 days. Also

[2]It has become fashionable in some circles to criticize homework in primary and secondary education. But Math, Music, and Reading require practice well outside the typical classroom time. This does not mean that I don't sympathize with many of the criticisms of homework, but we must recognize cases in which it, or something like it, is necessary.

[3]Computer programming languages often have what they call "integer type," but with rare exceptions they are limited in size. When we talk about integers in Mathematics (including the Cryptography branch of Mathematics) we mean integers in the unlimited mathematical sense.

you should be able to see that 7000 days from Friday will also be a Friday. (If you don't yet realize this, then practice with counting or using a calendar showing a month view working for 7, 14, and 28 days. Remember not to count the Friday you start on.)

If asked on a Friday, then any multiple of seven days from then will also be a Friday. Now suppose you are asked about eight days ahead of that Friday. You know that seven days ahead will be another Friday, and eight is one more than seven, so it will be one day after a Friday. It will Saturday. (If this isn't clear to you, go to a calendar that shows multiple weeks. Pick a Friday. Go seven days ahead to the next Friday and go one more day. Do this starting on other days of the week, going seven days ahead and then one more. Do not count the day you start on.)

Think not just about eight days ahead of some Friday but also about 15, 22, 29, and 7001 ahead. All of those are going to be a Saturday. And all of those numbers have a remainder of 1 when divided by 7.

Let's assign a number of the days of the week, starting with Sunday as 9 through Saturday as 6. Monday is 1, Tuesday is 2, and so on. In this scheme Friday is given the number 5. Now we will go over the same questions about days of the week, but using this way of numbering the days, dividing by 7 and looking at the remainder.

Friday (day 5) plus 7 is 12. When we divide 12 by 7 we get a remainder of 5, and 5 is our number from Friday. $5 + 21 = 26$. And when we divide 26 by 7 we get a remainder of 5. $5 + 7000 = 7005$, and when we divide 7005 by 7 we get a remainder of 5. These are all Friday.[4]

In our system, Saturday is given the number 6. If we are asked for one day from Friday, we have $5 + 1 = 6$, which gets the Saturday we expect. If we are asked about eight days from our Friday, we have $5 + 8 = 13$. When we divide 13 by 7, we get a remainder of 6. When we are asked about 7001 days from our Friday, we get $5 + 7001 = 7006$. When we divide 7006 by 7 we get a remainder of 6, which is a Saturday.

There is a really useful shortcut available. We can compute remainders earlier so that we get to work with smaller numbers. Instead of adding 5 + 7001 and then taking the remainder when divided by 7, we can take the remainder first. When we divide 7001 by 7 the remainder is 1. So we can just compute $5 + 1 = 6$ to get us to a Saturday. This trick of reducing some of numbers in a computation early becomes especially hand when we move on to multiplication instead of addition.

[4]If I were to waste enough time learning how to embed a video in a PDF using TEX you would now be watching a Katy Perry video.

**Modular notation** Modular arithmetic is just ordinary arithmetic, but focuses on the remainder of whole number division. In the examples above we have been doing everything in terms of remainders when divided by 7. That is, we did all of the arithmetic "modulo 7". And we can refer to seven as the "modulus" in those examples. When all of the arithmetic for some mathematical expression is to be done modulo 7, we would write "$(\text{mod} 7)$".

Using this notation and our numbering scheme for the days of the week, we would express the 7001 days from a Friday (5) as equation 2.2

$$6 \equiv 5 + 7001 \pmod 7 \tag{2.2}$$

There are many Fridays and Saturdays, but we aren't interested in which ones or how many weeks ahead (or behind) they might be. This is because we have only been asked which day of the week it will be, not how many weeks or in which particular week that day is. From our point of view for our computations, all Saturdays are equivalent to each other, even if they are in different weeks. This is why we use the congruence symbol '$\equiv$' instead of the equal sign in equation 2.2.

This notation and math should be sufficient for working through how to encrypt with a public key. So you can just skip ahead to section 2.3 if you don't want to think about private keys and decryption. Of course you will need to jump back here when you do want to have a sense of what is going on with private keys.

## 2.1.3 Division

We can say that 11 divided by 4 is 2.75, but that is not the kind of division we will be doing throughout. Instead we want to say that 4 goes into 11 twice with a remainder of 3. Some of the terminology that we will use is listed in table 2.1

We will make some use of being able to say whether some integer divides another. Here that means divides with no remainder:

- 5 divides 10 because $2 \times 5 = 10$.
- 1 divides any integer.
- 5 divides 5 because $1 \times 5 = 5$.
- 5 does not divide 12 because there is no integer we can multiply 5 by to get 12.

There is an unfortunate ambiguity about the word "divisor." When writing out a division expression, like 11/4 we call 4 the divisor. But we

| Term | Description | in “$11/4 = 2 \times 4 + 3$” |
|---|---|---|
| Dividend | The thing we are dividing | 11 |
| Divisor | What we are dividing the dividend by. (In the expression 11/4 we say that 4 is the divisor, but in general 4 is not a divisor of 11 because that division leaves a remainder.) | 4 |
| Quotient | The whole number amount of times the divisor goes into the dividend | 2 |
| Remainder | The amount left over between the dividend by the division | 3 |
| Divides | We say “$a$ divides $b$” when there there is no remainder of $b/a$. 4 does not divide 11, but 4 does divide 12. | False |

Table 2.1: Quotients, Remainders. Dividends and Divisors with example of $11/4 = 2(4) + 3$

will also use the word “divisor” to mean something that you can divide by without leaving a remainder. So 5 is not a divisor of 12, but 5 is a divisor of 10. 20 is a divisor of 80, but 20 is not a divisor of 85.

