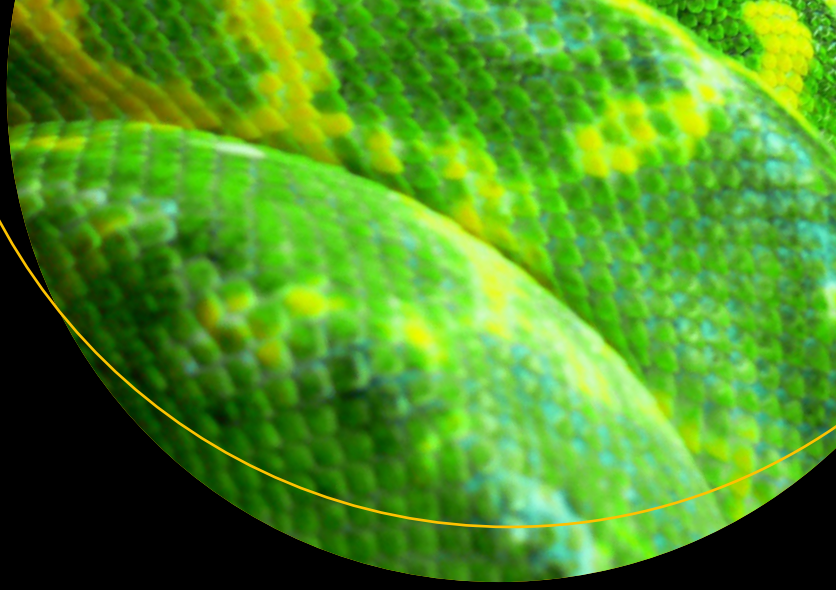

Practical Cryptography in Python

Learning Correct Cryptography by Example

Seth James Nielson
Christopher K. Monson

Apress®

Practical Cryptography in Python

Learning Correct Cryptography by Example

Seth James Nielson
Christopher K. Monson

Apress®

Practical Cryptography in Python: Learning Correct Cryptography by Example

Seth James Nielson
Austin, TX, USA

Christopher K. Monson
Hampstead, MD, USA

ISBN-13 (pbk): 978-1-4842-4899-7
ISBN-13 (electronic): 978-1-4842-4900-0
https://doi.org/10.1007/978-1-4842-4900-0

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Susan McDermott
Development Editor: Laura Berendson
Coordinating Editor: Rita Fernando

Cover designed by eStudioCalamar

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484248997. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

For Saige, who hopes to be a Computer Scientist like Daddy.
—Seth

To Mom and Dad, who valued good writing
and never settled for less than my best.
—Chris

Table of Contents

About the Authors xi

About the Technical Reviewer xiii

Introduction xv

Chapter 1: Cryptography: More Than Secrecy 1

Setting Up Your Python Environment 1

Caesar's Shifty Cipher 3

A Gentle Introduction to Cryptography 12

Uses of Cryptography 13

What Could Go Wrong? 14

YANAC: You Are Not A Cryptographer 15

"Jump Off This Cliff"—The Internet 16

The cryptodoneright.org Project 17

Enough Talk, Let's Sum Up 18

Onward 19

Chapter 2: Hashing 21

Hash Liberally with hashlib 21

Making a Hash of Education 25

Preimage Resistance 27

Second-Preimage and Collision Resistance 33

Digestible Hash 36

Pass Hashwords...Er...Hash Passwords 39

Pick Perfect Parameters 44

Cracking Weak Passwords 45
Proof of Work 48
Time to Rehash 52

Chapter 3: Symmetric Encryption: Two Sides, One Key 53

Let's Scramble! 53
What Is Encryption, Really? 57
AES: A Symmetric Block Cipher 58
ECB Is Not for Me 60
Wanted: Spontaneous Independence 70
Not That Blockchain 71
Cross the Streams 86
Key and IV Management 91
Exploiting Malleability 96
Gaze into the Padding 99
Weak Keys, Bad Management 107
Other Encryption Algorithms 109
finalize() 109

Chapter 4: Asymmetric Encryption: Public/Private Keys 111

A Tale of Two Keys 111
Getting Keyed Up 112
RSA Done Wrong: Part One 114
Stuffing the Outbox 122
What Makes Asymmetric Encryption Different? 126
Pass the Padding 128
Deterministic Outputs 129
Chosen Ciphertext Attack 131
Common Modulus Attack 135
The Proof Is in the Padding 138
Exploiting RSA Encryption with PKCS #1 v1.5 Padding 142

Step 1: Blinding 148
Step 2: Searching for PKCS-Conforming Messages 150
Step 3: Narrowing the Set of Solutions 156
Step 4: Computing the Solution 158
Additional Notes About RSA 160
Key Management 161
Algorithm Parameters 162
Quantum Cryptography 162
Really Short Addendum 163

Chapter 5: Message Integrity, Signatures, and Certificates 165

An Overly Simplistic Message Authentication Code (MAC) 165
MAC, HMAC, and CBC-MAC 168
HMAC 169
CBC-MAC 174
Encrypting and MACing 181
Digital Signatures: Authentication and Integrity 183
Elliptic Curves: An Alternative to RSA 193
Certificates: Proving Ownership of Public Keys 195
Certificates and Trust 208
Revocation and Private Key Protection 210
Replay Attacks 210
Summarize-Then-MAC 212

Chapter 6: Combining Asymmetric and Symmetric Algorithms 213

Exchange AES Keys with RSA 213
Asymmetric and Symmetric: Like Chocolate and Peanut Butter 217
Measuring RSA's Relative Performance 218
Diffie-Hellman and Key Agreement 227
Diffie-Hellman and Forward Secrecy 233
Challenge-Response Protocols 240
Common Problems 242

An Unfortunate Example of Asymmetric and Symmetric Harmony 244
That's a Wrap 248

Chapter 7: More Symmetric Crypto: Authenticated Encryption and Kerberos 249

AES-GCM 249
AES-GCM Details and Nuances 254
Other AEAD Algorithms 258
Working the Network 260
An Introduction to Kerberos 268
Additional Data 291

Chapter 8: TLS Communications 293

Intercepting Traffic 293
Digital Identities: X.509 Certificates 299
X.509 Fields 299
Certificate Signing Requests 302
Creating Keys, CSRs, and Certificates in Python 315
An Overview of TLS 1.2 and 1.3 320
The Introductory "Hellos" 322
Client Authentication 326
Deriving Session Keys 327
Switching to the New Cipher 330
Deriving Keys and Bulk Data Transfer 331
TLS 1.3 337
Certificate Verification and Trusting Trust 339
Certificate Revocation 340
Untrustworthy Roots, Pinning, and Certificate Transparency 341
Known Attacks Against TLS 344
POODLE 344
FREAK and Logjam 345
Sweet32 346
ROBOT 347

CRIME, TIME, and BREACH 347
Heartbleed 348
Using OpenSSL with Python for TLS 348
The End of the Beginning 359

Bibliography 361

Index 363

About the Authors

Seth James Nielson is the Founder and Chief Scientist of Crimson Vista, Inc., a boutique computer security research and consulting company. He is also an adjunct professor at Johns Hopkins University where he teaches network security and has also served as the Director of Advanced Research Projects in the Information Security Institute. As part of his Hopkins work, he co-founded the https://cryptodoneright.org knowledge base, through a generous grant from Cisco.

Christopher K. Monson has a PhD in machine learning and has spent over a decade at Google in various engineering, machine learning, and leadership roles. He has broad experience writing and teaching programming courses in multiple languages and has worked in document password recovery, malware detection, and large-scale secure computing. He is serving as the Chief Technology Officer at Data Machines Corp. and teaches Cloud Computing Security at the Johns Hopkins University Information Security Institute.

About the Technical Reviewer

Mike Ounsworth is a Software Security Architect at Entrust Datacard. He holds an undergraduate degree in computer science with concentrations in mathematics and physics and an MSc in computer science in robotics and artificial intelligence. Professionally, his day job is mainly application security architecture and penetration testing, with some research side projects in cryptography and post-quantum cryptography. Outside of work, he mentors teams competing in the high-school-age FIRST Robotics Competition.

Introduction

The interconnected world of the current era has drastically changed everything, including banking, entertainment, and even statecraft. Despite differences in users, purposes, and security profiles, these digital applications have at least one thing in common: they all require properly applied cryptography to work correctly.

Informally, cryptography is the mathematics of secrets. We need secret codes to make messages unreadable to unauthorized eyes, to make messages unchangeable, and to know who sent the message. Practical cryptography is the design and use of these codes in real systems.

This book is primarily for computer programmers with little or no previous background with cryptography. Although mathematics makes brief appearances in the book, the overall approach is to teach introductory cryptography concepts by example.

Our journey begins with some introductory components, including hashing algorithms, symmetric encryption, and asymmetric encryption. Next, we go beyond encryption and into the realm of digital certificates, signatures, and message authentication codes. The final chapters show how these various elements come together in interesting and useful combinations, such as Kerberos and TLS.