The fancy way of defining this is to say that $b$ is a divisor of $a$ (or $b$ divides $a$ if and only if there is a number $q$ such that $a = qb$ and $a$, $b$, and $q$ are integers. So 5 divides 10 because there is a number, the quotient q, such that $5q = 10$. In that instance $q$ is 2. But 5 is not a divisor of 12, because there is no quotient $q$ such that $5q = 12$. (Remember that we are restricting ourselves to integers, so 2.4 doesn't count as a possible quotient.)

## 2.2 Greatest Common Divisor

The greatest common divisor, gcd, of a set of numbers is the largest number that divides the numbers. Consider the numbers 12 and 30. Let's look at the divisors of each. The divisors of 12 are 1, 2, 4, 4, 6 and 12. The divisors of 30 are 1, 2, 3, 5, 6, 10, 15, and 30. The divisors that they have in common are 1, 2, 3, and 6. And the greatest (largest) of those common divisors of 12 and 30 is 6. And so we would write something like “$\gcd(12, 30) = 6$” to say “the greatest common divisor of 12 and 30 is 6.”

In example 2.1 we listed all of the divisors of 12 and all of the divisors

| Step: | Action | Result of Action |
|---|---|---|
| Start: | Seeking $\gcd(12, 30)$ | |
| 1: | Find all divisors of 12 | 1, 2, 3, 4, 6, 12 |
| 2: | Find all divisors of 30 | 1, 2, 3, 5, 6, 10, 15, 30 |
| 3: | Find *common* divisors | 1, 2, 3, 6 |
| 4: | Find the largest of the common divisors | 6 |

Example 2.1: Computing $\gcd(12, 30)$ with divisors

of 30, and then found the largest divisor they have in common to be 6. That is a fine way to do things for small numbers where it is easy to find all of the divisors. It is probably the method you were taught ages ago[5] when learning how to simplify fractions with the GCD possibly called the "greatest common *denominator*" or "greatest common *factor*." If you were required to simplify, say, $\frac{12}{30}$ you would find the greatest common factor of 12 and 30 to be 6, and then divide both 12 and 30 by 6 to give you $\frac{2}{5}$ as the simplified fraction.

### 2.2.1 Integer division in Python

Consider doing integer division of 23 by 5. We are not interested in a floating point answer of something like 4.6. Instead we are interested statements like 23 divided by 5 has a quotient of 4 and a remainder of 3. Another way of saying that is $23 = 5 \times 4 + 3$. We will sometimes use $q$ for quotient and $r$ for remainder. Getting at the quotient and remainder (or both) is done in python with `//`, `%`, and `divmod()`.

So, for example, `23 // 5` is 4; `23 % 5` is 3; and `divmod(23, 5)` is (4, 3). When we just need the quotient we will use `//`. When we just need the remainder we will use `%`. When we need both we will use `divmod()`.

### 2.2.2 Computing the GCD

In example 2.1 we listed all of the divisors of 12 and all of the divisors of 30, and then found the largest divisor they have in common. That is a fine way to do things for small numbers where it is easy to find all of the divisors, but it is not practical for big numbers where finding all of the divisors is hard. A great deal of computation is hidden in the "find all the divisors of $n$" steps of example 2.1. Instead, we will use (variations on) the Euclidean Algorithm.

The oldest known algorithm for finding the gcd was described by Euclid more than 2300 years ago. It is also the algorithm that is used today. The gist of it is that when looking for the GCD of two numbers, is that you keep

[5]Well it was ages ago for *me*. You, dear reader, are almost certainly younger than I am.

| Step: | Action | Result of Action | | |
|---|---|---|---|---|
| | | $a$ | $b$ | $a = b$? |
| Start: | Seeking $\gcd(5, 20)$ | 5 | 20 | No |
| 1: | Subtract smaller from larger, and set the latter to the this result. $b = 20 - 5$ | 5 | 15 | No |
| 2: | As before ($b$ is still the larger) $b = 15 - 5$ | 5 | 10 | No |
| 3: | As before ($b$ is still the larger) $b = 10 - 5$ | 5 | 5 | Yes |
| 5: | $a = b$, so $a$ (or equivalently $b$) is our answer | Return 5 | | |

Example 2.2: Computing $\gcd(5, 20)$

| Step: | Action | Result of Action | | |
|---|---|---|---|---|
| | | $a$ | $b$ | $a = b$? |
| Start: | Seeking $\gcd(12, 30)$ | 12 | 30 | No |
| 1: | Subtract smaller from larger, and set the latter to the this result. $b = 30 - 12$ | 12 | 18 | No |
| 2: | As before ($b$ is still the larger) $b = 18 - 12$ | 12 | 6 | No |
| 3: | As before, but $a$ is now the larger we subtract $b$ from $a$ $a = 12 - 6$ | 6 | 6 | Yes |
| 4: | $a = b$, so $a$ (or equivalently $b$) is our answer | Return 6 | | |

Example 2.3: Computing $\gcd(12, 30)$ using the Euclidean Algorithm

subtracting the smaller from the larger until they are both the same.

A few examples (with small numbers) will probably make more sense of that.

For our first example, let's take a really simple of of looking for the greatest common divisor of 5 and 20. In more mathy notation we are looking for $\gcd(5, 20)$.

In example 2.2 we get the answer 5. Five divides 5, and five divides 20, so it is a common divisor. And unless you want a proof that this algorithm works, you will have to take it on authority of those who have worked through the proof that it is also the *greatest* common divisor.