Another important part of cryptography by example is cryptography by bad example! In this book we will break things on purpose to help the reader appreciate what motivates accepted best practices. Exercises and examples include walk-throughs of real vulnerabilities that have afflicted the Internet. The bad examples will help the reader gain a greater intuition of what goes wrong in cryptography and why.

CHAPTER 1

Cryptography: More Than Secrecy

Welcome to the world of practical cryptography! The intent of this book is to teach you enough about cryptography that you can reason about what it does, when certain types can be effectively applied, and how to choose good strategies and algorithms. There are examples and exercises throughout each chapter, usually with a follow-along exercise right at the beginning to help you get your bearings. These examples are often accompanied by some fictitious stage setting to add some context. After you've had some exposure and experience, the technical terms that follow those examples should make more sense and be more memorable. We hope you like it.

Setting Up Your Python Environment

In order to dive in, we'll need a place to swim, and that's a Python 3 environment. If you are already a Python 3 pro and have no trouble installing modules that you discover you need, skip this section and do some actual diving. Otherwise, read on, and we'll get through the setup steps quickly.

All of the examples in this book are written using Python 3 and the third-party "cryptography" module.

If you do not want to mess around with your system Python environment, we suggest creating a Python virtual environment using the venv module. This will configure a selected directory with a Python interpreter and associated modules. Using an "activate" script, the shell is directed to use this custom environment for Python rather than the system-wide installation. Any modules you install are only locally installed.

S. J. Nielson and C. K. Monson, *Practical Cryptography in Python*,
https://doi.org/10.1007/978-1-4842-4900-0_1

We will walk through installing the system in Ubuntu Linux in this section. Installation will be slightly different for other versions of Linux or Unix and may be considerably different for Windows.

First, we need to install Python 3, Pip, and the venv module:

```
apt install python3 python3-venv python3-pip
```

Next, we use venv to set up the environment in an env directory:

```
python3 -m venv env
```

This will set up the interpreter and modules within the path. Once the installation is complete, the environment can be used at any time by the following command:

```
source env/bin/activate
```

You should now see a prefix to your shell prompt with the name of your environment. Once your environment is activated, install the cryptography module. Remember to activate your Python virtual environment first if you don't want cryptography installed system-wide.

```
pip install cryptography
```

We will be using the cryptography module throughout the book. Many times we will refer directly to the module's documentation that can be found online at https://cryptography.io/en/latest/.

For some practices, we will also need the gmpy2 module. This *does* require a few system-wide packages.

```
apt install libmpfr-dev libmpc-dev libgmp-dev python3-gmpy2
```

Once you have these packages installed, you can install the Python gmpy2 module within your virtual environment

```
pip install gmpy2
```

Note that within the virtual environment, you can use "python" instead of "python3" and "pip" instead of "pip3." This is because when you created the environment with venv, you did so using Python3. Within the virtual environment, Python3 is the only interpreter and there is no need to differentiate between version 2 and version 3. If you

install any of these packages system-wide, you may need to use pip3 instead of just pip. Otherwise, the packages might be installed for Python 2.

If you have trouble with `gmpy2` or do not wish to install all the system-wide packages, you can skip this step. There are only a few exercises you will not be able to complete.

Now let's get diving!

Caesar's Shifty Cipher

The two (made-up) countries of East Antarctica (EA) and West Antarctica (WA) don't like each other very much and are spying on each other incessantly. In this scenario, two spies from EA, with code names "Alice" and "Bob," have infiltrated their western neighbors and are sending messages back and forth through covert channels.

They don't like it when their adversaries in West Antarctica read their messages, so they communicate using a secret code.

Unfortunately, East Antarctica is not particularly advanced in the realm of cryptography. For a code, the East Antarctica Truth-Spying Agency (EATSA) creates a simple substitution by replacing each letter with another letter later in the alphabet. Both countries use the standard ASCII alphabet with the letters "A" through "Z."

Suppose for a moment that they choose to code their messages using this substitution technique with the *shift distance* set to 1. In that case, the letter "A" would be replaced with "B," the letter "B" would be replaced with "C," and so on. The last letter of the alphabet, "Z," would wrap around to the beginning and be replaced with "A." This table shows the whole (uppercase) mapping of **plaintext** (original, untouched) letters to **ciphertext** (coded) letters. Non-letters like spaces and punctuation are left intact.

A	B	C	D	E	F	G	H	I	J	K	L	M
B	C	D	E	F	G	H	I	J	K	L	M	N
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
O	P	Q	R	S	T	U	V	W	X	Y	Z	A

Using this table, `HELLO WORLD` encodes to `IFMMP XPSME`.