Now let's look at a slightly more complicated example in which we want to use the Euclid's algorithm to find $\gcd(12, 30)$. The steps are spelled out in example 2.3.

| Step: | Action | Result of Action | | |
|---|---|---|---|---|
| | | $a$ | $b$ | $a = b$? |
| Start: | Seeking gcd$(9, 22)$ | 9 | 22 | No |
| 1: | Subtract smaller from larger, and set the latter to the this result. $b = 22 - 9$ | 9 | 13 | No |
| 2: | $b = 13 - 9$ | 9 | 4 | No |
| 3: | Now $a$ is larger. $a = 9 - 4$ | 5 | 4 | No |
| 4: | $a = 5 - 4$ | 1 | 5 | No |
| 5: | $a$ is smaller (again). $b = 5 - 1$ | 1 | 4 | No |
| 6: | $b = 4 - 1$ | 1 | 3 | No |
| 7: | $b = 3 - 1$ | 1 | 2 | No |
| 8: | $b = 2 - 1$ | 1 | 1 | Yes |
| 9: | $a = b$, so $a$ (or equivalently $b$) is our answer | | Return 1 | |

Example 2.4: Computing gcd$(9, 22)$. When there is no common divisor greater than 1, the two numbers are coprime.

Now let's try this for gcd$(9, 22)$ as shown in example 2.4.

When the gcd of two numbers is 1, as it is in example 2.4, it means that nothing larger than 1 divides both numbers. We call such pairs of numbers "coprime" or "relatively prime" to each other.

### 2.2.3 Let a computer do it

It really is a good idea to get a sense of how the Euclidean Algorithm works, and so I strongly recommend that you work through the examples given and perhaps a few more of your own devising, but computers were invented for a reason, and this is one of them. Remember, however, that these are all toy implementations meant to illustrate the core concepts of the algorithms. Code for handling special cases (such as negative number input) is not included.

Listing 2.2 shows a function for doing this in Python.

**Listing 2.2** Euclid's Algorithm (with repeated subtraction)

```python
def gcd(a: int, b: int) -> int:
    """Greatest Common Divisor: Euclid's Algorithm"""
    while a != b:
        if a > b: a = a - b
        else: b = b - a
    return a
```

GCD through modular reduction

We can actually cut through (or hide) a whole bunch of the repeated subtraction by using the remainder option.

In example 2.3 on page 16, where we repeatedly subtracted 12 from 30 until we got 6, we could have simply used the fact that when you divide 30 by 12, you get the quotient 2 and a remainder of 6. We only need the remainder here, and the fancy name for this is "modular reduction". This can often be written as "$6 = 30 \bmod 12$". In many programming languages, including Python, the remainder operator is written as '`%`'.

We can skip a whole bunch of the individual subtractions by using modular reduction instead of repeated subtractions. This gives us a simpler way to write the algorithm shown in Listing 2.3.

**Listing 2.3** GCD using modular reduction

```python
def gcd_mod(a: int, b: int) -> int:
    """GCD with mod"""
    while a != 0:
        # make a the new remainder, b the old a
        a, b = b % a, a
    return b
```

The line `a, b = b % a, a` might be hard to read but it is saying, set `a` to `b % a`, and set `b` to the old value of $a$. That line could have been written as three lines:

```python
tmp = a
a = b % a
b = tmp
```

Instead of checking for when `a` and `b` are equal, we check for when the last (implicit) subtraction gets to zero. This goes one subtraction more than the original algorithm, and it returns $b$, which is the value we have before that last (implicit) subtraction. This version of the algorithm, using modular reduction, is the recommended one, and it also will serve as the basis of the Extended Euclidean Algorithm which we will get to in §2.2.6.

GCD through recursion

There is, of course, another way to write the Euclidean Algorithm. I list it in 2.4, but won't talk about it other than to say it is slow and consumes a great deal of memory. Don't use it. But I do feel that it is the best expression of the Euclidean Algorithm.

All three of these GCD implementations work, but `gcd_mod()` is the better one to use in practice. But don't even use it, as it behaves very badly on some input.

**Listing 2.4** Recursive GCD algorithm: If programming languages captured mathematical elegance, then this would be the right implementation. But as things stand, this is slow and consumes a great deal of memory.

```
def gcd_recursive(a: int, b: int) -> int:
    """Greatest Common Divisor: Recursive algorithm"""
    if b == 0:  return a
    else: return gcd_recursive(b, a % b)
```

Euclid versus common divisors

I mentioned that the method (illustrated in example 2.1 on page 15) which involves first finding all of the divisors is only suitable for small numbers. Suppose we are seeking to find the greatest common divisor of 57 130 249 371 827 and 7 433 019 835 183. Those numbers are still tiny by cryptographic standards, but they are big enough to illustrate the point. It takes 56 subtractions using the implementation of Euclid's Algorithm from listing 2.2 on page 17. Finding the divisors could take about 50 thousand multiplications.[6] The gcd in this case turns out to be 8 517 479. So even if Euclid's Algorithm is not what you learned in school[7] for simplifying fractions, you were rarely asked to find the factors of numbers greater than 100.

### 2.2.4 Least Common Multiple

We are later going to need the least (smallest) common multiple of two numbers. You did this back in school when learning how to add factions. Suppose you were tasked with adding $\frac{3}{10} + \frac{1}{25}$. You needed to convert that into something where the denominators are the same, and that meant finding the least common multiple of 10 and 25. One way to find a common multiple is to just multiply the two numbers together. That would get you 250 as a common multiple, and you would end up adding $\frac{75}{250} + \frac{10}{250}$.

But if you wanted to work with smaller numbers, the whole thing would be easier if you found a smaller common multiple of 10 and 25. The smallest common multiple is 50. So you can convert the problem to $\frac{15}{50} + \frac{2}{50}$ giving you an answer of $\frac{17}{50}$.

It turns out that if you can compute the gcd of the numbers easily it is easy to find the least common multiple. You multiply the two numbers and then divide by their greatest common divisor.

[6]I am estimating based on Fermat's factoring algorithm. There are more efficient ways to find divisors of big numbers, but Fermat's method is far more efficient than trial divisions.

[7]Meijer calls that the "high school method" [Mei16, §2.3], although it tends to be taught around 6th grade in the United States. Perhaps high school is the last time most people ever sought a gcd.

Consider the least common multiple of 12 and 20. Multiples of 12 include 12, 24, 36, 48, 60, 82, and more. Multiples of 20 include 20, 40, 60, 80, 100, 120, and so on. If we multiply 12 by 20 we get 240, which is a common multiple, but it is not the least common multiple. Looking at our list of multiples, we see that 60 is the smallest one in common. The gcd(12, 20) is 4. and $60 = (12 \times 20)/4$. That is not a coincidence.

We don't want negative answers, so we take the absolute value `abs()` of the product as shown in listing 2.5.

**Listing 2.5** Least Common Multiple

```python
def lcm(a: int, b: int) -> int:
    """Least common multiple"""
    # `//` is integer division in Python.
    return abs(a*b)//gcd(a, b)
```

The result should always be an integer, but we use the `//` operator to get an integer type in Python here.

### 2.2.5 Modular inverse

With ordinary arithmetic, there is a notion of reciprocal. The reciprocal of 3 is $\frac{1}{3}$. The reciprocal of $\frac{24}{5}$ is $\frac{5}{24}$. The defining characteristic of a reciprocal is that when you multiply a number by its reciprocal you get 1. In what follows (and much of anything you will read on this stuff), we write the reciprocal of $n$ as '$n^{-1}$'. It might occasionally be written as $\frac{1}{n}$, but it is more likely to be written as '$n^{-1}$'. Get used to it. Also the fancy name for reciprocal is "multiplicative inverse".

If we are only dealing with integers (as we are), then you would think that 1 is the only number that has an integer reciprocal. 1 is its own reciprocal. But when we do all of our arithmetic modulo some modulus, we might find that other numbers have multiplicative inverses.

In ordinary multiplication the reciprocal of 7 is $\frac{1}{7}$, which is not an integer. If, however, we are working modulo 15 we can find (as shown in from equation 2.3) that there is an integer (in this case 13) that you can multiply 7 by to get 1.

$$7 \times 13 \equiv 91 \equiv 1 \pmod{15} \tag{2.3}$$

and so 13 is the multiplicative inverse of 7 (mod 15). Because I am slowly trying to indoctrinate you all into algebraic notation, we will express that as in equation 2.4

$$7^{-1} \equiv 13 \pmod{15} \tag{2.4}$$

| $\times$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 2 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 1 | 3 | 5 | 7 | 9 | 11 | 13 |
| 3 | 3 | 6 | 9 | 12 | 0 | 3 | 6 | 9 | 12 | 0 | 3 | 6 | 9 | 12 |
| 4 | 4 | 8 | 12 | 1 | 5 | 9 | 13 | 2 | 6 | 10 | 14 | 3 | 7 | 11 |
| 5 | 5 | 10 | 0 | 5 | 10 | 0 | 5 | 10 | 0 | 5 | 10 | 0 | 5 | 10 |
| 6 | 6 | 12 | 3 | 9 | 0 | 6 | 12 | 3 | 9 | 0 | 6 | 12 | 3 | 9 |
| 7 | 7 | 14 | 6 | 13 | 5 | 12 | 4 | 11 | 3 | 10 | 2 | 9 | 1 | 8 |
| 8 | 8 | 1 | 9 | 2 | 10 | 3 | 11 | 4 | 12 | 5 | 13 | 6 | 14 | 7 |
| 9 | 9 | 3 | 12 | 6 | 0 | 9 | 3 | 12 | 6 | 0 | 9 | 3 | 12 | 6 |
| 10 | 10 | 5 | 0 | 10 | 5 | 0 | 10 | 5 | 0 | 10 | 5 | 0 | 10 | 5 |
| 11 | 11 | 7 | 3 | 14 | 10 | 6 | 2 | 13 | 9 | 5 | 1 | 12 | 8 | 4 |
| 12 | 12 | 9 | 6 | 3 | 0 | 12 | 9 | 6 | 3 | 0 | 12 | 9 | 6 | 3 |
| 13 | 13 | 11 | 9 | 7 | 5 | 3 | 1 | 14 | 12 | 10 | 8 | 6 | 4 | 2 |
| 14 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |

Figure 2.1: Multiplication table for 1 through 14 modulo 15

**Totients** This whole document is optional, but some sections are more optional than others. This is one of those more optional sections. So feel more than free to skip ahead to section 2.2.6. Or just skip the whole thing.[8]

Figure 2.1 shows a multiplication table for the numbers 1 through 14 modulo 15. If you stare at it enough, you might notice a few patterns. There will also be some patterns that you are not going to notice on your own, but do take a look at to see what you do notice.

1. We could have gotten by with only half of the table because $a \times b$ is the same as $b \times a$. For example, if you look *across* row 4 you should see the same sequence of numbers that you would get by looking *down* column 4.

   (Also this would be a good time for you to check work through $4 \times 9 \pmod{15}$ to make sure that you get 6.)

2. Not all rows (or columns) have a 1 in them. For example, there is nothing you can multiply 6 by to get 1. But there is something you can 8 by to get 1: $8 \times 2 \equiv 1 \pmod{15}$. This means that 8 has an inverse modulo 15, but 6 does not.

[8]Although I put in way too much work into creating this document and would love to have it appreciated, I recognize many of the limits of its usefulness.

Look at the row for 8 and the row for 6 (or the columns if you prefer) and see what else you notice. (Really, take some time to do that now instead of reading on.)

3. Each row (or column) with a 1 in it has all of the numbers 1 through 14; none of the other rows (or columns) do.

4. Which numbers do have modular inverses? There are eight of them. Can you catch them all?

5. None of the rows (or columns) that have a 1 in them have a 0 in them. All of the rows (and columns) that do not have a 1 in them do have a 0.