Now try it with distance 2, where "A" goes to "C," "B" goes to "D," and so on until "Y," which maps to "A," and "Z," which maps to "B."

A	B	C	D	E	F	G	H	I	J	K	L	M
C	D	E	F	G	H	I	J	K	L	M	N	O
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
P	Q	R	S	T	U	V	W	X	Y	Z	A	B

Now, the message `HELLO WORLD` is encoded as `JGNNQ YQTNF`.

Happy with their simple *shift cipher*, the East Antarctica Truth-Spying Agency (EATSA) decides to create a Python program to handle encoding and decoding messages.

Tip: Write Code

This book walks through a lot of sample Python programs. At the beginning of each one, we will list the requirements and perhaps a hint or an overview of a cryptographic API. You should go ahead and try to write the program yourself first. It's fine if you get stuck or make mistakes. Even if you can't figure everything out on your own, your experience with trying to write the program will help you understand the provided samples much better.

EXERCISE 1.1. SHIFT CIPHER ENCODER

Create a Python program that encodes and decodes messages using the shift cipher described in this section. The amount of shift must be configurable.

Let's walk through this exercise together. We use Python 3 for all exercises.

First, let's create a simple function for creating our substitution tables. For simplicity, we will create two Python dictionaries: one containing the encoding table and one creating the decoding table. We will also only encode and decode uppercase ASCII letters, as shown in Listing 1-1.

Listing 1-1. Creating Substitution Tables

```
# Partial Listing: Some Assembly Required

import string

def create_shift_substitutions(n):
    encoding = {}
    decoding = {}
    alphabet_size = len(string.ascii_uppercase)
    for i in range(alphabet_size):
        letter       = string.ascii_uppercase[i]
        subst_letter = string.ascii_uppercase[(i+n)%alphabet_size]

        encoding[letter]       = subst_letter
        decoding[subst_letter] = letter
    return encoding, decoding
```

Observe that this function is parameterized on `n`, the shift parameter. We don't have any error checking in this function; we will check parameters elsewhere. Note, though, that any integer value of `n` is valid because Python handles negative modulus in a reasonable way. Even the value 0 is okay: it just produces a mapping from each character to itself! Values larger than 26 also work fine because we apply a final modulus of `alphabet_size` before indexing into the alphabet.

Now, for encoding and decoding, we simply substitute each letter in a message for one in the corresponding dictionary, shown in Listing 1-2.

Listing 1-2. Shift Encoder

```
# Partial Listing: Some Assembly Required

def encode(message, subst):
    cipher = ""
    for letter in message:
        if letter in subst:
            cipher += subst[letter]
         else:
```

```
 9                cipher += letter
10        return cipher
11
12    def decode(message, subst):
13        return encode(message, subst)
```

Note: Compactness vs. Clarity

We tend to favor universal clarity over compactness when there is a conflict between them. We will even write things in ways that might not be widely considered idiomatic if it helps to illustrate what is happening.

The code in Listing 1-2 has a nice example of favoring clarity over common idioms. An idiomatic function body would probably be a one-liner:

```
def encode(message, subst):
    return "".join(subst.get(x, x) for x in message)
```

That's a lovely bit of Python if you're used to it, but we're trying not to make too many assumptions here.

In our implementation, the `encode` function takes an incoming message and a substitution dictionary. For each letter in the message, we replace it if a substitution is available. Otherwise, we just include the character itself with no transformation (preserving spaces and punctuation).

Obviously, the `decode` operation in this listing is completely unnecessary, but we have included it to emphasize that encoding and decoding in a substitution cipher work exactly the same. Only the dictionary needs to change.

These functions are sufficient to build an application, but for fun we will add in another function in Listing 1-3 to take a substitution dictionary and create a string that shows the mapping. This will allow us to print out our different tables created from different shift values.

Listing 1-3. Printable Substitutions

```
# Partial Listing: Some Assembly Required

def printable_substitution(subst):
    # Sort by source character so things are alphabetized.
    mapping = sorted(subst.items())

    # Then create two lines: source above, target beneath.
    alphabet_line = " ".join(letter for letter, _ in mapping)
    cipher_line = " ".join(subst_letter for _, subst_letter in mapping)
    return "{}\n{}".format(alphabet_line, cipher_line)
```

Using these functions, we can build a simple application for encoding and decoding messages, shown in Listing 1-4.

Listing 1-4. Shift Cipher Application

```
# Partial Listing: Some Assembly Required

if __name__ == "__main__":
    n = 1
    encoding, decoding = create_shift_substitutions(n)
    while True:
        print("\nShift Encoder Decoder")
        print("--------------------")
        print("\tCurrent Shift: {}\n".format(n))
        print("\t1. Print Encoding/Decoding Tables.")
        print("\t2. Encode Message.")
        print("\t3. Decode Message.")
        print("\t4. Change Shift")
        print("\t5. Quit.\n")
        choice = input(">> ")
        print()

        if choice == '1':
            print("Encoding Table:")
```

```
20              print(printable_substitution(encoding))
21              print("Decoding Table:")
22              print(printable_substitution(decoding))
23
24          elif choice == '2':
25              message = input("\nMessage to encode: ")
26              print("Encoded Message: {}".format(
27                 encode(message.upper(), encoding)))
28
29          elif choice == '3':
30              message = input("\nMessage to decode: ")
31              print("Decoded Message: {}".format(
32                  decode(message.upper(), decoding)))
33
34          elif choice == '4':
35              new_shift = input("\nNew shift (currently {}): ".format(n))
36              try:
37                  new_shift = int(new_shift)
38                  if new_shift < 1:
39                      raise Exception("Shift must be greater than 0")
40              except ValueError:
41                  print("Shift {} is not a valid number.".format(new_
                    shift))
42              else:
43                  n = new_shift
44                  encoding, decoding = create_shift_substitutions(n)
45
46          elif choice == '5':
47              print("Terminating. This program will self destruct in 5
                seconds .\n")
48              break
49
50          else:
51              print("Unknown option {}.".format(choice))
```

The encoding and decoding program completed, the East Antarctica Truth-Spying Agency (EATSA) sends Alice and Bob off to their covert destinations hopeful that their communications, if intercepted, will not be readable by the West Antarctica Central Knights Office (WACKO).

The problem is this code is quite easy to break. Can you see why? There are all kinds of ways to figure it out by clever guessing. For example, try to break this:

```
FA NQ AD ZAF FA NQ FTMF UE FTQ CGQEFUAZ
```

Using a couple of simple two-letter words such as "if," "or," "in," "to," and so forth, it quickly becomes obvious that this phrase is

```
TO BE OR NOT TO BE THAT IS THE QUESTION
```

The preserved spaces make it easy to figure out. For this reason, real spies before modern cryptography would typically remove all of the spaces in their messages, like this:

```
FANQADZAFFANQFTMFUEFTQCGQEFUAZ
```

With this change, at least it isn't obvious where to try easy word substitutions. But even if Alice and Bob remove all spaces and punctuation, it is still trivial to break their codes. Although this code is so trivial it can be broken with pen and paper, we are going to write a Python program to crack it. Do you already see how? If so, go ahead and do it yourself. If not, keep reading!

The problem with the substitution cipher used by EATSA is that there are only 25 unique and effective shifts. You can easily construct a Python program to try all possible 25 combinations.

How do we know when we are using the same shift as Alice and Bob? We'll know it when we see it because it will be readable.

Let's switch sides in this Antarctic cold war and work for the West Antarctica Central Knights Office (WACKO). They know that spies have infiltrated their country, and they are monitoring for communications between those spies and EATSA. One of their counter-intelligence agents, code named "Eve," has just come across the following message:

```
FANQADZAFFANQFTMFUEFTQCGQEFUAZ
```

With this message, Eve *also* has intel that EA agents are using substitution ciphers. She decides to construct a program for encoding and decoding such messages. In an amazing coincidence, she constructs a Python program just like EATSA!

Running the program, she tries decoding the message with a shift of 1, producing this:

```
EZMPZCYZEEZMPESLETDESPBFPDETZY
```

That doesn't look right. So Eve tries again with shifts 2, 3, and so forth.

```
1:  EZMPZCYZEEZMPESLETDESPBFPDETZY
2:  DYLOYBXYDDYLODRKDSCDROAEOCDSYX
3:  CXKNXAWXCCXKNCQJCRBCQNZDNBCRXW
4:  BWJMWZVWBBWJMBPIBQABPMYCMABQWV
5:  AVILVYUVAAVILAOHAPZAOLXBLZAPVU
6:  ZUHKUXTUZZUHKZNGZOYZNKWAKYZOUT
7:  YTGJTWSTYYTGJYMFYNXYMJVZJXYNTS
8:  XSFISVRSXXSFIXLEXMWXLIUYIWXMSR
9:  WREHRUQRWWREHWKDWLVWKHTXHVWLRQ
10: VQDGQTPQVVQDGVJCVKUVJGSWGUVKQP
11: UPCFPSOPUUPCFUIBUJTUIFRVFTUJPO
12: TOBEORNOTTOBETHATISTHEQUESTION
```

Using a shift of 12, Eve sees a string of obviously English text. This is clearly the message.