6. The numbers that have modular inverses modulo 15 are 1, 2, 4, 7, 8, 11, 13, 14. Do those numbers have a special relationship with 15 that the other numbers do not?

   (Yes. Yes they do. Hint: Look at the GCD of 6 and 15 and of 8 and 15.)

What you will not have noticed (unless you should be writing this document instead of reading it) it that the number (8) of numbers that have a modular inverse stands in a special relationship with the prime factors of 15. Fifteen is the product of 3 and 5, and eight is the product of $(3 - 1)$ and $(5 - 1)$. When some number $N$ is the product of two distinct primes, $p$ and $q$, then the number of numbers less than $N$ which are coprime to $N$ is $(p - 1)(q - 1)$. This quantity is called the (Euler) totient[9] or Euler's $\varphi$ (phi).[10] In the present case, we could write $\varphi(15) = 8$. To compute $\varphi(N)$ one needs the prime factorization of $N$.

If we toss out the numbers that are not coprime with 15 (and so don't have inverses mod 15) we get a nice tidy group of eight numbers modulo 15 with convenient mathematical properties.

[9]It is typically just called the totient, but when we need to distinguish it (as we will) from the Carmichael totient we say "Euler". Note also that the proper pronunciation of "Euler" sounds a lot like "oiler" and never like "yuler".

[10]The small Greek letter phi can be written as either '$\phi$' or '$\varphi$'. You may see both when people write about Euler's totient. I prefer the latter variant because it is prettier. In American English the Greek letter $\varphi$ is pronounced as the word "fie", while in British English it is pronounced "fee". I don't know what Canadians say, but you might encounter either.

| × | 1 | 2 | 4 | 7 | 8 | 11 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 2 | 4 | 7 | 8 | 11 | 13 | 14 |
| 2 | 2 | 4 | 8 | 14 | 1 | 7 | 11 | 13 |
| 4 | 4 | 8 | 1 | 13 | 2 | 14 | 7 | 11 |
| 7 | 7 | 14 | 13 | 4 | 11 | 2 | 1 | 8 |
| 8 | 8 | 1 | 2 | 11 | 4 | 13 | 14 | 7 |
| 11 | 11 | 7 | 14 | 2 | 13 | 1 | 8 | 4 |
| 13 | 13 | 11 | 7 | 1 | 14 | 8 | 4 | 2 |
| 14 | 14 | 13 | 11 | 8 | 7 | 4 | 2 | 1 |

Figure 2.2: Mod 15 multiplication table for the integers coprime with 15. Groups constructed this way have lots of properties that work out to be useful and interesting. This is called a "multiplicative group" and may be notated with something like '$\mathbb{Z}_{15}^{\times}$'.

### 2.2.6 Extended Euclidean algorithm

With a small number like 15, we can just create a multiplication table (as in Figure 2.1 on page 21) and look for the inverses. But this takes at least as much computation as factoring, so it won't work for big numbers. Fortunately there is a way to find the modular inverse of a number using an extension of the Euclidean algorithm for computing computing the greatest common divisor.

The extended Euclidean algorithm (EEA) computes the the $\gcd(a, b)$, but it also computes integers $x$ and $y$ such that

$$ax + by = \gcd(a, b) \tag{2.5}$$

In the case where $a$ and $b$ are coprime, the GCD will be 1. And with a bit more algebra than anyone wants to work through, if $ax + by = 1$, with all of the things being integers, then $x$ is the inverse of $a$ modulo $b$.

The function `egcd(a, b)` will give us not only the GCD of the input, but also the $x$ and $y$. We don't need the $y$.

The reason that we need to return `x % m` instead of just `x` is that there are many values for $x$ that are modular inverses when we consider negative numbers or numbers greater than $m$. Just as $3 \equiv 7 \pmod{10}$ so is 13, 23, 33, 2938473, as well as $-17$ and many more. We say that those numbers are "congruent" modulo 10, and typically use the symbol '$\equiv$' instead of '$=$'. It is convenient[11] to work with the smallest positive inverse, and that last remainder operations makes sure that we get this.

[11]When first implementing our web-client in Safari we encountered a bug that Rob Yoder

**Listing 2.6** Modular inverse

```
def modinv(a: int, m: int) -> int:
    """return x such that (x * a) % m == 1"""
    g, x, _ = egcd(a, m)
    if g != 1:
        raise Exception('gcd(a, m) != 1')
    return x % m
```

To get the $x$ from equation 2.5 on the preceding page, the algorithm needs to keep track of intermediate values of $x$ and $y$. And it has to recalculate these with every modular reduction based on prior values and the quotient from the most recent division. The bookkeeping for all of that makes my head spin, and while I see from the proofs that it does work, I don't have a firm intuition of why it works. In general, it is recomputing intermediate values of $x$ and $y$ to keep something like equation 2.5 true as the computation progresses.

Anyway, during the loop we have in `gcd_mod`, we need to keep track of $x$ and $y$, we also need to know what they were in the previous round. We can use a two-by-two matrix $A = \left(\begin{smallmatrix} y_0 & y_1 \\ x_0 & x_1 \end{smallmatrix}\right)$ to keep track of these. It's initial state is the identity matrix, $A_0 = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$. Each time through the loop after we compute a remainder and quotient we update $A$ as in equation 2.6. The $i$=th time through the loop, the new value for $A$ is computed from the previous value of $A$ (called '$A_{i-1}$') and the quotient, $q$, of $b/a$.

$$A_i = A_{i-1} \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix} \tag{2.6}$$

This gives us an implementation (in listing 2.7) of the extended Euclidean algorithm that should not be used in real life. It does help promote the intuition that each time through we are performing a linear transformation on our $x$s and $y$s.