This type of substitution cipher is often called a Caesar cipher because Julius Caesar used it for his secret messages [3]. This cipher is more than 2000 years old. Obviously, we've come a long way since then. This technology is quite obsolete.

Even so, there are a lot of principles of modern cryptography that can be discussed using the Caesar cipher, including

1. Key size
2. Block size
3. Preserved structure (structure that survives encoding)
4. Brute-force attacks

We will be learning about all of these concepts in this book in the context of modern cryptography. Mathematical advances have enabled new ciphers that are almost impossible to break *if used correctly*. Before we go on, though, here are a few additional exercises for the intellectually curious.

EXERCISE 1.2. AUTOMATED DECODING

In our example, Eve tried decoding various messages until she saw something that looked like English. Try automating this.

- Get a data structure containing a few thousand English words.[1]
- Create a program that takes in an encoded string, then try decoding it with all 25 shift values.
- Use the dictionary to try to automatically determine which shift is most likely.

Because you have to deal with messages with no spaces, you can simply keep a count of how many dictionary words show up in the decoded output. Occasionally, one or two words might appear by accident, but the correct decoding should have significantly more hits.

EXERCISE 1.3. A STRONG SUBSTITUTION CIPHER

What if instead of shifting the alphabet, you randomly jumbled the letters? Create a program that encodes and decodes messages using this kind of substitution.

Some newspapers publish puzzles like this called *cryptograms*.

EXERCISE 1.4. COUNT THE DICTIONARIES

How many substitution dictionaries are possible for the cryptogram-style substitution in the previous exercise?

[1]You can find lists of such words online, and your program can automatically populate your data structure with them.

EXERCISE 1.5. IDENTIFYING THE DICTIONARY

Modify your cryptogram program so that you can identify and pick the jumbled character substitution map with a number. That is, each mapping has a unique number that identifies it: picking substitution *n* should create the same substitution mapping every time. This exercise is a little tougher than the others. Do your best!

EXERCISE 1.6. BRUTE FORCE

Try having your cryptogram-decoding program brute force a message. How long would it take to test every possible mapping? Can you write a program that can speed this up with any kind of "smart guess"?

A Gentle Introduction to Cryptography

With the example out of the way, we are ready to get into some real cryptography. Welcome! Hopefully you had fun with the substitution cipher. As mentioned earlier, this particular form of encryption is called a "Caesar cipher" because it was used by Julius Caesar for protecting important documents.

Like Caesar, most of us have information that we would like to keep secret. In cryptography terms, we would like to keep it *confidential*. Encryption is a cornerstone of data confidentiality.

What do you think of Caesar's cipher? Even without a computer, how long do you think it would take you to break something like that? Perhaps in Caesar's time it was reasonably effective if Caesar's enemies were not well educated. This is an important lesson in cryptography and computer security. The effectiveness of cryptography is typically *dependent on context*. Good cryptography is effective no matter how well educated your adversaries are, how many computers they have, whether they know the algorithms you use, or how motivated they are.

In short, you're better off when you aren't too dependent on context, at least context that is out of your control.

Good security will *always* depend on *your choices*, however. The goal of this book is to help cryptographic beginners understand a little bit about how certain cryptographic algorithms work and a little bit about the contexts they are designed for. This book is directed at programmers and thus uses a lot of source code to teach and illustrate concepts. As we use the Python programming language, Python programmers will especially enjoy these exercises. However, the concepts are not language-dependent.

Thus, we assume some familiarity with programming. Python is easy enough to learn to read that it should be easy for anyone to at least follow the examples, and we try to stay away from very special Python idioms to facilitate that.

We do *not*, however, assume that the reader has any prior familiarity with cryptography. If you know cryptography a little, please be patient with some of the explanations in the book that may be directed to the absolute beginner. If you are a beginner, this book is for you. We hope that you enjoy getting your feet wet.

Uses of Cryptography

You are probably aware that cryptography is everywhere in today's modern interconnected world. The world's people are exchanging information in mind-boggling quantities and at mind-boggling speeds. A 2018 Forbes article reported the following statistics [10]:

1. 2.5 *quintillion* bytes of data are created each day, and that number is accelerating.
2. Google processes 3.5 billion searches each day.
3. Snapchat users share 500,000 photos *per second*.
4. More than 16 million text messages are sent every second.
5. More than 150 million email messages are sent every second.