Now we move on to the implementation that should be used. Instead of creating matrices (with all of the overhead that entails) it just unwinds the steps of the matrix multiplication.

traced to occurring when in the private key $p < q$. It wasn't Safari that was generating such private keys, but Safari at time was barfing on them. Apple optimized their modular inverse computation along with some other things, that worked fine when $p > q$. The standards were less than helpful, in that they seemed to assume that $p$ would always be greater than $q$, but they never stated it as a requirement.

**Listing 2.7** Extended Euclidean Algorithm with matrix multiplication

```python
def egcd_m(a: int, b: int) ->Tuple[int, int, int]:
    """Extended GCD using matrix multiplication"""

    # first set A as the identity matrix
    A = np.identity(2, dtype=object)
    while a != 0:
        (q, a), b = divmod(b, a), a
        R = np.array([[0, 1], [1, -q]], dtype=object)
        A = np.matmul(A, R)
    return b, A[1,0], A[0,0]
```

**Listing 2.8** Extended Euclidean Algorithm

```python
def egcd(a: int, b: int) ->Tuple[int, int, int]:
    """return (g, x, y) such that a*x + b*y = g = gcd(a, b)"""

    x0, x1, y0, y1 = 0, 1, 1, 0
    while a != 0:
        (q, a), b = divmod(b, a), a
        y0, y1 = y1, y0 - q * y1
        x0, x1 = x1, x0 - q * x1
    return b, x0, y0
```

## 2.3 RSA Public keys

The part of a public RSA key that is unique to each user is sometimes called "the public modulus". It is the product of two prime numbers, and is typically referred to as '$N$'.

There is another part, called "the public exponent", that is not unique. The public exponent, typically given the variable name '$e$', needs to have certain mathematical properties, but it is also convenient for it to have very few 1s in its binary representation. Back in the old days, 3 and 17 were typically used for the public exponent, but someone discovered some problems if it is too small. These days, it is usually set to 65537, but we will use default to 17, because we are illustrating things with small numbers.

A public RSA `PubKey`, then has two values in it. The `__init__()` function listed in 2.9 allows us to create a new public key with something like `MyPublicKey = PubKey(17, 4183)`.

### 2.3.1 Encryption

Encrypting something using a public key is very simple. The ciphertext, $c$, is simply the message $m$ you want to encrypt raised to the public exponent,

**Listing 2.9** Public key members

```
class PubKey:
    """An RSA public key"""
    def __init__(self, exp: int, N: int):
        self.e = exp
        self.N = N
```

all modulo the public modulus.

$$c = m^e \pmod{N} \tag{2.7}$$

Python has a built-in modular exponentiation function, `pow()`, that does – you guessed it – modular exponentiation. This makes the code for RSA encryption simple.

**Listing 2.10** `encrypt() method for PubKey`

```
    def encrypt(self, ptext: int) -> int:
        """Encrypt a plaintext (int) using the public key   """
        return pow(ptext, self.e, self.N)
```

## 2.4 Private keys

With the functions for computing modular inverses (listing 2.6 on page 24), which depends on the Extended Euclidean Algorithm (§2.2.6), and which in turn depends on the Euclidean Algorithm for computing Greatest Common Divisors (§2.2) which depends on integer division (§2.1.3) we now have everything we need to handle private keys and RSA decryption.[12] So if you skipped a bunch of the math to get straight to public keys, it is now time to pop back to §2.1.3 on page 13.

In one sense an RSA private key just needs to be the two prime factors $p$ and $q$ of the public modulus $N$ along with the public exponent. All of the other parts can be computed from those. However, it makes sense to pre-compute some of these, the most important of which is the decryption exponent $d$.

We can initialize it with the primes $p$ and $q$ along with the public exponent $e$ (default of 17). We can set up a `KeyPair` as something that has

[12]This assumes that we don't use the Chinese Remainder Theorem (CRT) in our decryption. An implementation designed for working with larger numbers would.

**Listing 2.11** Initializing a key pair from primes

```
class KeyPair(PubKey):
    """An RSA key pair"""
    def __init__(self, p: int, q: int, exp:int=17):
        primes = sorted([p, q])
        self.q, self.p = primes[0], primes[1]
        self.e = exp
        self.N = p * q
        try: self.d = self._compute_d()
        except Exception as e:
            raise Exception("Could not compute decryption exponent") from e
```

everything a `PubKey` has and more. See footnote 11 on page 23 for some hint into why we ensure that the prime we store as `p` is the larger of the two.

It will be useful to have a method get the public key out of a key pair. `KeyPair.pub_key()` is shown in listing 2.12

**Listing 2.12** Get a public key from a key pair

```
    def pub_key(self) -> PubKey:
        """Get public key"""
        return PubKey(self.e, self.N)
```

Before getting to how the decryption exponent ($d$) is computed, let me show how it is used in equation 2.8, in which $m$ is the decrypted message, $c$ is the ciphertext, $N$ is the public modulus, and $d$ is the decryption exponent.

$$m = c^d \pmod{N} \tag{2.8}$$

Once we have $d$ then the code to decrypt is simple (though see footnote 12), as shown in listing 2.13.