What's amazing from an information security perspective is that the vast majority of these transmissions are meant to be protected in some way. There are nearly 4 billion users of the Internet at the time of this writing, but almost all of the data transmitted is meant for a vanishingly small percentage of them. Even when someone posts to social media publicly for the world to see, they are posting *to a specific platform*. The communication is meant for Facebook, or Twitter, or Snapchat, or Instagram *first*, and the platform then makes it available publicly.

Cryptography is the primary tool for protecting information. Cryptography can be used to help provide the following protections:

Confidentiality: Only authorized parties can *read* the protected information. This is probably the first thing that you think of when you think about encryption or secret codes.

Authentication: You know that you are talking to the *right entity/person* and that they have *not delegated their identity* (they're "present"). Many people know that the little lock icon in their browser means that their data is encrypted, but fewer know that it also means the service's *identity* (e.g., your bank) has been verified by a trusted authority. That is pretty important, after all: encrypting data to the wrong party doesn't really help.

Integrity: A message hasn't been *changed* between the sender and receiver. This applies equally to plaintext and to encrypted messages. It may seem unintuitive in some cases, but it is possible to change an encrypted message without being able to read it, even in ways that "make sense" to the receiver.

While there are a lot of books on cryptography, not many of them are focused on programming as the primary method of teaching the algorithms and associated principles. Our goal is to walk you, the computer programmer, through hands-on exercises that will help make these concepts understandable and useful.

What Could Go Wrong?

Unfortunately, there are a lot of ways to use cryptography incorrectly. In fact, there are a lot more ways to use it incorrectly than correctly. There are many reasons for this, but two that we will focus on here.

First, cryptography is based on a lot of pretty esoteric mathematics that most programmers and IT professionals have little experience with. You don't have to know the mathematics to use the cryptography, but sometimes not knowing the math behind it makes it difficult to have correct intuition about what will work and what will not.

Second, and perhaps the biggest problem, is that correct usage is also dependent on context. It is rare to find a universal "this is how you should always do it under all

circumstances" algorithm. A big part of learning cryptography is learning how various *parameter settings* impact the operation.

We will talk about this a lot in the book. In fact, many of your exercises will be to *break* cryptography that has been set up incorrectly. Looking at something break is a great way to understand how it works. It is also a lot of fun.

YANAC: You Are Not A Cryptographer

Warning This Section Is Critical. Please Read It Carefully

To repeat, there are more ways to mess up cryptography than you can possibly imagine. The pages of cryptography history are filled with stories of very smart people that unintentionally created vulnerable algorithms and systems. Many times, non-experts learned just enough to be dangerous and threw together a cryptography-based module that provided little more than a false sense of security. Even some of the very best cryptographic minds have had to correct their protocols after finding out they overlooked a subtle edge case.

If this book is your first exposure to cryptography, you will still not be an expert by the time you finish. This book will not prepare you to create algorithms and protocols that provide industrial strength protections. Please, *please*, do not finish reading this book and then think that you are ready to slap together your own custom cryptography for a real application.

Even for experts, the current best thinking in the cryptography community is to *not* create new or custom mechanisms. This is typically stated as, "Don't roll your own crypto." Instead, find and use existing libraries, protocols, and algorithms that have been heavily tested and are both well documented and consistently maintained. When new algorithms are truly needed, these are typically created and tested to within an inch of their lives by committees of experts, then presented for peer review and public comment before ever being trusted to protect sensitive data.

So why read this book at all? If only the experts should develop cryptography, why should non-experts learn this stuff?

First and foremost, cryptography is fun! Regardless of how ready you are to secure data communications between an app you write and a back-end server, learning cryptography is interesting, enjoyable, and worthwhile. Moreover, maybe after you get

a taste for it you will want to do the hard work required to *become* an expert yourself! Perhaps this book will be the first step in your journey to becoming a cryptography wiz!

Second, we live in an imperfect world. You may be working on a project where former contributors (unfortunately) did roll their own cryptography. If you are in that situation, you need to encourage your organization to replace it as quickly as possible. Such situations are like a land mine just waiting to go off and may require a significant financial investment to fix. Your organization may need to hire a cryptography consultant to investigate and assess the risks. Without giving advance notice to the bad guys, you may need to send mandatory security patches to all of your customers. As bad as this situation is, it is still better to discover it yourself than to wait for the bad guys to find it for you. Reading this book can help you to recognize these issues and make a preliminary assessment of what you are dealing with.