**Listing 2.13** Decrypting with a private key

```
    def decrypt(self, ctext: int) -> int:
        """decrypt ctext"""
        return pow(ctext, self.d, self.N)
```

### 2.4.1 Decryption exponent

The decryption exponent, $d$, is a secret that should only be available to the holder of the private key. When $p$ and $q$ are sufficiently large primes and $N = pq$, it is easy (there is a polynomial time algorithm) to compute $d$ from $p$, $q$, and $e$. It can be proved that computing $d$ from just $e$ and $N$ is as hard as factoring $N$ into its prime factors.[13] So what is $d$? There isn't a really good order in which to explain this, so be prepared to read through this section multiple times. The decryption exponent ($d$) is the multiplicative inverse of the public exponent ($e$) modulo the the Carmichael totient of the public modulus $N$. The Carmichael totient of a number $N$ is typically written using the Greek letter lambda as '$\lambda(N)$'. We can define $d$ as in equation 2.9 and compute it as in listing 2.14.

$$d = e^{-1} \mod \lambda(N) \tag{2.9}$$

**Listing 2.14** Computing the decryption exponent

```
    def _compute_d(self) -> int:
        λ = lcm(self.p - 1, self.q - 1)
        try: return  modinv(self.e, λ)
        except Exception as e:
            raise Exception("Inverse of e mod λ does not exist") from e
```

The Carmichael totient can be computed as the least common multiple of one less than each of the distinct prime factors of a number. In our case, where $p$ and $q$ are the prime factors of $N$ that means that

$$\lambda(N) = \operatorname{lcm}(p-1, q-1) \quad (p \text{ and } q \text{ are distinct prime factors of } N) \tag{2.10}$$

Often times the Carmichael totient, $\lambda(N)$, will be the same as the Euler totient, $\varphi(N)$; but sometimes $\lambda(N)$ will be a factor of $\varphi(N)$.[14]

[13] Just as we can compute $d$ from $p$, $q$, and $e$, there is a probabilistic polynomial time algorithm to compute $p$ and $q$ from $d$, $N$, and $e$. Thus we can reduce finding $d$ to factoring $N$ and reduce factoring $N$ to finding $d$. So they are just as hard. This does not prove that the RSA algorithm is as hard to break as the factoring problem; it only proves that RSA *key recovery* is as hard as factoring. There may be ways to break RSA that do not involved finding $d$.

[14] If you correctly compute a Patty and Molly story for the 1Password white paper, but then provide the details of the computation using $\varphi(N)$ instead of $\lambda(N)$, you may find yourself in the embarrassing situation in which you have told people to work through the text and that the "the numbers are chosen to actually work" only to find that it doesn't actually work.

# Chapter 3

# Why does it work

If this chapter existed, it would contain an attempt at an explanation for why the math and algorithms described in Chapter 2 yield the kind of encryption which we see illustrated in Chapter 1.

But this chapter does not exist. You are hallucinating it.

## 3.1 Why totients matter

Remember the section 2.2.5 which you skipped over? Go back and read it now. Even if you didn't skip over it, go back and read it again.

Welcome back. In 1640 Pierre de Fermat stated (with some fixes, modern notation, and not in French) that if you raise a number to a prime power and you reduce it modulo that same prime you end up with the number you started with.

Theorem 1 **Fermat's Little Theorem** If $p$ is prime and $a$ is not a multiple of $p$ then

$$a^p \equiv a \pmod{p}$$

Theorem 1 is called "Fermat's *Little* Theorem" (FLT) to distinguish it from "Fermat's *Last* Theorem." The latter is famous for the fact that Fermat did not prove it; Fermat didn't prove the little theorem either, but it's not famous for that. Fermat's Little Theorem was proved by Euler in 1736. What really distinguishes the two is that the little theorem is enormously useful. Here we will use the abbreviation "FLT" to refer to the *little* theorem.

To illustrate it, let's set our prime $p$ to 7 and do a bunch of exponentiations as shown in Table 3.1.

If we divide both sides of the relation in theorem 1 by $a$ we get an alternate form of the theorem stated in 2. Recall that when we say "divide

| $a$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $a^7$ | 1 | 128 | 2187 | 16 384 | 78 125 | 279 936 |
| $a^7 \pmod 7$ | 1 | 2 | 3 | 4 | 5 | 6 |
| $a^{7-1} \pmod 7$ | 1 | 1 | 1 | 1 | 1 | 1 |

Table 3.1: Illustrating Fermat's Little Theorem with $p = 7$

by $a$" what we really mean is "multiply by the multiplicative inverse of $a$." That is, instead of saying something like $x/a$, we say, $xa^{-1}$.

$$\begin{aligned} a^p &\equiv a \pmod p && \text{(Statement of theorem 1)} \\ a^p a^{-1} &\equiv a a^{-1} \pmod p && \text{(Multiply both sides by } a^{-1}\text{)} \\ a^{p-1} &\equiv 1 \pmod p && \text{(After the multiplication)} \end{aligned}$$

This form of it is more common. If you don't see how theorems 1 and 2 are the same, it is worth spending time going over it until it makes sense. We will be switching back and forth between the two forms rapidly.

Corollary 2 **Also Fermat's Little Theorem** If $p$ is prime and $a$ is not $a$ multiple of $p$ then

$$a^{p-1} \equiv 1 \pmod p$$

Can we generalize Fermat's Little Theorem to cases when when $p$ is not prime? Well, we can't, but Euler could. When he proved FTL, he proved a generalization of it. Recall (I did tell you to go back and read §2.2.5) that the totient of a count of the coprimes less than it. There are eight numbers less than 15 which are coprime with 15. This gets written as "$\varphi(15) = 8$" using the greek letter phi.

Euler proved theorem 3, an extension of FLT, that works not just for prime moduli but for any modulus.

Theorem 3 **Euler's totient theorem** If $\gcd(a, n) = 1$ (that is, if $a$ and $n$ are coprime) then

$$a^{\varphi(n)} \equiv 1 \pmod n$$

where $\varphi(n)$ is the totient of $n$.

The numbers less than 10 which are coprime with 10 are 1, 3, 7, and 9. There are four of those, so $\varphi(10) = 4$. This means that if we raise any number (that is coprime with 10) to the 4-th power the result should be 1

| $a$ | 1 | 3 | 7 | 9 | 27 |
|---|---|---|---|---|---|
| $a^4$ | 1 | 81 | 2401 | 6561 | 531 441 |
| $a^4 \pmod{10}$ | 1 | 1 | 1 | 1 | 1 |

Table 3.2: Illustrating Euler's theorem with $n = 10$. $\varphi(10) = 4$, so 4 will be our exponent, while 10 is the modulus. We can set $a$ to something greater than 10 (as long $a$ and 10 are coprime), but for all computation mod10 we can first reduce $a \mod 10$. That is, because, say $27 \equiv 7 \pmod{10}$ we only need to test with 7 as 17, 27, 37, etc will have the same result.

more than a multiple of 10. So let's look at this for the numbers coprime with 10 that are less than 10 along with a number greater than 10.