Third, even when you are using a reputable algorithm (or better yet, third-party library), it is helpful to understand the underlying cryptography principles at least a little bit. It is handy to know how to use cryptography and particularly how to set parameters of various cryptographic methods. There is a big push from some in the cryptography community to create libraries with APIs that require minimal configuration and are nearly impossible to use incorrectly (we will talk about an example of this later in the book). Even for these, however, if a weakness is found *inside* these black boxes, an informed user can better understand how that weakness affects the security of the system and thus better select mitigation strategies.

Finally, an informed user is better able to recognize good advice and trustworthy experts. Let's discuss this point a little more in the next couple of sections.

"Jump Off This Cliff"—The Internet

Most of us that write code depend heavily on the Internet. It is common to search for API documentation, example code, and even best practices. But please be cautious when searching the Web for recommendations about cryptography. Many answers are good, but many more are terrible. If you're not an expert, it can be hard to recognize the difference.

For example, some researchers published a research paper in 2017 entitled "Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security" [5]. They detailed over 4000 posts on the Stack Overflow web site that included security-related code snippets. After forensically examining 1.3 million Android

applications, they found that a full 15% included code copied from these posts, most of which were insecure to some degree or another.

One of the first things you can do is educate yourself about cryptography in practice, and this is one of our goals in writing this book. You do not have to be an expert to be well-informed. Most of you reading this book know enough about computer hardware to not get taken advantage of by a pushy salesman even though you aren't personally designing circuit boards. Similarly, knowing just a little more about cryptography fundamentals can help you recognize good advice from bad. And it can help you know when you can figure it out yourself and when you should get expert help.

The cryptodoneright.org Project

One of the authors is a founding member of the Crypto Done Right project. The goal of this project is to bring together in one place the very best in practical cryptographic guidance. At the cryptodoneright.org web site, we are creating and maintaining a collection of cryptography recommendations designed for software developers, IT professionals, and managers. The goal is to bridge the gap between the crypto experts that know all the crazy math and the users of cryptography that just need an application to communicate securely with a cloud-based server.

Anyone can submit or suggest an entry to Crypto Done Right, but an editorial board of the very best experts ensure correct content. At the time of this writing, editorial control is still located with the Johns Hopkins University, but moving this into an independent, community-driven organization is on the road map.

We encourage you to use this web site as an authoritative source on cryptographic best practices, and we endorse the content. As a general knowledge base, it will never have everything that everyone needs or answer every question about every application. But it is a good start to understand how cryptographic algorithms work, which parameters matter, and what common problems to avoid. If you are trying to figure out what to do with cryptography in your development project, start there and then branch out to other sources for more detailed recommendations applicable to your situation. Crypto Done Right can sensitize you to the relevant issues so that you can recognize which sources are trustworthy.

Enough Talk, Let's Sum Up

This book is a Python programming book. We will write a lot of very fun, very interesting code to learn about cryptography. To keep things interesting, we are going to rely on Alice, Bob, and Eve throughout the book. Computer security people actually talk about scenarios this way where "Alice" represents "Party A," Bob represents "Party B," and Eve represents the "Eavesdropper." There are sometimes other common names, but these will be our three most common actors.

We will motivate a lot of our examples using a hypothetical cold war between East and West Antarctica, which are *totally fictitious*. Please do not read anything political into any of this. We use Antarctica because it was the least political place we could think of. If we have inadvertently offended you, we apologize in advance.

Although the sample code is written to be entertaining, it is also written to be relevant and illuminating. Take time to play around with the examples. Try out your own experiments. Learn from positive and negative examples.

Please be very careful not to ever use sample "bad" code in your projects. Even the "good" code should not just be copied and pasted into applications without carefully deciding that it is appropriate.

The rest of the book is organized as follows:

In Chapter 2, we will get started with *hashing*. You are probably familiar with hashes to some degree or another already, but we will do some interesting experiments in brute-force attacks against a hash algorithm and even talk a little about Proof of Work like what is used in Bitcoin. From a security perspective, hashes are extremely important for password protection. They are also useful for file integrity and will make a reappearance in later chapters when we talk about message integrity and digital signatures.

In Chapter 3, we really get into encryption with a discussion of *symmetric encryption*. If you have heard of AES, that is an example of a symmetric encryption scheme. It's called "symmetric" because the same key that encrypts the data is used to decrypt the data. These algorithms are fast and used almost exclusively for encrypting most data whether in transit or on disk.

In contrast to symmetric algorithms, Chapter 4 dives into *asymmetric encryption*. This kind of cryptography involves two keys that work together. What one encrypts, the other decrypts. These types of algorithms are used in certificates and digital signatures, although in that chapter we will focus on the algorithms themselves.