The last column of table 3.2 is to illustrate the case where $a$ (27) is larger than our modulus (10). But when working in a mod 10 world, we would just reduce 27 to 7 before doing any other computation. So because we have tested with 7 there is no point in testing with 27.[1]

A very useful fact that follows from Euler's Theorem is that

$$\begin{aligned} a^{\varphi(N)+1} &\equiv a \times a^{\varphi(N)} \pmod{N} \\ &\equiv a \times 1 \pmod{N} \\ &\equiv a \pmod{N} \end{aligned} \tag{3.1}$$

### 3.1.1 Computing totients

Computing the totient of a number $n$ requires knowing the prime factors of $n$. The general formula using the factors of $n$ to compute $\varphi(n)$ can be kind of messy. Fortunately we only need a simplified version. When we are talking about exactly two prime factors of $n$ and those factors not equaling each other, then computing the totient is simple and shown in equation 3.2.

$$\varphi(n) = (p-1)(q-1) \qquad \text{(When } p, q \text{ are prime and } p \neq q.) \tag{3.2}$$

If we look at 15, which is the product of 3 and 5, we can see that $\varphi(15) = (3-1)(5-1) = 8$.

In the special case when $n$ itself is prime the number of coprimes less than it will be $n - 1$. So in the case when $n$ is prime, Euler's theorem (3)

[1]Well, Euler's theorem is a theorem with a solid proof, so there really isn't any point of testing any of this other than as an illustration or to help see what it means.

reduces to Fermat's little theorem (2). Because FLT is just a special case of Euler's theorem, it will sometimes be referred to as "Euler's theorem." After all, Euler is the one who proved it. But there are so many things named for Euler, you will sometimes see Euler's theorem referred to as "Fermat's little theorem" as it is just a generalization of it. What you will never hear from a mathematician is it referred to as "you-ler's theorem" (see footnote 9 on page 22).

## 3.2 Putting it all together

Recall that encrypting a message $m$ is done by raising $m$ to public exponent $e$ power, all modulo the public key, $N$. $N$ is the product of two primes, $p$ and $q$.

$$c = m^e \mod N \tag{2.7}$$

A decryption (private) key is computed using the totient of $N$, as in equation 2.9,

$$d = e^{-1} \mod \varphi(N) \tag{2.9}$$

And decryption is

$$m = c^d \mod N \tag{2.8}$$

Recall what it means for something to be the modular inverse of something else. Anything times its modular inverse is 1. So that $d$ is computed as the inverse of $e$ modulo $\varphi(N)$ then we can see in (3.3) that $e$ times $d$ is one more than an integer multiple of the totient of $N$.

$$\begin{aligned} ed &= 1 \pmod{\varphi(N)} \\ ed &= k\varphi(N) + 1 \end{aligned} \tag{3.3}$$

From Euler's Theorem we know that

$$a^{\varphi(N)} = 1 \pmod{N}. \tag{3.4}$$

and we know that $a \times 1 = a \pmod{N}$ and $a \times a^x = a^{x+1}$. This gives us a fact that will be useful shortly.

$$a^{\varphi(N)+1} = a \pmod{N}. \tag{3.5}$$

And so

$$a^{k\varphi(N)+1} = a \pmod{N} \qquad \text{(for any positive integer } k\text{)} \tag{3.6}$$

If you don't see why (3.6) follows from (3.5) remember that $a^{2x} = a^x a^x$, and so $a^{2x+1} = a^x a^x a$.

Now let's pull apart what $c^d$ is, using the various useful facts discussed above.

$$\begin{aligned} c^d &= (m^e)^d \pmod N \\ &= m^{ed} \pmod N \\ &= m^{k\varphi(N)+1} \pmod N \\ &= m \pmod N \end{aligned} \tag{3.7}$$

And there we have it.

- $d$ allows one to compute the original message $m$ from the ciphertext $c$ and the public modulus $N$.
- $d$ is computed from the public exponent $e$ and the totient of $N$.
- The totient of $N$ is computed from the prime factors of $N$.

Is there a way to decrypt $c$ without learning $d$ or the factors of $N$? There is no proof to that effect.

# Bibliography

[Agi21] AgileBits. *1Password Security Design.* Version v0.4.0. Oct. 26, 2021. URL: `https://1password.com/teams/white-paper/` (visited on 12/21/2021).

[Mei16] Alko R Meijer. *Algebra for Cryptologists.* 1st ed. Springer Undergraduate Texts in Mathematics and Technology. Springer, Cham, 2016. ISBN: 9783319303963.

# Colophon

This document started out as an attempt to blend documentation, including rationales and explanations, into some Python code. This led me down a rabbit hole of experimenting with a variety of tools for literate programming with Python as well as far more discussion of code than originally anticipated. I personally learned a great deal, even if this was not the best use of my time.

The source for both the documentation and the Python code are in TeX Weave files, with the TeX files to be woven by `pweave` and the Python files to be tangled by `ptangle`. Both of those are from Pweave version 0.30.3. Unfortunately, characteristics of the Python language require more sophisticated weaving than Pweave offers.

Python 3.10.0 was used for all of the Python code. XeTeX 3.141592653-2.6-0.999993 is the TeX engine, with LaTeX $2_\varepsilon$ 2021-06-01. The fonts are all from the Computer Modern Unicode super family of typefaces.